Swivel v3 contest
Findings & Analysis Report

2022-09-29

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Swivel v3 smart contract system written in Solidity. The audit contest took place

between July 12—July 15 2022.

Wardens

80 Wardens contributed reports to the Swivel v3 contest:

1. scaraven
2. Franfran
3. GimelSec (rayn and sces60107)
4. panprog
5. 0x52
6. oyc_109
7. bardamu
8. devtooligan
9. hansfriese
10. csanuragjain
11. jonatascm
12. itsmeSTYJ
13. cccz
14. joestakey
15. Bahurum
16. 0xDjango
17. c3phas
18. rokinot
19. 0x1f8b
20. sashik_eth
21. ronnyx2017
22. bin2chen
23. Picodes
24. MadWookie
25. benbaessler

26. rbserver
27. 0xSky
28. auditor0517
29. kyteg
30. Bnke0x0
31. m_Rassska
32. Meera
33. hake
34. simon135
35. 8olidity
36. robee
37. 0xNazgul
38. fatherOfBlocks
39. _141345_
40. defsec
41. _Adam
42. Soosh
43. Sm4rty
44. slywaters
45. Waze
46. Funen
47. ElKu
48. Kaiziron
49. JC
50. ReyAdmirado
51. Avci (0xArshia and 0xdanial)
52. exd0tpy
53. arcoun
54. caventa

55. Lambda
56. Chom
57. cryptphi
58. 0xNineDec
59. gogo
60. sach1r0
61. aysha
62. ak1
63. Junnon
64. PaludoX0
65. rishabh
66. pashov
67. mektigboy
68. ajtra
69. 0xsam
70. durianSausage
71. 0x040
72. TomJ
73. samruna
74. Fitraldys
75. karanctf
76. ignacio
77. CRYP70
78. Aymen0909

This contest was judged by 0xean and bghughes.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 13 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 12 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 55 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 44 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Swivel v3 contest repository, and is composed of 5 smart contracts written in the Solidity programming language and includes 1,593 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (1)

[H-01] Mismatch in `withdraw()` between Yearn and other protocols can prevent Users from redeeming zcTokens and permanently lock funds

Submitted by scaraven, also found by Franfran

As defined in the docs for Euler, ERC4626, Compound and Aave, when withdrawing and depositing funds the `amount` specified corresponds excactly to how many of the underlying assets are deposited or withdrawn.

However, as specified by Yearn, the yearn withdraw `amount` parameter specifies how many `shares` are burnt instead of underlying assets retrieved. Two scenarios can occur from then on, if there are not enough shares then the transaction will revert and users will not be able to redeem the underlying assets from Yearn. If there are enough shares, then a higher number of assets will be withdrawn then expected (as assets per share will only increase). This means that once the user receives their underlying assets at a 1:1 peg the excess amounts will be permanently locked in the vault contract.

Proof of Concept

All scenarios use Yearn finance as the protocol, the other protocols are unaffected.

Scenario #1

1. A yearn finance vault contains 50 shares and 200 uTokens all owned by Swivel.sol
2. A user opens a zcToken position and deposits 100 uTokens into yearn receiving 100 zcTokens (and possibly a premium)
3. `100 / 200 * 50 = 25` shares are minted for the vault so we now have 75 shares and 300 uTokens in the yearn vault
4. After maturity is achieved, the user tries to redeem their 100 zcTokens for uTokens
5. The swivel.sol contract tries to withdraw the 100 uTokens, it instead withdraws 100 shares which is less than available so the transaction reverts and the user cannot withdraw all of their 100 uTokens and instead can only withdraw 75 uTokens.

Scenario #2

1. A yearn finance vault contains 50 shares and 100 uTokens all owned by Swivel.sol
2. A user opens a zcToken position and deposits 100 uTokens into yearn receiving 100 zcTokens (and possibly a premium)
3. `100 / 100 * 50 = 50` shares are minted for the vault so we now have 100 shares and 200 uTokens in the yearn vault
4. After maturity is achieved, the user tries to redeem their 100 zcTokens for uTokens
5. The contract tries to retrieve 100 uTokens but instead withdraws 100 shares which corresponds to 200 uTokens
6. User receives their 100 uTokens causing 100 uTokens to be left in the Swivel.sol contract which are now irretrievable
7. If any user tries to withdraw their funds then the transaction will fail as no shares are left in the yearn vault

Tools Used

VS Code

Recommended Mitigation Steps

In the `withdraw()` function in Swivel.sol, calculating the price per share and use that to retrieve the correct number of underlying assets e.g.

```
uint256 pricePerShare = IYearnVault(c).pricePerShare();
return IYearnVault(c).withdraw(a / pricePerShare) >= 0;
```

JTraversa (Swivel) commented:

> Duplicate of **#30**.

scaraven (warden) commented:

> I do not understand how this is a duplicate of #30, **#30** talks about a problem with `redeemUnderlying()` in Compound while this issue talks about a problem

> with `withdraw()` when using yearn.

JTraversa (Swivel) commented:

> They both cannot be true at once however. Either the lib expects shares, or the lib expects assets. His suggestion notes inconsistency and recommends changing the compound redeem to align with yearn, while you note the inconsistency and recommend fixing the yearn math. At the end of the day the issue is the "mismatch".

scaraven (warden) commented:

> Fair enough, that makes sense. I would still disagree with their interpretation of the issue, it is clear from the code that `a` represents the underlying assets. If `a` does represent the number of shares then other functions such as `authRedeemzcTokens` would be plain wrong because it would be redeeming each zcToken as if it was one share not one underlying asset.

JTraversa (Swivel) commented:

> Yeah I actually kind of agree with you. We intended the implementation to align more with your finding.

> That said it *might* be worth a judge's input.

> If they think they're different, the other finding is invalid/should be disputed and this one is correct.

bghughes (judge) commented:

> Confirmed, good issue and marked #30 as a duplicate.

robrobbins (Swivel) resolved:

> Addressed: https://github.com/Swivel-Finance/gost/pull/437.

Medium Risk Findings (12)

[M-01] With most functions in VaultTracker.sol, users can call them only once after maturity has been reached.

Submitted by hansfriese, also found by panprog

https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/VaultTracker/VaultTracker.sol#L124

https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/VaultTracker/VaultTracker.sol#L132

Impact

With most functions in VaultTracker.sol, users can call them only once after maturity has been reached.

So from the second call of any functions after maturity, it will revert and users might lose their funds or interests.

Proof of Concept

The main problem is that `vlt.exchangeRate` might be larger than maturityRate after maturity here.

Then it will revert from the second call with uint underflow here.

So such scenario is possible.

- Alice splits 1000 USDC into zcToken and nToken using splitUnderlying()
- After the market is matured, she withdraws back 1000 USDC using combineTokens()
- While combining tokens, removeNotional() is called and vlt.exchangeRate will be set as an exchangeRate after maturity here.
- From the explanation about Exchange Rate, we know this ratio increases over time and also exchangeRate after the market is matured is greater than market.maturityRate from this function for zcToken interest

- So after removeNotional() is called, vlt.exchangeRate > maturityRate.
- After that, Alice calls redeemVaultInterest() to claim interest, but it will revert here with uint underflow error because maturityRate < vlt.exchangeRate.
- So she can't claim her interest forever.
- Also, if she claims interest first before calling combineTokens(), she can't get paid back here 1000 USDC forever because removeNotional() will revert here for the same reason.

Tools Used

Solidity Visual Developer of VSCode

Recommended Mitigation Steps

We should save vlt.exchangeRate = maturityRate when maturityRate > 0 and exchangeRate > maturityRate.

```
vlt.exchangeRate = (maturityRate > 0 && maturityRate < exchangeR
```

There are several places to modify for safety.

- https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/VaultTracker/VaultTracker.sol#L73
- https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/VaultTracker/VaultTracker.sol#L104
- https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/VaultTracker/VaultTracker.sol#L132
- https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/VaultTracker/VaultTracker.sol#L176
- https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/VaultTracker/VaultTracker.sol#L199

- https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/VaultTracker/VaultTracker.sol#L231

JTraversa (Swivel) disputed, disagreed with severity and commented:

> I don't actually disagree with most of the warden's claims, just note that all of this would need to be intentional on the part of the user, and could not happen under any normal workflow of the protocol.
>
> E.g.:
>
> Alice splits 1000 USDC into zcToken and nToken using splitUnderlying() After the market is martured, she withdraws back 1000 USDC using combineTokens()
>
> This interaction relies on a user calling `combineTokens` past maturity, which is 1. not possible from our UI 2. not beneficial in any way whatsoever, as redemption is explicitly done post maturity through `redeemZcToken` , + not using `redeemZcToken` would also fail to accrue post maturity interest.
>
> The same can be said of:
>
> Also, if she claims interest first before calling combineTokens(), she can't get paid back here 1000 USDC forever because removeNotional() will revert here for the same reason.
>
> There is no use case to calling `combineTokens` after maturity, meaning this specifically isn't an issue in that direction either.
>
> Just disagreeing with severity because of the lack of potential impact in any reasonable operation of the protocol. A user would have to do this through a script, and the workflow would be unlikely to happen on accident.

robrobbins (Swivel) resolved:

> Addressed: https://github.com/Swivel-Finance/gost/pull/414.

