Sherlock
Findings & Analysis Report

2021-09-16

Table of contents

- Overview
  - About C4
  - Wardens
- Summary
- Scope
- Severity Criteria
- High Risk Findings (2)
  - [H-01] Single under-funded protocol can break paying off debt
  - [H-02] [Bug] A critical bug in `bps` function
- Medium Risk Findings (4)
  - [M-01] Incorrect internal balance bookkeeping
  - [M-02] `_doSherX` optimistically assumes premiums will be paid
  - [M-03] reputation risks with `updateSolution`
  - [M-04] Yield distribution after large payout seems unfair
- Low Risk Findings (31)

- [L-01] Gov.sol: Use `SafeERC20.safeApprove` in `tokenUnload()`
- [L-02] `withdraw` returns the final amount withdrawn
- [L-03] series of divs
- [L-04] ERC20 non-standard names
- [L-05] User's `calcUnderlyingInStoredUSD` value is underestimated
- [L-06] `PoolStrategy.sol` : Consider minimizing trust with implemented strategies
- [L-07] Unbounded iteration over all premium tokens
- [L-08] Unbounded iteration over all staking tokens
- [L-09] Unbounded iteration over all protocols
- [L-10] Missing verification on `tokenInit` 's lock
- [L-11] `_doSherX` does not return correct precision and it's confusing
- [L-12] Anyone can unstake on behalf of someone
- [L-13] Sanitize `_weights` in `setWeights` on every use
- [L-14] `initializeSherXERC20` can be called more than once
- [L-15] ERC20 can accidentally burn tokens
- [L-16] extra check `setUnstakeWindow` and `setCooldown`
- [L-17] delete `ps.stakeBalance`
- [L-18] prevent div by 0
- [L-19] unbounded loop in `getInitialUnstakeEntry`
- [L-20] prevent burn in `_transfer`
- [L-21] AaveV2 approves lending pool in the constructor
- [L-22] Inclusive checks
- [L-23] Group related data into separate structs
- [L-24] Re-entrancy mitigation
- [L-25] `getInitialUnstakeEntry` when `unstakeEntries` is empty
- [L-26] Loops may exceed gas limit
- [L-27] `SafeMath` library is not always used in `PoolBase`

    - [L-28] Missing non-zero address checks
    - [L-29] Possible divide-by-zero error in `PoolBase`
    - [L-30] Inconsistent block number comparison when deciding an unstaking entry is active
    - [L-31] Tokens cannot be reinitialized with new lock tokens
- Non-Critical Findings
- Gas Optimizations
- Disclosures

Overview

About C4

Code 432n4 (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 code 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 code contest outlined in this document, C4 conducted an analysis of the Sherlock smart contract system written in Solidity. The code contest took place between July 21—July 28.

Wardens

14 Wardens contributed reports to the Sherlock code contest:

1. cmichel
2. gpersoon
3. shw
4. pauliax
5. walker
6. hrkrshnn
7. jonah1005

8. hickuphh3
9. eriksal1217
10. patitonar
11. 0xsanson
12. tensors
13. a_delamo
14. bw

This contest was judged by ghoul.sol.

Final report assembled by moneylegobatman and ninek.

Summary

The C4 analysis yielded an aggregated total of 37 unique vulnerabilities. All of the issues presented here are linked back to their original finding

Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity, 4 received a risk rating in the category of MEDIUM severity, and 31 received a risk rating in the category of LOW severity.

C4 analysis also identified 19 non-critical recommendations and 36 gas optimizations.

Scope

The code under review can be found within the C4 Sherlock code contest repository which is comprised of 50 smart contracts written in the Solidity programming language and includes 3,063 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.

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] Single under-funded protocol can break paying off debt

Submitted by cmichel, also found by walker and gpersoon

The `SherXERC20.payOffDebtAll` function iterates over all protocols of the token. If *a single project* does not have enough funds to cover the premium payments, the transactions come to a halt, see `_payOffDebt`:

```
debt = _accruedDebt(ps, _protocol, _blocks);
// this can revert tx
ps.protocolBalance[_protocol] = ps.protocolBalance[_protocol].su
```

Many core functions require paying off debt first and can therefore revert when a single protocol cannot pay the token premium:

- `setTokenPrice`
- `setProtocolPremium`
- `withdrawProtocolBalance`
- `redeem`

- etc.

This scenario that a protocol is unable to pay a premium does not seem unlikely especially as there can be many protocols and each protocol can pay premiums in potentially many tokens and have to continuously re-deposit to their account to increase the balance. It is also rather involved to remove the protocol's coverage and remove the premium payments for the token. It requires governance interaction and potentially paying for the accumulated debt themselves.

[Evert0x (Sherlock) acknowledged]:

> This was a design tradeoff. As governance we can see it coming as the balance is slowly draining. But the fact the protocols are able to withdraw the full amount at any time could surprise the governance. (and make the reverts in the functions above happening)

> We are thinking to add a rule in the `withdrawProtocolBalance` to only allow withdrawals with at least 2 days of remaining balance. Allowing enough time for governance calls to remove the protocol.

