YIELD

Yield Witch v2 contest Findings & Analysis Report

2022-10-03

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 Yield Witch v2 smart contract system written in Solidity. The audit contest took place between July 14—July 17 2022.

Wardens

64 Wardens contributed reports to the Yield Witch v2 contest:

1. csanuragjain
2. antonttc
3. 0x52
4. hickuphh3
5. IIIIIII
6. horsefacts
7. hake
8. 0x29A (0x4non and rotcivegaf)
9. Meera
10. Waze
11. ak1
12. 0xNazgul
13. rajatbeladiya
14. __141345__
15. TomJ
16. rbserver
17. simon135
18. wastewa
19. Chom
20. kyteg
21. c3phas
22. Trumpero

23. ElKu
24. karanctf
25. cRat1st0s
26. rokinot
27. Deivitto
28. fatherOfBlocks
29. ReyAdmirado
30. pashov
31. SooYa
32. hyh
33. peritoflores
34. ladboy233
35. kenzo
36. exd0tpy
37. hansfriese
38. joestakey
39. asutorufos
40. reassor
41. delfin454000
42. Funen
43. m_Rassska
44. gogo
45. Limbooo
46. MadWookie
47. 0xKitsune
48. defsec
49. samruna
50. JC
51. JohnSmith

52. robee
53. sashik_eth
54. Aymen0909
55. tofunmi
56. ajtra
57. 0x1f8b
58. bulej93
59. Sm4rty
60. Rohan16
61. ignacio
62. durianSausage
63. Kaiziron

This contest was judged by PierrickGT.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 2 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 0 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 40 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 50 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 Yield Witch v2 contest repository, and is composed of 1 smart contract written in the Solidity programming language and includes 527 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 (2)

[H-01] Someone can create non-liquidatable auction if the collateral asset fails on transferring to `address(0)`

Submitted by antonttc, also found by 0x52

Witch.sol#L176
Witch.sol#L399

Might lead to systematic debt. Cause errors for liquidators to run normally.

Proof of Concept

In the function `auction`, there is an input validation around whether the `to` is `address(0)` or not. And if the `auctioneerReward` is set to an value > 0 (as default), each liquidate call will call `Join` module to pay out to `auctioneer` with the following line:

```
    if (auctioneerCut > 0) {
        ilkJoin.exit(auction_.auctioneer, auctioneerCut.u128());
    }
```

This line will revert if `auctioneer` is set to `address(0)` on some tokens (revert on transferring to address(0) is a default behaviour of the OpenZeppelin template). So if someone start an `auction` with `to = address(0)`, this auction becomes un-liquidatable.

A malicious user can run a bot to monitor his own vault, and if they got underwater and they don't have enough collateral to top up, they can immediately start an auction on their own vault and set actioneer to `0` to avoid actually being liquidated, which breaks the design of the system.

Recommended Mitigation Steps

Add check while starting an auction:

```
    function auction(bytes12 vaultId, address to)
        external
        returns (DataTypes.Auction memory auction_)
    {
        require (to != address(0), "invalid auctioneer");
                    ...
    }
```

alcueca (Yield) confirmed and commented:

> Best finding of the contest 🏆

PierrickGT (judge) commented:

> Most critical vulnerability found during the audit since a malicious user could open a vault and never get liquidated, it would force the protocol to take on bad debts. The warden did a great job of describing the issue and providing the sponsor with a detailed fix.

[H-02] Incorrect amount of Collateral moves for Auction

Submitted by csanuragjain

It was observed that the debt and collateral which moves for Auction is calculated incorrectly. In case where `line.proportion` is set to small value, chances are art will become lower than min debt. This causes whole collateral to go for auction, which was not expected.

Proof of Concept

1. Assume `line.proportion` is set to 10% which is a valid value
2. Auction is started on Vault associated with collateral & base representing line from Step 1
3. Now debt and collateral to be sold are calculated in _calcAuction

```
uint128 art = uint256(balances.art).wmul(line.proportion).u128()
        if (art < debt.min * (10**debt.dec)) art = balances.art;
        uint128 ink = (art == balances.art)
            ? balances.ink
            : uint256(balances.ink).wmul(line.proportion).u128()
```

