Cally contest
Findings & Analysis Report

2022-07-01

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 Cally smart contract system written in Solidity. The audit contest took place between May 10—May 14 2022.

Wardens

115 Wardens contributed reports to the Cally contest:

1. smiling_heretic
2. hubble (ksk2345 and shri4net)
3. sseefried
4. IllIllI
5. 0xDjango
6. Ruhum

7. Hawkeye (0xwags and 0xmint)
8. oyc_109
9. GimelSec (rayn and sces60107)
10. BondiPestControl (leastwood and kirk-baird)
11. WatchPug (jtp and ming)
12. 0xsanson
13. antonttc
14. VAD37
15. horsefacts
16. TrungOre
17. p4st13r4 (0x69e8 and 0xb4bb4)
18. hickuphh3
19. shung
20. MiloTruck
21. BowTiedWardens (BowTiedHeron and BowTiedPickle and m4rio_eth and Dravee and BowTiedFirefox)
22. MaratCerby
23. hake
24. ellahi
25. joestakey
26. dipp
27. 0xf15ers (remora and twojoy)
28. shenwilly
29. Kenshin
30. FSchmoede
31. cccz
32. minhquanym
33. Bludya
34. catchup

35. rfa
36. Cityscape
37. Picodes
38. reassor
39. pedroais
40. robee
41. pmerkleplant
42. rotcivegaf
43. TomFrenchBlockchain
44. csanuragjain
45. djxploit
46. 0x4non
47. CertoraInc (egjlmn1, OriDabush, ItayG, and shakedwinder)
48. 0x1f8b
49. fatherOfBlocks
50. sikorico
51. Waze
52. z3s
53. bobirichman
54. delfin454000
55. hansfriese
56. mics
57. Funen
58. Kumpa
59. jah
60. berndartmueller
61. 0x1337
62. Czar102
63. eccentricexit

64. MadWookie
65. hyh
66. seanamani
67. dirk_y
68. radoslav11
69. sorrynotsorry
70. Aits
71. 242
72. cryptphi
73. BouSalman
74. AlleyCat
75. JDeryl
76. defsec
77. RagePit
78. jayjonah8
79. m9800
80. gzeon
81. throttle
82. _Adam
83. TerrierLover
84. 0xkatana
85. samruna
86. 0xNazgul
87. simon135
88. DavidGialdi
89. ignacio
90. Tadashi
91. 0v3rf10w
92. Fitraldys

93. jonatascm
94. peritoflores
95. 0x52
96. ACai
97. kebabsec (okkothejawa and FlameHorizon)
98. ynnad
99. PPrieditis
100. crispymangoes

This contest was judged by HardlyDifficult.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 13 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 10 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 67 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 59 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 Cally contest repository, and is composed of 2 smart contracts written in the Solidity programming language and includes 439 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 (3)

[H-01] no-revert-on-transfer ERC20 tokens can be drained

*Submitted by Ruhum, also found by oyc*109 and smiling*heretic*

https://github.com/code-423n4/2022-05-cally/blob/main/contracts/src/Cally.sol#L198-L200

Impact

Some ERC20 tokens don't throw but just return false when a transfer fails. This can be abused to trick the `createVault()` function to initialize the vault without providing any tokens. A good example of such a token is *ZRX*: Etherscan code

When such a vault is initialized, another user can both buy and exercise the option without ever receiving any funds. The creator of the vault does receive the buyer's Ether tho. So it can cause a loss of funds.

Proof of Concept

The trick is to create a vault with an ERC20 token but use ERC721 as the vault's type. Then, instead of calling `safeTransferFrom()` the function calls `transferFrom()`

which won't catch the token returning false.

Here's a test that showcases the issue:

```
    // CreateVault.t.sol
        function testStealFunds() public {
            // address of 0x on mainnet
            address t = address(0xE41d2489571d322189246DaFA5ebDe1F46
            vm.startPrank(babe);
            require(ERC20(t).balanceOf(babe) == 0);
            uint vaultId = c.createVault(100, t, 1, 1, 1, 0, Cally.T
            // check that neither the Cally contract nor the vault c
            // had any 0x tokens
            require(ERC20(t).balanceOf(babe) == 0);
            require(ERC20(t).balanceOf(address(c)) == 0);

            // check whether vault was created properly
            Cally.Vault memory v = c.vaults(vaultId);
            require(v.token == t);
            require(v.tokenIdOrAmount == 100);
            vm.stopPrank();
            // So now there's a vault for 100 0x tokens although the
            // have any.
            // If someone buys & exercises the option they won't rec
            uint premium = 0.025 ether;
            uint strike = 2 ether;
            require(address(c).balance == 0, "shouldn't have any bal
            require(payable(address(this)).balance > 0, "not enough

            uint optionId = c.buyOption{value: premium}(vaultId);
            c.exercise{value: strike}(optionId);

            // buyer of option (`address(this)`) got zero 0x tokens
            // But buyer lost their Ether
            require(ERC20(t).balanceOf(address(this)) == 0);
            require(address(c).balance > 0, "got some money");
        }
```

To run it, you need to use forge's forking mode: `forge test --fork-url <alchemy/infura URL> --match testStealFunds`

Recommended Mitigation Steps

I think the easiest solution is to use `safeTransferFrom()` when the token is of type ERC721. Since the transfer is at the end of the function there shouldn't be any risk of reentrancy. If someone passes an ERC20 address with type ERC721, the `safeTransferFrom()` call would simply fail since that function signature shouldn't exist on ERC20 tokens.

outdoteth (Cally) confirmed and resolved:

> the fix for this issue is here; **https://github.com/outdoteth/cally/pull/4**

HardlyDifficult (judge) commented:

> This is a great report. I appreciate the clear test showcasing the issue well, and using a real token example.

[H-02] Inefficiency in the Dutch Auction due to lower duration

Submitted by hubble, also found by Hawkeye and sseefried

The vulnerability or bug is in the implementation of the function getDutchAuctionStrike() The AUCTION_DURATION is defined as 24 hours, and consider that the dutchAuctionReserveStrike (or reserveStrike) will never be set to 0 by user.

Now if a vault is created with startingStrike value of 55 and reserveStrike of 13.5 , the auction price will drop from 55 to 13.5 midway at ~12 hours. So, after 12 hours from start of auction, the rate will be constant at reserveStrike of 13.5, and remaining time of 12 hours of auction is a waste.

Some other examples :

```
startStrike, reserveStrike, time-to-reach-reserveStrike
55 , 13.5  , ~12 hours
55 , 5     , ~16.7 hours
55 , 1.5   , ~20 hours
5  , 1.5   , ~11 hours
```

Impact

The impact is high wrt Usability, where users have reduced available time to participate in the auction (when price is expected to change). The vault-Creators or the option-Buyers may or may not be aware of this inefficiency, i.e., how much effective time is available for auction.

