Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Amplol

Veridise Inc.
May 28, 2024

► **Prepared For:**

3Jane
https://www.3jane.xyz/

► **Prepared By:**

Ajinkya Rajput
Mark Anthony

► **Contact Us:** contact@veridise.com

► **Version History:**

May 29, 2024	V1
May 7, 2024	Initial Draft

Contents

Executive Summary 1

From May. 13, 2024 to May. 14, 2024, 3Jane engaged Veridise to review the security of their Amplol project. The review covered the on-chain contracts of Amplol which is a token issued to incentivize early depositors to 3Jane's 3JANE-EETH-X-C protocol which supports crypto-native derivatives market and its interaction with the vaults of the protocol. The 3JANE-EETH-X-C protocol allows market makers to create offers that users can utilize to swap their tokens. Additionally, the review covered the modifications introduced in commits `4e6e1cd` and `b4a96d2`. The modification in these commits allow the market makers to authorize a buyer to directly execute the swap. Veridise conducted the assessment over 2 person-days, with 2 engineers reviewing code over 1 days on code at commit `5851ed7`, `4e6e1cd` and `b4a96d2`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as an extensive manual code review.

Project summary. Amplol is an ERC-20 token that is issued to incentivze early depositors to the EtherFi vault of 3JANE-EETH-X-C protocol. The token is a rebasing token that implements rebasing mechanism by updating a monotonically increasing `base` value and keeping the balances of users same across rebases. The token overrides the `balanceOf()` function that divides the user's balance by `base` to return the number of AMPLOLs issued to the user.

Code assessment. The Amplol developers provided the source code of the Amplol contract for review. The code seems to be entirely developed by the 3Jane developers. To facilitate the Veridise auditors' understanding of the code, a write-up that summarized the intended behaviour of the token and examples of its usage was provided. The code also contained some in-line comments on structs and functions. The delivered source code also contained a test suite which the Veridise auditors noted tested many of the expected user-flows and much of the protocol's behavior.

During the audit, the 3Jane developers made several functional changes to the code. This is because the code was developed incrementally. The vaults were modified while the Amplol contracts were audited. Due to this, Veridise auditors executed the static analysis tools and extended the audit for to review the new code.

Summary of issues detected. The audit uncovered 13 issues, the most severe of which is a critical severity issue, V-AMP-VUL-001 which points out that an attacker can cause Denial of Service (DoS) to all the depositors that deposit after the attack. Another critical severity issue V-AMP-VUL-002 describes insufficient signature checks for EIP-712 structured data. The Veridise auditors also identified 3 high severity issues V-AMP-VUL-003 and V-AMP-VUL-004. V-AMP-VUL-003 reports a divide before multiply vulnerability that causes the protocol to lose money over time. V-AMP-VUL-004 identifies that `base` does not faithfully represent the TVL as intended.

The Veridise auditors also identified a medium severity issue V-AMP-VUL-006 that reports a possible use of a stale rebasing factor. Additionally Veridise auditors identified 2 low severity issues, 2 warnings, and 2 informational findings.

Among the 13 issues, 11 issues have been acknowledged and fixed by 3Jane, 0 issues are still unresolved, and 2 issues have been determined to be intended behavior after discussions with 3Jane.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the Amplol project and to avoid similar issues to those discovered in the audit in the future.

We recommend the developers to stress-test the tokenomics of `AMPLOL`; specifically, test the implications that can arise form the rebasing factor being monotonically increasing.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Project Dashboard 2

Table 2.1: Application Summary.

Name	Version	Type	Platform
Amplol	`5851ed7`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
May. 13 - May. 14, 2024	Manual & Tools	2	2 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	2	2	2
High-Severity Issues	3	3	3
Medium-Severity Issues	2	1	1
Low-Severity Issues	2	1	1
Warning-Severity Issues	2	2	2
Informational-Severity Issues	2	2	2
TOTAL	13	11	11

Table 2.4: Category Breakdown.