[H-02] [Bug] A critical bug in `bps` function

Submitted by hrkrshnn, also found by jonah1005 and walker

```
function bps() internal pure returns (IERC20 rt) {
  // These fields are not accessible from assembly
  bytes memory array = msg.data;
  uint256 index = msg.data.length;

  // solhint-disable-next-line no-inline-assembly
  assembly {
    // Load the 32 bytes word from memory with the address on th
    rt := and(mload(add(array, index)), 0xffffffffffffffffffffff
  }
}
```

The above function is designed to expect the token at the end of `calldata`, but a malicious user can inject extra values at the end of `calldata` and fake return values.

The following contract demonstrates an example:

```
pragma solidity 0.8.6;

interface IERC20 {}

error StaticCallFailed();

contract BadEncoding {
  /// Will return address(1). But address(0) is expected!
  function f() external view returns (address) {
    address actual = address(0);
    address injected = address(1);

    (bool success, bytes memory ret) = address(this).staticcall(

    if (!success) revert StaticCallFailed();

    return abi.decode(ret, (address));
  }
  function g(IERC20 _token) external pure returns (IERC20) {
    // to get rid of the unused warning
    _token;
    // Does it always match _token?
    return bps();
  }
  // From Sherlock Protocol: PoolBase.sol
  function bps() internal pure returns (IERC20 rt) {
    // These fields are not accessible from assembly
    bytes memory array = msg.data;
    uint256 index = msg.data.length;

    // solhint-disable-next-line no-inline-assembly
    assembly {
      // Load the 32 bytes word from memory with the address on
      rt := and(mload(add(array, index)), 0xffffffffffffffffffff
    }
  }
}
```

This example can be used to exploit the protocol:

```
function unstake(
```

```
    uint256 _id,
    address _receiver,
    IERC20 _token
  ) external override returns (uint256 amount) {
    PoolStorage.Base storage ps = baseData();
    require(_receiver != address(0), 'RECEIVER');
    GovStorage.Base storage gs = GovStorage.gs();
    PoolStorage.UnstakeEntry memory withdraw = ps.unstakeEntries[m
    require(withdraw.blockInitiated != 0, 'WITHDRAW_NOT_ACTIVE');
    // period is including
    require(withdraw.blockInitiated + gs.unstakeCooldown < uint40(
    require(
      withdraw.blockInitiated + gs.unstakeCooldown + gs.unstakeWir
      'UNSTAKE_WINDOW_EXPIRED'
    );
    amount = withdraw.lock.mul(LibPool.stakeBalance(ps)).div(ps.lo

    ps.stakeBalance = ps.stakeBalance.sub(amount);
    delete ps.unstakeEntries[msg.sender][_id];
    ps.lockToken.burn(address(this), withdraw.lock);
    _token.safeTransfer(_receiver, amount);
  }
```

State token `Token1`. Let's say there is a more expensive token `Token2`.

Here's an example exploit:

```
bytes memory exploitPayload = abi.encodeWithSignature(
  PoolBase.unstake.selector,
  (uint256(_id), address(_receiver), address(Token2), address(To
);
poolAddress.call(exploitPayload);
```

All the calculations on `ps` would be done on `Token2`, but at the end, because of, `_token.safeTransfer(_receiver, amount);`, `Token2` would be transferred. Assuming that `Token2` is more expensive than `Token1`, the attacker makes a profit.

Similarly, the same technique can be used at a lot of other places. Even if this exploit is not profitable, the fact that the computations can be done on two different tokens is buggy.

There are several other places where the same pattern is used. All of them needs to be fixed. I've not written an exhaustive list.

Evert0x (Sherlock) confirmed

Medium Risk Findings (4)

[M-01] Incorrect internal balance bookkeeping

Submitted by walker, also found by cmichel and shw

The sherlock smart contract system uses internal bookkeeping of arbitrary ERC20 token balances. It doesn't assert that the ERC20 doesn't implement some non-standard behavior. For example, deflationary tokens, or tokens with a transfer fee, will result in incorrect internal balances. In summary, an attacker can perform stake and deposit actions without actually depositing the amount that sherlock assumes. As a result, an attacker is unduly rewarded balance and yield.

> Balancer had a similar vulnerability in their system https://blog.1inch.io/balancer-hack-2020-a8f7131c980e.

An example location where such internal bookkeeping happens can be found here

Mitigating the issue is possible by requiring the amount to be added to the contracts' balance. Alternatively, it's possible to update the pool based on actual balance changes.

Evert0x (Sherlock) acknowledged and disagreed with severity:

> 2 med-risk, as extensive research will be done before adding certain tokens. This finding could even be noted a 0 non-critical if only 'standard' ERC20s are being used.

> med-risk because certain popular tokens are up-gradable and could potentially implement non-standard behavior

ghoul-sol (judge) commented:

> since there will be a curation process, I agree with sponsor, this is medium risk

[M-02] `_doSherX` optimistically assumes premiums will be paid

Submitted by cmichel

The `_doSherX` function does not attempt to pay off the accrued premiums ("pay off debt") for most tokens, only for the ones that would otherwise revert the tx:

```
// Expensive operation, only execute to prevent tx reverts
if (amounts[i] > ps.sherXUnderlying) {
  LibPool.payOffDebtAll(tokens[i]);
}
```

The `amounts = LibSherX.calcUnderlying(totalSherX)` array is an optimistic view assuming all outstanding, accrued premiums would indeed be paid until now. However, it could be that a protocol does not have enough balance to pay out these premiums and updating the state using `LibPool.payOffDebtAll(tokens[i]);` would fail for a token.

