yAudit Timeless Yield Daddy Review

Review Resources: None beyond the code repositories

Auditors:

- engn33r
- Benjamin Samuels

Table of Contents

1. Review Summary
2. Scope
3. Code Evaluation Matrix
4. Findings Explanation
5. High Findings
    a. 1. High - Incorrect order of operations in LibCompound.sol (engn33r)
        a. Technical Details
        b. Impact
        c. Recommendation
        d. Developer Response
    b. 2. High - User can set arbitrary `approveMaxIfNeeded()` target (engn33r)
        a. Technical Details
        b. Impact
        c. Recommendation
        d. Developer Response
    c. 3. High - Lack of slippage protection in Curve swaps (engn33r)
        a. Technical Details
        b. Impact
        c. Recommendation
        d. Developer Response

6 Medium Findings
  a 1. Medium - Accrued rewards may not be returned to depositors (engn33r)
    a Technical Details
    b Impact
    c Recommendation
    d Developer Response
  b 2. Medium - Bypass to add `cEtherAddress` to `underlyingToCToken` array (engn33r)
    a Technical Details
    b Impact
    c Recommendation
    d Developer Response
  c 3. Medium - Improper `wrapEthInput()` call can cause value loss (engn33r)
    a Technical Details
    b Impact
    c Recommendation
    d Developer Response
  d 4. Medium - ERC20 tokens sitting in contracts can be extracted (engn33r)
    a Technical Details
    b Impact
    c Recommendation
    d Developer Response
7 Low Findings
  a 1. Low - Difficult to use `withdraw()` with full deposit (engn33r)
    a Technical Details
    b Impact
    c Recommendation
    d Developer Response
  b 2. Low - Inaccurate function return values (engn33r)
    a Technical Details

    b Impact
    c Recommendation
    d Developer Response
c 3. Low - No fee-on-transfer token support (engn33r)
    a Technical Details
    b Impact
    c Recommendation
    d Developer Response
d 4. Low - Curve's `price_oracle()` may provide less effective sandwiching protection post-merge (Benjamin Samuels)
    a Technical Details
    b Impact
    c Recommendation
    d Developer Response
e 5. Low - Insufficient SwapArgs input validation (engn33r)
    a Technical Details
    b Impact
    c Recommendation
    d Developer Response
f 6. Low - Curve Swap design may generate unexpected NYT/PYT dust (Benjamin Samuels)
    a Technical Details
    b Impact
    c Recommendation
    d Developer Response
g 7. Low - xPYT deployment lacks trust mechanism (engn33r)
    a Technical Details
    b Impact
    c Recommendation
    d Developer Response

h 8. Low - Missing PYT/NYT lookup mechanism (engn33r)
  a Technical Details
  b Impact
  c Recommendation
  d Developer Response

8 Gas Savings Findings
  a 1. Gas - Replace require blocks with errors (engn33r)
    a Technical Details
    b Impact
    c Recommendation
    d Developer Response
  b 2. Gas - Declare variables internal, immutable, or constant when possible (engn33r)
    a Technical Details
    b Impact
    c Recommendation
    d Developer Response
  c 3. Gas - Remove `beforeWithdraw()` calls (engn33r)
    a Technical Details
    b Impact
    c Recommendation
    d Developer Response
  d 4. Gas - Replace duplicate code with internal function (engn33r)
    a Technical Details
    b Impact
    c Recommendation
    d Developer Response
  e 5. Gas - Declare functions external for gas savings (engn33r)
    a Technical Details
    b Impact

- c Recommendation
- d Developer Response

f 6. Gas - Use `claimComp()` with borrowers = false (engn33r)

- a Technical Details
- b Impact
- c Recommendation
- d Developer Response

g 7. Gas - Use unchecked if no underflow risk (engn33r)

- a Technical Details
- b Impact
- c Recommendation
- d Developer Response

h 8. Gas - Function arg should be `calldata` not `memory` (engn33r)

- a Technical Details
- b Impact
- c Recommendation
- d Developer Response

i 9. Gas - Move revert earlier in function (engn33r)

- a Technical Details
- b Impact
- c Recommendation
- d Developer Response

j 10. Gas - Internal function is cheaper than modifier (engn33r)

- a Technical Details
- b Impact
- c Recommendation
- d Developer Response

k 11. Gas - Zero check could save gas (engn33r)

- a Technical Details

- b Impact
- c Recommendation
- d Developer Response

l 12. Gas - Remove unnecessary approve (engn33r, Benjamin Samuels)

- a Technical Details
- b Impact
- c Recommendation
- d Developer Response

m 13. Gas - Unnecessary logic in `approveMaxIfNeeded()` function (engn33r)

- a Technical Details
- b Impact
- c Recommendation
- d Developer Response

9 Informational Findings

a 1. Informational - Direct transfer of aTokens impacts share value (engn33r)

- a Technical Details
- b Impact
- c Recommendation
- d Developer Response

b 2. Informational - aAMPL edge cases (engn33r)

- a Technical Details
- b Impact
- c Recommendation
- d Developer Response

c 3. Informational - Invert constant mask variables (engn33r)

- a Technical Details
- b Impact
- c Recommendation
- d Developer Response

d 4. Informational - Simplify logic in `maxDeposit()` and `maxMint()` (engn33r)
  a Technical Details
  b Impact
  c Recommendation
  d Developer Response
e 5. Informational - Add parentheses to avoid order of operations ambiguity (engn33r)
  a Technical Details
  b Impact
  c Recommendation
  d Developer Response
f 6. Informational - Inconsistent ERC4626 callback usage (engn33r)
  a Technical Details
  b Impact
  c Recommendation
  d Developer Response
g 7. Informational - Some hard coded variables might be better adjustable (engn33r)
  a Technical Details
  b Impact
  c Recommendation
  d Developer Response
h 8. Informational - Unclear upgrade mechanism (engn33r)
  a Technical Details
  b Impact
  c Recommendation
  d Developer Response
i 9. Informational - General lack of documentation and comments (engn33r)
  a Technical Details
  b Impact
  c Recommendation

d Developer Response

j 10. Informational - Typos (engn33r)

a Technical Details

b Impact

c Recommendation

d Developer Response

k 11. Informational - Aave LendingPool `getConfiguration()` can replace `getReserveData()` (engn33r)

a Technical Details

b Impact

c Recommendation

d Developer Response

l 12. Informational - Redundant dependency imports (engn33r)

a Technical Details

b Impact

c Recommendation

d Developer Response

m 13. Informational - Interface doesn't match mainnet contract (engn33r)

a Technical Details

b Impact

c Recommendation

d Developer Response

n 14. Informational - Consider upgrade to latest solidity release (engn33r)

a Technical Details

b Impact

c Recommendation

d Developer Response

o 15. Informational - Theoretical overflow of `convertToShares()` and `convertToAssets()` (engn33r)

a Technical Details

- b Impact
- c Recommendation
- d Developer Response

p 16. Informational - No zero address checks in constructor (engn33r)

- a Technical Details
- b Impact
- c Recommendation
- d Developer Response

q 17. Informational - Modify Gate.sol `claimYieldAndEnter()` function argument (engn33r)

- a Technical Details
- b Impact
- c Recommendation
- d Developer Response

r 18. Informational - CurveV2xPYT utilizes a non-TWAP price oracle (Benjamin Samuels)

- a Technical Details
- b Impact
- c Recommendation
- d Developer Response

s 19. Informational - Replace magic numbers with constants (engn33r)

- a Technical Details
- b Impact
- c Recommendation
- d Developer Response

t 20. Informational - No timelock or other protection around changing fee or fee recipient (engn33r)

- a Technical Details
- b Impact
- c Recommendation

- d Developer Response
- u 21. Informational - safeApprove is deprecated (Benjamin Samuels)
  - a Technical Details
  - b Impact
  - c Recommendation
  - d Developer Response
- v 22. Informational - Swapper repo foundry coverage failing (engn33r)
  - a Technical Details
  - b Impact
  - c Recommendation
  - d Developer Response
- w 23. Informational - StETHERC4626.t.sol test failing (Benjamin Samuels)
  - a Technical Details
  - b Impact
  - c Recommendation
  - d Developer Response

10 Final remarks

- a engn33r
- b Benjamin Samuels

Review Summary

Yield Daddy

Yield Daddy provides ERC4626 wrappers for common tokens from yield-generating protocols. This includes tokens received from Aave, Compound, Curve, and other lending protocols that provide users with tokens that increase in value over time.

The contracts of the Yield Daddy repo, xPYT repo, and swapper repo were reviewed over 16 days, 2 of which were used to create an initial overview of the contract. The code review was performed between August 31 and September 16, 2022. The code was reviewed by 2 auditors for a total of 60 review hours (engn33r 32 hours, Benjamin Samuels 28 hours). The

repository was under active development during the review, but the review was limited to the latest commit at the start of the review. This was commit 2c8f62681ab90325ede093ac9c9f94dbcc5f7e49 for the Yield Daddy repo, commit fabe9d7cb9700334b10a48ee5d081234eb98def9 for the xPYT repo, and commit 35db08558aa01e8473ada13c092e24857d01461b for the Swapper repo.