Name	Number
Logic Error	5
Maintainability	5
Flashloan	2
Data Validation	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Amplol's smart contracts. In our audit, we sought to answer questions such as:

- Can attackers exploit a monotonically increasing rebasing factor?
- Can an attacker exhibit a cost effective inflation attack?
- Can a rebase be triggered before the timer has expired?
- Can attackers perform replay attacks?
- Is the precision tracked consistently in all arithmetic calculations?
- Is the rebase mechanism vulnerable to flashloan attack?
- Is the signature verification performed correctly for the EIP-712 hashed data?
- Do the overridden ERC20 functions cause inconsistencies?
- Does the token accounting in mint, burn and balanceOf work as intended?
- Are the signatures vulnerable to signature malleability attacks?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard, which is designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

Scope. The scope of this audit is limited to the files

- `Amplol.sol`
- `AmplolStorage.sol`

The developers also provided `MockVault.sol` which is considered out of scope for this audit.

Methodology. Veridise auditors reviewed the reports of previous audits for Amplol, inspected the provided tests, and read the Amplol documentation. They then began a manual review of the code assisted by property-based testing. During the audit, the Veridise auditors regularly met with the Amplol developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

Vulnerability Report 4

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-AMP-VUL-001	Value of base can be inflated to cause DOS for . . .	Critical	Fixed
V-AMP-VUL-002	Wrong authorized checks	Critical	Fixed
V-AMP-VUL-003	Loss of precision in base	High	Fixed
V-AMP-VUL-004	Base does not promote increasing TVL	High	Fixed
V-AMP-VUL-005	Despitors can be griefed by inflating tvl at th. . .	High	Fixed
V-AMP-VUL-006	Possible use of stale base	Medium	Intended Behavior
V-AMP-VUL-007	Burning can have incorrect Amplol balance manag	Medium	Fixed
V-AMP-VUL-008	Timer not validated	Low	Intended Behavior
V-AMP-VUL-009	No storage gaps in AmplolSotrage	Low	Fixed
V-AMP-VUL-010	Missing totalSupply() override will cause incon. . .	Warning	Fixed
V-AMP-VUL-011	Base can be inflated to an extremely large valu. . .	Warning	Fixed
V-AMP-VUL-012	Amplol inherits ReentrancyGuardUpgradable but	Info	Fixed
V-AMP-VUL-013	Stray debug import	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-AMP-VUL-001: Value of base can be inflated to cause DOS for future users

Severity	Critical	**Commit**	5851ed7
Type	Flashloan	**Status**	Fixed
File(s)	Amplol.sol		
Location(s)	rebase()		
Confirmed Fix At	N/A		

In the `Amplol` contract the `mint(*` function is called when a depositor deposits `_amount` in the respective 3Jane vault. In the mint function the amount of tokens to be minted is calculated using the expression `_amount * 1e18 / base * FUN`. See snippet below.

```
function mint(address _recipient, uint256 _amount) external {
    if (msg.sender != address(vault)) revert BadMinter();
    _mint(_recipient, _amount * 1e18 / base * FUN);
}
```

Snippet 4.1: Snippet from `mint()`

As we can see `base` is in the denominator here. Meaning, the minimum `_amount` required for this expression to not evaluate to zero would be `base / 1e18`. And this minimum `_amount` would scale linearly with `base`, the larger `base` is the larger `_amount` needs to be. The value of base increases monotonically and can never decrease so once base reaches a large value it can never be decreased.

If the value of `base` is inflated to be large enough it will be very difficult for vault depositors to meet the minimum `_amount` requirements and more importantly the mint function will silently keep minting 0 for all deposits which are smaller than that. The process of inflating `base` is described in more detail in this V-AMP-VUL-011.

A malicious actor can make use of a flash loan to orchestrate this attack.

Impact An attacker can inflate the value of base to a very large value using a flash loan. This will cause deposits of value less than `base / 1e18` to always mint `0 Amplol`.

Future depositors will have to deposit at the very least `base / 1e18` to be able to mint non-zero `Amplol`. Depending on the value of inflated `base`, this amount required could be very large and therefore it may be practically impossible to achieve.

So, an attacker can effectively DOS future depositors by inflating base to a very large number.

Recommendation To avoid the possibility of inflation attacks, the initial few deposits and rebase should be done by the protocol itself. This ensures that any inflation attack is exponentially more costly to do.