An inflated amount is then paid out to the user based on the optimistic `calcUnderlying` call.

Evert0x (Sherlock) acknowledged:

> Fair point, the protocol is optimistic the protocols can payoff their debt.

[M-03] reputation risks with `updateSolution`

Submitted by gpersoon

`GovDev.so` l has a function `updateSolution` to upgrade parts of the contract via the Diamond construction. Via `updateSolution` , any functionality can be changed and all the funds can be accessed/rugged. Even if this is well intended the project could still be called out resulting in a reputation risk, see for [example(https://twitter.com/RugDocIO/status/1411732108029181960).

Note: there is a function `transferGovDev` which can be used to disable the `updateSolution`

```
25
26 function updateSolution(IDiamondCut.FacetCut[] memory _diamon
27   require(msg.sender == LibDiamond.contractOwner(), 'NOT_DEV'
28   return LibDiamond.diamondCut(_diamondCut, _init, _calldata)
29 }
```

Recommend applying extra safeguards for example to limit the time period where `updateSolution` can be used.

EvertOx (Sherlock) acknowledged:

> Fair point, although we are not anonymous, we still want to mitigate this risk.

> I'm thinking something like this

- update is pushed, everyone can review the code changes
- 14 days of waiting, people are able to get their funds out
- update is executed.

> Downside is that it doesn't allow us to fix potential critical issues fast.

[M-04] Yield distribution after large payout seems unfair

Submitted by gpersoon

When a large payout occurs, it will lower `unallocatedSherX`. This could mean some parties might not be able to get their Yield.

The first couple of users (for which harvest is called or which transfer tokens) will be able to get their full Yield, until the moment `unallocatedSherX` is depleted. The next users don't get any yield at all. This doesn't seem fair.

```
309
310 function doYield(ILock token,address from, address to, uint2
```

```
311 ...
312 ps.unallocatedSherX = ps.unallocatedSherX.sub(withdrawable_a
313
108
109  function payout( address _payout, IERC20[] memory _tokens,
110      // all pools (including SherX pool) can be deducted fmo
111      // deducting balance will reduce the users underlying va
112      // for every pool, _unallocatedSherX can be deducted, th
113      // for users that did not claim them (e.g materialized t
114 ....
115      // Subtract from unallocated, as the tokens are now allo
116          ps.unallocatedSherX = ps.unallocatedSherX.sub(unallo
```

Recommend that If `unallocatedSherX` is insufficient to provide for all the yields, only give the yields partly (so that each user gets their fair share).

EvertOx (Sherlock) disputed:

> Not only `unallocatedSherX` is subtracted but also `sWeight`, which is used to calculate the reward. I wrote some extra tests and in my experience the remaining SherX (in the unallocatedSherX variable) is splitted in a fair way.

EvertOx (Sherlock) confirmed:

> Together with gpersoon I discussed both issue #49 and #50 and based on both findings we found a med-risk issue. In case `payout()` is called with `_unallocatedSherX > 0` and a user called `harvest()` before the payout call. It blocks the user from calling `harvest()` again. + blocks the lock token transfer.

> Mitigations step is to stop calling `payout()` with `_unallocatedSherX > 0`

Low Risk Findings (31)

[L-01] Gov.sol: Use `SafeERC20.safeApprove` in `tokenUnload()`

Submitted by hickuphh3, also found by eriksal1217 and shw

This is probably an oversight since `SafeERC20` was imported and `safeTransfer()` was used for ERC20 token transfers. Nevertheless, note that `approve()` will fail for certain token implementations that do not return a boolean value (Eg. OMG and ADX tokens). Hence it is recommend to use `safeApprove()`.

Recommend updating to `_token.safeApprove(address(_native), totalToken)` in `tokenUnload()`.

Evert0x (Sherlock) confirmed

[L-02] `withdraw` returns the final amount withdrawn

Submitted by pauliax, also found by eriksal1217

function `withdraw` in ILendingPool returns the actual withdrawn amount, however, function `withdraw` in AaveV2 strategy does not check this return value so e.g. function strategyWithdraw may actually withdraw less but still add the full amount to the staked balance:

```
ps.strategy.withdraw(_amount);
ps.stakeBalance = ps.stakeBalance.add(_amount);
```

Recommend that function `withdraw` in `IStrategy` should return uint indicating the actual withdrawn amount and functions that use it should account for that.

Evert0x (Sherlock) confirmed:

> When looking at the LendingPool withdraw implementation: https://github.com/aave/protocol-v2/blob/master/contracts/protocol/lendingpool/LendingPool.sol#L142

> It will revert if the _amount > balance. It basically only returns a different value then _amount in case it is uint256(-1), correct?