Scope

The scope of the review consisted of the following contracts at the specific commit:

- Yield Daddy Code Repo @ 2c8f62681ab90325ede093ac9c9f94dbcc5f7e49
  - AaveV2ERC4626.sol
  - AaveV2ERC4626Factory.sol
  - AaveV3ERC4626.sol
  - AaveV3ERC4626Factory.sol
  - CompoundERC4626.sol
  - CompoundERC4626Factory.sol
  - EulerERC4626.sol
  - EulerERC4626Factory.sol
  - StETHERC4626.sol
- xPYT Code Repo @ fabe9d7cb9700334b10a48ee5d081234eb98def9
  - CurveV2xPYT.sol
  - CurveV2xPYTFactory.sol
- Swapper Code Repo @ 35db08558aa01e8473ada13c092e24857d01461b
  - CurveV2Swapper.sol
  - CurveV2Juggler.sol

No other protocol ERC4626 wrappers are to be reviewed in Yield Daddy, such as the Uniswap wrappers present in the repositories at the time of review. Contracts from the main Timeless repo that are integrated with the xPYT and swapper repositories, including Gate.sol, were out of scope and not fully reviewed, although some interactions with in-scope contracts were examined. After the findings were presented to the Yield Daddy team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Yield Daddy and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	No access control modifiers are present besides the use of the `onlyOwner` modifier in one location. Governance has no special privileges, which limits centralization risks and attack vectors targeting privileged addresses.
Mathematics	Good	The only math involved in the custom code of the protocol involves adding and subtracting account balances. There is no reliance on complex math.
Complexity	Low	The scope of the review is roughly 1200 lines of code. The ERC4626 wrappers are very concise. However, the xPYT and Swapper repositories are relatively complex and are very tightly coupled with several other contracts that were out of scope of the audit.
Libraries	Average	An average number of solmate libraries are imported into the project, as well as the `BoringOwnable.sol` library. This is common practice, but relying on external dependencies always increases the risk of vulnerabilities in these dependencies.
Decentralization	Good	Only `ownerSetProtocolFee()` in Swapper.sol provides extra

Category	Mark	Description
		functionality based on an access modifier. The rest of the code is permissionless and does not rely on governance, whitelisting, or other special admin requirements.
Code stability	Good	No changes were made to the code in the review scope after the review started and the code did not have TODO comments indicating unfinished code.
Documentation	Low	There are no developer docs for the project and many functions are missing NatSpec comments. Custom logic, like the value comparisons in `maxWithdraw()` and `maxRedeem()` in Yield Daddy, are missing comments explaining why the values are being compared (in these specific functions, the reason for the comparison is an edge case of undercollateralized Aave aTokens). Developer docs should be created, explaining a typical user's interaction with the protocol. NatSpec comments should be added to all functions, even functions that are part of the ERC4626 specification.
Monitoring	Average	Events exist in all functions involving state changes except the Swapper swap-related functions.
Testing and verification	Average	Tests are implemented in foundry but could use improved coverage. Specifically, the Compound ERC4626 wrapper and factory is at 0% code coverage, CurveV2xPYTFactory.sol in xPYT is also at 0% coverage, and the swapper repository returns a `CompilerError: Stack too deep when compiling inline assembly` error when `forge coverage` is run. Some contracts such as `CurveV2Juggler` are only tested indirectly through other contract's testing. There are also few integration tests for testing the system against a live network.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements
- Gas savings
  - Findings that can improve the gas efficiency of the contracts
- Informational
  - Findings including recommendations and best practices

High Findings

1. High - Incorrect order of operations in LibCompound.sol (engn33r)

Changes made to the LibCompound.sol dependency modify the order of operations. The original version of this file on github does not have the same order of operations. This changes the result of the math, causing incorrect calculations.

Technical Details

The problematic computations are for the variables `interestAccumulated` and the return value.

`interestAccumulated` calculation from transmissions11/libcompound repo:

```
uint256 interestAccumulated = (borrowRateMantissa * (block.number -
accrualBlockNumberPrior)).mulWadDown(
            borrowsPrior
);
```

`interestAccumulated` calculation from local LibCompound.sol contract:

```
uint256 interestAccumulated =
            borrowRateMantissa * block.number -
accrualBlockNumberPrior.mulWadDown(borrowsPrior);
```

Made up example values:

- borrowRateMantissa = 1000000000
- block.number = 15460000
- accrualBlockNumberPrior = 15459990
- borrowsPrior = 500000000000000000000000 `interestAccumulated` result from transmissions11/libcompound: 5000000000000000 `interestAccumulated` result from local LibCompound.sol: 15452270005000000

Return value calculation from transmissions11/libcompound repo:

```
(totalCash + totalBorrows - totalReserves).divWadDown(totalSupply);
```

Return value calculation from local LibCompound.sol contract:

```
totalCash + totalBorrows - totalReserves.divWadDown(totalSupply);
```

Example values loosely taken from cDAI:

- totalCash = 400000000000000000000000000
- totalBorrows = 300000000000000000000000000
- totalReserves = 200000000000000000000000000
- totalSupply = 3100000000000000000 Return value result from transmissions11/libcompound: 219354838709677419354838709 Return value result from local LibCompound.sol: 693548387096774193548387097

Impact

High. Incorrect order of operations leads to different calculation results for `totalAssets()`, which impacts key value management functions including `convertToShares()` and `convertToAssets()`.

Recommendation

Use the latest version of the transmissions11/libcompound repo after confirming its math is correct.

Developer Response

Fixed in commit 0ea9347d360dc6d226fa134c9c73d025d2fc6aa5.

2. High - User can set arbitrary `approveMaxIfNeeded()` target (engn33r)

The Swapper.sol contract is written in a way that a malicious contract could be written that:

1. is approved for `type(uint256).max` of the token types that pass through Swapper.sol
2. calls `safeTransferFrom()` on these ERC20 tokens it has been approved for

To be clear, this in itself would not steal value from Swapper.sol. But the `nonReentrant` modifiers in CurveV2Swapper.sol indicate there is in theory a non-zero chance that an external contract may execute malicious code to call back into a CurveV2Swapper.sol function partway through a function's execution. If such an action were possible, the `nonReentrant` modifier would not protect against a malicious contract calling `safeTransferFrom()`.

Technical Details

This finding is not a complete attack chain, but the missing link of the chain depends on a hypothetical weakness in an external contract, which is out of scope of this review. In order for value to be stolen from Swapper.sol, a malicious actor would:

1. Create a malicious contract that is approved by Swapper.sol for ERC20 tokens that pass through Swapper.sol, which enables it to `transferFrom()` those tokens
2. Use an external call from Swapper.sol to call the malicious contract while Swapper.sol holds value to take that value before it can be sent to the user calling the contract

The first step is possible because the input argument to `approveMaxIfNeeded()` in Swapper.sol code is a user-specified argument. Not only that, but in a function such as `_swapXpytToUnderlying()`, the `args.gate` parameter approved for `args.xPYT` here is only used for one function call, `exitToUnderlying()`. This function could be implemented in a malicious contract as an empty function and the rest of the `_swapXpytToUnderlying()` function would still succeed, allowing the `args.gate` user-specific argument to be set to an address under the malicious user's control. Even worse, this `exitToUnderlying()` could be where the `transferFrom()` call happens, so if an innocent user specifies this value for the `args.gate` value (either due to a mistake, a Timeless frontend hack, or any other means), the value would be transferred to the malicious gate contract and the innocent user would

lose the value that was going to be sent to them.

It is not even necessary for the user to provide a malicious value for `args.gate` for this to become a problem. Multiple external calls exist within the functions of Swapper.sol, and if any of these are insecure or have an external call to an insecure contract, loss of value can occur even if the user's transaction has the proper function argument values set.

Impact

High. The complete attack chain is not defined in the scope of this review. However, there is a non-zero risk of removal of value from Swapper.sol despite the `nonReentrant` modifiers on the external functions due to approval of arbitrary addresses.

Recommendation

As described in finding Low #5, there is insufficient input validation on the SwapArgs `args` function argument. For this specific issue, it would be best to:

1. Confirm that `args.gate` is a legitimate gate contract (and not a malicious contract that resembles the gate contract) by querying the factory contract or another contract that maintains a whitelist of legitimate gate contracts. If there is no such whitelist, it should be created. For example, Uniswap maintains a mapping of legitimate token pairs.
2. Confirm that the other values related to the `approveMaxIfNeeded()` calls, namely `args.xPYT`, `args.pyt`, and `args.nyt`, are legitimate Timeless tokens (and not malicious contracts) with the same solution of querying the relevant factory contract that is described above for the `args.gate` argument. Also perform a check the `args.underlying` corresponds to the Timeless tokens and is not an unrelated token.
3. If added protection against a malicious contract calling `transferFrom` in an external call is needed, consider checking that the token balance held by the contract before the external call and after the external call is unchanged, reverting if this is not the case.

Developer Response

This is fine. We assume that input validation is done on the frontend/by the caller, so user foot-gunning by giving invalid/malicious parameters is not in scope. The permissionlessness of the contract is more important to us.

3. High - Lack of slippage protection in Curve swaps (engn33r)

The Curve `exchange()` function has a `min_dy` parameter to specify the minimum amount of the output token to receive. This value is set to zero, meaning there is no slippage

requirement for the swap. Lack of slippage protection can result in loss of value from frontrunning and backrunning.

Technical Details

The `exchange()` call in `_swap()` in CurveV2xPYT.sol sets a minimum output token quantity of zero. This means the swap will happen regardless of how imbalanced the pool is. The only check that is applied to the quantity of tokens received is validating the amount is greater than `minXpytAmountOut`. The value of `minXpytAmountOut` does not help with slippage and is not user customizable, it only helps avoid a case where a very small (near zero) amount of of xPYT is returned.

This is in contrast to where `pool.exchange()` is used in `_swapFromUnderlying()` in the swapper repository. There is a check of `tokenAmountOut < args.minAmountOut` in Swapper.sol to check the return value of `pool.exchange()`. Whether the `args.minAmountOut` value will be calculated properly in the user interface is outside the scope of this review, but at least users have the option to specify a slippage tolerance. Similarly, the output of the `swapAmountOut` return value from `_swapFromYieldToken()` is compared against `args.minAmountOut` in Swapper.sol.