Oxean (judge) decreased severity to Medium and commented:

> I am somewhere between leaving this as High and downgrading to Medium. I understand the sponsor's points, but think it's a pretty nasty rough edge to leave out there if a user interacted with the contract through etherscan or some other UI with less guard rails.

> I am going to downgrade to Medium based on it being somewhat self inflicted and the probability of it therefore very low.

[M-02] Swivel.setFee() is implemented wrongly.

Submitted by hansfriese, also found by jonatascm

`Swivel.setFee()` is implemented wrongly. Swivel.feenominators won't be set as expected.

Proof of Concept

This function has a parameter "i" for the index of the new fee denomination but it isn't used during the update.

Tools Used

Solidity Visual Developer of VSCode

Recommended Mitigation Steps

This line should be modified like below.

```
    feenominators[i[x]] = d[x];
```

JTraversa (Swivel) disagreed with severity and commented:

> Given this allows us to change fees post initialization, and doesn't lead to the leakage of value or loss of funds, but a potential issue for admins solely (in a rare edge case where fees would even need to be changed), I might consider this low risk?

bghughes (judge) commented:

> I agree with the warden here given that this is an incorrect implementation of the intended functionality. The warden's suggestion shows that the function was not working as intended; if in production, for instance, this was not caught then calls to `setFee` would not work as intended and not set any fees across the markets. Instead, it would only populate the zero index of the 2nd-demensional array `uint16[4] public feenominators;` losing these values `[zcTokenInitiate, zcTokenExit, vaultInitiate, vaultExit]` and breaking functionality.

JTraversa (Swivel) commented:

> Yeah I was kind of on the fence on this one thinking the method wouldn't impact the current feenominators in the way you've stated, but because it actually could have an impact, and the event itself also would be misleading, removing the disagreement.

robrobbins (Swivel) resolved:

> Addressed: https://github.com/Swivel-Finance/gost/pull/419.

[M-03] Error in allowance logic

Submitted by 0xDjango, also found by 0x1f8b, 8olidity, arcoun, Bahurum, caventa, csanuragjain, hansfriese, joestakey, jonatascm, Lambda, oyc_109, and ronnyx2017

https://github.com/code-423n4/2022-07-swivel/blob/daf72892d8a8d6eaa43b9e7d1924ccb0e612ee3c/Tokens/ZcToken.sol#L112-L114

https://github.com/code-423n4/2022-07-swivel/blob/daf72892d8a8d6eaa43b9e7d1924ccb0e612ee3c/Tokens/ZcToken.sol#L132-L133

Impact

There is an error in the allowance functionality to allow a non-owner to withdraw or redeem ZcTokens for the owner. Taking `ZcToken.redeem()` as an example, behold

the following if/else block:

```
if (holder == msg.sender) {
    return redeemer.authRedeem(protocol, underlying, mat
}
else {
    uint256 allowed = allowance[holder][msg.sender];
    if (allowed >= principalAmount) { revert Approvals(a
    allowance[holder][msg.sender] -= principalAmount;
    return redeemer.authRedeem(protocol, underlying, mat
```

If the `msg.sender` is the holder, no check for allowance is needed. If the sender is not the holder, then their allowance is checked.

The error lies in the if statement `if (allowed >= principalAmount) { revert Approvals(allowed, principalAmount); }`. This states that if the sender has equal to or more allowance than the principalAmount, revert.

Therefore, if the sender has the proper allowance or more allowance than necessary, the transaction will revert. If the sender has less allowance than necessary, the transaction will still revert because of the `allowance[holder][msg.sender] -= principalAmount;` clause.

In conclusion, there is no way to `withdraw()` or `redeem()` on behalf of another user.

Recommended Mitigation Steps

The fix is to simply change `>=` to `<`.

JTraversa (Swivel) commented:

> Approval workflow doesnt leave funds at risk but is a nice to have, that plus scope and this *might* end up Low risk but I think Medium is appropriate as well.

bghughes (judge) commented:

> This is a good issue and I agree with the severity. I decided against making this High severity due to the fact that funds are not necessarily at risk; it's just intended allowance functionality will not behave as expected.

robrobbins (Swivel) resolved and commented:

> See #180.

bghughes (judge) commented:

> Making this the main issue for the allowance flipped sign.

[M-04] VaultTracker miscalculates compounding interest

Submitted by GimelSec

https://github.com/code-423n4/2022-07-swivel/blob/main/VaultTracker/VaultTracker.sol#L65

https://github.com/code-423n4/2022-07-swivel/blob/main/VaultTracker/VaultTracker.sol#L100

https://github.com/code-423n4/2022-07-swivel/blob/main/VaultTracker/VaultTracker.sol#L130

https://github.com/code-423n4/2022-07-swivel/blob/main/VaultTracker/VaultTracker.sol#L172

https://github.com/code-423n4/2022-07-swivel/blob/main/VaultTracker/VaultTracker.sol#L191

https://github.com/code-423n4/2022-07-swivel/blob/main/VaultTracker/VaultTracker.sol#L228

Impact

VaultTracker neglect previously accrued interest while attempting to calculate new interest. This causes `nToken` holders to receive less yield than they should.

All functions within VaultTracker that calculate interest are affected, including `addNotional`, `removeNotional`, `redeemInterest`, `transferNotionalFrom` and `transferNotionalFee`.

Proof of Concept

Consider the case where some user `N` tries to initiate a vault at 3 specific moments where `cToken` exchange rate is 5/10/20 respectively. The corresponding market stays active and has not reached maturity. Additionally, `N` selects his premium volume to make `principalFilled` match the `cToken` exchange rate during each call to `initiateVaultFillingZcTokenInitiate`. We recognize those exchange rates are most likely unrealistic, but we chose those for ease of demonstrating the bug. We also assume fees to be 0 for simplicity.

For the first call to `Swivel.deposit`

- `a` = 5
- `exchangeRate` = 5

Assuming no additional fees while minting `cToken`, `N` will receive `cToken\` for his 5 underlying tokens.

For the matching call to `VaultTracker.addNotional`

- `a` = 5
- `vlt.notional` = 0
- `exchangeRate` = 5

Since this is the first time adding `nToken` to the vault, there is no need to consider any accumulated interests, and we can assign `a` directly to `vlt.notional`.

The result will be

- `vlt.notional` = 5
- `vlt.redeemable` = 0
- `cToken` held by `Swivel` = 1

For the second call to Swivel.deposit, we have

- `a` = 10
- `exchangeRate` = 10

The matching call to `VaultTracker.addNotional` has

- `a` = 10
- `vlt.notional` = 5
- `vlt.redeemable` = 0
- `exchangeRate` = 10
- `vlt.exchangeRate` = 5

The `yield` is derived from `((exchangeRate * 1e26) / vlt.exchangeRate) - 1e26` = `((10 * 1e26) / 5) - 1e26` = 1e26 Applying this to `vlt.notional`, we get `interest` = `1e26 * 5 / 1e26` = 5

This results in

- `vlt.notional` = 5+10 = 15
- `vlt.redeemable` = 0+5 = 5
- `cToken` held by `Swivel` = 1+1 = 2

Now comes the last call to `Swivel.deposit`, where

- `a` = 20
- `exchangeRate` = 20

`VaultTracker.addNotional` has

- `a` = 20
- `vlt.notional` = 15
- `vlt.redeemable` = 5
- `exchangeRate` = 20

- `vlt.exchangeRate` = 10

`yield` = `((20 * 1e26) / 10) - 1e26` = 1e26 `interest` = `1e26 * 15 / 1e26` = 15

So we finally end up with

- `vlt.notional` = 15+20 = 35
- `vlt.redeemable` = 5+15 = 20
- `cToken` held by `Swivel` = 2+1 = 3

```
Swivel{
  ...
  function deposit(uint8 p, address u, address c, uint256 a) int
    ...
    if (p == uint8(Protocols.Compound)) { // TODO is Rari a drop
      return ICompound(c).mint(a) == 0;
    }
    ...
  }
  ...
}
```

```
VaultTracker{
  ...
  function addNotional(address o, uint256 a) external authorized
    uint256 exchangeRate = Compounding.exchangeRate(protocol, cT

    Vault memory vlt = vaults[o];

    if (vlt.notional > 0) {
      uint256 yield;

      // if market has matured, calculate marginal interest betw
      // otherwise, calculate marginal exchange rate between cur
      if (maturityRate > 0) { // Calculate marginal interest
        yield = ((maturityRate * 1e26) / vlt.exchangeRate) - 1e2
      } else {
        yield = ((exchangeRate * 1e26) / vlt.exchangeRate) - 1e2
      }
```

```
        uint256 interest = (yield * vlt.notional) / 1e26;
        // add interest and amount to position, reset cToken excha
        vlt.redeemable += interest;
        vlt.notional += a;
      } else {
        vlt.notional = a;
      }
      vlt.exchangeRate = exchangeRate;
      vaults[o] = vlt;

      return true;
    }
      ...
  }
```

Now take a step back and think about the actual value of 3 `cToken` when `exchangeRate = 20`, it should be pretty obvious that the value tracked by `VaultTracker` = 35 + 20 = 55 is lesser than the actual value of `cToken` held by `Swivel` = 20*3 = 60.