[L-03] series of divs

Submitted by gpersoon, also found by hickuphh3 and shw

The function `payout` contains an expression with 3 sequential divs. This is generally not recommended because it could lead to rounding errors / loss of precision. Also, a div is usually more expensive than a mul. Also, an intermediate division by 0 (if `SherXERC20Storage.sx20().totalSupply` == 0) could occur.

```
108
109 function payout(
110 ..
111 uint256 deduction =  excludeUsd.div(curTotalUsdPool.div(Sher
```

Recommend verifying the formula and replace with something like:

```
uint256 deduction =  excludeUsd.mul(SherXERC20Storage.sx20().tot
```

EvertOx (Sherlock) confirmed

[L-04] ERC20 non-standard names

Submitted by cmichel, also found by shw

Usually, the functions to increase the allowance are called `increaseAllowance` and `decreaseAllowance` but in `SherXERC20` they are called `increaseApproval` and `decreaseApproval`

Recommend renaming these functions to the more common names.

EvertOx (Sherlock) confirmed sponsor confirmed- EvertOx (Sherlock) labeled disagree with severity

EvertOx (Sherlock) confirmed:

> Good point
> https://docs.openzeppelin.com/contracts/2.x/api/token/erc20#ERC20

EvertOx (Sherlock) disagreed with severity:

> 0 non-critical

ghoul-sol (judge) commented:

> I agree with warden, low risk looks reasonable here.

[L-05] User's `calcUnderlyingInStoredUSD` value is underestimated

Submitted by shw

The `calcUnderlyingInStoredUSD()` function of `SherX` should return `calcUnderlyingInStoredUSD(getSherXBalance())` instead of `calcUnderlyingInStoredUSD(sx20.balances[msg.sender])` since there could be `SherX` unallocated to the user at the time of the function call. A similar function, `calcUnderlying()`, calculates the user's underlying tokens based on the user's current balance plus the unallocated ones.

Recommend changing `sx20.balances[msg.sender]` to `getSherXBalance()` at L141 in `SherX.sol`.

Evert0x (Sherlock) confirmed:

> 1 (low risk); as the function is called '..inStored..', at it is using the stored variables. I agree it is a confusing function name.

ghoul-sol (judge) disagreed with severity:

> agree with sponsor, low risk

[L-06] `PoolStrategy.sol` : Consider minimizing trust with implemented strategies

Submitted by hickuphh3, also found by shw

`PoolStrategy` trusts the implemented strategy `ps.strategy` (Eg. `AaveV2.sol`) to:

- return the right amount for `ps.strategy.balanceOf()`
- have sent back the withdrawn funds when `ps.strategy.withdraw()` is called
- report the correct withdrawn amount when `ps.strategy.withdrawAll()` is called

While `ps.strategy` is assumed to have been scrutinized and its code verified before adding it as a strategy, and can therefore be trusted, consider minimizing trust between `PoolStrategy` and `ps.strategy`, since strategies are themselves reliant on other protocols and therefore subject to external risk.

- Verify the amount sent back to `PoolStrategy` for withdrawals instead
- The reliance on `balanceOf()` can be mitigated slightly by using a counter `uint256 depositedAmount` that increments / decrements upon deposits and withdrawals to the strategy respectively. This value can then be used in lieu of `ps.strategy.balanceOf()`. However, the downsides to this are that
  - this counter does not account for yield amounts from the strategy and
  - it increases complexity

A simple implementation to checking the withdrawal amounts is provided below.

```
function strategyWithdraw(uint256 _amount, IERC20 _token) exterr
  ...
  uint256 balanceBefore = _token.balanceOf(address(this));
  ps.strategy.withdraw(_amount);
  require(balanceBefore.add(_amount) == _token.balanceOf(address
  ps.stakeBalance = ps.stakeBalance.add(_amount);
}

function strategyWithdrawAll(IERC20 _token) external override {
  PoolStorage.Base storage ps = baseData();
  _enforceGovPool(ps);
  _enforceStrategy(ps);

  uint256 balanceBefore = _token.balanceOf(address(this));
  ps.strategy.withdrawAll();
  // alternatively, verify amount returned by withdrawAll() meth
  uint256 amount = _token.balanceOf(address(this)).sub(balanceBe
  ps.stakeBalance = ps.stakeBalance.add(amount);
```

```
    }
```

EvertOx (Sherlock) acknowledged

ghoul-sol (judge) commented:

> This looks like low risk issue.

[L-07] Unbounded iteration over all premium tokens

Submitted by cmichel

The `Gov.protocolRemove` function iterates over all elements of the `tokensSherX` array.

The transactions could fail if the arrays get too big and the transaction would consume more gas than the block limit. This will then result in a denial of service for the desired functionality and break core functionality.

The severity is low as only governance can whitelist these tokens but not the protocols themselves.

Recommendation is to keep the array size small.

EvertOx (Sherlock) acknowledged

[L-08] Unbounded iteration over all staking tokens

Submitted by cmichel

The `SherX.getTotalSherXUnminted` function iterates over all elements of the `tokensStaker` array.

The transactions could fail if the arrays get too big and the transaction would consume more gas than the block limit. This will then result in a denial of service for the desired functionality and break core functionality.

The severity is low as only governance can whitelist these tokens but not the protocols themselves.

Recommend keeping the array size small.

EvertOx (Sherlock) acknowledged

[L-09] Unbounded iteration over all protocols

Submitted by cmichel

The `LibPool.payOffDebtAll` function iterates over all elements of the `ps.protocols` array.