4. Now lets say **debt (art)** on this vault was **amount 10**, **collateral (ink)** was **amount 9**, debt.min * (10**debt.dec) was **amount 2**
5. Below calculation occurs

```
uint128 art = uint256(balances.art).wmul(line.proportion).u128()
        if (art < debt.min * (10**debt.dec)) art = balances.art;
        uint128 ink = (art == balances.art)
            ? balances.ink
            : uint256(balances.ink).wmul(line.proportion).u128()
```

6. So full collateral and full debt are placed for Auction even though only 10% was meant for Auction. Even if it was lower than min debt, auction amount should have only increased up to the point where minimum debt limit is reached

Recommended Mitigation Steps

Revise the calculation like below

```
uint128 art = uint256(balances.art).wmul(line.proportion).u128()
uint128 ink=0;
        if (art < debt.min * (10**debt.dec))
{
art = debt.min * (10**debt.dec);
(balances.ink<art) ? (ink=balances.ink) : (ink=art)
} else {
ink=uint256(balances.ink).wmul(line.proportion).u128();
}
```

hickuphh3 (warden) commented:

> debt.min * (10**debt.dec) was amount 2
>
> Only way for this to happen is for the token's decimals to be 0, which is an edge case.
>
> Anyway, the issue is invalid because it is intended for the full collateral to be offered if it is below the minimum debt amount, ie. vault proportion is to be disregarded: `// We store the proportion of the vault to auction, which is the whole vault if the debt would be below dust.`

alcueca (Yield) confirmed and commented:

> The finding is valid, but it is a bit complicated.
>
> The behaviour should be:
>
> 1. If the part of the vault for auction is below `dust`, increase to `dust`.
> 2. If the remaining part of the vault is below `dust`, increase to 100%.

PierrickGT (judge) commented:

> This is the second most critical vulnerability found during the audit.
> This issue is less critical than H-01 (#116) since the protocol would not take on bad

> debts but users may lose their entire collateral when only part of their collateral should have been put to auction.

Low Risk and Non-Critical Issues

For this contest, 40 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **hickuphh3** received the top score from the judge.

The following wardens also submitted reports: horsefacts, lllllll, 0x29A, Waze, ak1, 0xNazgul, wastewa, rajatbeladiya, Meera, __141345__, hyh, peritoflores, simon135, Chom, rbserver, ladboy233, 0x52, kenzo, exd0tpy, hansfriese, Trumpero, karanctf, c3phas, csanuragjain, asutorufos, rokinot, Deivitto, cRat1st0s, reassor, TomJ, delfin454000, hake, kyteg, ReyAdmirado, ElKu, pashov, Funen, fatherOfBlocks, *and* SooYa.

Codebase Impressions & Summary

Functionality

The revised witch (liquidation engine) contract includes the following improvements over the previous version. As stated in the README, they are:

1. Greater flexibility in exploring different liquidation models.
2. Making liquidations more profitable for liquidators by allowing payments in `fyToken`.
3. Introduce a mechanism to reward starting an auction.
4. Allow fine-tuning of all parameters for any collateral/underlying pair.
5. Correct bugs.

The liquidations flow was quite easy to follow as it consists of the following:

1. Liquidation parameters are defined by governance functions (auction duration, vault proportion, auctioneer reward etc.)
2. Starting an auction: `auction()`
3. Liquidators executing the liquidations: `payBase()` and `payFYToken()`

4. Either the entire vault collateral has been auctioned off, or `cancel()` is called to prematurely end the auction

Documentation

The README was very extensive and thorough, and succinctly explained design considerations made. Flow diagrams were provided to help visualise the interactions required between different contracts. Inline comments were appropriate too, aided in understanding the functionality.

Tests

All foundry tests passed as expected. One area of improvement is to have mainnet forking tests, since mocking is used for the external contracts. Running `forge coverage` unfortunately didn't work. I suspect it is due to the instability of the feature rather than the fault of the tests.

Gas Optimizations

Casting could be avoided if input / output params were defined appropriately. For instance, `inkOut`, `artIn` in `_updateAccounting()`, and `liquidatorCut` and `auctioneerCut` could have been `uint128` instead of `uint256`.

[L-01] Vaults that are over-collateralised after partial liquidation are possibly subject to further liquidations

If a vault becomes over-collateralised after a partial liquidation, it is still subject to further liquidation as the auction isn't closed. The vault owner has to call `cancel()` himself, or trust other altruistic actors to perform this action on his behalf. Liquidators will unlikely do it because they are economically incentivised not to do so.