This is due to `VaultTracker` neglecting that previously accrued interest should also be considered while calculating new interest.

Recommended Mitigation Steps

For all interest calculations, use `vlt.notional + vlt.redeemable` instead of just `vlt.notional` as `yield` base.

[JTraversa (Swivel) disputed and commented](#):

> I believe that this would be valid if the `redeemable` was not redeemable by the user at any point in time.

> While interest accrues, it accrues to the `redeemable` balance which is withdrawn at any time.

> That said, in most cases, the math required to store and do the additional calculation marginal interest on the `redeemable` balance is largely a UX

> consideration? Should users be required to redeem their `redeemable` to earn interest or should it be compounded naturally though bloat tx costs?

> Should we force additional costs on a per transaction basis (that likely costs more than the interest itself for the vast majority of users), or should we assume that once significant enough `redeemable` is accrued to earn reasonable further interest, it will be redeemed by the user.

> (e.g. with example being 400% interest, and assuming an (optimistic) 5% APY is possible, the example would take ~28 years to replicate, and gas costs for transactions in that 28 years would be significant).

JTraversa (Swivel) disagreed with severity and commented:

> Thought about this one a bit more and it might slip in as acknowledged and disagreed with severity as it's more value leakage than anything else.

> It's a design decision, but could still be considered a detriment so perhaps not worth disputing all together?

robrobbins (Swivel) commented:

> This is a design feature. We could just as easily dismiss this all together by stating *it is now documented that .notional is non compounding*. I don't think we should do that however as I agree with @JTraversa here that some acknowledgement should happen for bringing this up. But it is not a **risk** in any situation.

robrobbins (Swivel) resolved:

> Addressed: https://github.com/Swivel-Finance/gost/pull/427

Oxean (judge) decreased severity to Medium and commented:

> Going to reduce the severity on this to Medium as it has some pretty large external factors to end up in a scenario where it leaks any real value.

[M-05] Should use >= instead of >

Submitted by oyc_109

https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/VaultTracker/VaultTracker.sol#L86

https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/VaultTracker/VaultTracker.sol#L158

The comparison should be 'a >= vlt.notional' instead of a > vlt.notional.

Otherwise dust amounts will always be left in vlt.notional when calling `removeNotional()` or `transferNotionalFrom()`.

JTraversa (Swivel) confirmed, but disagreed with severity and commented:

> While the leakage is potentially ... 1 wei, its leakage nonetheless and a frontend would surely provide the max amount most of the time leading to reverts.

> Maybe a quick severity review from wardens but happy with where its at.

bghughes (judge) commented:

> I agree that this is quite low stakes, given that there is non-zero potential leakage I think the warden should get credit for this as a Medium Risk vulnerability. Per the C4 juging criteria:

> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

robrobbins (Swivel) resolved:

> Addressed: https://github.com/Swivel-Finance/gost/pull/423.

[M-06] Yearn vault integration is broken

Submitted by bardamu

https://github.com/code-423n4/2022-07-swivel/blob/fbf94f87994d91dce75c605a1822ec6d6d7e9e74/Marketplace/Compounding.sol#L55-L56

https://github.com/code-423n4/2022-07-swivel/blob/fbf94f87994d91dce75c605a1822ec6d6d7e9e74/Marketplace/MarketPlace.sol#L64-L73

Impact

Yearn integration is broken, making it impossible to create a `MarketPlace` associated with a Yearn vault.

Proof of Concept

`Compounding.sol` attempts to resolve the `underlying` token implemented by supported protocols:

https://github.com/code-423n4/2022-07-swivel/blob/fbf94f87994d91dce75c605a1822ec6d6d7e9e74/Marketplace/Compounding.sol#L50-L64

```
function underlying(uint8 p, address c) internal view returns (a
    if (p == uint8(Protocols.Compound)) {
      return ICompoundToken(c).underlying();
    } else if (p == uint8(Protocols.Rari)) {
      return ICompoundToken(c).underlying();
    } else if (p == uint8(Protocols.Yearn)) {
      return IYearnVault(c).underlying();
    } else if (p == uint8(Protocols.Aave)) {
      return IAaveToken(c).UNDERLYING_ASSET_ADDRESS();
    } else if (p == uint8(Protocols.Euler)) {
      return IEulerToken(c).underlyingAsset();
    } else {
      return IErc4626(c).asset();
    }
  }
```

As can be seen in the excerpt above, the contract will attempt to access `IYearnVault(c).underlying();` when integrating with a Yearn vault. **Problem is this property is not present in Yearn vaults.** Therefore this operation will always revert.

In particular, this would not allow to create a `MarketPlace` associated with a Yearn vault even though the protocol is stated as supported.

https://github.com/code-423n4/2022-07-swivel/blob/fbf94f87994d91dce75c605a1822ec6d6d7e9e74/Marketplace/MarketPlace.sol#L64-L73

```
    function createMarket(
      uint8 p,
      uint256 m,
      address c,
      string memory n,
      string memory s
    ) external authorized(admin) unpaused(p) returns (bool) {
      if (swivel == address(0)) { revert Exception(21, 0, 0, address

      address underAddr = Compounding.underlying(p, c);
```

Correct underlying token address in Yearn vaults is stored in the `token` property.

https://github.com/yearn/yearn-vaults/blob/beff27908bb2ae017ed73b773181b9b93f7435ad/contracts/Vault.vy#L74

Tools Used

VIM

Recommended Mitigation Steps

Access the underlying token of a Yearn vault through the `token` property instead of the non-existent `underlying` which will cause a revert and the inability to create marketplaces associated with the protocol.

JTraversa (Swivel) disagreed with severity and commented:

> Given no funds would be at risk as this is an issue that would exist at the time of admin's market creation, I'd probably say that this issue is likely ~Low or so.

bghughes (judge) decreaased severity to Medium and commented:

> Agreed that funds are not necessarily at risk but this is a broken functionality of the protocol. Given:
>
> There are a few primary targets for concern:
>
> 1. Ensuring the new `Compounding.sol` library properly calculates the exchangeRate for each external protocol.
> 2. Ensuring the new `withdraw` and `deposit` methods on `Swivel.sol` properly encapsulate external protocol interactions.
>
> I believe this should be a Medium Risk issue.

robrobbins (Swivel) commented:

> The example given is incorrect. `Compounding.Underlying()` is correct as `Compounding` is our library that abstracts the actual call to get the asset in question.
>
> The actual error is in the Compounding lib where `.Protocol` resolves to `Protocols.Yearn`.

robrobbins (Swivel) resolved:

> Addressed: https://github.com/Swivel-Finance/gost/pull/422.

[M-07] ERC20 Incorrect check on returnedAddress in permit() results in unlimited approval of zero address

Submitted by devtooligan

When creating ERC20.sol from Solmate, a `require()` in `permit()` was converted to a custom error incorrectly.

It now reads:

```
    if (recoveredAddress != address(0) && recoveredAddress != owner)
        revert Invalid(msg.sender, owner);
    }
```

So if the `recoveredAddress` is non-zero and the `recoveredAddress` is not `owner` it will error. But if the `recoveredAddress == 0x0` then it will pass this check.

Anyone can permit themselves as a `spender` using the zero address for any token which inherits this ERC20 implementation. So, for example, someone could redeem some zcTokens for underlying, then transfer the burned tokens back to themselves and repeat, draining the protocol.

Proof of Concept

```
    function attack(IZcToken zctoken) {
        zctoken.permit(
            address(0x0),
            address(this),
            type(uint).max,
            block.timestamp,
            uint8(0),  // using 0 for r,s,v returns address(0x0) frc
            bytes32(0),
            bytes32(0)
        );
        assert(zctoken.allowance(address(0x0), address(this)), type(

        uint amount = zctoken.balanceOf(address(0x0)

        zctoken.transferFrom(
            address(0x0),
            address(this),
            amount
        );
```

```
        // assumes attacker has previously acquired notional
        IMarketPlace(mp).burnZcTokenRemovingNotional(
            zctoken.protocol(),
            zctoken.underlying(),
            zctoken.maturity(),
            address(this),
            amount
        );

        // .. repeat until drained, then move on to next token

    }
```

Recommended Mitigation Steps

```
diff --git a/Creator/Erc20.sol b/Creator/Erc20.sol
index a1f72b0..8464626 100644
--- a/Creator/Erc20.sol
+++ b/Creator/Erc20.sol
@@ -162,7 +162,7 @@ contract Erc20 {
                    s
                );

-               if (recoveredAddress != address(0) && recoveredAddr
+               if (recoveredAddress == address(0) || recoveredAddr
                    revert Invalid(msg.sender, owner);
                }
```

JTraversa (Swivel) disagreed with severity and commented:

> This would rely on the zcTokens themselves to sit on address(0) right?