Impact

High. Lack of slippage protection is likely to result in loss of value from MEV on larger swaps, especially because the `pound()` function can be called by anyone. The existing assumption is that MEV will cause this function to be called frequently so the extracted value amount is small. But this function may not be called frequently by users, which could lead to a sizeable loss of value when the function is finally called and MEV bots extract value.

Recommendation

Because this function is designed to be called relatively frequently, hard code a reasonable slippage tolerance for this external function depending on the pool that is traded. The cost/benefit of using keepers from the Keeper Network or Gelato should be examined, because it may preserve more value for users (less MEV extracting value from the `pound()` process).

Developer Response

This is by design.

The description

> The value of minXpytAmountOut does not help with slippage and is not user customizable, it only helps avoid a case where a very small (near zero) amount of of xPYT is returned.

is incorrect. minXpytAmountOut is computed using the EMA price oracle of the Curve pool to ensure the experienced slippage is not too high. It does not "only helps avoid a case where a very small (near zero) amount of of xPYT is returned", it works as a generalized & permissionless way to limit slippage & sandwiching attacks. Because pound() is permissionless by design, allowing the caller to specify slippage parameters does not help.

Medium Findings

1. Medium - Accrued rewards may not be returned to depositors (engn33r)

The `claimRewards()` function sends all accrued rewards for the vault to `rewardRecipient`. These rewards are liquidity mining incentives that sometimes provided in addition to the base APY of the pool. The contracts in the scope of review did not show how (or if) rewards from `rewardRecipient` would be returned to depositors. This is a possible rug pull vector where the contract deployer could steal value from users.

Technical Details

The `claimRewards()` function is an external function without access controls, so anyone can call it. When it is called, the liquidity mining incentive rewards accrued to the aTokens held by the ERC4626 vault are sent to the address `rewardRecipient`. The `rewardRecipient` address may be a previously audited contract that is out of scope of this review, but because it is not in the scope of this review, it is not clear whether users will receive rewards claimed from this function.

Impact

Medium. Depositors could lose accrued rewards that they expected to receive.

Recommendation

The protocol should make it clear how users receive rewards that are swept by the `claimRewards()` function. Rewards should be distributed to users fairly and the contract code should demonstrate how this fairness is calculated. If users that deposited in the

ERC4626 vault earlier than other users should receive more rewards, the code should illustrate this.

Developer Response

This is by design. We have considered building a reward distribution system inside the Yield Daddy wrapper contracts, but decided against it. The reason is that liquidity mining incentives are usually temporary, and it's not worth it to add significant complexity to the contracts just for it to be useless after the incentives are over. The reward tokens will be handled by Timeless as it sees fit, for example using them to incentivize yield token liquidity.

2. Medium - Bypass to add `cEtherAddress` to `underlyingToCToken` array (engn33r)

The CompoundERC4626Factory.sol constructor has an if statement to prevent `cEtherAddress` from being added to the `underlyingToCToken` array. This if statement does not exist in `updateUnderlyingToCToken()`, a function with no access controls, so any user can added `cEtherAddress` to `underlyingToCToken`, which bypasses the check in the constructor.

Technical Details

The CompoundERC4626Factory.sol constructor has an if statement that prevents `cEtherAddress` from getting added to the `underlyingToCToken` array. The `updateUnderlyingToCToken()` function has a for loop very similar to the constructor, but the if statement check is missing. Because `updateUnderlyingToCToken()` has no access controls, anyone can use `updateUnderlyingToCToken()` to add the cEtherAddress to the array. This will cause problems because the CompoundERC4626.sol code assumes ERC20 tokens when using imported SafeTransferLib functions and does not use `safeTransferETH()` for the case where ETH is the underlying asset.

Impact

Medium. The protocol is designed to support only ERC20 tokens and not ether. Adding cEther to the `underlyingToCToken` array could result in a denial of service problem when functions handling ether are called.

Recommendation

Add the same `if (address(cToken) != cEtherAddress)` check from the constructor to the `updateUnderlyingToCToken()` function to prevent `cEtherAddress` from being added to the `underlyingToCToken` array.

Developer Response

Fixed in commit 46e528a99773cf4060331cfe065cb1d781b2ff55.

3. Medium - Improper `wrapEthInput()` call can cause value loss (engn33r)

`wrapEthInput()` in Swapper.sol must be used in a multicall when an external contract is depositing ETH. If this function is called on its own or in a separate transaction, this can cause loss of value for the user who called `wrapEthInput()` improperly. A footgun of this type should have the risks of improper use more clearly documented or add protections against improper use.

Technical Details

If a user calls `wrapEthInput()` as a separate transaction, ether will be converted to WETH and deposited to the Swapper.sol address. Even if the user intends to call `swapUnderlyingToXpyt()` or `swapUnderlyingToNyt()` in their next transaction, the user can be frontrun. Whoever calls `swapUnderlyingToXpyt()` or `swapUnderlyingToNyt()` while Swapper.sol holds value can extract that value because those functions use `args.underlying.balanceOf(address(this))` for the value of `tokenAmountIn` (here and here).

Impact

Medium. The risk only exists if a user improperly uses the protocol, but the protocol could add additional protections to avoid the improper usage scenario.

Recommendation

Two options are possible depending if the developers want to remove the footgun entirely or leave the footgun with a better warning sign:

1. To remove the footgun from the code, create `_swapUnderlyingETHToXpyt()` and `_swapUnderlyingETHToNyt()` functions that are specifically designed to handle the case where ether is sent to Swapper.sol as the underlying asset. This is the approach taken by Uniswap, which has `swapExactTokensForETH()` and `swapETHForExactTokens()` functions.
2. While not the recommended approach, the code can remain as is if the users are assumed to be trusted, but clearer warnings around `wrapEthInput()` should be added to indicate that this function can lead to loss of value if used improperly. If or when a website with developer documentation is created, add a colored “Warning” text box (similar to those in the solidity documentation) when documenting this function.

Developer Response

This is fine.

4. Medium - ERC20 tokens sitting in contracts can be extracted (engn33r)

Any ERC20 tokens left in Swapper.sol can be extracted. Any unaccounted for `asset` tokens in xPYT.sol can be extracted with `sweep()`. This is similar to Uniswap's V2 router, which is not intended to "hold token balances" per Uniswap documentation. This is unlikely a concern because tokens are not designed to be stored in the Swapper contract, but if the protocol is misused it can lead to loss of value for users.

Technical Details

The Swapper.sol contract uses the code `args.TOKEN.balanceOf(address(this))` in several places. The value of `args.xPYT`, `args.pyt`, or `args.nyt` can be any address, which means any token balance stored by the contract can be queried, and soon after extracted, with this code. The contract is not designed to store value, but if it does store value (say if a user accidentally sends tokens directly to the contract), the value could be extracted in a backrun transaction. A similar scenario exists with Uniswap's V2 router, which is not designed to store any token balances but it does have non-zero token balances as seen on etherscan.

Separately, `sweep()` in xPYT.sol is described with `Uses the extra asset balance of the xPYT contract to mint shares`. This can remove any extra assets of token `asset` that sits in the xPYT.sol contract. While less generalize than the Sweeper.sol contract, a more specific case of the same issue exists.

Impact

Medium. No edge case that forcibly traps tokens in the Swapper contract was found so it is unlikely a user will expect a different result.

Recommendation

Clearly document that the Swapper contract should not store any token balances, ideally in developer documentation that is more reader friendly that reading NatSpec comments directly from the source code.

Developer Response

This is fine.

Low Findings

1. Low - Difficult to use `withdraw()` with full deposit (engn33r)

The `withdraw()` function allows users to specify the amount of assets they wish to withdraw. If a user wants to withdraw all the assets they have deposited in Yield Daddy without leaving dust, it can be hard to accurately specify their full assets value because of the constantly rebasing nature of Aave aTokens.

Technical Details

Aave allows users to submit the value `type(uint256).max` to withdraw the user's full balance. No similar feature is present in the Yield Daddy ERC4626 vault. This makes it hard to use `withdraw()` while specifying the correct value of assets to remove all aTokens from the vault because the aTokens are rebasing and increasing over time. In contrast, the `redeem()` function allows a user a specify the number of shares they want to withdraw. Shares are not always rebasing and are easier for a user to specify an accurate number for a complete withdrawal.

Impact

Low.

Recommendation

If the users are intended to call `redeem()` instead of `withdraw()` when withdrawing their entire vault deposit, this should at minimum be made clear in the documentation. If it should be possible to allow users to call `withdraw()` to receive their entire deposit, consider replacing this line in `withdraw()` with the following code:

```
shares = assets == type(uint256).max ? balanceOf[owner] : previewWithdraw(assets); //
No need to check for rounding error, previewWithdraw rounds up.
```

A similar feature to redeem all shares may be useful for `redeem()`. Although it may not be worth the gas cost because a user's shares do not rebase in the same way that the underlying aTokens do, so `redeem()` can already allow a relatively simple way of withdrawing the entire deposit for a user.

Developer Response

This is fine. The recommendation would make the wrappers not conform to the ERC-4626

specs.

2. Low - Inaccurate function return values (engn33r)

`maxDeposit()` and `maxMint()` return `type(uint256).max` even though that value may not be possible to deposit or mint without a revert. Most likely these functions were copied from a default ERC4626 implementation and not customized for the underlying tokens.

Technical Details

The `maxDeposit()` and `maxMint()` functions only return the value `type(uint256).max` when the underlying pool can accept a deposit or mint. The value `type(uint256).max` does not accurately represent the maximum value that can be deposited into the vault when some ERC4626 tokens have already been minted. This is because the vault normally can mint a maximum of `type(uint256).max`, otherwise the `totalSupply` state variable will overflow and cause a revert, so the existing token supply should be subtracted. The same is true for aTokens, which should have the existing supply excluded. EIP4626 defines this requirement for `maxMint()`:

> MUST return the maximum amount of shares mint would allow to be deposited to receiver and not cause a revert, which MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary).