Proof of Concept

Contract : Cally.sol Function : getDutchAuctionStrike ()

Recommended Mitigation Steps

The function getDutchAuctionStrike() can be modified such that price drops to the reserveStrike exactly at 24 hours from start of auction.

```
        /*
            delta = max(auctionEnd - currentTimestamp, 0)
            progress = delta / auctionDuration
            auctionStrike = progress^2 * (startingStrike - reser
            strike = auctionStrike + reserveStrike
        */
        uint256 delta = auctionEndTimestamp > block.timestamp ?
        uint256 progress = (1e18 * delta) / AUCTION_DURATION;
        uint256 auctionStrike = (progress * progress * (starting

        strike = auctionStrike + reserveStrike;
```

outdoteth (Cally) confirmed, disagreed with severity and commented:

> We think this should be bumped to high severity. It would be easy for a user to create an auction that declines significantly faster than what they would have assumed - even over 1 or 2 blocks. It makes no sense for the auction to ever behave in this way and would result in options getting filled at very bad prices for the creator of the vault.

outdoteth (Cally) resolved:

> The fix for this issue is here: https://github.com/outdoteth/cally/pull/2

HardlyDifficult (judge) increased severity to High and commented:

> The sponsor comment here makes sense. Agree with (1) High since this can potentially be very detrimental to the promise of this protocol.

[H-03] [WP-H0] Fake balances can be created for not-yet-existing ERC20 tokens, which allows attackers to set traps to steal funds from future users

Submitted by WatchPug, also found by 0xsanson, BondiPestControl, and IllIllI

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L158-L201

```
    function createVault(
        uint256 tokenIdOrAmount,
        address token,
        ...
    ) external returns (uint256 vaultId) {
        ...
        Vault memory vault = Vault({
            ...
        });

        // vault index should always be odd
        vaultIndex += 2;
        vaultId = vaultIndex;
        _vaults[vaultId] = vault;

        // give msg.sender vault token
        _mint(msg.sender, vaultId);

        emit NewVault(vaultId, msg.sender, token);

        // transfer the NFTs or ERC20s to the contract
        vault.tokenType == TokenType.ERC721
            ? ERC721(vault.token).transferFrom(msg.sender, address(t
            : ERC20(vault.token).safeTransferFrom(msg.sender, addres
    }
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L23-L34

```
import "solmate/utils/SafeTransferLib.sol";

...

contract Cally is CallyNft, ReentrancyGuard, Ownable {
    using SafeTransferLib for ERC20;
    ...
```

When creating a new vault, solmate's `SafeTransferLib` is used for pulling `vault.token` from the caller's account, this issue won't exist if OpenZeppelin's SafeERC20 is used instead.

That's because there is a subtle difference between the implementation of solmate's `SafeTransferLib` and OZ's `SafeERC20`:

OZ's `SafeERC20` checks if the token is a contract or not, solmate's `SafeTransferLib` does not.

See: https://github.com/Rari-Capital/solmate/blob/main/src/utils/SafeTransferLib.sol#L9

> Note that none of the functions in this library check that a token has code at all! That responsibility is delegated to the caller.

As a result, when the token's address has no code, the transaction will just succeed with no error.

This attack vector was made well-known by the qBridge hack back in Jan 2022.

For our project, this alone still won't be a problem, a vault created and wrongfully accounted for a certain amount of balance for a non-existing token won't be much of a problem, there will be no fund loss as long as the token stays that way (being non-existing).

However, it’s becoming popular for protocols to deploy their token across multiple networks and when they do so, a common practice is to deploy the token contract from the same deployer address and with the same nonce so that the token address can be the same for all the networks.

For example: $1INCH is using the same token address for both Ethereum and BSC; Gelato's$GEL token is using the same token address for Ethereum, Fantom and Polygon.

A sophisticated attacker can exploit it by taking advantage of that and setting traps on multiple potential tokens to steal from the future users that deposits with such tokens.

Proof of Concept

Given:

- ProjectA has TokenA on another network;
- ProjectB has TokenB on another network;
- ProjectC has TokenC on another network;
- The attacker `createVault()` for `TokenA`, `TokenB`, and `TokenC` with `10000e18` as `tokenIdOrAmount` each;
- A few months later, ProjectB lunched `TokenB` on the local network at the same address;
- Alice created a vault with `11000e18 TokenB`;
- The attacker called `initiateWithdraw()` and then `withdraw()` to receive `10000e18 TokenB`.

In summary, one of the traps set by the attacker was activated by the deployment of `TokenB` and Alice was the victim. As a result, `10000e18 TokenB` was stolen by the attacker.

Recommendation

Consider using OZ’s `SafeERC20` instead.

outdoteth (Cally) confirmed and resolved:

> this issue has been fixed here: https://github.com/outdoteth/cally/pull/5

[HardlyDifficult (judge) commented]():

> Great catch and the potential attack is very clearly explained. Although the window for an attack like this would not be common, it's an easy trap to setup and likely would occur as some point if Cally is planning to support multiple networks.

Medium Risk Findings (10)

[M-01] Owner can modify the feeRate on existing vaults and steal the strike value on exercise

*Submitted by IllIIl, also found by Adam, 0x52, 0xf15ers, 0xsanson, berndartmueller, Bludya, BondiPestControl, catchup, crispymangoes, Czar102, eccentricexit, ellahi, GimelSec, hake, horsefacts, hubble, joestakey, Kumpa, pedroais, peritoflores, reassor, shenwilly, shung, smiling*heretic, sseefried, and throttle_

Owner can steal the exercise cost which should have gone to the option seller

Proof of Concept

There are no restrictions on when the owner can set the `feeRate`:

```
File: contracts/src/Cally.sol   #1

117       /// @notice Sets the fee that is applied on exercise
118       /// @param feeRate_ The new fee rate: fee = 1% = (1 /
119       function setFee(uint256 feeRate_) external onlyOwner {
120           feeRate = feeRate_;
121       }
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L117-L121

By using a rate that consumes the exercise cost, the owner can steal Ether from the seller:

```
File: contracts/src/Cally.sol   #2

282             uint256 fee = 0;
283             if (feeRate > 0) {
284                 fee = (msg.value * feeRate) / 1e18;
285                 protocolUnclaimedFees += fee;
286             }
287
288             // increment vault beneficiary's ETH balance
289             ethBalance[getVaultBeneficiary(vaultId)] += msg.va
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L282-L289

The owner can wait for a particularly large-value NFT, snipe that one option, then retire

Recommended Mitigation Steps

Fix the fee rate per vault during vault creation

outdoteth (Cally) confirmed and resolved:

> issue is fixed here: https://github.com/outdoteth/cally/pull/8

HardlyDifficult (judge) commented:

> This is a valid concern. The owner could abuse this to capture much more ETH than was intended. It assumes that the owner is malicious or compromised and does not introduce any more risk than an upgradeable contract would have.

[M-02] It shouldn’t be possible to create a vault with Cally’ own token

Submitted by p4st13r4, also found by GimelSec, TrungOre, and VAD37

https://github.com/code-423n4/2022-05-cally/blob/main/contracts/src/Cally.sol#L193

https://github.com/code-423n4/2022-05-cally/blob/main/contracts/src/Cally.sol#L199

Impact

Affected code:

- https://github.com/code-423n4/2022-05-cally/blob/main/contracts/src/Cally.sol#L193
- https://github.com/code-423n4/2022-05-cally/blob/main/contracts/src/Cally.sol#L199

Currently it's possible to create an ERC-721 vault using Cally' own address as `token`, and using the freshly minted vault id as `tokenIdOrAmount`. This results in a new vault whose ownership is passed to Cally contract immediately upon creation.

The vault allows users to perform `buyOption` and increase the ETH balance of the Cally contract itself, which is still the vault beneficiary. As soon as an user calls `exercise`, she will receive the `vault.tokenIdOrAmount` in exchange, which in this case coincides with the vault nft. However this is of no good because the final user may just initiate a withdrawal, which will:

- always fail because the vault id is burned (https://github.com/code-423n4/2022-05-cally/blob/main/contracts/src/Cally.sol#L335) and then transferred back to the user (https://github.com/code-423n4/2022-05-cally/blob/main/contracts/src/Cally.sol#L344)
- leave all the ETH unredemable in Cally contract

So the vault will be unusable and the ETH deposited by users to buy/exercise options will remain locked in Cally contract

Proof of Concept

- Current vault id is, let's say, 11

- User deploys a vault with Cally' address as `token` and `13` as `tokenIdOrAmount`
- Since `createVault()` mints the vault token to the user, and then transfers the underlying address from the user, an user is able to create a vault with something she doesn't own at the moment of the `createVault()` function call, because it's created while the function runs
- The vault `13` is pretty limited in functionality, because Cally' smart contract is the owner
- However, users can still buy options: so Alice and Bob deposit their premiums
- Whoever `exercise` the active option, becomes the vault owner now; this is of no good because no one can actually call `withdraw()` as it will always revert, and no one can recover the ETH deposited by Alice and Bob as they are locked forever

Recommended Mitigation Steps

Add the following check at the start of `createVault()` :