> So the big thing would be that whatever burn implementation the token uses must actually remove the tokens from the supply rather than sending them to address(0). (which this implementation does)

> That said, its clear the statement wasn't as intended (or that block would have been removed).

> With scope questions this should probably be low-med or disputed?

robrobbins (Swivel) resolved:

> Addressed: https://github.com/Swivel-Finance/gost/pull/415.

bghughes (judge) commented:

> It's an interesting scope question as it questions a dependency - that said, this sort of low-level dependency, if used throughout the protocol, could jeopardize funds in a vault and break ERC-20 assumptions. I think the issue should stand given the risk.

JTraversa (Swivel) commented:

> Yea agreed.
>
> That said I'm more emphasizing the other part for the severity claim?:
>
> So the big thing would be that whatever burn implementation the token uses must actually remove the tokens from the supply rather than sending them to address(0). (which this implementation does)
>
> Though I think we are ameliorating this anyways, the issue would only arise if somehow address(0) was broken, as discussed by the warden himself in this twitter thread: https://twitter.com/devtooligan/status/1554709426652688384

0xean (judge) decreased severity to Medium and commented:

> Given the current implementation does not send these tokens to address(0) to burn them I think this should be downgraded to Medium severity.
>
> The "external requirement" would be someone / some protocol trying to burn tokens be sending them to address(0) instead of calling burn.
>
> I don't see a direct attack path given the current implementation but do think this is above QA, Med seems right.

[M-08] ZcToken.withdraw will send user 0 tokens if called after maturity deadline but before market is set mature

Submitted by panprog

https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/Tokens/ZcToken.sol#L99

https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/Tokens/ZcToken.sol#L92

Impact

If `maturityRate` is still `0` after maturity deadline (because no transactions setting `maturityRate` have been executed yet), then `previewWithdraw` calculated amount (used by `ZcToken.withdraw` function) is `0` and thus `withdraw` function will send `0` underlying tokens to user, which might be very confusing to user. Subsequent call to the same function will send him correct amount.

The same problem applies to all view functions in `ZcToken` contract - they use saved market `maturityRate`, which can be `0` even past deadline time and functions revert or return `0` in this case.

Incorrect withdrawal behaviour:

1. Bob has some `ZcToken`s.
2. Right at the time of maturity Bob tries to withdraw his underlying tokens by calling `ZcToken.withdraw` with some underlying amount.
3. Instead of receiving corresponding amount, Bob receives nothing (but transaction still succeeds and he uses gas for it).

Proof of Concept

1. `withdraw`: calculates `previewAmount` from `previewWithdraw`

https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/Tokens/ZcToken.sol

#L99

2. `previewWithdraw` : multiplication by `maturityRate` returns 0

https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/Tokens/ZcToken.sol#L92

Recommended Mitigation Steps

Add `getMaturityRate` function to `ZcToken` , which will return either market's `maturityRate` or (if it's `0`) current market's `exchangeRate` . Use this function instead of `maturityRate` everywhere across `ZcToken` .

JTraversa (Swivel) confirmed

robrobbins (Swivel) resolved:

> Addressed: https://github.com/Swivel-Finance/gost/pull/421

[M-09] VaultTracker has the wrong admin

Submitted by itsmeSTYJ, also found by GimelSec

https://github.com/code-423n4/2022-07-swivel/blob/main/Marketplace/MarketPlace.sol#L77

https://github.com/code-423n4/2022-07-swivel/blob/main/Creator/Creator.sol#L41

https://github.com/code-423n4/2022-07-swivel/blob/main/VaultTracker/VaultTracker.sol#L32

Description

`MarketPlace.createMarket()` calls `Creator.create()` which creates an instance of `ZcToken` and a `VaultTracker` . `VaultTracker` takes `msg.sender` as the admin. We know that if contract A calls contract B which calls contract C,

`msg.sender` in contract C is the address of B i.e. the `msg.sender` in VaultTracker is the address of the creator contract. However, the creator contract is not able (and not supposed to) interact with the VaultTracker unlike the marketplace contract.

Recommended Mitigation Steps

Modify the constructor of the VaultTracker contract so that the creator contract can pass in msg.sender (MarketPlace's address) to be used as admin.

JTraversa (Swivel) confirmed

bghughes (judge) commented:

> Using this as the main `admin` constructor issue. Given it is admin related, I believe going with the Medium issue instead of #134 makes sense.

robrobbins (Swivel) resolved:

> See **#134**.

[M-10] unpaused(p) modifier missing in authRedeem function

Submitted by csanuragjain, also found by cccz

Due to missing modifier, User will be able to redeem zcTokens and withdraw underlying even in paused Market. This happens due to missing unpaused(p) modifier

Proof of Concept

1. Lets see function definition for authRedeem function

```
function authRedeem(uint8 p, address u, uint256 m, address f, ac
```

2. Observe that unpaused(p) modifier is missing
3. This means if Marketplace is placed under paused state by Admin, then also User can call authRedeem at Marketplace via withdraw/redeem at ZcToken

contract.

4. This will allow Users to withdraw in paused state which is incorrect

Recommended Mitigation Steps

Add unpaused(p) modifier in authRedeem function

```
function authRedeem(uint8 p, address u, uint256 m, address f, ac
...
}
```

JTraversa (Swivel) confirmed

robrobbins (Swivel) resolved:

> Addressed: https://github.com/Swivel-Finance/gost/pull/420

[M-11] Loss of funds in an underlying protocol would cause catostrophic loss of funds for swivel

Submitted by 0x52

Loss of all user funds.

Proof of Concept

This exploit stems from a quirk in the way that exchange rate is tracked for matured positions. We first need to breakdown how interest is calculate for a matured position.

https://github.com/code-423n4/2022-07-swivel/blob/daf72892d8a8d6eaa43b9e7d1924ccb0e612ee3c/VaultTracker/VaultTracker.sol#L123-L132

In L124 the yield for a matured position is calculated as the difference between the previous exchange ratio and the maturity rate. This counts on the fact the exchange rate of the underlying protocol never decreases, as it always set the previous

exchange rate to the current exchange rate after yield is calculated regardless of whether it is mature or not. The assumption is that the current exchange rate will always be greater than or equal to the maturity exchange rate. If it is higher then L124 will revert due to underflow and if it is equal then L124 will return yield = 0. The issue comes in when this assumption is invalidated. This would happen were the underlying protocol to lose value (hacked, UST depeg, etc.). With the loss of value, the exchange rate would drop, allowing a user to repeatedly redeem their matured position until all funds from the affected protocol are drained.

Example: Imagine a yearn vault that takes USDC as the underlying token. It's current price per share is 1.25e6 (1.25 USDC per vault share). Swivel has a recently expired VaultTracker.sol for this yearn vault for which mature() is called, setting maturityRate = 1.25e6. Now let's imagine that one small strategy deployed by the vault is compromised and the vault loses 4% of it's USDC. Now the vault will return 1.2e6 as the price per share. When calling redeemInterest for the first time, vlt.exchangeRate will be updated to 1.2e6 in L132. The next time redeemInterest is called it will pay the difference between 1.25e6 (maturityRate) and 1.2e6 (vlt.exchangeRate). redeemInterest can be called repeatedly like this until all USDC deposited in the yearn vault by Swivel users has been drained.

Additionally the user in question would not even need to have an expired position before the loss of funds occurred. SplitUnderlying in Swivel.sol has no checks to keep a user from minting previously expired market. After the loss of funds occurred the malicious user could use SplitUnderlying to create nTokens for the expired market, then carry out the exploit

Recommended Mitigation Steps

The impact of such an event could be decreased with changes. In splitUnderlying add:

```
require(block.timestamp < m)
```

This prevents nTokens from being created after expiration which dramatically reduces the ability to take advantage of the opportunity. As for redeemInterest, add the following line after L124:

```
vlt.notional = 0
```

This would clear the notional balance of the user when redeeming after maturity, making it impossible to call repeatedly and reduces the chances that any users have a notional balance to exploit it should an event like this happen.

JTraversa (Swivel) disputed, disagreed with severity and commented:

> I'm unsure if this is a proper report / should be accepted given similar issues exist when you integrate nearly any DeFi primitive.

> You have risks that your integrated protocols could break, but make assumptions that they will not while placing failsafes like pause mechanism in your protocol.

> That said, the suggestions dont really solve anything, though we may end up taking them to heart?