The output of `maxDeposit()` cannot be passed into `withdraw()` without a revert, which does not satisfy this ERC4626 requirement. A more accurate return value for `maxDeposit()` in Aave V2 and Aave V3 is:

```
return type(uint256).max - aToken.totalSupply();
```

The `totalSupply` value limitation of the ERC4626 vault can be ignored because the `totalSupply` of the vault will always be less than or equal to `aToken.totalSupply()`, so the aToken totalSupply is the limiting factor.

The same issue is relevant for `maxMint()`, but the return value of `maxMint()` should be in units of shares instead of assets because the ERC4626 `mint()` function takes a shares value as input. The more accurate return value for `maxMint()` in Aave V2 and Aave V3 is therefore:

```
return convertToShares(type(uint256).max - aToken.totalSupply());
```

This issue is present in the Compound ERC4626 wrapper as well. A similar solution should be applied.

Impact

Low. The `maxDeposit()` and `maxMint()` are view functions that do not directly impact any state variables, but the value returned from these functions does not match the intended purposes of these functions.

Recommendation

Use the above code instead of returning the constant `type(uint256).max` in AaveV2ERC4626.sol and AaveV3ERC4626.sol. If a vault does not allow a max of `type(uint256).max` in deposits, use custom math for that vault, which is already done with the supplyCap in AaveV3ERC4626.sol but not used for Aave V2. Fix the Compound wrapper with a similar approach.

Developer Response

This is fine. While the ERC-4626 specs are not completely clear about what to do in this case, I've consulted transmissions11 and the return values of maxMint() and maxDeposit() are only supposed to differ from type(uint256).max when the ERC-4626 contract itself (or the contracts it interacts with) implements an explicit limit on deposits. Implicit limits, such as it being impossible to deposit a token with an amount greater than the token's total supply, or it being impossible to deposit enough tokens to make the wrapper's total supply exceed 256 bits, should not be reflected.

3. Low - No fee-on-transfer token support (engn33r)

The `beforeWithdraw()` and `afterDeposit()` functions are not implemented with fee-on-transfer tokens in mind. If one of the supported lending platforms adds a fee-on-transfer token, or an existing supported tokens enables the fee-on-transfer feature, some ERC4626 wrapper functions in yield daddy will revert. The same lack of fee-on-transfer support exists in the swapper repo.

Technical Details

Fee-on-transfer tokens have the property where the value that is sent is not the same as the value that is received. This difference in logic compared to standard ERC20 tokens

requires special handling in solidity to avoid a revert or loss of value. No fee-on-transfer ERC20 appears to exist on the supported lending platforms at the time of the review, but if one is added in the future or if the fee is enabled on an existing token that supports fee-on-transfer, the ERC4626 wrappers for protocols that require two separate transfers for depositing and withdrawing (Euler, Compound, Aave only for depositing) would not function with fee-on-transfer tokens because the deposit or withdraw process would revert because the fee is not accounted for in between the two transfers.

The swapper repo also lacks fee-on-transfer support. This is because the `tokenAmountIn` value used in the `safeTransferFrom()` call is assumed to equal the value held by the Swapper.sol contract after the transfer takes place. This will revert when `args.gate.enterWithUnderlying()` is called because the `underlying.safeTransferFrom()` call in Gate.sol's `enterWithUnderlying()` will not have sufficient balance to transfer.

Impact

Low. It is unlikely that fee-on-transfer tokens would be added to these lending protocols, but if this does occur, the ERC4626 wrapper as designed would not support these fee-on-transfer tokens.

Recommendation

Add support for fee-on-transfer tokens if these wrappers are intended to remain in use without modifications for an extended period of time. A solution similar to Uniswap's `swapExactTokensForTokensSupportingFeeOnTransferTokens()` could enable this support. Otherwise, be mindful that future changes to tokens supported by the lending protocols could cause compatibility issues with yield daddy.

Developer Response

This is fine. We have zero intentions to support fee-on-transfer tokens.

4. Low - Curve's `price_oracle()` may provide less effective sandwiching protection post-merge (Benjamin Samuels)

CurveV2xPYT uses Curve's `price_oracle()` function to prevent swaps from being sandwiched.

Curve's `price_oracle()` function is implemented using an Exponential Moving Average (EMA) filter. This filter prevents MEV extractors from manipulating the result of the price oracle, since doing so would require either a prohibitive amount of capital, or full control

over multiple blocks in a row.

Technical Details

In the upcoming Merge of the Ethereum mainnet, block proposers and validators are assigned randomly 2 epochs in advance. This enables an attacker with a sufficient amount of capital to fully manipulate the contents of two or more blocks in a row, undermining previous the security assumption that arbitrageurs could disrupt a multi-block oracle manipulation attack.

This risk is compounded by price oracles that put more weight on recent observations, such as Curve's EMA oracle.

Impact

Low. Given the undermining of several security assumptions in TWAP/EMA price oracles, it is possible that given the correct liquidity conditions and validator assignments, an attacker can sufficiently manipulate an oracle to enable an effective, multi-block sandwich attack.

While this attack has not been demonstrated in the wild yet, research is ongoing on its viability.

Recommendation

An alternate mechanism for permissionless compounding is to use Keep3r Network, which operates semi-permissionlessly and can be configured to require transactions be routed through Flashbots for sandwiching protection. Given the lack of demonstrated attacks, implementing a measure such as Keep3r may be better left off to later protocol versions.

Developer Response

Acknowledged.

5. Low - Insufficient SwapArgs input validation (engn33r)

`swapUnderlyingToNyt()` and `swapUnderlyingToXpyt()` in CurveV2Swapper allows a user to input a SwapArgs struct that could be custom crafted. There is no validation of the data from this struct, which could lead to unexpected consequences.

Technical Details

There is no validation in `_swapUnderlyingToNyt()` that confirms that the addresses of `args.underlying`, `args.nyt`, `args.pyt`, `args.xPYT`, and `args.vault` are all related to the same underlying asset. Similarly, the value `args.gate` may not hold the address of the actual

Gate.sol contract in the timeless protocol, but instead may be the address of an attacker's contract. An external call is made to `args.gate` in `_swapUnderlyingToNyt()` and `_swapUnderlyingToXpyt()`, and although the external functions `swapUnderlyingToNyt()` and `swapUnderlyingToXpyt()` have the nonReentrant modifier, there is no global reentrancy lock across the Timeless protocol(s), so the risk of a hack involving reentrancy is not zero.

Impact

Low. No direct attack vectors were observed, but lack of user input validation is often a cause for concern.

Recommendation

Add multiple validation checks to confirm the SwapArgs function argument complies with certain assumptions. When possible, do not trust the function argument fully, but instead derive the necessary values using external calls to look up the necessary values using `args.underlying` or similar values.

To reduce the number of inputs that need to be validated, at least three parameters in the `SwapArgs` struct can be removed and derived from existing parameters:

1. `args.gate` is unnecessary because the proper gate can be derived using `args.xPYT.gate` or from `args.pyt.gate`
2. `args.vault` can be derived from `args.xPYT.vault` or from `args.pyt.vault`
3. `args.nyt` can be derived from `args.xPYT.nyt`

Developer Response

This is by design, validation is not present to save on the gas costs of swapping.

6. Low - Curve Swap design may generate unexpected NYT/PYT dust (Benjamin Samuels)

`CurveV2Swapper` will leave behind unexpected dust due the use of the off-chain oracle implemented by `CurveV2Juggler`.

Technical Details

According to the documentation, CurveV2Swapper's `swapNytToUnderlying()` and `swapXpytToUnderlying()` functions are intended to be used in conjunction with `CurveV2Juggler.juggleXpytInput()` and `CurveV2Juggler.juggleNytInput()` respectively. These two juggle function are intended to operate as off-chain oracles which are used to

calculate the correct quantities of NYT/PYT tokens to be swapped. The goal of the swap is for the user to end up with a nominally equivalent number of NYT+PYT tokens for a vault, which are then redeemed in equal amounts for xPYT tokens, which is then redeemed for the underlying vault's token.

The decoupling of the off-chain price oracle with the on-chain swap introduces a chance that the swap is executed at a different price that the price oracle expected. If a swap is executed at a different price than the oracle predicted, then the user will have differing quantities of NYT and PYT, and will have some amount of dust remaining after they redeem their NYT+PYT for xPYT.

Impact

Low. As usage of the protocol increases, an increasingly large fraction of NYT/PYT redemption flows utilizing Curve swaps will generate NYT or PYT dust.

In scenarios where the Curve pool's parameters are being modified (gamma, fees, amp parameter, etc.), *all* swap-based redemption flows will generate dust.

Recommendation

CurveJuggler currently implements a numeric solution for calculating how much NYT/PYT should be swapped. This approach consumes a large amount of gas, which is why it can only be used as an off-chain oracle. If CurveJuggler implemented an analytic solution, gas consumption may be reduced enough that the oracle can be consulted from the on-chain swap transaction.

If an analytic solution is not possible, then an alternate AMM should be considered for NYT/PYT swaps.

Developer Response

This is fine, since the extra dust is returned to the user. There is no analytical solution for Curve swaps, as even Curve itself uses Newton's method to approximate the solution.

7. Low - xPYT deployment lacks trust mechanism (engn33r)

xPYT.sol is designed to allow for many xPYTs to exist for a single underlying vault. `deployCurveV2xPYT()` in CurveV2xPYTFactory.sol does not use CREATE2 to deploy the xPYT contract, not does it store an array or mapping of deployed xPYT contracts. This means there is no way to determine whether an address is a legitimate xPYT contract deployed

by CurveV2xPYTFactory.sol or a malicious contract deployed by a third party with the same function names and interfaces. Any contracts that take a xPYT address as a function argument, including third-party contracts that want to integrate with xPYT tokens, cannot validate that the address is legitimate and must treat the contract as potentially malicious. This also means that a xPYT contract call could add risk of reentrancy or other attack vectors.