One can however argue that this is mitigated by the fact that protocol (governance) sets the vault proportion that can be auctioned. Regardless of whether the fact that the vault is over-collateralised after partial liquidations, the liquidators arguably are given the right to carry out further liquidations up to the proportion set.

Nevertheless, a reason for a revised liquidations witch contract is that "More often than not, liquidated users have lost all their collateral as we have failed to make

liquidations competitive.”. Hence, it might make sense to ensure that users need not lose more collateral than necessary.

Recommended Mitigation Steps

Consider checking if the vault is over-collateralized (maybe in `_updateAccounting()`) and close the auction if it is. This however adds complexity to the liquidation logic, as you have to update the cauldron first `cauldron.slurp()` before checking and updating the collateralization status. It will also break the CEI pattern, which might be unfavourable.

[L-02] Comparison in `_calcPayout()` should include equality

Witch.sol#L586

TLDR

```
- else if (elapsed > duration)
+ else if (elapsed >= duration)
```

Description

In the case where `elapsed == duration`, `proportionNow` evaluates to `1e18`, which is the same result when `elapsed > duration`. Proof below.

```
proportionNow =
  uint256(initialProportion) +
  uint256(1e18 - initialProportion).wmul(elapsed.wdiv(duration))

// = initialProportion + (1e18 - initialProportion).wmul(1e18)
// = initialProportion + (1e18 - initialProportion) * 1e18 / 1e1
// = initialProportion + 1e18 - initialProportion
// = 1e18
```

Of slightly greater importance, this handles the edge case when `elapsed = duration = 0`, ie. the liquidation transaction is included in the same block / has the

same timestamp as the auction initialization transaction

Recommended Mitigation Steps

As per the TLDR.

P.S. Regarding zero duration auctions

Since the proportion given for zero duration auctions is `1e18`, it is equivalent to an auction of infinite duration with 100% initial offer: `duration == type(uint32).max` and `line_.initialOffer = 1e18`.

[L-03] Incorrect description for `auctioneerCut`

Witch.sol#L284

Witch.sol#L342

Description

Technically, the `auctioneerCut` goes to the `to` address specified by the auctioneer when `auction()` is called, which, while unlikely, may not be the auctioneer himself. Also, the comparison is done against the `to` address specified, not the caller / `msg.sender` as the comment implies.

Recommended Mitigation Steps

```
- Amount paid to whomever started the auction. 0 if it's the san
+ Amount paid to address specified by whomever started the aucti
```

[L-04] Incorrect natspec for `setLimit()`

Witch.sol#L118-L122

Description

The comments seem outdated as the only parameter that is updated by the function is the maximum collateral that can be concurrently auctioned off.

```
///  - the auction duration to calculate liquidation prices
///  - the proportion of the collateral that will be sold at auc
///  - the maximum collateral that can be auctioned at the same
///  - the minimum collateral that must be left when buying, unl
///  - The decimals for maximum and minimum
```

Recommended Mitigation Steps

Suggest removing / updating the referenced comments.

[N-01] Modify comment to soft limit check for clarity

[Witch.sol#L194-L196]

[Witch.sol#L200]

[Witch.sol#L204]

Description

The limit check is done before the summation to the total collateral allowable for liquidation. One may consider this to be a bug, but the README explains why this is the case:

> Note that the first auction to reach the limit is allowed to pass it, so that there is never the situation where a vault would be too big to ever be auctioned.

The inline comments have this as well, but isn't as clearly put as the README.

```
// There is a limit on how much collateral can be concurrently p
// If the limit has been surpassed, no more vaults of that colla
// This avoids the scenario where some vaults might be too large
```

Recommended Mitigation Steps

For greater clarity, I would suggesting modifying the inline comment to be worded similar as the README.

```
// There is a limit on how much collateral can be concurrently p
- // If the limit has been surpassed, no more vaults of that col
+ // The first auction to reach or exceed the limit is allowed t
// This avoids the scenario where some vaults might be too large
```

[N-02] Typos

```
- bellow
+ below

- differente
+ different

// Extra spacing
- The Join  then dishes out
+ The Join then dishes out

- quoutes hoy much ink
+ quotes how much ink
```

alcueca (Yield) confirmed and commented:

> This is a great QA report ❤️

PierrickGT (judge) commented:

> Best QA report of this contest with clear description of the problems and comprehensive suggestions provided.