The transactions could fail if the arrays get too big and the transaction would consume more gas than the block limit. This will then result in a denial of service for the desired functionality and break core functionality.

The severity is low as only governance can whitelist protocols per token but not the protocols themselves.

Recommendation is to keep the array size small.

EvertOx (Sherlock) acknowledged

[L-10] Missing verification on `tokenInit`'s lock

Submitted by cmichel

The `Gov.tokenInit` skips the underlying token check if the `_token` is SHERX:

```
if (address(_token) != address(this)) {
  require(_lock.underlying() == _token, 'UNDERLYING');
}
```

This check should still be performed even for `_token == address(this) // SHERX`, otherwise, the lock can have a different underlying and potentially pay out

wrong tokens.

Recommendation is to verify the underlying of all locks.

EvertOx (Sherlock) confirmed

EvertOx (Sherlock) commented:

> Good catch!

[L-11] `_doSherX` does not return correct precision and it's confusing

Submitted by cmichel

The `_doSherX` function does not return the correct precision of `sherUsd` and it is **not** the "Total amount of USD of the underlying tokens that are being transferred" that the documentation mentions.

```
sherUsd = amounts[i].mul(sx.tokenUSD[tokens[i]]);
```

Instead, the amount is inflated by `1e18`, it should divide the amount by `1e18` to get a USD value with 18 decimal precision.

The severity is low as the calling site in `payout` makes up for it by dividing by `1e18` in the `deduction` computation.

We still recommend returning the correct amount in `_doSherX` already to match the documentation and avoid any future errors when using its unintuitive return value.

EvertOx (Sherlock) confirmed

[L-12] Anyone can unstake on behalf of someone

Submitted by cmichel

The `PoolBase.unstakeWindowExpiry` function allows unstaking tokens of other users. While the tokens are sent to the correct address, this can lead to issues with smart contracts that might rely on claiming the tokens themselves.

For example, suppose the `_to` address corresponds to a smart contract that has a function of the following form:

```
function withdrawAndDoSomething() {
    uint256 amount = token.balanceOf(address(this));
    contract.unstakeWindowExpiry(address(this), id, token);
    amount = amount - token.balanceOf(address(this));
    token.transfer(externalWallet, amount)
}
```

Recommend considering that, If the contract has no other functions to transfer out funds, they may be locked forever in this contract.

EvertOx (Sherlock) confirmed

[L-13] Sanitize `_weights` in `setWeights` on every use

Submitted by cmichel

The `setWeights` function only stores the `uint16` part of `_weights[i]` in storage (`ps.sherXWeight = uint16(_weights[i])`). However, to calculate `weightAdd/weightSub` the full value (not truncated to 16 bits) is used. This can lead to discrepancies as the actually added part is different from the one tracked in the `weightAdd` variable.

EvertOx (Sherlock) confirmed:

> Your recommendation is to do `.add(uint16(_weights[i]))` for both weightAdd and weightSub?

[L-14] `initializeSherXERC20` can be called more than once

Submitted by cmichel, also found by pauliax

The `SherXERC20.initializeSherXERC20` function has `initialize` in its name which indicates that it should only be called once to initialize the storage. But it can be repeatedly called to overwrite and update the ERC20 name and symbol.

Recommend considering an `initializer` modifier or reverting if `name` or `symbol` is already set.

EvertOx (Sherlock) confirmed

[L-15] ERC20 can accidentally burn tokens

Submitted by cmichel, also found by shw

The `SherXERC20.transfer` / `transferFrom` actions allow transferring tokens to the zero address. This is usually prohibited to accidentally avoid "burning" tokens by sending them to an unrecoverable zero address.

EvertOx (Sherlock) acknowledged:

> Does it make more sense to include an extra `burn()` function? As removing the possibility to send to zero address removes the ability to burn.

[L-16] extra check `setUnstakeWindow` and `setCooldown`

Submitted by gpersoon

The function `setUnstakeWindow` and `setCooldown` don't check that the input parameter isn't 0. So the values could accidentally be set to 0 (although unlikely). However you wouldn't want the to be 0 because that would allow attacks with flashloans (stake and unstake in the same transaction)

```
124 https:
125  function setUnstakeWindow(uint40 _unstakeWindow) external o
126    require(_unstakeWindow < 25000000, 'MAX'); // ~ approxim
127    GovStorage.gs().unstakeWindow = _unstakeWindow;
128   }
129
130   function setCooldown(uint40 _period) external override onl
131    require(_period < 25000000, 'MAX'); // ~ approximate 10
```

```
132       GovStorage.gs().unstakeCooldown = _period;
133     }
```

Recommend checking the input parameter of `setUnstakeWindow` and `setCooldown` isn't 0

EvertOx (Sherlock) confirmed

[L-17] delete `ps.stakeBalance`

Submitted by gpersoon

In the function `tokenUnload`, `ps.stakeBalance` is only deleted if balance >0. e.g it is deleted if `ps.stakeBalance` > ps.firstMoneyOut So if `ps.stakeBalance` == ps.firstMoneyOut then `ps.stakeBalance` will not be deleted. And then a call to tokenRemove will revert, because it checks for `ps.stakeBalance` to be 0

```
271
272   function tokenUnload( IERC20 _token, IRemove _native, addre
273 ...
274       uint256 balance = ps.stakeBalance.sub(ps.firstMoneyOut);
275       if (balance > 0) {
276         _token.safeTransfer(_remaining, balance);
277         delete ps.stakeBalance;
278       }
279 ..
280     delete ps.firstMoneyOut;
281
282   function tokenRemove(IERC20 _token) external override onlyG
283     ...
284       require(ps.stakeBalance == 0, 'BALANCE_SET');
285   }
```

Recommend checking what to do in this edge case and add the appropriate code.