> If the issue is that funds are lost when integrated protocol funds are lost and an attacker can then abuse these methods then the solutions provided just make someone spend more on gas in whatever flash loan tx they use to drain the protocol or slightly reduce the likelihood than an attacker will be in the correct position.

bghughes (judge) commented:

> Note that this builds on **#79** and I believe it is a good issue. The warden lays out a clear case in which Vault funds can be drained if these conditions occur:
>
> - A market has matured
> - The exchange rate of the underlying vault token drops below the set `maturityRate`.

> I agree that integrations can break @JTraversa, but given that Swivel aims to be the place people can go to deposit vault tokens and get ZCB + yield tokens (including allowing 4626 vaults) this is an attack vector IMO. Moreover, Yearn uses some discrete strategies in which the price per share realistically could reduce or even be manipulated in atomicity in a flash loan attack (which may seem counterintuitive bc generally lending vaults are supposed to be "up only").

> Confirming this as a High-Risk issue as an attack vector to drain all vault funds is present. Please let me know Sponsor if you have any questions.

robrobbins (Swivel) resolved and commented:

> Much of this is addressed with: https://github.com/Swivel-Finance/gost/pull/417 (cannot mint post maturity).

0xean (judge) decreased severity to Medium and commented:

> Downgrading to Medium as it requires external factors for this situation to present itself and the attack to occur.

[M-12] Interface definition error

Submitted by bin2chen, also found by 0x52, 0xDjango, 0xSky, auditor0517, Picodes, rokinot, ronnyx2017, and scaraven

https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/Marketplace/Interfaces.sol#L52

https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/Marketplace/MarketPlace.sol#L164

https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/Swivel/Swivel.sol#L620

https://github.com/Swivel-Finance/gost/blob/a76ac859df049527c3e5df85e706dec6ffa0e2bb/test/swivel/Swivel.sol#L10

Impact

`MarketPlace.authRedeem()` call interface `ISwivel.authRedeem()` but Swivel contract does not have this method only method “authRedeemZcToken()“.

The result will cause `MarketPlace.authRedeem()` to fail forever, thus causing `ZcToken.withdraw()` to fail forever.

Proof of Concept

MarketPlace.sol call `ISwivel.authRedeem()`

```
function authRedeem(uint8 p, address u, uint256 m, address f,
   .....

    ISwivel(swivel).authRedeem(p, u, market.cTokenAddr, t, a);

   .....
  } else {

   .....
    ISwivel(swivel).authRedeem(p, u, market.cTokenAddr, t, amo
   ....
  }
```

Swivel.sol does not have `authRedeem()` ,only `authRedeemZcToken()`

```
function authRedeemZcToken(uint8 p, address u, address c, addr
  // redeem underlying from compounding
  if (!withdraw(p, u, c, a)) { revert Exception(7, 0, 0, addre
  // transfer underlying back to msg.sender
  Safe.transfer(IErc20(u), t, a);

  return (true);
}
```

Recommended Mitigation Steps

Swivel contract need declare “is ISwivel” and change method name.

Other contracts should also declare “is Interfacename” to avoid method name errors like IMarketPlace.

JTraversa (Swivel) commented:

> Duplicate of #186.

bghughes (judge) commented:

> Main issue for the non-existent function call and interface definition error.

Oxean (judge) increased severity to High and commented:

> This shows a direct path to funds being locked, upgrading the severity (and all duplicates) to high.

JTraversa (Swivel) commented:

> Hmm, I think this could go either way.

> Technically because this is a redundant method meant to add compliance with EIP-5095, there aren't actually user funds locked because they can follow the normal `redeem` workflow. So it is specifically that EIP-5095 compliance that would be broken.

> Plus personally I'd also want to reward more interesting or critical errors rather than a mis-spelled or defined interface.

> That said, I'd of course err towards the judge's decision, but maybe that extra context helps !

Oxean (judge) commented:

> @JTraversa - appreciate the note and context.

> Can we walk through that flow to confirm the code path? I would downgrade to M if there is an alternate flow.

> We both agree this function doesn't work - https://github.com/code-423n4/2022-07-swivel/blob/fd36ce96b46943026cb2dfcb76dfa3f884f51c18/Marketplace/MarketPlace.sol#L148

> Which afaict also means that these calls are not functional either.

> https://github.com/code-423n4/2022-07-swivel/blob/daf72892d8a8d6eaa43b9e7d1924ccb0e612ee3c/Creator/ZcToken.sol#L98

> https://github.com/code-423n4/2022-07-swivel/blob/daf72892d8a8d6eaa43b9e7d1924ccb0e612ee3c/Creator/ZcToken.sol#L124

> Can you direct me to the alternate flow for redemption and I will confirm and move back to M

JTraversa (Swivel) commented:

> For sure!

> So there are two ways to redeem. Given custody sits on Swivel.sol, one is to call the zcToken (ERC-5095), which then itself calls authorized methods that bubble up from zcToken -> marketplace -> swivel. This is the path that appears to be broken, the `authRedeem`.

> That said, the primary path (the one we've used in previous Swivel versions) is one that starts with a redemption method directly on Swivel.sol, and has the instruction to burn bubbled down from swivel -> marketplace -> zcToken.

> That is available on Swivel.sol here: Link

> And its call to Marketplace.sol here: Link

> Hopefully that clarifies a bit :)

Oxean (judge) decreased severity back to Medium and commented:

> Yup, thank you. Downgraded back to M as user funds are not locked given the alternate path.

Low Risk and Non-Critical Issues

For this contest, 55 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **joestakey** received the top score from the judge.

The following wardens also submitted reports: Bahurum, c3phas, sashik_eth, benbaessler, 0x1f8b, defsec, simon135, hake, Meera, robee, BnkeOx0, fatherOfBlocks, oyc_109, 0xNazgul, rbserver, Chom, Sm4rty, Soosh, kyteg, csanuragjain, GimelSec, Funen, cryptphi, Kaiziron, scaraven, 0xNineDec, 0x52, _141345_, _Adam, 0xDjango, jonatascm, hansfriese, itsmeSTYJ, 8olidity, bin2chen, gogo, cccz, slywaters, sach1r0, aysha, rokinot, Picodes, ak1, Junnon, exd0tpy, JC, Waze, ReyAdmirado, ElKu, PaludoX0, rishabh, pashov, Avci, *and* mektigboy.

[L-01] Contracts should inherit their interfaces

Contract implementations should inherit their interfaces. Extending an interface ensures that all function signatures are correct, and catches mistakes introduced (e.g. through errant keystrokes)

Proof of Concept

Marketplace.sol

contract MarketPlace {

Does not inherit `IMarketPlace` , defined here

Recommended Mitigation Steps

```
-8: contract MarketPlace {
+8: contract MarketPlace is IMarketPlace {
```

[L-02] Immutable addresses lack zero-address check

Constructors should check the address written in an immutable address variable is not the zero address.

Note: while it has been indicated by the sponsor input validation will be on the front-end side to relieve users from unnecessary gas spendings, this issue here concerns constructor functions, ie when the contract is deployed by the team.

Proof of Concept

11 instances include:

Swivel.sol

marketPlace = m

MarketPlace.sol

creator = c

VaultTracker.sol

protocol = p
maturity = m
cTokenAddr = c
swivel = s

ZcToken.sol

protocol = _protocol
underlying = _underlying
maturity = _maturity
cToken = _cToken
redeemer = IRedeemer(_redeemer)

Recommended Mitigation Steps

Add a zero address check for the immutable variables aforementioned.

[L-03] Safe.sol does not check contract existence

`Safe.sol` is used to perform ERC20 transfers in the protocols contracts. It does not check for contract existence, meaning any call to an address with no bytecode (the address zero or an EOA) would not revert upon `Safe.transfer`.

If there is a market with an underlying token being an empty bytecode address, this means a user can initiate a position without actually transferring any token to `Swivel` - as all protocols except Aave do not check for the underlying token.

As only an admin can add a market, the risk of this issue affecting the protocols is low.

Proof of Concept

In `Safe.transfer` () and `Safe.transferFrom`, `success` is set to 1 if the call returns `1`, which is the case if it made to an address with no bytecode.

Recommended Mitigation Steps

Use `extcodesize()` in `Safe.sol` transfer functions to ensure the destination contract has bytecode.

[L-04] Setters should check the input value

Setters should check the input value - i.e. make revert if it is the zero address or zero.

Note: while it has been indicated by the sponsor input validation will be on the front-end side to relieve users from unnecessary gas spendings, this issue here concerns constructor functions or setters called by authorized admins, not users, and most likely by interacting with the contract directly (for instance using Etherscan or Hardhat) and not with a front-end.

Proof of Concept

5 instances:

Swivel.sol

function setAdmin(address a)

MarketPlace.sol

function setSwivel(address s)
function setAdmin(address a)

Creator.sol

function setAdmin(address a)

function setMarketPlace(address m)

Recommended Mitigation Steps

Add non-zero checks to the setters aforementioned.

[N-01] `2**256 - 1` can be re-written

Proof of Concept

VaultTracker.sol

uint256 max = 2**256 - 1

[N-02] Constants instead of magic numbers

It is best practice to use constant variables rather than literal values to make the code easier to understand and maintain.