> **[L-01] Vaults that are over-collateralised after partial liquidation are possibly subject to further liquidations**

> Great suggestion and explanation of the risks induced by the smart contract architecture and design of the Witch contract. As mentioned by the warden, their suggestion may complexify the code but they did a good job of outlining the risks and the sponsor can now decide to implement or not the suggestion.

> **[L-02] Comparison in _calcPayout() should include equality**

> Great find and recommendation that will cover the edge case when `elapsed = duration = 0`.

> **[L-03] Incorrect description for auctioneerCut**

> The comment was indeed not entirely clear and the suggestion provided by the warden should avoid any confusion in the future.

> **[L-04] Incorrect natspec for setLimit()**

> Good recommendation, it's always best to write up to date Natspect documentations to avoid any confusion during a future refactor of the code.

> **[N-01] Modify comment to soft limit check for clarity**

> Not critical, but the suggestion from the warden adds a bit of clarity to the comment.

> **[N-02] Typos**

> It's always best to avoid typos in the documentation of the code.

Gas Optimizations

For this contest, 50 reports were submitted by wardens detailing gas optimizations. The report highlighted below by IIIIIII received the top score from the judge.

The following wardens also submitted reports: hake, Meera, antonttc, joestakey, m_Rassska, csanuragjain, TomJ, gogo, rbserver, Limbooo, MadWookie, hickuphh3, kyteg, ak1, 0xKitsune, defsec, Waze, ElKu, samruna, c3phas, JC, rajatbeladiya, cRat1st0s, JohnSmith, robee, sashik_eth, __141345__, simon135, Aymen0909, fatherOfBlocks, Deivitto, tofunmi, 0xNazgul, ajtra, 0x1f8b, ReyAdmirado, pashov, bulej93, Trumpero, Sm4rty, Rohan16, rokinot, ignacio, durianSausage, 0x29A, karanctf, Kaiziron, Chom, *and* SooYa.

Summary

	Issue	Instances
[G-01]	Multiple `address` /ID mappings can be combined into a single `mapping` of an `address` /ID to a `struct` , where appropriate	1
[G-02]	Using `storage` instead of `memory` for structs/arrays saves gas	1
[G-03]	The result of external function calls should be cached rather than re-calling the function	3
[G-04]	Optimize names to save gas	1
[G-05]	Using `bool` s for storage incurs overhead	2
[G-06]	`>=` costs less gas than `>`	3
[G-07]	Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead	2
[G-08]	`require()` or `revert()` statements that check input arguments should be at the top of the function	1
[G-09]	Use custom errors rather than `revert()` / `require()` strings to save gas	17
[G-10]	Functions guaranteed to revert when called by normal users can be marked `payable`	6

Total: 37 instances over 10 issues

[G-01] Multiple `address` /ID mappings can be combined into a single `mapping` of an `address` /ID to a `struct` , where appropriate

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (**20000 gas**) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot. Finally, if both fields are accessed in the same function, can save **~42 gas per access** due to [not having to recalculate the key's keccak256 hash](#) (Gkeccak256 - 30 gas) and that calculation's associated stack operations.

There is 1 instance of this issue:

```
File: contracts/Witch.sol

66          mapping(bytes6 => mapping(bytes6 => DataTypes.Line)) p
67          mapping(bytes6 => mapping(bytes6 => DataTypes.Limits))
68          mapping(address => bool) public otherWitches;
69:         mapping(bytes6 => mapping(bytes6 => bool)) public ignc
```

[Witch.sol#L66-L69]

[G-02] Using `storage` instead of `memory` for structs/arrays saves gas

When fetching data from a storage location, assigning the data to a `memory` variable causes all fields of the struct/array to be read from storage, which incurs a Gcoldsload (**2100 gas**) for *each* field of the struct/array. If the fields are read from the new memory variable, they incur an additional `MLOAD` rather than a cheap stack read. Instead of declearing the variable with the `memory` keyword, declaring the variable with the `storage` keyword and caching any fields that need to be re-read in stack variables, will be much cheaper, only incuring the Gcoldsload for the fields actually read. The only time it makes sense to read the whole struct/array into a `memory` variable, is if the full struct/array is being returned by the function, is being passed to a function that requires `memory`, or if the array/struct is being read from another `memory` array/struct

There is 1 instance of this issue:

```
File: contracts/Witch.sol

197             DataTypes.Limits memory limits_ = limits[vault.ilk
198                 series.baseId
199:            ];
```