EvertOx (Sherlock) confirmed

[L-18] prevent div by 0

Submitted by gpersoon

On several locations in the code precautions are taken not to divide by 0, because this will revert the code. However on some locations this isn't done.

Especially in `doYield` a first check is done for totalAmount >0, however a few lines later there is an other div(totalAmount) which isn't checked.

The proof of concept show another few examples.

```
309
310 function doYield(ILock token,address from,address to,uint256
311 ..
312     uint256 totalAmount = ps.lockToken.totalSupply();
313 ..
314     if (totalAmount > 0) {
315       ineglible_yield_amount = ps.sWeight.mul(amount).div(to
316     } else {
317       ineglible_yield_amount = amount;
318     }
319     if (from != address(0)) {
320       uint256 raw_amount = ps.sWeight.mul(userAmount).div(to
321
295
296 function activateCooldown(uint256 _amount, IERC20 _token) ex
297 ...   uint256 tokenAmount = fee.mul(LibPool.stakeBalance(ps)
298
351
352  function unstake( uint256 _id, address _receiver, IERC20 _t
353 ...    amount = withdraw.lock.mul(LibPool.stakeBalance(ps)).
354
 67
 68  function stake( PoolStorage.Base storage ps,uint256 _amount
 69 ...      lock = _amount.mul(totalLock).div(stakeBalance(ps))
```

Recommend making sure division by 0 won't occur by checking the variables beforehand and handling this edge case.

EvertOx (Sherlock) confirmed:

> if (from != address(0)) { uint256 raw_amount = ps.sWeight.mul(userAmount).div(totalAmount); // totalAmount could be 0, see lines above

> If totalAmount == 0, from is always address(0). As no one holds this lockToken and it's being minted

EvertOx (Sherlock) commented:

> function activateCooldown(uint256 _amount, IERC20 _token) external override returns (uint256) { ... uint256 tokenAmount = fee.mul(LibPool.stakeBalance(ps)).div(ps.lockToken.totalSupply()); // ps.lockToken.totalSupply() might be 0

> Can not be 0, as there is lockToken being transferred `require(_amount > 0, 'AMOUNT');` , so the value is at least 1.

EvertOx (Sherlock) commented:

> function unstake(uint256 _id, address _receiver, IERC20 _token) external override returns (uint256 amount) { ... amount = withdraw.lock.mul(LibPool.stakeBalance(ps)).div(ps.lockToken.totalSupply()); // // ps.lockToken.totalSupply() might be 0

> Can not be 0, as there is lockToken in the contract, waiting to be transferred back to the user

EvertOx (Sherlock) commented:

> function stake(PoolStorage.Base storage ps,uint256 _amount, address _receiver) external returns (uint256 lock) { ... lock = _amount.mul(totalLock).div(stakeBalance(ps)); // stakeBalance(ps) might be 0

> Can not be 0 (most of the times), as there are already lockTokens in circulation, which means someone has deposited BUT the balance could be fully depleted because of a `payout()` call which could make it 0.

> Thanks!

[L-19] unbounded loop in `getInitialUnstakeEntry`

Submitted by gpersoon

The functions `getInitialUnstakeEntry` contains a for loop that can be unbounded. This would mean it could run out of gas and the function would revert. The array `unstakeEntries` can be made arbitrarily large by repeatedly calling activateCooldown with a small amount of tokens.

The impact is very low because the array `unstakeEntries` is separated per user and links to `mgs.sender` , so you can only shoot yourself in your foot.

Additionally the function `getInitialUnstakeEntry` isn't used in the smart contracts.

```
123
124  function getInitialUnstakeEntry(address _staker, IERC20 _to
125 ...
126 for (uint256 i = 0; i < ps.unstakeEntries[_staker].length; i
127       if (ps.unstakeEntries[_staker][i].blockInitiated == 0)
128
129 function activateCooldown(uint256 _amount, IERC20 _token) ex
130     require(_amount > 0, 'AMOUNT');
131 ...
132     ps.unstakeEntries[msg.sender].push(PoolStorage.UnstakeEn
```

Recommend probably accepting the situation and add a comment in the function `getInitialUnstakeEntry`

EvertOx (Sherlock) acknowledged

[L-20] prevent burn in `_transfer`

Submitted by gpersoon

The function `_transfer` in SherXERC20.sol allow transfer to address 0. This is usually considered the same as burning the tokens and the `Emit` is indistinguishable from an `Emit` of a burn.