```
    require(token != address(this), "Cant use Cally as token");
```

outdoteth (Cally) confirmed, disagreed with severity and commented:

> This is an exploit that requires users to actively make a very precise and niche mistake. should be medium severity in our opinion.

outdoteth (Cally) resolved:

> fix for this issue is here: https://github.com/outdoteth/cally/pull/10

HardlyDifficult (judge) decreased severity to Medium and commented:

> Copying in the POC from GimelSec in #244 because it's an interesting attack to consider for this issue as well.

1. > Alice (Attacker) pack 2 transactions into same block:

> - first transaction: calls `createVault` to vault a NFT which worth 100 ETH, with parameters:
>
>   - `dutchAuctionStartingStrikeIndex` is set to 0 (which `strikeOptions` is 1 ETH)
>   - a long `durationDays`, e.g. `255` days
>
> - then Alice will get a `vaultId 1` token, and Alice do another transaction: call `buyOption(1)` to get a `optionId 2` token

2. Alice re-vault the `vaultId 1` token with strike 89 ETH, and get a `vaultId 3` token
3. Bob see that the auction of `vaultId 3` token is 89 ETH, but the `vaultId 3` token can get the NFT which worth 100 ETH, so Bob pays 89 ETH, calls `buyOption(3)`, and `exercise(4)` to get the `vaultId 1` token. Then, Bob calls `initiateWithdraw(1)` and waits for the `optionId 2` token to expire (which `durationDays` is set to `255` days in step 1).
4. Alice monitors that someone bought the `vaultId 1` token, then Alice quickly calls `exercise(2)`. Finally, Alice just pays Bob 1 ETH, and gets the NFT back. Alice also gets 89 ETH which is paid by Bob from the `vaultId 3` token.

> I agree with (2) Medium for this issue. It can be abused, but the impacted parties can clearly see this is an attempt to subvert the system in some way (a vault of a vault with an NFT, instead of a single value with an NFT as expected). That should be a red flag for Bob in the example above.

[M-03] User's may accidentally overpay in `buyOption()` and the excess will be paid to the vault creator

Submitted by BondiPestControl, also found by 0xf15ers, berndartmueller, cccz, csanuragjain, dipp, GimelSec, hake, horsefacts, IllIllI, jayjonah8, m9800, MadWookie, MiloTruck, pedroais, Ruhum, throttle, and VAD37

It is possible for a user purchasing an option to accidentally overpay the premium during `buyOption()`.

Any excess funds paid for in excess of the premium will be transferred to the vault creator.

The premium is fixed at the time the vault is first created by `vault.premiumIndex`. Hence there is no need to allow users to overpay since there will be no benefit.

Proof of Concept

`buyOption()` allows `msg.value > premium`

```
        uint256 premium = getPremium(vaultId);
        require(msg.value >= premium, "Incorrect ETH amount sent
```

Recommended Mitigation Steps

Consider modifying the check such that the `msg.value` is exactly equal to the `premuim`. e.g.

```
        uint256 premium = getPremium(vaultId);
        require(msg.value == premium, "Incorrect ETH amount sent
```

outdoteth (Cally) confirmed and resolved:

> this issue is fixed in: https://github.com/outdoteth/cally/pull/9

HardlyDifficult (judge) commented:

> Agree with 2 (Medium) for this. The issue doesn't really open the door for an attack, except for maybe via a malicious frontend. But it could potentially leak value in terms of over compensating the vault creator.

HickupHH3 (warden) commented:

> QA report #182 should have its issue bumped up and marked as a duplicate IMO

[M-04] Vaults steal rebasing tokens' rewards

*Submitted by IllIllI, also found by horsefacts and smiling*heretic_

Rebasing tokens are tokens that have each holder's `balanceof()` increase over time. Aave aTokens are an example of such tokens.

Impact

If rebasing tokens are used as the vault token, rewards accrue to the vault and cannot be withdrawn by either the option seller or the owner, and remain locked forever.

Proof of Concept

The amount 'available' for withdrawal comes from an input parameter and is stored for later operations:

```
File: contracts/src/Cally.sol   #1

        Vault memory vault = Vault({
            tokenIdOrAmount: tokenIdOrAmount,
            token: token,
            premiumIndex: premiumIndex,
            durationDays: durationDays,
            dutchAuctionStartingStrikeIndex: dutchAuctionS
            currentExpiration: uint32(block.timestamp),
            isExercised: false,
            isWithdrawing: false,
            tokenType: tokenType,
            currentStrike: 0,
            dutchAuctionReserveStrike: dutchAuctionReserve
        });

        // vault index should always be odd
        vaultIndex += 2;
        vaultId = vaultIndex;
        _vaults[vaultId] = vault;

        // give msg.sender vault token
        _mint(msg.sender, vaultId);

        emit NewVault(vaultId, msg.sender, token);
```

```
196
197             // transfer the NFTs or ERC20s to the contract
198             vault.tokenType == TokenType.ERC721
199                 ? ERC721(vault.token).transferFrom(msg.sender,
200                 : ERC20(vault.token).safeTransferFrom(msg.sen
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L173-L200

The amount actually available grows over time and is only known at the time of withdrawal. The option withdrawal/exercise use the original amount:

```
File: contracts/src/Cally.sol   #2

345                 : ERC20(vault.token).safeTransfer(msg.sender,
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L345

```
File: contracts/src/Cally.sol   #3

296                 : ERC20(vault.token).safeTransfer(msg.sender,
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L296

Recommended Mitigation Steps

Track total amounts currently deposited and allow vault creators to withdraw excess on a pro-rata basis

outdoteth (Cally) acknowledged and commented:

> This is technically an issue however we have no intention of supporting rebase tokens

[M-05] Expiration calculation overflows if call option duration ≥ 195 days

Submitted by hickuphh3, also found by BondiPestControl, GimelSec, sseefried, and VAD37

`vault.durationDays` is of type `uint8`, thus allowing a maximum value of 255. `1 days = 86400`, thus fitting into a `uint24`. Solc creates a temporary variable to hold the result of the intermittent multiplication `vault.durationDays * 1 days` using the data type of the larger operand.