Developer Response Developers fixed the issue by pegging the number of AMPLOLs minted to the TVL of the vault.

```
contract AmplolTest is Test {
  uint256 public constant account = 1;
  uint256 public constant account2 = 2;

  Amplol public amplol;
  MockVault public vault;

  uint256 private constant FUN = 1000;
  string public name = "AMPLOL";
  string public symbol = "AMPLOL";
  uint256 public timer = 3600;
  uint256 public pTVL = 100;
  uint256 private start;
  uint256 public startTotalBalance = 100;

  address public amplolImplementation;

  event NewTimer(uint256 timer);
  event ToggleTransfer(bool canTransfer);
  event Rebase(uint256 base, uint256 pTVL, uint256 pRebase);
  event Transfer(address indexed from, address indexed to, uint256 value);

  function setUp() public {
      vault = new MockVault();

      amplolImplementation = address(new Amplol());

      amplol = Amplol(
          address(
              new ERC1967Proxy(
                  amplolImplementation,
                  abi.encodeWithSelector(
                      Amplol.initialize.selector, name, symbol, address(vault), timer
    , pTVL, address(this)
                  )
              )
          )
      );

      start = block.timestamp;
      vault.setTotalBalance(startTotalBalance);
  }

// Aim - Test if inflation can cause mint() to mint 0 tokens and also identify the
    minAmount threshold as per given values for
// pTVL and base inflation.
function testInflateAndMint() public {
    // Initial pTVL = 100 wei.
    vault.setTotalBalance(startTotalBalance);
    vm.warp(amplol.nRebase());
    amplol.rebase(); // base = 10 ** 18, pTVL = 100
    uint256 base = amplol.base();
    assertEq(base, 1 * 1e18);

    // We define 100 ETH as an amount which is too costly for depositors to cross. In
     reality the amount will likely be
    // smaller than 100 ETH.
    uint256 inflateAmount = startTotalBalance * 1e20; // So The amount by which we
    want to inflate base is 1e20 (derived from 100 ETH).
    vault.setTotalBalance(inflateAmount);
    vm.warp(amplol.nRebase());
    amplol.rebase(); // base = 1e38, pTVL = inflateAmount
    base = amplol.base();
```

4.1.2 V-AMP-VUL-002: Wrong authorized checks

Severity	Critical	**Commit**	4e6e1cd
Type	Logic Error	**Status**	Fixed
File(s)	Swap.sol		
Location(s)	_swap()		
Confirmed Fix At	bf3c7c1		

_swap() implements the logic for swapping the sellers otoken with buyers biddingToken. This function takes a EIP712 structured hash. The function performs a few validations at the beginning as shown below.

```
function _swap(
    OfferDetails memory details,
    Offer storage offer,
    Bid calldata bid
) internal {
    require(DOMAIN_CHAIN_ID == getChainId(), "CHAIN_ID_CHANGED");

    address signatory = _getSignatory(bid);

    require(signatory != address(0), "SIGNATURE_INVALID");

    if (bid.signerWallet != signatory) {
        require(authorized[bid.signerWallet] == signatory, "UNAUTHORIZED");
    }
```

Snippet 4.3: Snippet from _swap()

The function first get the signer of the bid by calling _getSignatory() and stores it in signatory. Then the function checks that the signature is valid and if the bid.signerWallet is the signotary . If that is not the case the function additionally checks if the signatory is authorised for bid.signerWallet.

After performing a few more validations the function transfers the oToken to the bid.buyer and transfers the biddingToken to the seller and a fee to bid.referrer

Impact An attacker can steal funds from any account that has set non-zero allowance to the Swap contract using the following attack scenario.

1. An attacker identifies some benign Alice, has set allowance for the Swap contract.
2. An attacker crafts a bid with
   a) bid.signerWallet set to his own address.
   b) bid.referrer set to his own address
   c) bid.buyer set to address for Alice, the address of benign user from Step 1.
3. Attacker signs the crafted bid
4. Attacker makes a bid using the crafted bid, the flow of control eventually reaches _swap()
5. During the execution of _swap() , the swap is forced on Alice
6. The fees will be deposited to attacker.

Recommendation Check if the `bid.buyer` has authorised the signatory.

4.1.3 V-AMP-VUL-003: Loss of precision in base

Severity	High	**Commit**	5851ed7
Type	Logic Error	**Status**	Fixed
File(s)	`Amplol.sol`		
Location(s)	rebase()		
Confirmed Fix At	835f613		

The protocol issues `Amplol` ERC20 tokens to incentivize early depositors into the corresponding 3jane vault. This token is an approximation of rebasing token in which the balance is user remains constant on rebasing but the `balanceof()` function is overridden to divide the balance of user by the monotonically increasing `base` variable. This will issue more `Amplol` to depositors that deposit early in protocol.