However the burn function in LibSherXERC20.sol has extra functionality, which `_transfer` doesn't have. `sx20.totalSupply = sx20.totalSupply.sub(_amount);`

So it is safer to prevent `_transfer` to address 0 (which is also done in the openzeppelin erc20 contract) See: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol#L226

Note: minting from address 0 will not work because that is blocked by the `safemath` sub in: `sx20.balances[_from] = sx20.balances[_from].sub(_amount);`

```
118 https:
119 function _transfer(address _from, address _to, uint256 _amou
120   SherXERC20Storage.Base storage sx20 = SherXERC20Storage.sx
121   sx20.balances[_from] = sx20.balances[_from].sub(_amount);
122   sx20.balances[_to] = sx20.balances[_to].add(_amount);
123   emit Transfer(_from, _to, _amount);
124 }
125
 29
 30 function burn(address _from, uint256 _amount) internal {
 31   SherXERC20Storage.Base storage sx20 = SherXERC20Storage.sx
 32   sx20.balances[_from] = sx20.balances[_from].sub(_amount);
 33   sx20.totalSupply = sx20.totalSupply.sub(_amount);
 34   emit Transfer(_from, address(0), _amount);
 35 }
```

Recommend adding something like to following to `_transfer` of SherXERC20.sol:

```
require(_to!= address(0), "Transfer to the zero address'
```

Or, updating `sx20.totalSupply` if burning a desired operation.

Evert0x (Sherlock) confirmed

[L-21] AaveV2 approves lending pool in the constructor

Submitted by pauliax

Contract AaveV2 does not cache the lending pool, it retrieves it when necessary by calling a function `getLp()`. This is great as the implementation may change, however, this contract also approves an unlimited amount of want in the constructor:

```
    ILendingPool lp = getLp();
    want.approve(address(lp), uint256(-1));
so if the implementation changes, the approval will reset. This
```

For reference, function `setLendingPoolImpl`.

Not sure how likely is that lending pool implementation will change so marking this as 'Low'.

Recommend that before calling `lp.deposit` check that the approval is sufficient and increase otherwise.

EvertOx (Sherlock) confirmed

[L-22] Inclusive checks

Submitted by pauliax

I think these checks should be inclusive:

```
require(_unstakeWindow < 25000000, 'MAX');
require(_period < 25000000, 'MAX');
if (_amount > oldValue) // >= will reduce gas here
```

```
require(_unstakeWindow <= 25000000, 'MAX');
require(_period <= 25000000, 'MAX');
if (_amount >= oldValue)
```

[L-23] Group related data into separate structs

Submitted by pauliax

In Base struct having 3 separate fields that map from _protocol is error-prone. If you later introduce new fields, etc, you need not forget to delete them in function protocolRemove, etc. I think it would be better to have a separate struct for protocol-related data and map to that.

An example solution, replace:

```
    mapping(bytes32 => address) protocolManagers;
    mapping(bytes32 => address) protocolAgents;
    mapping(bytes32 => bool) protocolIsCovered;
```

with:

```
  struct ProtocolInfo {
    address manager;
    address agent;
    bool covered;
  }
  struct Base {
    ...
    mapping(bytes32 => ProtocolInfo) protocolInfo;
    ...
  }
```

Then you can delete all fields this way: delete `gs.protocolInfo[_protocol]`; Similar solution may be applied to `PoolStorage` (`protocolBalance`, `protocolPremium`, `isProtocol`).

EvertOx (Sherlock) acknowledged

[L-24] Re-entrancy mitigation

Submitted by pauliax

I see no re-entrancy mitigations. Contracts interact with various outside sources (tokens, aave pools, other possible strategies that may be added in the future, etc). so, for instance, now you have to be careful and do not allow tokens that have a receiver callback (e.g. erc777) or untrustable sources of yield (strategies).

Consider using ReentrancyGuard on main action functions.

EvertOx (Sherlock) acknowledged

[L-25] `getInitialUnstakeEntry` when `unstakeEntries` is empty

Submitted by pauliax

When the address has no unstake entries, function `getInitialUnstakeEntry` still returns 0 index. This function is external but can still confuse the outside consumers.

Recommend considering requiring `ps.unstakeEntries[_staker].length > 0;`

EvertOx (Sherlock) acknowledged

[L-26] Loops may exceed gas limit

Submitted by pauliax

Probably you are aware of this, but as I see many for loops throughout the code iterating over dynamic arrays I suggest being very careful as the execution may exceed the block gas limit, consume all the gas provided, and fail. Some arrays have removal functions, but there is, for instance, `unstakeEntries` array that is never actually removed as 'delete ps.unstakeEntries[msg.sender][_id];' only resets the values to default.

You can consider introducing max limits on items in the arrays or make sure that elements can be removed from dynamic arrays in case it becomes too large.

EvertOx (Sherlock) acknowledged

[L-27] `SafeMath` library is not always used in `PoolBase`

Submitted by shw

`SafeMath` library functions are not always used in arithmetic operations in the `PoolBase` contract, which could potentially cause integer underflow/overflows. Although in the reference lines of code, there are upper limits on the variables to ensure an integer underflow/overflow could not happen, using `SafeMath` is always a best practice, which prevents underflow/overflows completely (even if there were no assumptions on the variables) and increases code consistency as well.

Recommend considering using the `SafeMath` library functions in the referenced lines of code.