Proof of Concept

30 instances:

VaultTracker.sol

yield = ((maturityRate * 1e26) / vlt.exchangeRate) - 1e26

yield = ((exchangeRate * 1e26) / vlt.exchangeRate) - 1e26

uint256 interest = (yield * vlt.notional) / 1e26

yield = ((maturityRate * 1e26) / vlt.exchangeRate) - 1e26

yield = ((exchangeRate * 1e26) / vlt.exchangeRate) - 1e26

uint256 interest = (yield * vlt.notional) / 1e26

yield = ((maturityRate * 1e26) / vlt.exchangeRate) - 1e26

yield = ((exchangeRate * 1e26) / vlt.exchangeRate) - 1e26

uint256 interest = (yield * vlt.notional) / 1e26

yield = ((maturityRate * 1e26) / from.exchangeRate) - 1e26

yield = ((exchangeRate * 1e26) / from.exchangeRate) - 1e26

uint256 interest = (yield * from.notional) / 1e26

yield = ((maturityRate * 1e26) / to.exchangeRate) - 1e26
yield = ((exchangeRate * 1e26) / to.exchangeRate) - 1e26
uint256 newVaultInterest = (yield * to.notional) / 1e26
yield = ((maturityRate * 1e26) / sVault.exchangeRate) - 1e26
yield = ((exchangeRate * 1e26) / sVault.exchangeRate) - 1e26
uint256 newVaultInterest = (yield * sVault.notional) / 1e26

Recommended Mitigation Steps

Define constant variables for the literal values aforementioned.

[N-03] Events indexing

Events should use the maximum amount of indexed fields: up to three parameters. This makes it easier to filter for specific values in front-ends.

Proof of Concept

3 instances:

lending-market/Comptroller.sol

event ScheduleWithdrawal(address indexed token, uint256 hold)
event ScheduleApproval(address indexed token, uint256 hold)
event ScheduleFeeChange(uint256 hold)

Recommended Mitigation Steps

Add indexed fields to these events so that they have the maximum number of indexed fields possible.

[N-04] Event should be emitted in setters

Setters should emit an event so that Dapps can detect important changes to storage.

Proof of Concept

5 instances:

Swivel.sol

function setAdmin(address a)

MarketPlace.sol

function setSwivel(address s)
function setAdmin(address a)

Creator.sol

function setAdmin(address a)
function setMarketPlace(address m)

Recommended Mitigation Steps

Emit an event in all setters.

[N-05] Function order

Functions should be ordered following the Solidity conventions.

Proof of Concept

Swivel.sol

- The modifier `authorized` is placed at the end of the contract, while it should be placed before every other function:

```
Inside each contract, library or interface, use the following or

Type declarations

State variables

Events

Modifiers

Functions
```

MarketPlace.sol

- the modifiers `authorized` and `unpaused` are placed at the end of the contract, while it should be placed before every other function
- the internal function `calculateReturn()` is placed before several external functions, while it `external` functions should be declared before `internal` ones:

```
Functions should be grouped according to their visibility and or

constructor

receive function (if exists)

fallback function (if exists)

external

public

internal

private
```

Recommended Mitigation Steps

Place the functions in the correct order.

[N-06] Long lines

Source codes lines should be limited to a certain number of characters. A good practice is to ensure the code does not require a horizontal scroll bar on GitHub. The lines mentioned below have that problem

Proof of Concept

14 instances:

Swivel.sol

[if (!mPlace.custodialInitiate(o.protocol, o.underlying, o.maturity, o.maker, msg.sender, principalFilled)) { revert Exception(8, 0, 0, address(0), address(0)); }]

```
if (!IMarketPlace(marketPlace).p2pZcTokenExchange(o.protocol, o.underlying, o.maturity, o.maker, msg.sender, a)) { revert Exception(11, 0, 0, address(0), address(0)); }
if (!mPlace.p2pVaultExchange(o.protocol, o.underlying, o.maturity, o.maker, msg.sender, principalFilled)) { revert Exception(12, 0, 0, address(0), address(0)); }
if (!IMarketPlace(marketPlace).p2pZcTokenExchange(o.protocol, o.underlying, o.maturity, msg.sender, o.maker, principalFilled)) { revert Exception(11, 0, 0, address(0), address(0)); }
if (!IMarketPlace(marketPlace).p2pVaultExchange(o.protocol, o.underlying, o.maturity, msg.sender, o.maker, a)) { revert Exception(12, 0, 0, address(0), address(0)); }
if (!mPlace.custodialExit(o.protocol, o.underlying, o.maturity, msg.sender, o.maker, principalFilled)) { revert Exception(9, 0, 0, address(0), address(0)); }
```

MarketPlace.sol

```
function authRedeem(uint8 p, address u, uint256 m, address f, address t, uint256 a) public authorized(markets[p][u][m].zcToken) returns (uint256 underlyingAmount) {
```

ZcToken.sol

```
constructor(uint8 _protocol, address _underlying, uint256 _maturity, address _cToken, address _redeemer, string memory _name, string memory _symbol, uint8 _decimals)
return (principalAmount * IRedeemer(redeemer).getExchangeRate(protocol, cToken) / IRedeemer(redeemer).markets(protocol, underlying, maturity).maturityRate);
return (underlyingAmount * IRedeemer(redeemer).markets(protocol, underlying, maturity).maturityRate / IRedeemer(redeemer).getExchangeRate(protocol, cToken));
return (principalAmount * IRedeemer(redeemer).getExchangeRate(protocol, cToken) / IRedeemer(redeemer).markets(protocol, underlying, maturity).maturityRate);
return (balanceOf[owner] * IRedeemer(redeemer).getExchangeRate(protocol, cToken) / IRedeemer(redeemer).markets(protocol, underlying, maturity).maturityRate);
return (underlyingAmount * IRedeemer(redeemer).markets(protocol, underlying, maturity).maturityRate / IRedeemer(redeemer).getExchangeRate(protocol,
```

cToken));
/// @notice At or after maturity, burns exactly `principalAmount` of Principal Tokens from `owner` and sends underlyingAmount of underlying tokens to `receiver`.

Recommended Mitigation Steps

Split the lines to avoid needing a scroll bar to look through the code.

[N-07] Natspec

Important functions should have a @notice comment to describe what they perform.

Proof of Concept

MarketPlace.sol

function authRedeem(uint8 p, address u, uint256 m, address f, address t, uint256 a) public authorized(markets[p][u][m].zcToken) returns (uint256 underlyingAmount)

Recommended Mitigation Steps

Add a @notice comment to this function.

[N-08] Non-library files should use fixed compiler versions

Contracts should be compiled using a fixed compiler version. Locking the pragma helps ensure that contracts do not accidentally get deployed using a different compiler version with which they have been tested the most.

Proof of Concept

ZcToken.sol

pragma solidity ^0.8.4

Recommended Mitigation Steps

Used a fixed compiler version.

[N-09] Non-library files should use the same compiler version

Contracts within the scope should be compiled using the same compiler version.

Proof of Concept

All the files in scope have the compiler version set to `0.8.13`, while `ZcToken.sol` has it set to `^0.8.4`.

Recommended Mitigation Steps

Use the same compiler version throughout the contracts.

[N-10] Open TODOs

There are open TODOs in the code. Code architecture, incentives, and error handling/reporting questions/issues should be resolved before deployment.

Proof of Concept

16 instances:

lending-market/Comptroller.sol

// TODO immutable?
// TODO cheaper to assign amount here or keep the ADD?
// TODO assign amount or keep the ADD?
// TODO assign amount or keep ADD?
// TODO assign amount or keep the ADD?
// TODO assign amount or keep the ADD?
// TODO assign amount or keep the ADD?
// TODO assign amount or keep the ADD?
// TODO as stated elsewhere, we may choose to simply return true in all and not attempt to measure against any expected return
// TODO is Rari a drop in here?
// TODO explain the Aave deposit args
// TODO explain the 0 (primary account)
// TODO as stated elsewhere, we may choose to simply return true in all and not attempt to measure against any expected return

// TODO is Rari a drop in here?
// TODO explain the withdraw args
// TODO explain the 0

Recommended Mitigation Steps

Remove the TODOs.

[N-11] Public functions can be external

It is good practice to mark functions as `external` instead of `public` if they are not called by the contract where they are defined.

Proof of Concept

1 instance:

MarketPlace.sol

function authRedeem(uint8 p, address u, uint256 m, address f, address t, uint256 a) public

Recommended Mitigation Steps

Declare it as `external` instead of `public`.

[N-12] Tautological code

Remove tautologies.

Proof of Concept

5 instances:

Swivel.sol

return IYearn(c).deposit(a) >= 0
return IErc4626(c).deposit(a, address(this)) >= 0
return IYearn(c).withdraw(a) >= 0
return IAave(aaveAddr).withdraw(u, a, address(this)) >= 0
return IErc4626(c).withdraw(a, address(this), address(this)) >= 0

Recommended Mitigation Steps

Remove the tautologies. Simply replace them with a `return true` statement.