The value of base is updated in the `rebase()` function which is an external function that can be called by any one.

```
function rebase() public {
    // Too early
    if (block.timestamp < nRebase()) revert EarlyRebase();
    uint256 cTVL = vault.totalBalance();
    // numba up-only LOL
    if (cTVL < pTVL) revert BadRebase();
    base *= cTVL / pTVL;
    //@audit recommendation
    //base = base * cTVL / pTVL;
    pTVL = cTVL; // Update the last recorded TVL
    pRebase = block.timestamp; // Update last rebase time
    emit Rebase(base, pTVL, pRebase);
}
```

Snippet 4.4: Snippet from `rebase()`

The value of TVL at last rebase is stored in `pTVL` variable. The `rebase()` function first fetches the current TVL in `cTVL` , and calculates `base *= cTVL / pTVL`. In this operation the value `cTVL/pTVL` is calculated first and then multiplied by base.

Since cTVL and pTVL have are same quantities measured at different times, they have the same decimals. This division before multiplication causes a loss of precision

Impact The value of base will be calculated lower than the intended value. This will cause protocol to issue more `Amplols`

Recommendation Calculate the value of `base` as `base = base * cTVL/pTVL`

4.1.4 V-AMP-VUL-004: Base does not promote increasing TVL

Severity	High	**Commit**	5851ed7
Type	Logic Error	**Status**	Fixed
File(s)	`Amplol.sol`		
Location(s)	rebase()		
Confirmed Fix At	N/A		

The protocol issues `Amplol` ERC20 tokens to incentivize early depositors into the corresponding 3jane vault. This token is an approximation of rebasing token in which the balance is user remains constant on rebasing but the `balanceof()` function is overridden to divide the balance of user by the monotonically increasing `base` variable. This will issue more `Amplol` to depositors that deposit early in protocol.

The value of base is calculated as shown below

```
function rebase() public {
    // Too early
    if (block.timestamp < nRebase()) revert EarlyRebase();
    uint256 cTVL = vault.totalBalance();
    // numba up-only LOL
    if (cTVL < pTVL) revert BadRebase();
    base *= cTVL / pTVL;
    pTVL = cTVL; // Update the last recorded TVL
    pRebase = block.timestamp; // Update last rebase time
    emit Rebase(base, pTVL, pRebase);
}
```

Snippet 4.5: Snippet from `rebase()`

The function get the current TVL in `cTVL`. The TVL at the time of previous rebase is stored in `pTVL`. The function reverts if current TVL, `cTVL` is less than previous TVL, `pTVL`. This causes the `base` to increase monotonically.

An attacker can use a flash loan to

1. Deposit a huge amount,
2. Call `rebase()`
3. Mint `AMPLOL`
4. Withdraw her funds and burn all her `AMPLOLs`

This will cause the `base` and `pTVL` to be set to a very high value, while vault does not have the corresponding TVL.

Impact Depositors after the attack described above get less `AMPLOL` compared to the intended amount. This reduces their incentives even if they are depositing at early stages of TVL

Recommendation We are still working on the recommendation

Developer Response Developers fixed the issue by pegging the number of AMPLOLs minted to the TVL of the vault.

4.1.5 V-AMP-VUL-005: Despitors can be griefed by inflating tvl at the start of every new time period

Severity	High	**Commit**	2d566c5
Type	Flashloan	**Status**	Fixed
File(s)	`Amplol.sol`		
Location(s)	mint()		
Confirmed Fix At	9bfcf9d		

The amount of amplol a user is minted depends entirely on the `deposit_amount` and the vault `tvl`. The tvl of the vault is rebased every `timer` seconds.

An attacker can inflate the `tvl` to a very large amount at the beginning of a `timer` period by using a flashloan. Although the `tvl` value can decrease it cannot be rebased until the `timer` duration expires. And so, within this duration the users who deposit to the vault will be griefed as they will be minted `0` amplol unless they deposit more than a very large amount which is `(tvl / FUN)`. See snippet below for the mint calculations.