Technical Details

Often the CREATE2 opcode is used to deploy contracts that depend on underlying parameters, because the address of the contract can be derived from those underlying parameters. This is the approach used in `deployYieldTokenPair()` of Factory.sol to deploy new NYT/PYT pairs and by Uniswap V2 to deploy new pairs. But because multiple xPYT tokens can exist for a single underlying vault, perhaps CREATE2 cannot be used in this case. There is no state variable in CurveV2xPYTFactory.sol that stores the address of deployed xPYT contracts, which means that functions that accept a xPYT address as a function argument, including several functions in Swapper.sol that have a `SwapArgs.xPYT` function argument, cannot determine whether the xPYT address is a legitimate xPYT token or a malicious contract. This is unlikely to impact most users who use a web interface in their browser, but it may open up an attack vector of a malicious contract posing as a legitimate xPYT token.

Impact

Low. No specific proof of concept attacks were found during the allocated review time, but the lack of trust for xPYT contracts is likely a concern that should be mitigated.

Recommendation

Use CREATE2 to deploy new xPYT contracts if possible, otherwise add a mechanism to validate whether an address is a legitimate xPYT token. A mapping of xPYT addresses stored in `CurveV2xPYTFactory.sol` is a reasonable solution for this, as found in Uniswap V2.

Developer Response

Fixed in commit 5bcc2e5f053b7e7c31610b532f29720625429aa2.

8. Low - Missing PYT/NYT lookup mechanism (engn33r)

There is no good way to list all valid PYT/NYT token pairs. `deployYieldTokenPair()` has no access controls so anyone can call the function with their own vault and a malicious gate-like contract. If a frontend or bot creates a list of existing PYT/NYT tokens by collecting all

the `DeployYieldTokenPair` events emitted from `deployYieldTokenPair()`, the list would then need to be filtered to contain only trusted `gate` addresses and validate the gate type corresponded to the vault type.

Note that this finding is outside the scope of this review, but was still highlighted.

Technical Details

The PerpetualYieldToken.sol and NegativeYieldToken.sol contracts contain many external calls to the gate contract, notably the `gate.beforePerpetualYieldTokenTransfer()` call in PYT's `transfer()` and `transferFrom()` functions and the gate-protected `gateMint()` and `gateBurn()` in BaseERC20.sol. If the gate is malicious, it could leverage these callbacks for a variety of attack vectors.

Impact

Low. A malicious gate address can be specified which increases the risk of several attack vectors.

Recommendation

A whitelist of valid gate addresses or a similar mechanism should be added to Factory.sol to prevent PYT/NYT token pair creation with arbitrary gate addresses. An additional check should exist to validate that the vault is matched with a proper gate, otherwise an ERC20Gate may be paired with a ERC4626 token. For reference, Uniswap V2's `createPair()` function only allows users to specify the two token addresses and nothing else, so users only need to trust the underlying tokens. To interact with Timeless PYT/NYT tokens as it is designed now, the user must trust the vault AND the gate, PLUS verify the gate type and vault type match.

Developer Response

This is fine. Gate contracts are immutable, so malicious gate contracts are easy to avoid. We do not intend to add whitelists for contracts in order to keep the protocol permissionless. There's no good way that can ensure a vault matches the Gate type on the smart contract level, it is assumed that the validation is done in the frontend; furthermore, there's no incentive for people to deploy yield tokens for a mismatched Gate and vault pair.

Gas Savings Findings

1. Gas - Replace require blocks with errors (engn33r)

One require block can be replaced with a custom error for gas savings. Errors were introduced in Solidity 0.8.4 and offer a gas savings. Custom errors are already used elsewhere in the project but are not applied everywhere.

Technical Details

Instances of require that can be replace with custom errors are found in:

- AaveV2
- AaveV3
- xPYT

Impact

Gas savings.

Recommendation

Use solidity custom errors instead of require statements.

Developer Response

This is fine.

2. Gas - Declare variables internal, immutable, or constant when possible (engn33r)

Declaring a constant with internal visibility enables a cheaper contract deployment than public constants because public variables automatically receive a getter function.

Technical Details

Many public immutable variables exist in the Euler, Aave V2, Aave V3, and Compound wrappers. If possible, make these internal instead of public for gas savings.

There is one variable in xPYT.sol, `assetBalance`, that can be made private because it already has a public getter function in the form of `totalAssets()`.

A variable in CurveV2xPYT.sol, `curvePool`, can be made immutable because it is only changed once, in the initializer.

Impact

Gas savings.

Recommendation

Make variables internal or private for gas savings on contract deployment.

Developer Response

It's not possible to make curvePool in CurveV2xPYT immutable, as it would create a circular dependency between the xPYT contract and the Curve pool. We're ok with the other variables being public.

3. Gas - Remove `beforeWithdraw()` calls (engn33r)

An unnecessary function call can be removed.

Technical Details

`withdraw()` and `redeem()` call `beforeWithdraw()`, but `beforeWithdraw()` is never implemented with code. `beforeWithdraw()` is declared in ERC4626.sol as an empty virtual function. The calls to `beforeWithdraw()` can be removed because it doesn't run any code.

Impact

Gas savings.

Recommendation

Remove function calls to functions that are never implemented, specifically this line and this line.

Developer Response

This is by design. It makes it easier for other contracts to inherit from it and still implement withdraw hooks.

4. Gas - Replace duplicate code with internal function (engn33r)

Certain code is duplicated. This code can be replaced with an internal function that could reduce gas costs on deployment.

Technical Details

The code below is found in two places in AaveV2ERC4626.sol (here and here):

```
// check if pool is paused
if (lendingPool.paused()) {
            return 0;
}
```

```
// check if asset is paused
uint256 configData = lendingPool.getReserveData(address(asset)).configuration.data;
if (!_getActive(configData)) {
            return 0;
}
```

This code could be moved into an internal view function to reduce contract deployment costs. The same could be done with the similar duplicate code block (here and here). The Aave V3 wrapper has duplicate code in the same functions of `maxDeposit()`, `maxMint()`, `maxWithdraw()`, and `maxRedeem()`.

Impact

Gas savings.

Recommendation

Replace duplicate code with internal functions to reduce contract deployment costs.

Developer Response

This won't save gas since it's in a view function.

5. Gas - Declare functions external for gas savings (engn33r)

The boostedBalanceOf function and deposit function of Gauge.sol should be declared external, not public, for gas savings.

Technical Details

There is a public function `updateUnderlyingToCToken()` in CompoundERC4626Factory.sol that can be made external.

Impact

Gas savings.

Recommendation

Declare functions as an external functions for gas savings.

Developer Response

Fixed in commit 6722541c7ae9a2c152bc90dcdb39f204af809312.

6. Gas - Use `claimComp()` with borrowers = false (engn33r)

There are three `claimComp()` functions in the Compound Comptroller contract. The one called by CompoundERC4626.sol sets `borrowers = true` and `suppliers = true`. Because the vault does not borrow from Compound, the `borrowers` bool can be set to `false`, which will save gas by skipping logic in this `claimComp()` function.

Technical Details

`claimComp()` is called in `claimRewards()`. It can be replaced with a call to the other `claimComp()` function with `borrowers` set to `false`.

Impact

Gas savings.

Recommendation

Replace this call to `claimComp()` with the following code:

```
address[] memory holders = new address[](1);
holders[0] = address(this);
comptroller.claimComp(holders, cTokens, false, true);
```

Developer Response

Fixed in commit a53c2333f40a7e3056a7dfeec3fa994e4da37993.

7. Gas - Use unchecked if no underflow risk (engn33r)

There is a subtraction that can be made unchecked for gas savings.

Technical Details

There are two subtraction operations (here and here) that should not underflow because the `supplyCap` should be greater than the value of `totalSupply()`. The Aave V3 code should be checked to confirm that there is a valid comparison to `supplyCap` when the supply increases.

Another subtraction operation can be made unchecked in CurveV2Juggler.sol (and the out of scope UniswapV3Juggler.sol contract), found here and here. This operation can be unchecked because `tokenAmountIn >= (tokenAmountIn >> 1)`, and because `swapAmountIn = (tokenAmountIn >> 1)`, then `tokenAmountIn >= swapAmountIn` and `tokenAmountIn - swapAmountIn >= 0`.

Swapper.sol has two operations repeated four times that can be unchecked to save gas. A similar operation that already is unchecked is found in xPYT.sol.

1. Division here, here, here, and here can be unchecked because `(protocolFeeInfo_.fee / 10000) < 1` so it cannot overflow.

```
uint256 feeAmount = (tokenAmountIn * protocolFeeInfo_.fee) / 10000;
```

2. Subtraction here, here, here, and here can be unchecked because `feeAmount < tokenAmountIn` so it cannot underflow.

```
tokenAmountIn -= feeAmount;
```

Impact

Gas savings.

Recommendation

Use unchecked where there is no overflow or underflow risk.

Developer Response

Many of the suggested operations that can be made unchecked are in view functions meant to be called off chain, so gas optimization is meaningless for them.

For the first point, the intermediate multiplication may overflow, so unchecked cannot be used here.

The second point has been fixed in commit 9fbd20056c2bf6261524741d62ddeef2c6067d08.

8. Gas - Function arg should be `calldata` not `memory` (engn33r)

Array function arguments that are `calldata` are cheaper to read than arguments that are `memory`.

Technical Details

The Compound factory contract has a `memory` argument that can be changed to `calldata` for gas savings.