In this case, the intermittent data type used would be `uint24`, which has a maximum value of `2**24 - 1 = 16777215`. The maximum number allowable before overflow achieved is therefore `(2**24 - 1) / 86400 = 194`.

Proof of Concept

Insert this test case into BuyOption.t.sol

```
function testCannotBuyDueToOverflow() public {
  vm.startPrank(babe);
  bayc.mint(babe, 2);
  // duration of 195 days
  vaultId = c.createVault(2, address(bayc), premiumIndex, 195, s
  vm.stopPrank();

  vm.expectRevert(stdError.arithmeticError);
  c.buyOption{value: premium}(vaultId);
}
```

Then run

```
forge test --match-contract TestBuyOption --match-test testCannc
```

Tidbit

This was the 1 high-severity bug that I wanted to mention at the end of the C4 TrustX showcase but unfortunately could not due to a lack of time :(It can be found in the vulnerable lottery contract on L39. Credits to Pauliax / Thunder for the recommendation and raising awareness of this bug =p

Reference

Article

Recommended Mitigation Steps

Cast the multiplication into `uint32`.

```
    vault.currentExpiration = uint32(block.timestamp) + uint32(vault
```

outdoteth (Cally) confirmed and commented:

> Agree that this is high risk - a user can unintentionally create vaults that would never have been able to have been filled and result in them losing funds because the vault creation was useless. They then also have to initiate a withdraw and then actually withdraw before they can create another vault.

> In terms of gas prices at 100 gwei (which it frequently was a few months ago) the total gas cost of this bug/incorrect vault creation is not insignificant.

outdoteth (Cally) resolved:

> This issue is fixed here; https://github.com/outdoteth/cally/pull/3

HardlyDifficult (judge) decreased severity to Medium and commented:

> This is an easy way someone could create a vault where it's not possible to buy an option, and without using an unreasonably high duration value. If this were to occur, the vault creator could immediately `initiateWithdraw` and then `withdraw`. No time delay is required and the only funds lost is the gas cost of those 3 transactions.

> Lowering to 2 (Medium) since there's an easy recovery and no assets lost.

[M-06] Owner can set the feeRate to be greater than 100% and cause all future calls to `exercise` to revert

Submitted by IllIllI, also found by 0xDjango, ACai, antonttc, BowTiedWardens, Cityscape, defsec, dipp, FSchmoede, GimelSec, gzeon, hickuphh3, hubble, joestakey, Kenshin, m9800, MiloTruck, RagePit, Ruhum, shenwilly, TomFrenchBlockchain, and WatchPug

Impact

The owner can force options to be non-exercisable, collecting premium without risking the loss of their NFT/tokens

Proof of Concept

After a buyer buys an option owned by the owner, the owner can change the fee rate to be close to `type(uint256).max`, which will cause the subtraction below to always underflow, preventing the exercise of the option. Once the option expires, the owner can change the fee back and wait for another buyer

```
File: contracts/src/Cally.sol   #1

288            // increment vault beneficiary's ETH balance
289            ethBalance[getVaultBeneficiary(vaultId)] += msg.va
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L288-L289

Recommended Mitigation Steps

Add reasonable fee rate bounds checks in the `setFee()` function

outdoteth (Cally) confirmed and resolved:

> this issue is fixed here; https://github.com/outdoteth/cally/pull/7

[M-07] Lack of 0 amount check allows malicious user to create infinite vaults

Submitted by 0xDjango

A griefer is able to create as many vaults as they want by simply calling `createVault()` with `tokenIdOrAmount = 0`. This will most likely pose problems on the front-end of the Cally protocol because there will be a ridiculously high number of malicious vaults displayed to actual users.

I define these vaults as malicious because it is possible that a user accidently buys a call on this vault which provides 0 value in return. Overall, the presence of zero-amount vaults is damaging to Cally's product image and functionality.

Proof of Concept

- User calls `createVault(0,,,,);` with an ERC20 type.
- There is no validation that `amount > 0`
- Function will complete successfully, granting the new vault NFT to the caller.
- Cally protocol is filled with unwanted 0 amount vaults.

Recommended Mitigation Steps

Add the simple check `require(tokenIdOrAmount > 0, "Amount must be greater than 0");`

outdoteth (Cally) confirmed and commented:

> This check should only be applied on ERC20 tokens because ERC721 tokens can still have tokenIds that have ID's with a value of 0.

outdoteth (Cally) resolved:

> this issue is fixed here: https://github.com/outdoteth/cally/pull/12

[M-08] Vault is Not Compatible with Fee Tokens and Vaults with Such Tokens Could Be Exploited

*Submitted by 0x1337, also found by 0x52, 0xDjango, 0xsanson, berndartmueller, BondiPestControl, BowTiedWardens, cccz, dipp, GimelSec, hake, hickuphh3, horsefacts, hubble, IllIIII, MaratCerby, MiloTruck, minhquanym, PPrieditis, reassor, shenwilly, smiling*heretic, TrungOre, VAD37, and WatchPug_

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L198-L200

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L294-L296

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L343-L345

Impact

Some ERC20 tokens charge a transaction fee for every transfer (used to encourage staking, add to liquidity pool, pay a fee to contract owner, etc.). If any such token is used in the `createVault()` function, either the token cannot be withdrawn from the contract (due to insufficient token balance), or it could be exploited by other such token holders and the `Cally` contract would lose economic value and some users would be unable to withdraw the underlying asset.

Proof of Concept

Plenty of ERC20 tokens charge a fee for every transfer (e.g. Safemoon and its forks), in which the amount of token received is less than the amount being sent. When a fee token is used as the `token` in the `createVault()` function, the amount received by the contract would be less than the amount being sent. To be more precise, the increase in the `cally` contract token balance would be less than `vault.tokenIdOrAmount` for such ERC20 token because of the fee.