[Witch.sol#L197-L199]

```
diff --git a/contracts/Witch.sol b/contracts/Witch.sol
index f98dd6a..ccf9822 100644
--- a/contracts/Witch.sol
+++ b/contracts/Witch.sol
@@ -194,15 +194,15 @@ contract Witch is AccessControl {
         // There is a limit on how much collateral can be concu
         // If the limit has been surpassed, no more vaults of t
         // This avoids the scenario where some vaults might be
-        DataTypes.Limits memory limits_ = limits[vault.ilkId][
+        DataTypes.Limits storage limits_ = limits[vault.ilkId]|
             series.baseId
         ];
-        require(limits_.sum <= limits_.max, "Collateral limit r
+        uint128 lsum_ = limits_.sum;
+        require(lsum_ <= limits_.max, "Collateral limit reachec

         auction_ = _calcAuction(vault, series, to, balances, de

-        limits_.sum += auction_.ink;
-        limits[vault.ilkId][series.baseId] = limits_;
+        limits_.sum = lsum_ + auction_.ink;

         auctions[vaultId] = auction_;
```

```
diff --git a/gas_before b/gas_after
index 68d894d..749b496 100644
--- a/gas_before
+++ b/gas_after
@@ -3,11 +3,11 @@
 ╠═══════════════════════════════════════════╬═══════════════════╬═══
 ┃ Deployment Cost                           ┃ Deployment Size   ┃
 ┣-------------------------------------------------------------------┫
-┃ 3076398                                   ┃ 15658             ┃
+┃ 3062982                                   ┃ 15591             ┃
 ┣-------------------------------------------------------------------┫
 ┃ Function Name                             ┃ min               ┃ avg
 ┣-------------------------------------------------------------------┫
-┃ auction                                   ┃ 4219              ┃ 7011
+┃ auction                                   ┃ 4219              ┃ 6995
 ┣-------------------------------------------------------------------┫
 ┃ auctioneerReward                          ┃ 426               ┃ 426
 ┣-------------------------------------------------------------------┫
```

[G-03] The result of a function call should be cached rather than re-calling the function

The instances below point to the second+ call of the function within a single function

There are 3 instances of this issue:

```
File: contracts/Witch.sol

/// @audit inkOut.u128()
450:                  limits_.sum -= inkOut.u128();

/// @audit inkOut.u128()
/// @audit artIn.u128()
458:          cauldron.slurp(vaultId, inkOut.u128(), artIn.u128(
```

[Witch.sol#L450]

```
diff --git a/contracts/Witch.sol b/contracts/Witch.sol
index f98dd6a..c2196e8 100644
--- a/contracts/Witch.sol
+++ b/contracts/Witch.sol
@@ -421,6 +421,7 @@ contract Witch is AccessControl {
         ];

         // Update local auction
+        uint128 io128_;
         {
             if (auction_.art == artIn) {
                 // If there is no debt left, return the vault w
@@ -428,6 +429,7 @@ contract Witch is AccessControl {

                 // Update limits - reduce it by the whole aucti
                 limits_.sum -= auction_.ink;
+                io128_ = inkOut.u128();
             } else {
                 // Ensure enough dust is left
                 DataTypes.Debt memory debt = cauldron.debt(
@@ -439,15 +441,16 @@ contract Witch is AccessControl {
                     "Leaves dust"
```

```
                );

+               io128_ = inkOut.u128();
                // Update the auction
-               auction_.ink -= inkOut.u128();
+               auction_.ink -= io128_;
                auction_.art -= artIn.u128();

                // Store auction changes
                auctions[vaultId] = auction_;

                // Update limits - reduce it by whatever was bo
-               limits_.sum -= inkOut.u128();
+               limits_.sum -= io128_;
            }
        }

@@ -455,7 +458,7 @@ contract Witch is AccessControl {
        limits[auction_.ilkId][auction_.baseId] = limits_;

        // Update accounting at Cauldron
-       cauldron.slurp(vaultId, inkOut.u128(), artIn.u128());
+       cauldron.slurp(vaultId, io128_, artIn.u128());
    }

    /// @dev Logs that a certain amount of a vault was liquidat



diff --git a/gas_before b/gas_after
index 68d894d..7867bad 100644
--- a/gas_before
+++ b/gas_after
@@ -3,7 +3,7 @@
 ╠═══════════════════════════════════╬═══════════════════╬═══════
 ┆ Deployment Cost                   ┆ Deployment Size   ┆
 ├-----------------------------------┼-------------------┼-
-┆ 3076398                           ┆ 15658             ┆
+┆ 3075198                           ┆ 15652             ┆
 ├-----------------------------------┼-------------------┼-
 ┆ Function Name                     ┆ min               ┆ avg
 ├-----------------------------------┼-------------------┼-
@@ -29,9 +29,9 @@
 ├-----------------------------------┼-------------------┼-
 ┆ otherWitches                      ┆ 570               ┆ 570
 ├-----------------------------------┼-------------------┼-
```

```
-| payBase                                                    ¦ 7632          ¦ 2043
+| payBase                                                    ¦ 7632          ¦ 2037
 |-------------------------------------------------------------+---------------+-
-| payFYToken                                                 ¦ 7507          ¦ 1921
+| payFYToken                                                 ¦ 7504          ¦ 1916
 |-------------------------------------------------------------+---------------+-
 | point                                                      ¦ 2934          ¦ 5098
 |-------------------------------------------------------------+---------------+-
```