The same change can be made for `_deployCurvePool()` in CurveV2xPYTFactory.sol.

Impact

Gas savings.

Recommendation

Replace `memory` with `calldata` for the Compound factory function.

Developer Response

The point about the Compound factory is fixed in commit 6722541c7ae9a2c152bc90dcdb39f204af809312.

It makes no sense to update _deployCurvePool(), since the coins array is indeed initialized in memory and not from calldata.

9. Gas - Move revert earlier in function (engn33r)

If a function reverts, the remaining gas is returned to the user. This means that the earlier a function reverts, the more gas that could be returned to the user. Revert checks should happen as early in the function as possible.

Technical Details

This revert check in the xPYT constructor can take place earlier in the constructor because it only relies on a function argument and does not rely on any other calculations. Move this if statement to immediately follow the other if statement revert check.

Impact

Gas savings.

Recommendation

Move revert checks earlier in functions by combining the related checks into one:

```
if (minOutputMultiplier_ > ONE || pounderRewardMultiplier_ > ONE) {
            revert Error_InvalidMultiplierValue();
}
```

Developer Response

This is fine.

10. Gas - Internal function is cheaper than modifier (engn33r)

One access control modifier in Swapper.sol could be replaced with an internal function for gas savings.

Technical Details

This onlyOwner modifier could be replaced with a require check in this function or an internal function. For example, add this line to `ownerSetProtocolFee()` to replace the modifier on the function.

```
require(msg.sender == owner, "UNAUTHORIZED");
```

Impact

Gas savings.

Recommendation

Replace modifiers with internal functions or require checks for gas savings.

Developer Response

Acknowledged.

11. Gas - Zero check could save gas (engn33r)

There are two places where the claimable yield is calculated, but in one place the code does not confirm that `yieldAmount != 0`. Applying this zero check in both places where yield is calculated can provide a gas savings when `yieldAmount` is zero. While this finding relies on code outside the scope of this review, the in-scope contacts use this function, so it was reviewed in excess of the requested scope.

Technical Details

`getClaimableYieldAmount()` is called once in xPYT.sol, but this function is missing a zero check for `yieldAmount`. The first function that calculates the claimable yield, `claimYieldAndEnter()`, confirms that `yieldAmount != 0`. The other locations where the claimable yield is calculated, `getClaimableYieldAmount`, does not confirm that `yieldAmount != 0`. This zero check could provide a gas savings in the case that `yieldAmount` is zero.

Impact

Gas savings.

Recommendation

Add the missing zero check. A longer term improvement would be to write the yield calculation logic only in one place inside a function and call this function as needed. This would avoid similar pieces of code with minor differences that could result in unexpected differences in behavior.

Developer Response

Acknowledged.

12. Gas - Remove unnecessary approve (engn33r, Benjamin Samuels)

There is an unnecessary `approve()` call in CurveV2xPYT.sol which can be removed.

Technical Details

This `approve()` call is designed to initialize the storage slot to a non-zero value in order to save gas on future changes of this slot. The problem is that the `approve()` call that happens before each swap overwrites the initial allowance value, which is subsequently consumed by the Curve swap and re-set to zero.

Impact

Gas savings.

Recommendation

In order for the initially approved value of 1 to remain after the swap's `transferFrom()` call, the `increaseAllowance()` function can be used as implemented in OpenZeppelin's ERC20 implementation.

Alternatively, the approval at line 93 can be modified so it grants an approval on nytAmountIn + 1 wei;

```
nyt.approve(address(curvePool_), nytAmountIn + 1);
```

Developer Response

Fixed in commit 59c3d152da5c4af50b24edbce68db053c6bbac05.

13. Gas - Unnecessary logic in `approveMaxIfNeeded()` function (engn33r)

The `approveMaxIfNeeded()` differs from a simple `token.safeApprove(spender, type(uint256).max);` call by setting the allowance to zero first if the existing allowance is non-zero. This is to handle the edge case of USDT, which requires this series of actions. Because there is no other approve in the contracts that use `approveMaxIfNeeded()`, these

extra steps are not necessary because the only approve that can happen is approving `type(uint256).max`.

Technical Details

The contracts that call `approveMaxIfNeeded()` include Swapper.sol and CurveV2Swapper.sol. No other `approve()` or `safeApprove()` calls happen in these contracts, so the only allowance being set is an allowance of `type(uint256).max`. If the allowance is set to `type(int256).max`, USDT will not decrease the allowance when a transfer happens. So the process of checking the allowance and setting it to zero if the value is not `type(int256).max` is unnecessary and can be replaced with a simple `token.safeApprove(spender, type(uint256).max);` call. There might be an edge case other than USDT that the `approveMaxIfNeeded()` was designed for, but the function should not be necessary for USDT or standard ERC20 tokens.

Impact

Gas savings.

Recommendation

Remove unnecessary checks in `approveMaxIfNeeded()` or use a simpler `safeApprove()` if approval value is never set to a value other than `type(uint256).max`.

Developer Response

Fixed in commit 14c6928f9d0efb37c4d63d9bf6f5f3b4dcb53d99.

Informational Findings

1. Informational - Direct transfer of aTokens impacts share value (engn33r)

The `totalAssets()` function returns the balance of aTokens held by the ERC4626 vault. Any user can transfer aTokens to the vault directly, without depositing, and alter the output values of other functions. This would be equivalent to an instantaneous spike in the aToken yield. This scenario may cause unexpected consequences with other protocols integrating with the Yield Daddy ERC4626 vaults. There is a comment for this edge case in the code.

Technical Details

```
function totalAssets() public view virtual override returns (uint256) {
    // aTokens use rebasing to accrue interest, so the total assets is just the
```

```
aToken balance
    return aToken.balanceOf(address(this));
}
```

The `totalAssets()` function in AaveV2ERC4626.sol and AaveV3ERC4626.sol is an important function. It is called in the calculations of `convertToShares()`, `convertToAssets()`, `previewMint()`, and `previewWithdraw()`. It is possible for anyone to transfer Aave aTokens directly to the vault, which would change the value returned by `totalAssets()`. Although such an increase in value happens normally as deposits take place or yield is earned, an instantaneous change may impact how other protocols interface with Yield Daddy ERC4626 tokens depending on how the other protocols are implemented. This comment is absent from the Compound ERC4626 wrapper, but the same edge case of a user sending tokens directly to the vault can apply.

Impact

Informational.

Recommendation

If a contract modification is warranted, created an internal `aTokenBalance` variable that is increased in `afterDeposit()` and decreased in `withdraw()` and `redeem()`. Most importantly, documentation should clearly highlight that a change in `totalSupply()` is possible from a direct transfer so other protocols interfacing with Yield Daddy ERC4626 tokens can implement their protocols accordingly. If there is a significant risk from a direct transfer of aTokens, measures can be taken to deal with this, but there is no contract upgrade process implemented in the contracts.

Developer Response

Acknowledged.

2. Informational - aAMPL edge cases (engn33r)

AMPL is a rebasing token, and aAMPL is a rebasing token. The Yield Daddy vault will be exposed to two layers of rebasing, which could lead to unexpected edge cases. Aave even has a special docs page about AMPL caveats, and a similar documentation page listing caveats for Yield Daddy may be useful.

Technical Details

AMPL is a rebasing token, but unlike Aave aTokens, AMPL can increase or decrease the balance that user's hold. This website shows some recent rebasing activity in both the positive and negative directions. The ability to rebalance in either direction means that a user depositing into the Yield Daddy aAMPL vault may receive less tokens when withdrawing than when they deposited. If there are very few depositors in the aAMPL vault, a user with a large fraction of vault ownership may be able to take advantage of the timing of the AMPL rebasing (which happens once per day) at the cost of the other vault shareholders.

Impact

Informational.

Recommendation

Although AMPL owners should be aware of the potential for positive and negative rebasing, Yield Daddy is exposed to two levels of rebasing. Proper testing of this 2nd order effect should take place and users should be warned of possible side effects, including withdrawing fewer tokens than were deposited.

Developer Response

Acknowledged.

3. Informational - Invert constant mask variables (engn33r)

There are five mask variables used only once, in the `_getDecimals()`, `_getActive()`, `_getFrozen()`, `_getPaused()`, and `_getSupplyCap()` functions. The mask values are constants but are only used in their inverted or negated form. Instead of negating the constants each time, the constant value should be the negated value of the current constant. This will improve readability.

Technical Details

Modify the constant masks in the Aave V3 wrapper to the following:

```
-   uint256 internal constant DECIMALS_MASK =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00FFFFFFFFFFFF;
+ uint256 internal constant DECIMALS_MASK = 0xFF << 48;
-   uint256 internal constant ACTIVE_MASK =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFF;
```

```
+ uint256 internal constant ACTIVE_MASK = 1 << 56;
-   uint256 internal constant FROZEN_MASK =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFDFFFFFFFFFFFFFF;
+ uint256 internal constant FROZEN_MASK = 1 << 57;
-   uint256 internal constant PAUSED_MASK =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFF;
+ uint256 internal constant PAUSED_MASK = 1 << 60;
-   uint256 internal constant SUPPLY_CAP_MASK =
0xFFFFFFFFFFFFFFFFFFFFFFFF000000000FFFFFFFFFFFFFFFFFFFFFFFFFFF;
+ uint256 internal constant PAUSED_MASK = 0xFFFFFFFFF << 116;
```

Remember to remove the negation where the variables are used, so `~DECIMALS_MASK` will become `DECIMALS_MASK`.

Impact

Informational.

Recommendation

Change the constant variables to their inverse values.

Developer Response

This is fine.

4. Informational - Simplify logic in `maxDeposit()` and `maxMint()` (engn33r)

The logic reused in several functions in AaveV3ERC4626.sol can be simplified.