Evert0x (Sherlock) acknowledged

[L-28] Missing non-zero address checks

Submitted by shw

Adding non-zero address checks on the following function's parameters can help ensure the ownership of contracts is not lost or the contracts do not need to be redeployed if any of them is provided as zero accidentally.

Recommend considering adding non-zero address checks on the parameters.

Evert0x (Sherlock) acknowledged:

> `GovDev.sol#L19-L23` is used to eventually renounce the role, but maybe it makes sense to create a different function for that.

[L-29] Possible divide-by-zero error in `PoolBase`

Submitted by shw

A possible divide-by-zero error could happen in the `getSherXPerBlock(uint256, IERC20)` function of `PoolBase` when the `totalSupply` of `lockToken` and `_lock` are both 0. Recommend checking if

`baseData().lockToken.totalSupply().add(_lock)` equals to 0 before line 214. If so, then return 0.

[Evert0x (Sherlock) confirmed]

[L-30] Inconsistent block number comparison when deciding an unstaking entry is active

Submitted by shw

The `getInitialUnstakeEntry` function of `PoolBase` returns the first active unstaking entry of a staker, which requires the current block to be strictly before the last block in the unstaking window. However, the `unstake` function allows the current block to be exactly the same as the last block (same logic in `unstakeWindowExpiry`).

Recommend changing the `<=` comparison at line 136 to `<` for consistency.

[Evert0x (Sherlock) confirmed]

[L-31] Tokens cannot be reinitialized with new lock tokens

Submitted by shw

A token cannot be reinitialized with a new lock token once it is set to a non-zero address. If the lock token needs to be changed (for example, because of implementation errors), the token must be removed and added again.

Consider removing the `if` condition at line 219 to allow the lock token to be reinitialized.

[Evert0x (Sherlock) acknowledged]:

> Upgrading the lockToken a pretty complex procedure. As old lockTokens suddenly become worthless.

Non-Critical Findings

- [N-01] [PoolBase.sol] Calculations are being divided before being multiplied
- [N-02] [ForeignLock.sol] Local Variables Shadowing other variables
- [N-03] transferFrom when from = to
- [N-04] Different solidity pramas
- [N-05] Poorly Named variables
- [N-06] NatSpec typo in `_doSherX` @return
- [N-07] Confusing exponentiation (10e17)
- [N-08] typo: `ineglible_yield_amount` -> `ineligible_yield_amount`
- [N-09] Define Global Constants
- [N-10] confusing comment in protocolUpdate
- [N-11] don't use add(add.sub(sub)
- [N-12] Gov.sol: Non-intuitive comment in tokenRemove()
- [N-13] PoolBase.sol: Consider returning 0 instead of reverting in LockToToken()
- [N-14] SherX.sol: Change variable names weightSub and weightAdd to totalWeightOld and totalWeightNew
- [N-15] SherX.sol: Redeeming SherX may run out of gas
- [N-16] SherX.sol: Unsafe casting of _weights in setWeights()
- [N-17] General suggestions
- [N-18] Use EnumerableSet to store protocols
- [N-19] Check _aaveLmReceiver and _sherlock are not empty

Gas Optimizations

- [G-01] gas reduction in `calcUnderlying`
- [G-02] uncheckable math in `payout()`
- [G-03] Uncheckable math in `redeem()`
- [G-04] [Gas optimizations] - Public functions that are public, but could be external
- [G-05] Make variables immutable or constant

- [G-06] `SherX.setWeights` only accrue _tokens
- [G-07] `payout` does token transfers twice
- [G-08] `increaseApproval` gas improval
- [G-09] [Optimization] Caching variable
- [G-10] [Optimization] Packing various structs carefully
- [G-11] Two functions with the same implementation
- [G-12] x > 0 ==> x!=0
- [G-13] Gov.sol: Consider abstracting protocolUpdate() and protocolDepositAdd() to avoid duplicate checks
- [G-14] Gov.sol: Optimise protocolRemove()
- [G-15] Gov.sol: Small refactoring of tokenInit() to save gas
- [G-16] [Optimization] Setting higher value for optimize-runs
- [G-17] LibSherX.sol: Optimise calcUnderlying()
- [G-18] Manager.sol: Can avoid `safemath` sub in usdPerBlock and usdPool calculations
- [G-19] Manager.sol: Pass ps.sherXUnderlying instead of ps into updateData()
- [G-20] [Optimization] A branchless version of an if else statement
- [G-21] [Optimization] Use at least 0.8.4
- [G-22] [Optimization] Caching in for loops
- [G-23] [Optimization] Changing memory to calldata and again caching in loops
- [G-24] PoolStrategy unused parameter `_token`
- [G-25] Use calldata is a little more gas efficient
- [G-26] Avoid storing lp in AaveV2 constructor
- [G-27] Aav2V2 is Ownable but not owner capabilites are used
- [G-28] Declare NativeLock underlying variable as immutable
- [G-29] Functions aBalance and balanceOf
- [G-30] Call to LibDiamond.contractOwner() can be cached
- [G-31] Gas optimization on calculating the storage slot of a token

- [G-32] Avoid repeating storage reads in a loop to save gas
- [G-33] Saving gas by checking the last-recorded block number
- [G-34] Unused functions and storage cost gas.
- [G-35] Unnecessary require + if combination.
- [G-36] `transferFrom` gas improval

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