```
            vault.tokenType == TokenType.ERC721
            ? ERC721(vault.token).transferFrom(msg.sender, addre
            : ERC20(vault.token).safeTransferFrom(msg.sender, ac
```

The implication is that both the `exercise()` function and the `withdraw()` function are guaranteed to revert if there's no other vault in the contract that contains the same fee tokens, due to insufficient token balance in the `Cally` contract.

When an attacker observes that a vault is being created that contains such fee tokens, the attacker could create a new vault himself that contains the same token, and then withdraw the same amount. Essentially the `Cally` contract would be paying the transfer fee for the attacker because of how the token amount is recorded. This causes loss of user fund and loss of value from the `Cally` contract. It would make economic sense for the attacker when the fee charged by the token accrue to the attacker. The attacker would essentially use the `Cally` contract as a conduit to generate fee income.

Recommended Mitigation Steps

Recommend disallowing fee tokens from being used in the vault. This can be done by adding a `require()` statement to check that the amount increase of the `token` balance in the `Cally` contract is equal to the amount being sent by the caller of the `createVault()` function.

[outdoteth (Cally) confirmed and commented](https://github.com/code-423n4/2022-05-cally-findings/issues/39):

> reference issue: https://github.com/code-423n4/2022-05-cally-findings/issues/39

HardlyDifficult (judge) decreased severity to Medium and commented:

> This is a good description of the potential issue when a fee on transfer token is used.

> Lowing to 2 (Medium). See https://github.com/code-423n4/org/issues/3 for some discussion on how to consider the severity for these types of issues.

> The attack described does leak value, but the vault could be recovered by transferring in the delta balance so that the contract has more than enough funds

> in order to exercise or withdraw. That plus these types of tokens are relatively rare is why I don't think this warrants a High severity.

[M-09] Use safeTransferFrom instead of transferFrom for ERC721 transfers

Submitted by hickuphh3, also found by antonttc, berndartmueller, catchup, cccz, dipp, FSchmoede, GimelSec, hake, jah, jayjonah8, joestakey, kebabsec, Kenshin, Kumpa, MiloTruck, minhquanym, peritoflores, rfa, shenwilly, WatchPug, and ynnad

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L199

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L295

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L344

Details & Impact

The `transferFrom()` method is used instead of `safeTransferFrom()`, presumably to save gas. I however argue that this isn't recommended because:

- OpenZeppelin's documentation discourages the use of `transferFrom()`, use `safeTransferFrom()` whenever possible
- Given that any NFT can be used for the call option, there are a few NFTs (here's an example) that have logic in the `onERC721Received()` function, which is only triggered in the `safeTransferFrom()` function and not in `transferFrom()`

Recommended Mitigation Steps

Call the `safeTransferFrom()` method instead of `transferFrom()` for NFT transfers. Note that the `CallyNft` contract should inherit the `ERC721TokenReceiver` contract as a consequence.

```
abstract contract CallyNft is ERC721("Cally", "CALL"), ERC721Tok
```

outdoteth (Cally) confirmed and resolved:

> the fix for this issue is here; https://github.com/outdoteth/cally/pull/4

[M-10] `createVault()` does not confirm whether `tokenType` and `token`'s type are the same

Submitted by GimelSec, also found by antonttc

https://github.com/code-423n4/2022-05-cally/blob/main/contracts/src/Cally.sol#L158-L201

https://github.com/code-423n4/2022-05-cally/blob/main/contracts/src/Cally.sol#L296

https://github.com/code-423n4/2022-05-cally/blob/main/contracts/src/Cally.sol#L345

Impact

When calling `createVault()`, `tokenType` could be different from `token`'s type. If a user accidentally used the wrong `tokenType`, it could lead to two different results.

If `token` is an ERC20 token and the user uses `TokenType.ERC721` as `tokenType`. It is less harmful, since `ERC721(vault.token).transferFrom(msg.sender, address(this), vault.tokenIdOrAmount)` still works when `vault.token` is actually ERC20 token.

However, if `token` is an ERC721 token and the user uses `TokenType.ERC20` as `tokenType`. When doing `creatVault()`, `ERC20(vault.token).safeTransferFrom(msg.sender, address(this), vault.tokenIdOrAmount)` works fine. But when doing `exercise()` or `withdraw()`, `ERC20(vault.token).safeTransfer(msg.sender,` `vault.tokenIdOrAmount);` doesn't work since ERC721 doesn't implement `safeTransfer()` function. In consequence, the ERC721 token is frozen in the vault.

Proof of Concept

`createVault()` does not confirm whether `tokenType` and `token`'s type are the same. But the token can still be transferred into this contract. Since `transferFrom()` is implemented in ERC20 and `safeTransferFrom()` is implemented in ERC721 https://github.com/code-423n4/2022-05-cally/blob/main/contracts/src/Cally.sol#L158-L201

```
    function createVault(
        uint256 tokenIdOrAmount,
        address token,
        uint8 premiumIndex,
        uint8 durationDays,
        uint8 dutchAuctionStartingStrikeIndex,
        uint256 dutchAuctionReserveStrike,
        TokenType tokenType
    ) external returns (uint256 vaultId) {
        require(premiumIndex < premiumOptions.length, "Invalid p
        require(dutchAuctionStartingStrikeIndex < strikeOptions.
        require(dutchAuctionReserveStrike < strikeOptions[dutchA
        require(durationDays > 0, "durationDays too small");
        require(tokenType == TokenType.ERC721 || tokenType == To

        Vault memory vault = Vault({
            tokenIdOrAmount: tokenIdOrAmount,
            token: token,
            premiumIndex: premiumIndex,
            durationDays: durationDays,
            dutchAuctionStartingStrikeIndex: dutchAuctionStartin
            currentExpiration: uint32(block.timestamp),
            isExercised: false,
            isWithdrawing: false,
            tokenType: tokenType,
            currentStrike: 0,
            dutchAuctionReserveStrike: dutchAuctionReserveStrike
        });

        // vault index should always be odd
        vaultIndex += 2;
        vaultId = vaultIndex;
        _vaults[vaultId] = vault;

        // give msg.sender vault token
        _mint(msg.sender, vaultId);

        emit NewVault(vaultId, msg.sender, token);

        // transfer the NFTs or ERC20s to the contract
        vault.tokenType == TokenType.ERC721
            ? ERC721(vault.token).transferFrom(msg.sender, addre
            : ERC20(vault.token).safeTransferFrom(msg.sender, ac
    }
```

However when doing `exercise()` or `withdraw()` , it always reverts since ERC721 doesn't implement `safeTransfer()` . The ERC721 token is frozen in the contract.

https://github.com/code-423n4/2022-05-cally/blob/main/contracts/src/Cally.sol#L296

```
    function exercise(uint256 optionId) external payable {
        …
        // transfer the NFTs or ERC20s to the exerciser
        vault.tokenType == TokenType.ERC721
            ? ERC721(vault.token).transferFrom(address(this), ms
            : ERC20(vault.token).safeTransfer(msg.sender, vault.
    }
```

https://github.com/code-423n4/2022-05-cally/blob/main/contracts/src/Cally.sol#L345