[N-13] Typos

There are a few typos in the contracts.

Proof of Concept

3 instances:

Swivel.sol

Varifies
it's signature.
withraw

Recommended Mitigation Steps

Correct the typos.

robrobbins (Swivel) commented:

> Contracts should explicitly implement their interface (if available). Agreed. Addressed here: https://github.com/Swivel-Finance/gost/pull/424

> Reference to VaultTracker having `2**256-1` is incorrect. Is in Swivel.sol. agree with assessment tho, changed.

> Casting int returns as booleans is not tautological, particularly in the context of what that method is doing (normalizing returns).

> MarketPlace.authRedeem as `external` agree - changed.

> Removing TODOS agree: removing.

> Various issues above addressed here: https://github.com/Swivel-Finance/gost/pull/425.

> Event is now in place for `setAdmin` the only one that is needed. the other setters are one-time only by the deploying admin.

Gas Optimizations

For this contest, 44 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **joestakey** received the top score from the judge.

The following wardens also submitted reports: MadWookie, m_Rassska, rbserver, kyteg, Bnke0x0, Meera, c3phas, hake, ajtra, _141345_, 0xsam, _Adam, 0xNazgul, 0x1f8b, slywaters, Soosh, Waze, rokinot, robee, sashik_eth, ElKu, durianSausage, Funen, Sm4rty, 0x040, oyc_109, benbaessler, JC, ReyAdmirado, TomJ, simon135, samruna, Avci, fatherOfBlocks, csanuragjain, exd0tpy, Fitraldys, karanctf, Kaiziron, 0xDjango, ignacio, CRYP70, *and* Aymen0909.

[G-01] Address mappings can be combined in a single mapping

Combining mappings of `address` into a single mapping of `address` to a `struct` can save a `Gssset` (20000 gas) operation per mapping combined. This also makes it cheaper for functions reading and writing several of these mappings by saving a `Gkeccak256` operation- 30 gas.

Proof of Concept

1 instance:

Swivel.sol

mapping (address => uint256) public withdrawals
mapping (address => uint256) public approvals

Recommended Mitigation Steps

Combine the `address` mappings aforementionned in a single `address => struct` mapping, for instance

```
- mapping (address => uint256) public withdrawals;
  /// @dev maps a token address to a point in time, a hold, afte
```

```
- mapping (address => uint256) public approvals;
+ struct Hold {
+     uint256 withdrawals;
+     uint256 approvals;
+ }
+ mapping (address => Hold) public holds;
```

[G-02] Bytes constant are cheaper than string constants

If the string can fit into 32 bytes, then `bytes32` is cheaper than `string`. `string` is a dynamically sized-type, which has current limitations in Solidity compared to a statically sized variable. This means extra gas spent upon deployment and every time the constant is read.

Proof of Concept

Instances:

Swivel.sol

string constant public NAME = 'Swivel Finance';
string constant public VERSION = '3.0.0';

Recommended Mitigation Steps

```
- string constant public NAME = 'Swivel Finance';
- string constant public VERSION = '3.0.0';
+ bytes32 constant public NAME = 'Swivel Finance';
+ bytes32 constant public VERSION = '3.0.0';
```

[G-03] Caching storage variables in local variables to save gas

Anytime you are reading from storage more than once, it is cheaper in gas cost to cache the variable: a SLOAD cost 100gas, while MLOAD and MSTORE cost 3 gas.

In particular, in `for` loops, when using the length of a storage array as the condition being checked after each loop, caching the array length can yield significant gas

savings if the array length is high

Proof of Concept

2 instances:

Swivel.sol

scope: `setFee()`

- `feeChange` is read twice. Unless `feeChange == 0`, but given that it is an admin function, it is expected that when the admin invokes this function, all the required conditions are met, hence the state variable will be read twice from storage.

[if (feeChange == 0)]
[if (block.timestamp < feeChange)]

MarketPlace.sol

scope: `createMarket()`

- `swivel` is read twice. Unless `swivel == address(0)`, but given that it is an admin function, it is expected that when the admin invokes this function, all the required conditions are met, hence the state variable will be read twice from storage.

[if (swivel == address(0))]
[(address zct, address tracker) = ICreator(creator).create(p, underAddr, m, c, swivel, n, s, IErc20(underAddr).decimals())]

Recommended Mitigation Steps

Cache these storage variables using local variables.

[G-04] Caching mapping accesses in local variables to save gas

Anytime you are reading from a mapping value more than once, it is cheaper in gas cost to cache it, by saving one `gkeccak256` operation - `30` gas.

Proof of Concept

8 instances:

Swivel.sol

scope: `initiateVaultFillingZcTokenInitiate()`

- `filled[hash]` is read twice:

uint256 amount = a + filled[hash];
filled[hash] += a;

scope: `initiateZcTokenFillingVaultInitiate()`

- `filled[hash]` is read twice:

uint256 amount = a + filled[hash];
filled[hash] += a;

scope: `initiateZcTokenFillingZcTokenExit()`

- `filled[hash]` is read twice:

uint256 amount = a + filled[hash];
filled[hash] += a;

scope: `initiateVaultFillingVaultExit()`

- `filled[hash]` is read twice:

uint256 amount = a + filled[hash];
filled[hash] += a;

scope: `exitZcTokenFillingZcTokenInitiate()`

- `filled[hash]` is read twice:

uint256 amount = a + filled[hash];
filled[hash] += a;

scope: `exitVaultFillingVaultInitiate()`

- `filled[hash]` is read twice:

uint256 amount = a + filled[hash];
filled[hash] += a;

scope: `exitVaultFillingZcTokenExit()`

- `filled[hash]` is read twice:

uint256 amount = a + filled[hash];
filled[hash] += a;

scope: `exitZcTokenFillingVaultExit()`

- `filled[hash]` is read twice:

uint256 amount = a + filled[hash];
filled[hash] += a;

Recommended Mitigation Steps

Cache these `filled[hash]` accesses using local variables.

Note: this will not save gas on the first call for a specific `hash` - only on subsequent calls.

With that in mind, considering all these functions will only be called once for a specific `hash`, you can do the following change to save `SLOAD` operations:

```
-uint256 amount = a + filled[hash];

-if (amount > o.premium) { revert Exception(5, amount, o.premium
+if (a > o.premium) { revert Exception(5, a, o.premium, address(

-filled[hash] += a;
+filled[hash] = a;
```

[G-05] Calldata instead of memory for RO function parameters

If a reference type function parameter is read-only, it is cheaper in gas to use calldata instead of memory. Calldata is a non-modifiable, non-persistent area where function arguments are stored, and behaves mostly like memory,but it alleviates the compiler from the `abi.decode()` step that copies each index of the calldata to the memory index, each iteration costing `60` gas.

Proof of Concept

2 instances:

Swivel.sol

function setFee(uint16[] memory i, uint16[] memory d)

Recommended Mitigation STeps

Replace `memory` with `calldata`

[G-06] Clones for cheap contract deployment

There's a way to save a significant amount of gas on deployment using Clones: https://www.youtube.com/watch?v=3Mw-pMmJ7TA .

With the standard way using the `new` keyword, each contract created contains the entire logic. Using proxies allow only the first implementation to contain the logic, saving deployment costs on subsequent instances deployed.

PROOF OF CONCEPT

Instances:

Creator.sol

address zct = address(new ZcToken(p, u, m, c, marketPlace, n, s, d))
address tracker = address(new VaultTracker(p, m, c, sw))

Recommended Mitigation Steps

Use a proxy system, see here for an example.

[G-07] Constants can be private

Marking constants as `private` save gas upon deployment, as the compiler does not have to create getter functions for these variables. It is worth noting that a `private` variable can still be read using either the verified contract source code or the bytecode.

For immutable variables written via constructor parameters, you can also look the contract deployment transaction.

Proof of Concept

10 instances:

Swivel.sol

uint256 constant public HOLD = 3 days
uint16 constant public MIN_FEENOMINATOR = 33

Marketplace.sol

address public immutable creator

VaultTracker.sol

address public immutable admin
address public immutable cTokenAddr
address public immutable swivel
uint256 public immutable maturity
uint8 public immutable protocol

ZcToken.sol

uint8 public immutable protocol
address public immutable cToken

Recommended Mitigation Steps

Make these constants `private` instead of `public`

[G-08] Constructor parameters should be avoided when possible

Constructor parameters are expensive. The contract deployment will be cheaper in gas if they are hard coded instead of using constructor parameters.

Proof of Concept

4 instances:

Swivel.sol

marketPlace = m

aaveAddr = a

feenominators = [200, 600, 400, 200]

MarketPlace.sol

creator = c

Recommended Mitigation Steps

Hardcode these variables with their initial value instead of writing them during contract deployment with constructor parameters.