[G-04] Optimize names to save gas

`public` / `external` function names and `public` member variable names can be optimized to save gas. See this link for an example of how it works. Below are the interfaces/abstract contracts that can be optimized so that the most frequently-called functions use the least amount of gas possible during method lookup. Method IDs that have two leading zero bytes can save **128 gas** each during deployment, and renaming functions to have lower method IDs will save **22 gas** per call, per sorted position shifted.

There is 1 instance of this issue:

```
File: contracts/Witch.sol

/// @audit point(), setLine(), setLimit(), setAnotherWitch(), se
19:   contract Witch is AccessControl {
```

Witch.sol#L19

[G-05] Using `bool`s for storage incurs overhead

```
// Booleans are more expensive than uint256 or any type that
// word because each write operation emits an extra SLOAD to
// slot's contents, replace the bits taken up by the boolean
// back. This is the compiler's defense against contract upg
// pointer aliasing, and it cannot be disabled.
```

OpenZeppelin/ReentrancyGuard.sol#L23-L27

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas) for the extra SLOAD, and to avoid Gsset (**20000 gas**) when changing from `false` to `true`, after having been `true` in the past

There are 2 instances of this issue:

```
File: contracts/Witch.sol

68:       mapping(address => bool) public otherWitches;

69:       mapping(bytes6 => mapping(bytes6 => bool)) public ignc
```

Witch.sol#L68

[G-06] `>=` costs less gas than `>`

The compiler uses opcodes `GT` and `ISZERO` for solidity code that uses `>`, but only requires `LT` for `>=`, which saves 3 gas

There are 3 instances of this issue:

```
File: contracts/Witch.sol

308:           artIn = artIn > auction_.art ? auction_.art : artI

361:           artIn = maxArtIn > auction_.art ? auction_.art : m

556:           artIn = maxArtIn > auction_.art ? auction_.art : m
```

Witch.sol#L308

[G-07] 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

Each operation involving a `uint8` costs an extra 28 gas as compared to ones involving `uint256`, due to the compiler having to clear the higher bits of the memory word before operating on the `uint8`, as well as the associated stack operations of doing so. Use a larger size then downcast where needed

There are 2 instances of this issue:

```
File: contracts/Witch.sol

/// @audit uint128 art
233:          if (art < debt.min * (10**debt.dec)) art = balance

/// @audit uint128 artIn
308:          artIn = artIn > auction_.art ? auction_.art : artI
```