This attack can be replayed each time period to grief the protocol 24/7. The costs attached for the attack are relatively cheaper - flashloan_fee (which is usually 0.09% of flashloan amount) + tx_fees. But, because there are a lot of elements required for a successful attack the likelihood of it happening is reduced a little.

```
function mint(address _recipient, uint256 _amount) external {
    if (msg.sender != address(vault)) revert BadMinter();
    _mint(_recipient, _amount * FUN / tvl);
}
```

Snippet 4.6: Snippet from `mint()`

Impact An attacker can continuously grief the depositors by inflating the tvl at the start of each period such that the depositors are always minted `0` amplol unless they deposit a very large amount.

Recommendation Having a timer based access mechanism to `rebase()` would not allow decrease of tvl until the timer expires which can be exploited to grief depositors by inflating the tvl.

Instead `rebase()` should be triggered internally with after each `mint()` and `burn()` which would ensure the state of tvl is always in sync with the actual state.

Developer Response [If applicable]

```
function testOneHourAttack() public {

    // Initial pTVL = 100 wei.
    vault.setTotalBalance(startTotalBalance);
    vm.warp(amplol.nRebase());
    amplol.rebase(); // TVL = 100
    uint256 tvl = amplol.tvl();
    assertEq(tvl, 1e20); // tvl = 1e20

    // We define 100 ETH as an amount which is too costly for depositors to cross. In reality the amount will likely be
    // smaller than 100 ETH. The aim is to make the cost for next depositor > 100 ETH by making the denominator ‘tvl‘ large so that if deposit amount
    // is not > 100 ETH the amplol minted on deposit is 0.

    // Attacker uses flashloan to inflate and then immediately removes his added deposits to pay back flashloan
    uint256 inflateAmount = startTotalBalance * 1e6; // So The amount by which we want to inflate base is 1e26 (derived from tvl * FUN).
    vault.setTotalBalance(inflateAmount);
    vm.warp(amplol.nRebase());
    amplol.rebase(); // TVL = inflateAmount - 1e26
    tvl = amplol.tvl();
    assertEq(tvl, 1e26); // tvl = 1e26

    // Now mimic remove in the same tx. Rebase() cannot be called for another ‘timer‘ seconds.
    inflateAmount = startTotalBalance; // Attacker has removed his flashloan.
    vault.setTotalBalance(inflateAmount);

    // For the next 1 hour rebase cannot be called. This may not exactly be 1 hour, unless attacker times it perfectly

    // Now untill rebse() is callable the following will exist within the conditions already defined before.

    // 1) Check that just below value of deposit 1e20 the amount of tokens minted is 0. This exists only till next rebase().
    uint256 minAmountMinusOne = 1e20 - 1; // just 1 less than 1e18 will mint 0 tokens.
    address alice = vm.addr(account);
    vm.prank(address(vault));
    amplol.mint(alice, minAmountMinusOne);
    assertEq(amplol.balanceOf(alice), 0); // Mints 0 to Alice


    // 2) Check that just exactly at 1e20 the amount of tokens minted is 1e26.
    // Test for exact minimum value
    uint256 minAmount = 1e20; // Exact minimum.
    address bob = vm.addr(account2);
    vm.prank(address(vault));
    amplol.mint(bob, minAmount);

    assertEq(amplol.balanceOf(bob), 1e26); // gets minted FUN * minAmount


    // Check after next rebase() that the vector does not prevail.

    vm.warp(amplol.nRebase());
    amplol.rebase();
    tvl = amplol.tvl();
    assertEq(tvl, 1e20); // tvl = 1e20

    // 1) Check that just below value of deposit 1e20 the amount of tokens minted is
```

4.1.6 V-AMP-VUL-006: Possible use of stale base

Severity	Medium	**Commit**	5851ed7
Type	Logic Error	**Status**	Intended Behavior
File(s)	Amplol.sol		
Location(s)	mint(), burn()		
Confirmed Fix At	N/A		

The protocol implements approximation of the rebasing mechanism in which the protocol does not update the balances of addresses holding tokens on rebases. Instead the protocol updates the value of the `base` variable based on current TVL in the corresponding 3jane vault and TVL at the time of previous rebase as shown below.

```
function rebase() public {
    // Too early
    if (block.timestamp < nRebase()) revert EarlyRebase();
    uint256 cTVL = vault.totalBalance();
    // numba up-only LOL
    if (cTVL < pTVL) revert BadRebase();
    base *= cTVL / pTVL;
    //@audit recommendation
    //base = base * cTVL / pTVL;
    pTVL = cTVL; // Update the last recorded TVL
    pRebase = block.timestamp; // Update last rebase time
    emit Rebase(base, pTVL, pRebase);
}
```