[G-09] Event fields are redundant

`block.timestamp` and `block.number` are added to event information by default, explicitly adding them is a waste of gas.

Proof of Concept

1 instance:

MarketPlace.sol

emit Mature(p, u, m, exchangeRate, block.timestamp)

Recommended Mitigation Steps

Remove the `uint256 matured` event field, as it always corresponds to `block.timestamp`.

[G-10] Functions with access control cheaper if payable

A function with access control marked as payable will lbe cheaper for legitimate callers: the compiler removes checks for `msg.value`, saving approximately `20` gas per function call.

Proof of Concept

36 instances:

Swivel.sol

function setAdmin(address a) external authorized(admin)
function scheduleWithdrawal(address e) external authorized(admin)
function blockWithdrawal(address e) external authorized(admin)
function withdraw(address e) external authorized(admin)
function scheduleFeeChange() external authorized(admin)
function blockFeeChange() external authorized(admin)
function setFee(uint16[] memory i, uint16[] memory d) external authorized(admin)
function scheduleApproval(address e) external authorized(admin)
function blockApproval(address e) external authorized(admin)
function approveUnderlying(address[] calldata u, address[] calldata c) external authorized(admin)
function authRedeemZcToken(uint8 p, address u, address c, address t, uint256 a) external authorized(marketPlace)

MarketPlace.sol

function setSwivel(address s) external authorized(admin)
function setAdmin(address a) external authorized(admin)
function createMarket(uint8 p,uint256 m,address c,string memory n,string memory s) external authorized(admin)
function mintZcTokenAddingNotional(uint8 p, address u, uint256 m, address t, uint256 a) external authorized(swivel)
function burnZcTokenRemovingNotional(uint8 p, address u, uint256 m, address t,

uint256 a) external authorized(swivel)
function authRedeem(uint8 p, address u, uint256 m, address f, address t, uint256 a) public authorized(markets[p][u][m].zcToken)
function redeemZcToken(uint8 p, address u, uint256 m, address t, uint256 a) external authorized(swivel)
function redeemVaultInterest(uint8 p, address u, uint256 m, address t) external authorized(swivel)
function custodialInitiate(uint8 p, address u, uint256 m, address z, address n, uint256 a) external authorized(swivel)
function custodialExit(uint8 p, address u, uint256 m, address z, address n, uint256 a) external authorized(swivel)
function p2pZcTokenExchange(uint8 p, address u, uint256 m, address f, address t, uint256 a) external authorized(swivel)
function p2pVaultExchange(uint8 p, address u, uint256 m, address f, address t, uint256 a) external authorized(swivel)
function transferVaultNotionalFee(uint8 p, address u, uint256 m, address f, uint256 a) external authorized(swivel)
function pause(uint8 p, bool b) external authorized(admin)

Creator.sol

function create (//args) external authorized(marketPlace)
function setAdmin(address a) external authorized(admin)
function setMarketPlace(address m) external authorized(admin)

VaultTracker.sol

function addNotional(address o, uint256 a) external authorized(admin)
function removeNotional(address o, uint256 a) external authorized(admin)
function redeemInterest(address o) external authorized(admin)
function matureVault(uint256 c) external authorized(admin)
function transferNotionalFrom(address f, address t, uint256 a) external authorized(admin)
function transferNotionalFee(address f, uint256 a) external authorized(admin)

ZcToken.sol

function burn(address f, uint256 a) external onlyAdmin(address(redeemer))
function mint(address t, uint256 a) external onlyAdmin(address(redeemer))

Recommended Mitigation Steps

Remove these event fields.

[G-11] Immutable variables save storage

If a variable is set in the constructor and never modified afterwards, marking it as `immutable` can save a storage slot - `20,000` gas. This also saves `97` gas on every read access of the variable.

Proof of Concept

1 instance:

Swivel.sol

[address public aaveAddr]

`aaveAddr` is set in the constructor and read in two functions, but never modified.

Recommended Mitigation Steps

Mark `aaveAddr` as `immutable`.

[G-12] Prefix increments

Prefix increments are cheaper than postfix increments - `6` gas. This can mean interesting savings in `for` loops.

Proof of Concept

5 instances:

xTRIBE.sol

[unchecked {i++;}]
[unchecked {i++;}]
[unchecked {i++;}]
[unchecked {x++;}]
[unchecked {i++;}]

Recommended Mitigation Steps

Change `variable++` to `++variable`.

[G-13] Storage cheaper than memory

Reference types cached in memory cost more gas than using storage, as new memory is allocated for these variables, copying data from storage to memory for each field of the struct or array: this means every field of the struct/array is read.

Proof of Concept

18 instances:

MarketPlace.sol

Market memory market = markets[p][u][m]
Market memory market = markets[p][u][m]
Market memory market = markets[p][u][m]
Market memory market = markets[p][u][m]
Market memory market = markets[p][u][m]
Market memory market = markets[p][u][m]
Market memory market = markets[p][u][m]
Market memory market = markets[p][u][m]
Market memory market = markets[p][u][m]
Market memory market = markets[p][u][m]

VaultTracker.sol

Vault memory vlt = vaults[o]
Vault memory vlt = vaults[o]
Vault memory vlt = vaults[o]
Vault memory from = vaults[f]
Vault memory to = vaults[t]
Vault memory oVault = vaults[f]
Vault memory sVault = vaults[swivel]
Vault memory vault = vaults[o]

Recommended Mitigation Steps

Use `storage` instead of `memory`. Cache any field read more than once onto the stack to avoid unnecessary `SLOAD` operations.

[G-14] Unchecked arithmetic

The default “checked” behavior costs more gas when adding/diving/multiplying, because under-the-hood those checks are implemented as a series of opcodes that, prior to performing the actual arithmetic, check for under/overflow and revert if it is detected.

If it can statically be determined there is no possible way for your arithmetic to under/overflow (such as a condition in an if statement), surrounding the arithmetic in an `unchecked` block will save gas.

Proof of Concept

Instances:

VaultTracker.sol

vlt.notional -= a

Because of the check here, it cannot underflow

from.notional -= a

Because of the check here, it cannot underflow

ZcToken.sol

allowance[holder][msg.sender] -= previewAmount

Because of the check here, it cannot underflow

allowance[holder][msg.sender] -= principalAmount

Because of the check here, it cannot underflow.

Recommended Mitigation Steps

Place the arithmetic operations in an `unchecked` block.

[G-15] Unnecessary computation

Unnecessary stack variables can be removed to save gas.

Proof of Concept

Instances:

ZcToken.sol

```
111:             uint256 allowed = allowance[holder][msg.sender]
112:             if (allowed >= previewAmount) {
113:                 revert Approvals(allowed, previewAmount);
114:             }
115:             allowance[holder][msg.sender] -= previewAmount;
```

```
132:             uint256 allowed = allowance[holder][msg.sender]
133:             if (allowed >= principalAmount) { revert Approv
134:             allowance[holder][msg.sender] -= principalAmour
```

Both `allowed` are not necessary. They only save gas if the function reverts

Recommended Mitigation Steps

Remove both `allowed` and read the storage variables directly in the condition blocks.

[G-16] Upgrade Solidity compiler version

0.8.10 removes contract existence checks if the external call has a return value - `700` gas

Proof of Concept

Instances:

ERC20Gauges.sol

pragma solidity ^0.8.4

Recommended Mitigation Steps

Upgrade `ZcToken.sol` compiler version.

[G-17] use Assembly for simple setters

Where it does not affect readability, using assembly allows to save gas not only on deployment, but also on function calls.

This is the case for instance for simple admin setters.

Proof of Concept

Instances:

Swivel.sol

```
428:   function setAdmin(address a) external authorized(admin) r
429:     admin = a;
430:
431:     return true;
432:   }
```

MarketPlace.sol

```
53:   function setAdmin(address a) external authorized(admin) re
54:     admin = a;
55:     return true;
56:   }
```

Creator.sol

```
47:   function setAdmin(address a) external authorized(admin) re
48:     admin = a;
49:     return true;
50:   }
```

Recommended Mitigation Steps

```
-  admin = a;
+  assembly {
+    sstore(admin.slot, a)
+  }
```

[G-18] Writing zero wastes gas

`Swivel.sol` uses a timelock system for changing the fees, which consists in setting `feeChange` to zero every time fees are changed, then setting it to `block.timestamp + HOLD` upon the next scheduled fee change.

As `SSTORE` operations are more expensive when setting a state variable from zero to a non-zero value than setting it from a non-zero value to another non-zero value, gas can be saved by not setting `feeChange` to zero upon a fee update.

Proof of Concept

Instances:

Swivel.sol

- in `blockFeeChange`:

feeChange = 0

- in `setFee`:

feeChange = 0

Recommended Mitigation Steps

Change to `feeChange = 1`, and change the following line in setFee:

```
- if (feeChange == 0)
+ if (feeChange == 1)
```

robrobbins (Swivel) commented:

> Well-written report.

> But items either addressed elsewhere or won't be fixed.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top