Witch.sol#L233

```
diff --git a/contracts/Witch.sol b/contracts/Witch.sol
index f98dd6a..066e912 100644
--- a/contracts/Witch.sol
+++ b/contracts/Witch.sol
@@ -229,11 +229,11 @@ contract Witch is AccessControl {
     ) internal view returns (DataTypes.Auction memory) {
         // We store the proportion of the vault to auction, whi
         DataTypes.Line storage line = lines[vault.ilkId][series
-        uint128 art = uint256(balances.art).wmul(line.proportic
+        uint256 art = uint256(balances.art).wmul(line.proportic
         if (art < debt.min * (10**debt.dec)) art = balances.art
-        uint128 ink = (art == balances.art)
+        uint256 ink = (art == balances.art)
             ? balances.ink
-            : uint256(balances.ink).wmul(line.proportion).u128(
+            : uint256(balances.ink).wmul(line.proportion);

         return
             DataTypes.Auction({
@@ -242,8 +242,8 @@ contract Witch is AccessControl {
                 seriesId: vault.seriesId,
```

```diff
                     baseId: series.baseId,
                     ilkId: vault.ilkId,
-                    art: art,
-                    ink: ink,
+                    art: art.u128(),
+                    ink: ink.u128(),
                     auctioneer: to
                 });
         }
```

```diff
diff --git a/gas_before b/gas_after
index 68d894d..4f7212a 100644
--- a/gas_before
+++ b/gas_after
@@ -3,17 +3,17 @@
 ╞═════════════════════════════════════╪══════════════════╪═══════
 | Deployment Cost                     ┆ Deployment Size  ┆
 ├╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌
-| 3076398                             ┆ 15658            ┆
+| 3077398                             ┆ 15663            ┆
 ├╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌
 | Function Name                       ┆ min              ┆ avg
 ├╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌
-| auction                             ┆ 4219             ┆ 7011
+| auction                             ┆ 4219             ┆ 7008
 ├╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌
 | auctioneerReward                    ┆ 426              ┆ 426
 ├╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌
 | auctions                            ┆ 1244             ┆ 1244
 ├╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌
-| calcPayout                          ┆ 3627             ┆ 1042
+| calcPayout                          ┆ 3627             ┆ 1041
 ├╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌
 | cancel                              ┆ 2736             ┆ 9683
 ├╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌╌┼╌
```

[G-08] `require()` or `revert()` statements that check input arguments should be at the top of the function

Checks that involve constants should come before checks that involve state variables, function calls, and calculations. By doing these checks first, the function is

able to revert before wasting a Gcoldsload (**2100 gas***) in a function that may ultimately revert in the unhappy case.

There is 1 instance of this issue:

```
File: contracts/Witch.sol

/// @audit expensive op on line 105
108:          require(proportion >= 0.01e18, "Proportion below 1
```

[Witch.sol#L108]

[G-09] Use custom errors rather than `revert()` / `require()` strings to save gas

Custom errors are available from solidity version 0.8.4. Custom errors save [~50 gas] each time they're hit by [avoiding having to allocate and store the revert string]. Not defining the strings also save deployment gas

There are 17 instances of this issue:

```
File: contracts/Witch.sol

84:           require(param == "ladle", "Unrecognized");

102:          require(initialOffer <= 1e18, "InitialOffer above

103:          require(proportion <= 1e18, "Proportion above 100%

104           require(
105               initialOffer == 0 || initialOffer >= 0.01e18,
106               "InitialOffer below 1%"
107:          );

108:          require(proportion >= 0.01e18, "Proportion below 1

189:          require(cauldron.level(vaultId) < 0, "Not undercol

200:          require(limits_.sum <= limits_.max, "Collateral li
```

```
255:         require(auction_.start > 0, "Vault not under aucti

256:         require(cauldron.level(vaultId) >= 0, "Undercollat

300:         require(auction_.start > 0, "Vault not under aucti

313:         require(liquidatorCut >= minInkOut, "Not enough bo

328:             require(baseJoin != IJoin(address(0)), "Join r

358:         require(auction_.start > 0, "Vault not under aucti

365:         require(liquidatorCut >= minInkOut, "Not enough bo

395:             require(ilkJoin != IJoin(address(0)), "Join no

416:         require(auction_.start > 0, "Vault not under aucti
```

```
437                 require(
438                     auction_.art - artIn >= debt.min * (10
439                     "Leaves dust"
440:                 );
```

[Witch.sol#L84]

[G-10] 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 6 instances of this issue:

```
File: contracts/Witch.sol

83:     function point(bytes32 param, address value) external
```

```
95          function setLine(
96              bytes6 ilkId,
97              bytes6 baseId,
98              uint32 duration,
99              uint64 proportion,
100             uint64 initialOffer
101:        ) external auth {

126         function setLimit(
127             bytes6 ilkId,
128             bytes6 baseId,
129             uint128 max
130:        ) external auth {

141:        function setAnotherWitch(address value, bool isWitch)

150         function setIgnoredPair(
151             bytes6 ilkId,
152             bytes6 baseId,
153             bool ignore
154:        ) external auth {

161:        function setAuctioneerReward(uint128 auctioneerReward_
```

[Witch.sol#L83]

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