Snippet 4.8: Snippet from `rebase()`

The protocol further overrides `balanceOf()`, `mint()` and `burn()` as shown below.

```
function mint(address _recipient, uint256 _amount) external {
    if (msg.sender != address(vault)) revert BadMinter();
    _mint(_recipient, _amount * 1e18 / base * FUN);
    //@audit
    //_mint(_recipient, _amount * FUN * 1e18 / base );
}
```

Snippet 4.9: Snippet from `mint()`

```
function burn(address _recipient, uint256 _amount) external {
    if (msg.sender != address(vault)) revert BadBurner();
    _burn(_recipient, _amount * 1e18 / base * FUN);
}
```

Snippet 4.10: Snippet from `burn()`

```
function balanceOf(address account) public view override returns (uint256) {
    return super.balanceOf(account) * base / 1e18;
}
```

Snippet 4.11: Snippet from `balanceOf()`

If deposits are made after the last rebase, the TVL increases but the value of base is not updated until rebase is called. Therefore, it is possible that the `mint()`, `burn()` and `balanceOf()` may use the stale value of the `base`

Impact Users or downstream protocols that call `balanceOf()` may an inflated `Amplol` balance.

Since, vault is the only possible caller of `mint()` and `burn()` , if vault does not call rebase before calling these functions, it may mint or burn more tokens than required.

Recommendation Call rebase() in the beginning of `mint()`, `burn()` and `balanceOf()`

Developer Response The developers informed us that it is okay for a stale base to be read as rebasing will be performed at a regular interval using openzeppelins timer.

4.1.7 V-AMP-VUL-007: Burning can have incorrect Amplol balance management

Severity	Medium	**Commit**	5851ed7
Type	Logic Error	**Status**	Fixed
File(s)	`Amplol.sol`		
Location(s)	burn(), 3Jane Vault		
Confirmed Fix At	N/A		

In the Amplol contract the `burn()` takes in an amount, adjusts it according to the current base and burns the resulting amount of Amplol's.

But if the amount of `eETH` is passed to the function directly then there will be residual balance left. This occurs because the Amplol balance of user inflates with each rebase therefore the correlation between the user `eETH` deposit amount and user Amplol balance is not completely 1:1.

Let us illustrate this with an example.

Consider `base` has value `2e18` with `pTVL=10M`. If a user deposits `1M`, he will be issued `500K` Amplol. The cTVL is now `11M`. The user can now call `rebase()` and `base` will become `2*11/10 = 2.2e18`, so the amplol balance of user automatically becomes `500K * 2 * 1.1` without any deposits in between. Now, if user withdraws all `eETH` he will have to burn `1M/2.2` amplol i.e. `454K`, which will leave `46k` amplol from the original `500k` as residual balance. See the snippet below for a detailed flow with numbers.

```

PHASE - I
----------------------------------------
initial_base = 2e18;
initial_total_deposit = 10M;
user_deposit = 1M;
mint(user, user_deposit);
user_amplol_minted_internal = 500k;
balanceOf(user) = 1e9; [500k * 2 * FUN]

PHASE - II
----------------------------------------
final_total_deposit = 11M; // 10M + 1M
rebase(); // Trigger rebase
final_base = initial_base * 1.1 = 2.2e18;

balanceOf(user) = 1.1e9; // pre-burn balance
burn(user, user_deposit); // burning amount equal to user_deposit
balanceOf(user) = 100001000 // post-balance residue = 1.00001e8

// To avoid residue amount to be burnt should be balanceOf(user) = 1.1e9
// in this case.
```