Technical Details

`maxDeposit()` and `maxMint()` have logic which can be simplified:

```
- if (!(_getActive(configData) && !_getFrozen(configData) &&
!_getPaused(configData))) {
+ if (!_getActive(configData) || _getFrozen(configData) || _getPaused(configData)) {
```

Impact

Informational.

Recommendation

Simplify boolean logic to make the code easier to read.

Developer Response

This is fine.

5. Informational - Add parentheses to avoid order of operations ambiguity (engn33r)

`_getActive()`, `_getFrozen()`, and `_getPaused()` rely on bitwise operations taking place left to right. Because solidity version updates can sometimes alter this process, such as solidity 0.8.0 changing exponentiation from left associative to right associative, adding parentheses would help remove any ambiguity and make it clearer to readers of the code what is happening.

Technical Details

These three functions have boolean logic that relies on order of operations. If this changes in a future solidity version, an incorrect value may be calculated. Applying parentheses would remove any ambiguity and make it easier to upgrade to future solidity releases with reduced concerns.

```
    function _getActive(uint256 configData) internal pure returns (bool) {
-       return configData & ~ACTIVE_MASK != 0;
+       return (configData & ~ACTIVE_MASK) != 0;
    }

    function _getFrozen(uint256 configData) internal pure returns (bool) {
-       return configData & ~FROZEN_MASK != 0;
+       return (configData & ~FROZEN_MASK) != 0;
    }

    function _getPaused(uint256 configData) internal pure returns (bool) {
-       return configData & ~PAUSED_MASK != 0;
+       return (configData & ~PAUSED_MASK) != 0;
    }
```

Impact

Informational.

Recommendation

Removing ambiguity from important logic is a good practice.

Developer Response

This is fine.

6. Informational - Inconsistent ERC4626 callback usage (engn33r)

The `afterDeposit()` callback is implemented in AaveV2ERC4626.sol but the `beforeWithdraw()` callback is not implemented. Instead of implementing the `beforeWithdraw()` callback, the `withdraw()` and `redeem()` function have a custom implementation to remove the need for the `beforeWithdraw()` callback. Ideally AaveV2ERC4626.sol should be consistent and implement both callbacks or neither.

Technical Details

The default `withdraw()` and `redeem()` function implementations are overridden in AaveV2ERC4626.sol, possibly to save gas from one ERC20 token transfer. The same approach could be used for `deposit()` and `mint()` by having the user send their reserve asset directly to Aave with an onBehalfOf value in the `deposit()` call of the Yield Daddy vault. This may require the user to approval the Aave pool, but it would improve consistency in the Yield Daddy code.

Impact

Informational.

Recommendation

Consider using one approach consistently throughout the contracts. Using the more gas efficient option can improve gas efficiency.

Developer Response

This is more gas efficient.

7. Informational - Some hard coded variables might be better adjustable (engn33r)

Some variables are hard coded in Yield Daddy, but making these values modifiable in functions with the `onlyGovernance()` modifier may make the protocol more adaptable.

Technical Details

The `rewardRecipient` is an address that receives accrued rewards from the ERC4626 vault.

This address cannot be changed in the existing AaveV2ERC4626.sol contract. Scenarios may arise where this value needs to be changed, but the contract does not currently allow this to happen.

Impact

Informational.

Recommendation

Making certain variables adjustable could make the contract easier to upgrade, maintain, or simply make the contract more versatile to unexpected scenarios. Another option is to set `rewardRecipient` to a proxy address so that the logic behind the proxy can be modified if needed.

Developer Response

The contracts are not intended to be upgradeable. If a new version of the wrappers become available, existing wrapper holders can easily migrate to the new wrapper if desired.

8. Informational - Unclear upgrade mechanism (engn33r)

There is no clear upgrade mechanism if the Yield Daddy contracts needs to be upgraded. This may cause problems later on if an upgrade to the vault needs to happen.

Technical Details

The Yield Daddy contracts have no clear upgrade mechanism even though some hard coded values might need changing in the future. Some examples of upgrade use cases include:

1. If extra features are planned for a future Yield Daddy vault version, there is no clear path for migrating the aTokens from the existing vault.
2. The Yield Daddy vaults do not have a mechanism to borrow from Aave using the aTokens the vault holds. Borrowing with some amount of this collateral could allow Yield Daddy to increase the yield generated and improve the value proposition to users. Introducing such a mechanism in the future would require the vault to approve the aTokens to another address or borrowing directly from Aave, and neither option is available with the contract logic.
3. Aave had a referral program in the past which is now inactive, and a governance proposal could theoretically bring the program back. Even though this scenario is

unlikely, if it were to happen the Aave `deposit()` call in Yield Daddy has a hard coded value of 0 for the referral code, so no referral rewards could be collected if the reward program is resumed by Aave.

Impact

Informational.

Recommendation

Consider the current strategy for handling a scenario where the contracts need to be upgraded and the probability of such scenarios surfacing. If necessary, make it easier to upgrade the contracts. If upgrades are not necessary, the code can be left as it is.

Developer Response

The contracts are not intended to be upgradeable. If a new version of the wrappers become available, existing wrapper holders can easily migrate to the new wrapper if desired.

9. Informational - General lack of documentation and comments (engn33r)

There is no developer docs for Yield Daddy. Not all functions have NatSpec documentation. Improved documentation should be added before deployment.

Technical Details

To give one example of some unclear code that should have comments, the `maxRedeem()` function checks how much of the reserve asset is held by the aToken address. It may not be obvious to all readers that the reason that an external protocol is queried before returning a result for Yield Daddy users is to cover the edge case where the Aave pool is undercollateralized (which is common due to Aave's lending feature) and the Yield Daddy withdrawal would be larger than the entire Aave aToken reserve balance. A comment in the code or developer docs explaining this reasoning could expedite understanding for developers interfacing with this protocol or future code reviewers. Another place where NetSpec comments would help is clarifying that the return value of `maxWithdraw()` is in units of shares while the return value of `maxRedeem()` is in units of the underlying asset.

Impact

Informational.

Recommendation

Improve the Yield Daddy project documentation.

Developer Response

Acknowledged.

10. Informational - Typos (engn33r)

There were some typos found in the repo.

Technical Details

The Aave V2 factory contract has a comment accidentally copied from Aave V3. This should be modified to reference Aave V2.

This comment about `burn the xPYT` should be removed or changed. It was erroneously copied from a similar function that does burn xPYT. The same applieds to this other comment in the same function.

Impact

Informational.

Recommendation

Fix typos.

Developer Response

Fixed in commit 6c679121d2b4bacc5376c69d676928958a2b602b.

11. Informational - Aave LendingPool `getConfiguration()` can replace `getReserveData()` (engn33r)

In the Aave V2 and V3 wrappers, `lendingPool.getReserveData()` is used, which returns many pieces of data. These calls can be replaced with `lendingPool.getConfiguration()`, which returns less data. There are no gas savings because this is a view function.

Technical Details

`getReserveData()` returns many pieces of data in Aave V2 and Aave V3. The only data needed in the wrapper contracts can be received from `getConfiguration()`, which returns less data in Aave V2 and Aave V3. This efficiency boost won't save gas because it is used in a view function.

Impact

Informational.

Recommendation

Use more efficient external Aave call when possible.

Developer Response

Acknowledged.

12. Informational - Redundant dependency imports (engn33r)

The factory contracts for each protocol (Aave V2, Aave V3, Compound, Euler) import ERC4626Factory.sol and two solmate files. The ERC4626Factory.sol contract already imports the two solmate files, so they are imported twice. A similar instance is found in StETHERC4626.sol.

Technical Details

The ERC4626 factory contracts have the same redundant import of two solmate files. For example, Aave V3 imports ERC4626Factory.sol and two solmate files. But the imported base factory contract has the solmate files imported, making the import of these files redundant.

StETHERC4626.sol has a similar instance of this where StETHERC4626.sol imports "solmate/tokens/ERC20.sol" and "./external/IStETH.sol", but "solmate/tokens/ERC20.sol" is already imported by IStETH.sol making the separate import unnecessary.

CurveV2Swapper.sol has a similar instance of this where CurveV2Swapper.sol imports "../Swapper.sol" and "../lib/ApproveMaxIfNeeded.sol", but "../lib/ApproveMaxIfNeeded.sol" is already imported by Swapper.sol. The same applies to the solmate and timeless imports in CurveV2Swapper.sol.

Impact

Informational.

Recommendation

Remove redundant solmate file imports from Aave V2, Aave V3, Compound, and Euler factory contracts. Separately, remove StETHERC4626.sol and CurveV2Swapper.sol redundant import.

Developer Response

This is by design. I want it to be obvious where each imported contract comes from in every file, so each symbol is explicitly imported from the file in which they were defined instead of importing all symbols from some other files.

13. Informational - Interface doesn't match mainnet contract (engn33r)

The interface for `mintGuardianPaused()` in IComptroller.sol doesn't match the actual Compound contract. When yield daddy ERC4626 functions that rely on this external call are called on mainnet, they will revert because the ICERC20 type must be cast to an address first.

Technical Details

This is the Compound code defining the `mintGuardianPaused` mapping as `mapping(address => bool) public mintGuardianPaused`. The interface for this mapping is defined as `function mintGuardianPaused(ICERC20 cToken) external view returns (bool)`. This is problematic because the `address` type is not equal to the `ICERC20` type.

A foundry test was created to test if this type mismatch caused a revert, but the test passed without a revert. However, the custom `maxMint()` and `maxDeposit()` functions are not tested with existing tests for Compound or Aave V3.

Impact

Informational.

Recommendation

Modify the interface file IComptroller.sol to match Compound's code. Add foundry tests to confirm these functions work as expected.