```
    function withdraw(uint256 vaultId) external nonReentrant {
        …
        // transfer the NFTs or ERC20s back to the owner
        vault.tokenType == TokenType.ERC721
            ? ERC721(vault.token).transferFrom(address(this), ms
            : ERC20(vault.token).safeTransfer(msg.sender, vault.
    }
```

Recommended Mitigation Steps

Confirm whether `tokenType` and `token`'s type are the same in `createVault()`.

outdoteth (Cally) disputed, disagreed with severity and commented:

> ref; https://github.com/code-423n4/2022-05-cally-findings/issues/38

HardlyDifficult (judge) decreased severity to Medium and commented:

> There were a lot of reports recommending a similar change, but this is one of the few that points our a critical issue that could arise in the current state.

> Although the issue only occurs when the original vault creator makes a user error, the fact that their NFT becomes unrecoverable makes this a Medium Risk concern.

Low Risk and Non-Critical Issues

For this contest, 67 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **hubble** received the top score from the judge.

The following wardens also submitted reports: VAD37, IllIllI, shung, MiloTruck, ellahi, MaratCerby, WatchPug, reassor, Picodes, shenwilly, 0xDjango, Bludya, pmerkleplant, hake, joestakey, robee, cccz, hyh, sseefried, hickuphh3, seanamani, djxploit, dirk_y, Kumpa, fatherOfBlocks, pedroais, catchup, radoslav11, csanuragjain, BowTiedWardens, Cityscape, sorrynotsorry, z3s, 0xsanson, minhquanym, sikorico, Czar102, jah, 0x4non, bobirichman, antonttc, BondiPestControl, 0x1f8b, FSchmoede, hansfriese, rfa, 0xf15ers, TrungOre, Aits, 242, Waze, horsefacts, Funen, Kenshin, mics, CertoraInc, Hawkeye, Ruhum, dipp, 0x1337, cryptphi, BouSalman, eccentricexit, AlleyCat, delfin454000, *and* JDeryl.

[L-01] Wrong error message string in function createVault()

Function : createVault() in Cally.sol

line 169 require(dutchAuctionReserveStrike < strikeOptions[dutchAuctionStartingStrikeIndex], "Reserve strike too small");

Impact

If the current error message is followed, user will never be able to successfully createVault()

Recommended Mitigation Steps

Correct error message string

> require(dutchAuctionReserveStrike < strikeOptions[dutchAuctionStartingStrikeIndex], “Reserve strike more than Starting strike”);

[L-02] Function vaults() can return misleading information

VaultId are odd in number; if a valultId of event number is given, the function valuts() will return misleading information.

Function valuts() in Cally.sol

Recommended Mitigation Steps

Check if the valutID parameter is of vault type, by adding a require statement

```
    function vaults(uint256 vaultId) external view returns (Vault me
        require(vaultId % 2 != 0, "Not vault type");
        return _vaults[vaultId];
    }
```

[L-03] initiateWithdraw() should not be callable , if option already exercised

If the option is already exercised, the vault owner should not be allowed to call the initiateWithdraw() function.

Recommended Mitigation Steps

Add a require statement in the function initiateWithdraw()

> require(vault.isExercised == false, “Vault already exercised”);

[L-04] Ambiguous error message in createVault() for durationDays

If a value of 0 is given for durationDays in the createVault() function, the transaction will revert with an ambigous message "durationDays too small" It can better stated as given below

Recommended Mitigation Steps

Error message string can be changed as below.

> line 170 require(durationDays > 0, "durationDays cannot be zero");

[L-05] The available values in premiumOptions[] & strikeOptions[] are too restrictive

To reduce gas and storage, the protocol has currently designed to store the index of the premiumOptions[] & strikeOptions[] in the Vault structure.

Impact

This is too restrictive and may not be future proof.

Recommended Mitigation Steps

One suggestion is to add another member unit8 premiumMultiplier (with default value of 1) in the Vault struct, and users can have combination of values of the premiumMultiplier and premiumIndex to define more range of premium values if required.

Same suggestion applies for adding a multiplier for the strikeOptions[]

[L-06] No event is raised when feeRate is changed

Impact

When feeRate is changed at setFee(Cally.sol), no event is raised. It would be important to raise this event for any external integration with this system.

Proof of Concept

Contract : Cally.sol Function : setFee

Recommended Mitigation Steps

event definition

```
    event FeeRateUpdated(uint256 newFeeRate);
```

event emit at setFee function

```
    require(feeRate_ != feeRate, "new feeRate should be different");
    feeRate = feeRate_;
    emit FeeRateUpdated(feeRate_);
```

[L-07] No event is raised when vault beneficiary is changed

Impact

When beneficiary is changed at setVaultBeneficiary(Cally.sol), no event is raised. It would be important to raise this event for any external integration with this system.

Proof of Concept

Contract : Cally.sol Function : setVaultBeneficiary

Recommended Mitigation Steps

event definition

```
    event VaultBeneficiaryUpdated(uint256 indexed vaultId, address i
```

event emit at setVaultBeneficiary function

```
    emit VaultBeneficiaryUpdated(vaultId, beneficiary);
```

[N-01] Consistency in fetching vault values

In Cally.sol, function buyOption the following is the order of lines. line 208 require(vaultId % 2 != 0, “Not vault type”); line 211 Vault memory vault = _vaults[vaultId];

This can be made consistent with other functions by changing the order.

Vault memory vault = _vaults[vaultId]; require(vaultId % 2 != 0, “Not vault type”);

[N-02] valutIndex = 1 is never used

The value of vaultIndex = 1 is never assigned to any vault.

```
            // vault index should always be odd
            vaultIndex += 2;
            vaultId = vaultIndex;
            _vaults[vaultId] = vault;
```

This can be changed to as below, so that vaultID = 1 is also used

```
            vaultId = vaultIndex;
            // vault index should always be odd
            vaultIndex += 2;
            _vaults[vaultId] = vault;
```

outdoteth (Cally) commented:

> high quality report

Gas Optimizations

For this contest, 59 reports were submitted by wardens detailing gas optimizations. The report highlighted below by IllIllI received the top score from the judge.

The following wardens also submitted reports: BowTiedWardens, shung, rotcivegaf, TomFrenchBlockchain, MaratCerby, MiloTruck, VAD37, hickuphh3, Bludya, WatchPug, robee, rfa, RagePit, defsec, TerrierLover, joestakey, 0x4non, CertoraInc, horsefacts, 0x1f8b, djxploit, Kenshin, gzeon, ellahi, minhquanym, sikorico, antonttc, 0xf15ers, Waze, fatherOfBlocks, pmerkleplant, 0xkatana, Cityscape, MadWookie, bobirichman, catchup, oyc_109, samruna, 0xDjango, _Adam, delfin454000, 0xNazgul, 0xsanson, Picodes, hansfriese, mics, simon135, DavidGialdi, ignacio, Tadashi, Funen, hake, z3s, 0v3rf10w, Hawkeye, Fitraldys, jonatascm, *and* FSchmoede.

Summary

	Issue	Instances
1	Full vault details unnecessarily fetched twice in each call to `buyOption()`	1
2	Re-creation of short `memory` arrays is cheaper than fetching indecies from a storage array	2
3	Use existing stack cache variable rather than re-fetching state variable	1
4	Cheaper to split struct if only part of it is updated frequently	1
5	Store actual beneficiary rather than deciding every time	2
6	Use `unchecked {}` for calculations that cannot overflow	1
7	Also `_burn()` the vault to get a gas refund	1
8	State variables should be cached in stack variables rather than re-reading them from storage	3
9	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables	2
10	`<array>.length` should not be looked up in every loop of a `for`-loop	1
11	`++i`/`i++` should be `unchecked{++i}`/`unchecked{++i}` when it is not possible for them to overflow, as is the case when used in `for`- and `while`-loops	1
12	Offsets should only be calculated once per loop	1
13	Access mappings directly rather than using accessor functions	5
14	Cheaper input valdiations should come before expensive operations	2
15	Not using the named return variables when a function returns, wastes deployment gas	2

	Issue	Instances
16	Use a more recent version of solidity	1
17	Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement	1
18	It costs more gas to initialize variables to zero than to let the default of zero be applied	2
19	`++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)	1
20	Usage of `uints`/`ints` smaller than 32 bytes (256 bits) incurs overhead	10
21	Using `private` rather than `public` for constants, saves gas	1
22	Don't compare boolean expressions to boolean literals	3
23	Duplicated `require()`/`revert()` checks should be refactored to a modifier or function	4
24	Multiplication/division by two should use bit shifting	3
25	Use custom errors rather than `revert()`/`require()` strings to save deployment gas	32
26	Functions guaranteed to revert when called by normal users can be marked `payable`	2

Total: 86 instances over 26 issues

[G-01] Full vault details unnecessarily fetched twice in each call to `buyOption()`

`getPremium()` should be changed to have its argument be an index into the array, rather than looking up the vault again. Since this is in one of the two frequently-called functions, it'll save a lot of gas

There is 1 instance of this issue:

```
File: contracts/src/Cally.sol   #1
```

```
223:         uint256 premium = getPremium(vaultId);
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L223

[G-02] Re-creation of short `memory` arrays is cheaper than fetching indecies from a storage array

Since the arrays are relatively short, it's cheaper to have a pure virtual function that re-creates them every time just to fetch a specific index, rather than incuring a Gcoldsload (2100 gas). Since this is in one of the two frequently-called functions, it'll save a lot of gas

There are 2 instances of this issue:

```
File: ./contracts/src/Cally.sol   #1

90:      uint256[] public premiumOptions = [0.01 ether, 0.025 et
```

./https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L90

```
File: contracts/src/Cally.sol   #2

92:      uint256[] public strikeOptions = [1 ether, 2 ether, 3 e
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L92

[G-03] Use existing stack cache variable rather than re-fetching state variable

Using `auctionStartTimestamp` defined above will save a Gwarmaccess (100 gas). Since this is in one of the two frequently-called functions, it'll save a lot of gas

There is 1 instance of this issue:

```
File: contracts/src/Cally.sol   #1

233:                vault.currentExpiration + AUCTION_DURATION,
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L233

[G-04] Cheaper to split struct if only part of it is updated frequently

The `currentStrike` and `currentExpiration` fields are updated frequently so they should be in a separate struct rather than re-writing the whole struct every time

There is 1 instance of this issue:

```
File: contracts/src/Cally.sol   #1

230          // set new currentStrike
231          vault.currentStrike = getDutchAuctionStrike(
232              strikeOptions[vault.dutchAuctionStartingStrike]
233              vault.currentExpiration + AUCTION_DURATION,
234              vault.dutchAuctionReserveStrike
235          );
236
237          // set new expiration
238          vault.currentExpiration = uint32(block.timestamp) +
239
240          // update the vault with the new option expiration
241:         _vaults[vaultId] = vault;
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.s ol#L230-L241

[G-05] Store actual beneficiary rather than deciding every time

If the beneficiary array assigns the owner every time the NFT is minted or transferred, the mapping can be used directly, saving the gas overhead of function calls

There are 2 instances of this issue:

```
File: contracts/src/Cally.sol   #1

249:         address beneficiary = getVaultBeneficiary(vaultId);
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L249

```
File: contracts/src/Cally.sol   #2

289:         ethBalance[getVaultBeneficiary(vaultId)] += msg.val
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L289

[G-06] Use `unchecked {}` for calculations that cannot overflow

The subtraction below can be `unchecked {}` because of the check that comes before it

There is 1 instance of this issue:

```
File: contracts/src/Cally.sol   #1
```

```
417:         uint256 delta = auctionEndTimestamp > block.timesta
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L417

[G-07] Also `_burn()` the vault to get a gas refund

There is 1 instance of this issue:

```
File: contracts/src/Cally.sol   #1

275:         _burn(optionId);
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L275

[G-08] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching will replace each Gwarmaccess (100 gas) with a much cheaper stack read. Less obvious fixes/optimizations include having local storage variables of mappings within state variable mappings or mappings within state variable structs, having local storage variables of structs within mappings, having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

There are 3 instances of this issue:

```
File: contracts/src/Cally.sol   #1

/// @audit strikeOptions
169:          require(dutchAuctionReserveStrike < strikeOptions[
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L169

```
File: contracts/src/Cally.sol   #2

/// @audit feeRate
284:                fee = (msg.value * feeRate) / 1e18;
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L284

```
File: contracts/src/Cally.sol   #3

/// @audit vaultIndex
189:            vaultId = vaultIndex;
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L189

[G-09] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

There are 2 instances of this issue:

```
File: contracts/src/Cally.sol   #1

188:            vaultIndex += 2;
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L188

```
File: contracts/src/Cally.sol   #2
```

```
285:                protocolUnclaimedFees += fee;
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L285

[G-10] `<array>.length` should not be looked up in every loop of a `for`-loop

The overheads outlined below are *PER LOOP*, excluding the first loop

- storage arrays incur a Gwarmaccess (100 gas)
- memory arrays use `MLOAD` (3 gas)
- calldata arrays use `CALLDATALOAD` (3 gas)

Caching the length changes each of these to a `DUP<N>` (3 gas), and gets rid of the extra `DUP<N>` needed to store the stack offset

There is 1 instance of this issue:

```
File: contracts/src/CallyNft.sol   #1

244:            for (uint256 i = 0; i < data.length; i++) {
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/CallyNft.sol#L244

[G-11] `++i`/`i++` should be `unchecked{++i}`/`unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for`- and `while`-loops

The `unchecked` keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves 30-40 gas [***PER LOOP***]

There is 1 instance of this issue:

```
File: contracts/src/CallyNft.sol   #1

244:          for (uint256 i = 0; i < data.length; i++) {
```

[https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/CallyNft.sol#L244]

[G-12] Offsets should only be calculated once per loop

`i * 2` is calculated twice per loop, wasting gas

There is 1 instance of this issue:

```
File: contracts/src/CallyNft.sol   #1

244          for (uint256 i = 0; i < data.length; i++) {
245              str[2 + i * 2] = alphabet[uint256(uint8(data[i]
246              str[3 + i * 2] = alphabet[uint256(uint8(data[i]
247:         }
```

[https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/CallyNft.sol#L244-L247]

[G-13] Access mappings directly rather than using accessor functions

Saves having to do two JUMP instructions, along with stack setup

There are 5 instances of this issue:

```
File: contracts/src/Cally.sol

214:        require(ownerOf(vaultId) != address(0), "Vault does

263:        require(msg.sender == ownerOf(optionId), "You are r

307:        require(msg.sender == ownerOf(vaultId), "You are nc

323:        require(msg.sender == ownerOf(vaultId), "You are nc

354:        require(msg.sender == ownerOf(vaultId), "Not owner'
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol

[G-14] Cheaper input valdiations should come before expensive operations

There are 2 instances of this issue:

```
File: contracts/src/Cally.sol   #1

211:        require(vaultId % 2 != 0, "Not vault type");
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L211

```
File: contracts/src/Cally.sol   #2

214:        require(ownerOf(vaultId) != address(0), "Vault does
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L214

[G-15] Not using the named return variables when a function returns, wastes deployment gas

There are 2 instances of this issue:

```
File: contracts/src/Cally.sol   #1

382:            return currentBeneficiary == address(0) ? ownerOf
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L382

```
File: contracts/src/Cally.sol   #2

396:            return premiumOptions[vault.premiumIndex];
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L396

[G-16] Use a more recent version of solidity

Use a solidity version of at least 0.8.0 to get overflow protection without `SafeMath`