Snippet 4.12: Pseudocode for the example

The issue has also been confirmed with a POC. See snippet below.

```
function testIssue() public {

  MockVault vault2 = new MockVault();

  address amplolImplementation2 = address(new Amplol());
  uint256 pTVL2 = 500000;

  Amplol amplol2 = Amplol(
      address(
          new ERC1967Proxy(
              amplolImplementation2,
              abi.encodeWithSelector(
                  Amplol.initialize.selector, name, symbol, address(vault2), timer,
    pTVL2, address(this))
              )
          )
      )
  );

  start = block.timestamp;
  uint256 initialBalance = 1e6;
  vault2.setTotalBalance(initialBalance);
  vm.warp(amplol2.nRebase());
  amplol2.rebase(); // base = 2 * 1e18, pTVL = 1M

  uint256 base = amplol2.base();
  assertEq(base, 2 * 1e18);

  uint256 amount = 1e6; // Add 1M
  address alice = vm.addr(account);
  vm.prank(address(vault2));
  amplol2.mint(alice, amount); // alice actual balance = 500k
  console.log(amplol2.balanceOf(alice)); // 1e9

  vault2.setTotalBalance(11e5);
  vm.warp(amplol2.nRebase());
  amplol2.rebase(); // base = 2 * 1e18, pTVL = 1M
  console.log(amplol2.balanceOf(alice)); // 1e9
  console.log(amplol2.base());

  vm.prank(address(vault2));
  amplol2.burn(alice, amount); // Burning deposited amount
  console.log(amplol2.balanceOf(alice)); // 100001000 - residual balance
}
```

Snippet 4.13: A POC to illustrate that using vault balance to burn amplol tokens can result in incorrect accounting of amplol

Impact Unless the amount passed into the burn function is adjusted correctly for rebased values of balance, the burned balance will not match the actual balance intended to be burnt.

Recommendation The value of Amplol to be burnt should be calculated as a percentage of user `eETH` balance. If a user has `1M` eETH deposit and withdraws `500k` then the amount of amplol to be burnt can be calculated using `balanceOf(user) * 50%` which will ensure that the equivalent percentage of users Amplol is burnt.

Developer Response Developers fixed this issue by pegging the value of AMPLOL to the TVL of the vault

4.1.8 V-AMP-VUL-008: Timer not validated

Severity	Low	**Commit**	5851ed7
Type	Data Validation	**Status**	Intended Behavior
File(s)	`Amplol.sol`		
Location(s)	initialize()		
Confirmed Fix At	N/A		

The `rebase()` function updates the value of the `base`. This function can only be called after `timer` amount of time is elapsed after last rebase.

```
function rebase() public {
  // Too early
  if (block.timestamp < nRebase()) revert EarlyRebase();
```

Snippet 4.14: Snippet from `rebase()`

The `timer` is set in `initialize()` and can be updated in `setTimer()` as shown below.

```
function setTimer(uint256 _timer) external onlyOwner {
    timer = _timer;
    emit NewTimer(_timer);
}
```

Snippet 4.15: Snippet from `setTimer()`

```
timer = _timer;
```

Snippet 4.16: Snippet from `initialize()`

The value of `timer` is updated without any validation.

Impact

- Timer may be set to very large value, which will effectively stop deposits
- Timer may be set to zero, which will allow rebases to happen any time

Recommendation Add a validations when updating the value of `timer`

Developer Response [If applicable]

4.1.9 V-AMP-VUL-009: No storage gaps in AmplolSotrage

Severity	Low	**Commit**	eecd846
Type	Maintainability	**Status**	Fixed
File(s)	`AmplolStorage.lol`		
Location(s)	See Issue Description		
Confirmed Fix At	4da9aa8		

The `AmplolStore` contract is an abstract contract that allocates storage for upgradable `Amplol` contract.

```
abstract contract AmplolStore is IAmplol {
    // Previous TVL
    uint256 internal pTVL;
    // Previous Rebase
    uint256 internal pRebase;
    // This will act as the scaling factor for balance calculations
    uint256 public base;
    // Can transfer
    bool public canTransfer;
    // Timer between rebases
    uint256 public timer;
    // 3Jane vault
    IVault public vault;
}
```

Snippet 4.17: Snippet from `AmplolStore`

There is no storage gap allocated for further upgrade.

Impact The protocol will not be allowed to allocate new storage variables in further upgrades.

Recommendation Allocate sufficient storage gaps at the end of the `AmplolStore`

4.1.10 V-AMP-VUL-010: Missing totalSupply() override will cause inconsistency

Severity	Warning	**Commit**	5851ed7
Type	Maintainability	**Status**	Fixed
File(s)	`Amplol.sol`		
Location(s)	ERC20Upgradeable		
Confirmed Fix At	0abc3cd		

The `Amplol` contract overrides `balanceOf()` in the snippet below to show the correctly adjusted rebased value on its Frontend as well as on a blockchain explorer. But it does not override `totalSupply()`.