Developer Response

This is by design. Contract types are treated the same as address types in external calls, so it's good to use contract types to enforce extra type checking and avoid passing in the wrong parameters.

14. Informational - Consider upgrade to latest solidity release (engn33r)

The ERC4626 wrapper contracts use solidity 0.8.13. The latest solidity release at the time of writing, 0.8.16, has improvements that could be useful to take advantage of by upgrading the release version.

Technical Details

The release notes of solidity releases since 0.8.13 describe code generation bugfixes and gas optimizations that have been introduced in the more recent releases. It would be best to use the latest release unless there is a good reason to continue using 0.8.13.

Impact

Informational.

Recommendation

Upgrade contracts to 0.8.16 to use the recent bug fixes and optimizations.

Developer Response

The Solidity pragma used is pragma solidity ^0.8.13;, which means if a newer version of Solidity has been released it would automatically be used during compilation.

15. Informational - Theoretical overflow of `convertToShares()` and `convertToAssets()` (engn33r)

The ERC4626 functions `convertToShares()` and `convertToAssets()` could overflow and revert if a very large assets input parameter value is used. This is unlikely because it requires a small denominator and a very large numerator and most tokens don't have a supply cap that large.

Technical Details

Below is the custom `convertToShares()` function from StETHERC4626.sol:

```
    function convertToShares(uint256 assets) public view virtual override returns
(uint256) {
        uint256 supply = stETH().totalSupply();

        return supply == 0 ? assets : assets.mulDivDown(stETH().getTotalShares(),
supply);
    }
```

If the value of the `assets` input parameter or the value of `stETH().getTotalShares()` is very close to `type(uint256).max` at the same time that `supply` is a very small value, this function could revert during the mulDivDown library call. The same is true of the `convertToShares()` function in the standard solmate implementation. The `convertToAssets()` function suffers

from the same side effect in this extreme case.

Impact

Informational.

Recommendation

Most likely no change is needed because the revert condition is on a view function and only happens in extreme edge cases.

Developer Response

Acknowledged.

16. Informational - No zero address checks in constructor (engn33r)

There is no zero address checks in the constructor of xPYT.sol. This is acceptable if it is assumed that no mistake will occur during the contract deployment, but for security purposes it is best to err on the side of caution.

Technical Details

The xPYT.sol constructor sets three state variables to specific external contract addresses. No zero address check is performed, making it possible that one of these state variables could be set to the zero address.

Impact

Informational.

Recommendation

Add the following lines at the end of the xPYT.sol constructor. Note that storing `nyt` in a temporary `nyt_` variable like is done with the other state variables may save gas.

```
require(address(gate_) != address(0));
require(address(vault_) != address(0));
require(address(nyt) != address(0));
```

Developer Response

This is fine.

17. Informational - Modify Gate.sol `claimYieldAndEnter()` function argument (engn33r)

`claimYieldAndEnter()` accepts four function arguments. The fourth argument is of type `IxPYT`, but this value is cast to an address four out of five times that it is used in the function. Consider changing this input type of `address` to cast this value only one time instead of four times.

Technical Details

Most locations where xPYT is used, it is cast to an address with `address(xPYT)`. Only once is it used as `IxPYT` type. Instead, take in a function argument of type `address` and cast to `IxPYT` the one time that this type is needed.

Impact

Informational.

Recommendation

Modify `claimYieldAndEnter()` to take a fourth argument of type `address` instead of type `IxPYT`.

Developer Response

This is by design. This adds more type checking for function parameters.

18. Informational - CurveV2xPYT utilizes a non-TWAP price oracle (Benjamin Samuels)

The CurveV2xPYT implements the `_getTwapQuote` function, which `curvePool.price_oracle()` to acquire a TWAP estimation, however, curve's price_oracle() function implements an exponential moving average (EMA) oracle.

Technical Details

Curve's `price_oracle()` implementation uses an exponential moving average for its oracle. EMA oracles have different properties from TWAP oracles, most notable of which is they are more sensitive to high price volatility near the time of query.

Impact

Informational.

Recommendation

Modify xPYT's `_getTwapQuote` function name to reflect that the implementer may use a different averaging mechanism than TWAP, such as `_getOracleQuote`

Developer Response

This is fine.

19. Informational - Replace magic numbers with constants (engn33r)

Constant variables should be used in place of magic numbers to prevent typos. For one example, the magic number 10000 is found in multiple places in Swapper.sol and should be replaced with a constant. Using a constant also adds a description to the value to explain the purpose of the value. This will not change gas consumption.

Technical Details

There are many instances of the value 10000. Consider replacing this magic number with a constant internal variable named `FEE_BASE`. Instances are found here, here, here, and here.

Impact

Informational.

Recommendation

Use constant variables instead of magic numbers.

Developer Response

Acknowledged.

20. Informational - No timelock or other protection around changing fee or fee recipient (engn33r)

`ownerSetProtocolFee()` allows the owner to change the fee percentage and the recipient of fees. This is a centralization risk because the owner can make this change without any delay. If holders of a timeless token are promised a fraction of these fees, the owner can redirect the fees elsewhere, effectively rugging the token holders.

Technical Details

`ownerSetProtocolFee()` allows the owner to set the fee percentage and fee recipient. While this function has access control to only allow the owner to call this function, there is no further controls to reduce centralization risk. Whether this is necessary depends on the parties who are intended to receive the fees. If users are intended to receive a fraction of the fees from Swapper.sol, they may want additional guarantees that the fees will not be redirected somewhere else without notice.

Impact

Informational.

Recommendation

Consider reducing centralization risk by adding a timelock to fee changes and clarifying how the multisig works.

Developer Response

Acknowledged.

21. Informational - safeApprove is deprecated (Benjamin Samuels)

The `safeApprove` function is used in several locations across the scoped & unscoped contracts. According to the OpenZeppelin documentation, it is currently deprecated.

Technical Details

This function is deprecated because it can be used to manipulate a user's allowances using specific transaction ordering.

Impact

Informational.

Recommendation

Consider replacing instances of safeApprove with safeIncreaseAllowance/safeDecreaseAllowance respectively.

Developer Response

This is by design. Some major tokens such as USDT will revert if safeIncreaseAllowance or safeDecreaseAllowance is used, since calling approve() with a non-zero input & non-zero existing allowance is not allowed.

22. Informational - Swapper repo foundry coverage failing (engn33r)

Using `foundry coverage` for the swapper repo results in a `CompilerError` message.

Technical Details

While increasing code coverage does not necessarily reduce the risk of security issues, it is generally a good idea to improve code coverage for projects. Running `foundry coverage` in the swapper repo returns a `CompilerError: Stack too deep when compiling inline assembly: Variable value0 is 1 slot(s) too deep inside the stack.` error. This implies that code coverage is not being monitored or improved for the swapper issue.

Impact

Informational.

Recommendation

Fix the Swapper repo to enable foundry code coverage.

Developer Response

Acknowledged.

23. Informational - StETHERC4626.t.sol test failing (Benjamin Samuels)

The testVaultInteractionsForSomeoneElse test for StETHERC4626.t.sol in the yield-daddy repo is failing.

Technical Details

This issue is caused by the `mintUnderlying` call on line 188, which mints 1e18+2 of the underlying token for alice. 1e18 of these tokens are deposited to the vault on line 201, leaving behind 2 underlying tokens in alice's account when the following assertion is made on line 205.

Impact

Informational.

Recommendation

Fix the test and add automatic test-on-push using Github Actions to prevent similar test failures in the future.

Developer Response

Acknowledged.

Final remarks

engn33r

The swapper and xPYT repositories have many code paths that involve calls to contracts that were outside the scope of this review, making it difficult to guarantee the security of the in-scope contracts given the involvement of extensive out of scope code. Additionally, the swapper and xPYT contracts are written in a way that involves many code paths that make it difficult to follow for every edge case. Refactoring this code, replacing duplicated

line of code with functions and making it more linear where possible, would go a long way to making the code easier to understand in future reviews. In contrast, the ERC4626 wrappers in the yield-daddy repository were relatively straightforward to review.

Beyond the complexity of the code, one area in need of improvement is documentation. The documentation for the all the repositories, including the core timeless repository (which was not in the scope of this review) could be improved, especially for the xPYT and swapper repositories. Because swaps are happening between three different Timeless tokens, it is important to keep track of what units/token each variable refers to, and the lack of documentation around what units/token each variable uses makes it harder to verify the contract logic.

Benjamin Samuels

Swapper, xPYT, and yield-daddy have adequate testing that would benefit greatly from an integration test suite that runs against mainnet/other network state. Integration tests currently run against locally-deployed variants, however it should be noted that many of the protocols being integrated against utilize upgradable proxies(Lido), or have different versions of their protocol deployed on different networks (Aave). Some level of mainnet integration testing can help avoid unexpected issues such as the cETH Price Oracle incident that recent impacted Compound.

If future passes on the testing suite are planned, it would be advisable to standardize the invariants that are tested in the ERC4626 wrappers. The current test suite for the ERC4626 wrappers is comprehensive for some wrappers, but relatively lacking for others. Equal testing coverage would help ensure each wrapper is conforming to the properties of the official spec.

The in-code naming conventions for NYT, PYT, and xPYT made auditing the system much more difficult. This convention will likely make it more challenging for integrators to develop on top of Yield Daddy, and it would be advisable to choose a more concise naming convention in future protocol versions.

The Swapper repository exposes the most dynamic behavior, so some additional time was spent focusing on verifying its behavior. Several Echidna invariants were written to verify the CurveJuggler component of Swapper, the code for which is attached to this report.

While Yield Daddy's architecture is relatively complex, it is worth noting that some care

was taken to decouple its components using ERC4626. This decoupling made auditing the ERC4626 contracts much simpler, and other parts of the codebase may benefit from a similar decoupling pass.