Use a solidity version of at least 0.8.2 to get compiler automatic inlining

Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads

Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings

Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

There is 1 instance of this issue:

```
File: contracts/lib/base64/base64.sol   #1

3:    pragma solidity >=0.6.0;
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/lib/base64/base64.sol#L3

[G-17] Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement

This change saves 6 gas per instance

There is 1 instance of this issue:

```
File: contracts/src/Cally.sol   #1

170:          require(durationDays > 0, "durationDays too small'
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L170

[G-18] It costs more gas to initialize variables to zero than to let the default of zero be applied

There are 2 instances of this issue:

```
File: contracts/src/Cally.sol   #1

282:          uint256 fee = 0;
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L282

```
File: contracts/src/CallyNft.sol   #2

244:            for (uint256 i = 0; i < data.length; i++) {
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/CallyNft.sol#L244

[G-19] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)

Saves 6 gas *PER LOOP*

There is 1 instance of this issue:

```
File: contracts/src/CallyNft.sol   #1

244:            for (uint256 i = 0; i < data.length; i++) {
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/CallyNft.sol#L244

[G-20] Usage of `uints`/`ints` smaller than 32 bytes (256 bits) incurs overhead

> When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html Use a larger size then downcast where needed

There are 10 instances of this issue:

```
File: contracts/src/Cally.sol

76:            uint8 premiumIndex; // indexes into `premiumOptio

77:            uint8 durationDays; // days

78:            uint8 dutchAuctionStartingStrikeIndex; // indexes

79:            uint32 currentExpiration;

87:        uint32 public constant AUCTION_DURATION = 24 hours;

161:           uint8 premiumIndex,

162:           uint8 durationDays,

163:           uint8 dutchAuctionStartingStrikeIndex,

227:           uint32 auctionStartTimestamp = vault.currentExpir

408:           uint32 auctionEndTimestamp,
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol

[G-21] Using `private` rather than `public` for constants, saves gas

If needed, the value can be read from the verified contract source code. Savings are due to the compiler not having to create non-payable getter functions for deployment calldata, and not adding another entry to the method ID table

There is 1 instance of this issue:

```
File: contracts/src/Cally.sol   #1

87:        uint32 public constant AUCTION_DURATION = 24 hours;
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.s ol#L87

[G-22] Don’t compare boolean expressions to boolean literals

`if (<x> == true)` => `if (<x>)`, `if (<x> == false)` => `if (!<x>)`

There are 3 instances of this issue:

```
File: contracts/src/Cally.sol   #1

217:          require(vault.isExercised == false, "Vault already
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L217

```
File: contracts/src/Cally.sol   #2

220:          require(vault.isWithdrawing == false, "Vault is be
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L220

```
File: contracts/src/Cally.sol   #3

328:          require(vault.isExercised == false, "Vault already
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L328

[G-23] Duplicated `require()` / `revert()` checks should be refactored to a modifier or function

Saves deployment costs

There are 4 instances of this issue:

```
File: contracts/src/Cally.sol   #1

304:            require(vaultId % 2 != 0, "Not vault type");
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L304

```
File: contracts/src/Cally.sol   #2

328:            require(vault.isExercised == false, "Vault already
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L328

```
File: contracts/src/Cally.sol   #3

323:            require(msg.sender == ownerOf(vaultId), "You are n
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L323

```
File: contracts/src/CallyNft.sol   #4

42:            require(to != address(0), "INVALID_RECIPIENT");
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/CallyNft.sol#L42

[G-24] Multiplication/division by two should use bit shifting

`<x> * 2` is equivalent to `<x> << 1` and `<x> / 2` is the same as `<x> >> 1`. The `MUL` and `DIV` opcodes cost 5 gas, whereas `SHL` and `SHR` only cost 3 gas

There are 3 instances of this issue:

```
File: contracts/src/CallyNft.sol   #1

241:          bytes memory str = new bytes(2 + data.length * 2);
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/CallyNft.sol#L241

```
File: contracts/src/CallyNft.sol   #2

245:              str[2 + i * 2] = alphabet[uint256(uint8(data[i
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/CallyNft.sol#L245

```
File: contracts/src/CallyNft.sol   #3

246:              str[3 + i * 2] = alphabet[uint256(uint8(data[i
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/CallyNft.sol#L246

[G-25] Use custom errors rather than `revert()` / `require()` strings to save deployment gas

Custom errors are available from solidity version 0.8.4. The instances below match or exceed that version

There are 32 instances of this issue:

```
File: contracts/src/Cally.sol

167:            require(premiumIndex < premiumOptions.length, "Inv

168:            require(dutchAuctionStartingStrikeIndex < strikeOp

169:            require(dutchAuctionReserveStrike < strikeOptions[

170:            require(durationDays > 0, "durationDays too small'

171:            require(tokenType == TokenType.ERC721 || tokenType

211:            require(vaultId % 2 != 0, "Not vault type");

214:            require(ownerOf(vaultId) != address(0), "Vault doe

217:            require(vault.isExercised == false, "Vault already

220:            require(vault.isWithdrawing == false, "Vault is be

224:            require(msg.value >= premium, "Incorrect ETH amour

228:            require(block.timestamp >= auctionStartTimestamp,

260:            require(optionId % 2 == 0, "Not option type");

263:            require(msg.sender == ownerOf(optionId), "You are

269:            require(block.timestamp < vault.currentExpiration,

272:            require(msg.value == vault.currentStrike, "Incorre

304:            require(vaultId % 2 != 0, "Not vault type");

307:            require(msg.sender == ownerOf(vaultId), "You are n

320:            require(vaultId % 2 != 0, "Not vault type");

323:            require(msg.sender == ownerOf(vaultId), "You are n

328:            require(vault.isExercised == false, "Vault already

329:            require(vault.isWithdrawing, "Vault not in withdra
330:            require(block.timestamp > vault.currentExpiration,

353:            require(vaultId % 2 != 0, "Not vault type");

354:            require(msg.sender == ownerOf(vaultId), "Not owne

436:            require(from == _ownerOf[id], "WRONG_FROM");

437:            require(to != address(0), "INVALID_RECIPIENT");

438             require(
439                 msg.sender == from || isApprovedForAll[from][m
440                 "NOT_AUTHORIZED"
441:            );

456:            require(_ownerOf[tokenId] != address(0), "URI quer
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol

```
File: contracts/src/CallyNft.sol

15:             require(to != address(0), "INVALID_RECIPIENT");

16:             require(_ownerOf[id] == address(0), "ALREADY_MINTE

36:             require(owner != address(0), "ZERO_ADDRESS");

42:             require(to != address(0), "INVALID_RECIPIENT");
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/CallyNft.sol

[G-26] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVERT` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about 21 gas per call to the function, in addition to the extra deployment cost

There are 2 instances of this issue:

```
File: contracts/src/Cally.sol   #1

119:       function setFee(uint256 feeRate_) external onlyOwner {
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L119

```
File: contracts/src/Cally.sol   #2

124:       function withdrawProtocolFees() external onlyOwner ret
```

https://github.com/code-423n4/2022-05-cally/blob/1849f9ee12434038aa80753266ce6a2f2b082c59/contracts/src/Cally.sol#L124

outdoteth (Cally) commented:

> this is the best gas report imo - hats off to you!

Disclosures

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

Top