In addition to `balanceOf()`, the function `totalSupply()` should also be overridden to ensure that it reflects adjusted values for total supply.

```
function balanceOf(address account) public view override returns (uint256) {
    return super.balanceOf(account) * base / 1e18;
}
```

Snippet 4.18: Snippet of `balanceOf()`

Impact A common user checking balance and total supply of tokens will notice the discrepancy and be confused. It would also cause issues for any 3rd party integration which uses metrics and such.

Recommendation Override `totalSupply()` in a similar manner to `balanceOf()` to reflect correctly rebased values.

4.1.11 V-AMP-VUL-011: Base can be inflated to an extremely large value in first few deposits

Severity	Warning	**Commit**	5851ed7
Type	Maintainability	**Status**	Fixed
File(s)	`Amplol.sol`		
Location(s)	rebase()		
Confirmed Fix At	N/A		

The `Amplol` token rebases itself on the basis of the TVL in its respective 3Jane vault. See snippet below.

The first 2 transactions to the vault can inflate the value of `base` at will. For example, an attacker can deposit `1` wei and call `rebase()` afterwards. Then they can deposit any `amount > 1 ether` and the value of `base` would be inflated to be greater than `1e36`. The above works assuming `ptvl` is set to `1`.

Therefore the above attack scenario also depends on the initial value of `pTVL` which is set in the initialiser. Depending on that value the amount needed to cause the inflation would increase.

```
function rebase() public {
    // Too early
    if (block.timestamp < nRebase()) revert EarlyRebase();
    uint256 cTVL = vault.totalBalance();
    // numba up-only LOL
    if (cTVL < pTVL) revert BadRebase();
    base *= cTVL / pTVL;
    pTVL = cTVL; // Update the last recorded TVL
    pRebase = block.timestamp; // Update last rebase time
    emit Rebase(base, pTVL, pRebase);
}
```

Snippet 4.19: Snippet from `rebase()`

Impact A sudden extreme increase in `base` can cause unforeseen issues and is undesirable for the protocol.

Recommendation To protect against inflation of `base` the initial few deposits and `rebase()` should be done by the protocol itself to ensure any sort of inflation attack is exponentially more costly for an attacker to do

Developer Response Developers fixed the issue by pegging the number of AMPLOLs minted to the TVL of the vault.

4.1.12 V-AMP-VUL-012: Amplol inherits ReentrancyGuardUpgradable but never uses nonReentrant modifier

Severity	Info	**Commit**	5851ed7
Type	Maintainability	**Status**	Fixed
File(s)	`Amplol.sol`		
Location(s)	Amplol		
Confirmed Fix At	fc9f4b8		

The contract `Amplol` inherits `ReentrancyGuardUpgradeable` but never makes use of the `nonReentrant` modifier. See snippet.

Furthermore `__ReentrancyGuard_init_unchained()` is also called inside `initialize()` but as `nonReentrant()` is never used, these will just add to gas costs.

```
contract Amplol is ERC20Upgradeable, OwnableUpgradeable, ReentrancyGuardUpgradeable,
    UUPSUpgradeable, AmplolStore {
uint256 private constant FUN = 1000;

constructor() {
    _disableInitializers();
}
```

Snippet 4.20: Snippet from `Amplol`

Impact The deployment gas costs for the contract will increase.

Recommendation Remove `ReentrancyGuardUpgradeable` inheritance and also remove `__ReentrancyGuard_init_u`
`()` from `initialize()`.

4.1.13 V-AMP-VUL-013: Stray debug import

Severity	Info	**Commit**	5851ed7
Type	Maintainability	**Status**	Fixed
File(s)	`Amplol.sol`		
Location(s)	See description		
Confirmed Fix At	5229607		

Foundry build system provides a utility contract `console.sol` for debugging. `Amplol.sol` imports this contract. This import should not be present in deployed contract.

```
import "forge-std/console.sol";
```

Snippet 4.21: Snippet from `Amplol.sol`

Recommendation Remove the line that imports `forge-std/console.sol`