BLOCKSEC

Security Audit

Report for Bedrock DAO

Date: May 31, 2024 **Version:** 1.1

Contact: contact@blocksec.com

Contents

Chapter 1 Introduction **1**

1.1 About Target Contracts . . . 1
1.2 Disclaimer . . . 1
1.3 Procedure of Auditing . . . 2
1.3.1 Software Security . . . 2
1.3.2 DeFi Security . . . 2
1.3.3 NFT Security . . . 3
1.3.4 Additional Recommendation . . . 3
1.4 Security Model . . . 3

Chapter 2 Findings **5**

2.1 DeFi Security . . . 5
2.1.1 Potential inability to claim rewards . . . 5
2.1.2 Incompatible `addBribeERC20` interface . . . 7
2.1.3 Incorrect `voteUsed` for event `GaugeVoted` . . . 8
2.1.4 Unnecessary unlimited approval . . . 9
2.2 Additional Recommendation . . . 10
2.2.1 Ensure consistency between `rewardToken` and `VotingEscrow.assetToken` . . . 10
2.2.2 Maintain consistency in the style of code implementation and usage . . . 11
2.2.3 Remove redundant code . . . 11
2.2.4 Optimize gas usage . . . 11
2.2.5 Add a non-zero check for important addresses . . . 12
2.2.6 Remove the `resetAllowance` function . . . 12
2.3 Note . . . 13
2.3.1 Potential centralization risks . . . 13
2.3.2 Concerns regarding reward distribution . . . 13
2.3.3 Lack of pause/unpause mechanisms . . . 13

Report Manifest

Item	Description
Client	Bedrock
Target	Bedrock DAO

Version History

Version	Date	Description
1.0	May 7, 2024	First release
1.1	May 31, 2024	Token symbol rebranding

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-05-31 07:39:16+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The focus of this audit is on Bedrock DAO [1] of Bedrock. Bedrock DAO is a project where users can lock their BR tokens [2], vote on different gauges, and get weekly rewards based on their vote weight. Additionally, the project leverages the *veTokennomics* model of Pendle Finance by using the weekly rewards of gauges as incentives for the Pendle markets.

Please note that this audit is limited to the smart contracts located within the `contracts` folder of the repository. Files intended for testing purposes, specifically those found in the `contracts/mocks` directory, are not within the scope of the audit.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Bedrock DAO	`Version 1`	`48f873b8771055465b331b837c79faf0ddbb76e3`
	`Version 2`	`a199bde00b17ad341eee1fef96da2bc90f13e460`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

[1] https://github.com/Bedrock-Technology/bedrock-dao/

[2] The token's symbol has been rebranded from `BRT` to `BR`, and so has `veBRT` to `veBR`.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [3] and Common Weakness Enumeration [4]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.

[3] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[4] https://cwe.mitre.org/

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **four** potential security issues. Besides, we have **six** recommendations and **three** notes.

- High Risk: 1
- Medium Risk: 1
- Low Risk: 2
- Recommendation: 6
- Note: 3

ID	Severity	Description	Category	Status
1	High	Potential inability to claim rewards	DeFi Security	Fixed
2	Medium	Incompatible `addBribeERC20` interface	DeFi Security	Fixed
3	Low	Incorrect `voteUsed` for event `GaugeVoted`	DeFi Security	Fixed
4	Low	Unnecessary unlimited approval	DeFi Security	Fixed
5	-	Ensure consistency between `rewardToken` and `VotingEscrow.assetToken`	Recommendation	Fixed
6	-	Maintain consistency in the style of code implementation and usage	Recommendation	Fixed
7	-	Remove redundant code	Recommendation	Fixed
8	-	Optimize gas usage	Recommendation	Fixed
9	-	Add a non-zero check for important addresses	Recommendation	Fixed
10	-	Remove the `resetAllowance` function	Recommendation	Fixed
11	-	Potential centralization risks	Note	-
12	-	Concerns regarding reward distribution	Note	-
13	-	Lack of pause/unpause mechanisms	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential inability to claim rewards

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `claim` function of the `VeRewards` contract, the variable `userLastSettledWeek` will only be updated if `profits` > 0, where `profits` is calculated in the `_calcProfits` function.

```
92    function claim(bool restake) external nonReentrant whenNotPaused {
93        _updateReward();
94
95        // calc profits and update settled week
96        (uint256 profits, uint256 settleToWeek) = _calcProfits(msg.sender);
97
98        if (profits == 0) return;
```

```
 99
100       userLastSettledWeek[msg.sender] = settleToWeek;
101
102       if (restake) {
103           IERC20(rewardToken).safeApprove(votingEscrow, profits);
104           IVotingEscrow(votingEscrow).depositFor(msg.sender, uint128(profits));
105       } else {
106           // transfer profits to user
107           IERC20(rewardToken).safeTransfer(msg.sender, profits);
108       }
109
110       // track balance decrease
111       _balanceDecrease(profits);
112
113       // log
114       emit Claimed(msg.sender, restake, profits);
115   }
```

Listing 2.1: contracts/ve_rewards.sol

Specifically, in the `_calcProfits` function, the `nextWeek` variable is calculated as the the maximum value between `userLastSettledWeek[account]` and `getFirstUserPoint(account)`, and the calculation loop will break after `MAXWEEKS` weeks. This could result in a scenario where users are unable to claim rewards after `MAXWEEKS` weeks without profits.

```
150   function _calcProfits(address account) internal view returns (uint256 profits, uint256
          settleToWeek) {
151       // load user's latest settled week
152       settleToWeek = userLastSettledWeek[account];
153       if (settleToWeek < genesisWeek) {
154           settleToWeek = genesisWeek;
155       }
156
157       // lookup user's first ve deposit timestamp
158       (,,uint256 ts) = IVotingEscrow(votingEscrow).getFirstUserPoint(account);
159       if (settleToWeek < ts) {
160           settleToWeek = _getWeek(ts);
161       }
162
163       // loop throught weeks to accumulate profits
164       for (uint i=0; i<MAXWEEKS;i++) {
165           uint256 nextWeek = settleToWeek + WEEK;
166           if (nextWeek > block.timestamp || nextWeek > lastProfitsUpdate) {
167               break;
168           }
169           settleToWeek = nextWeek;
170           uint256 preSettleWeek = settleToWeek - WEEK;
171
172           // get total supply of the week
173           uint256 totalSupply = IVotingEscrow(votingEscrow).totalSupply(preSettleWeek);
174           if (totalSupply > 0) { // avert division by zero
175               profits += weeklyProfits[settleToWeek]
176                       * IVotingEscrow(votingEscrow).balanceOf(account, preSettleWeek)
```

```
177                         / totalSupply;
178             }
179         }
180
181         return (profits, settleToWeek);
182     }
```

Listing 2.2: contracts/ve_rewards.sol

Impact Users can't claim rewards.

Suggestion In the `claim` function, the `userLastSettledWeek[msg.sender]` should be updated even if `profits` is zero.

2.1.2 Incompatible `addBribeERC20` interface

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `_updateReward` function of the `PenpieAdapter` contract, the `IBribeManager` interface is incompatible with Penpie's current implementation. Specifically, the invocation to the `addBribeERC20` function lacks a parameter named `_forPreviousEpoch`, resulting in the failure to update gauge rewards.

```
103     function _updateReward() internal {
104         // get current balance for the reward token
105         uint256 balance = IERC20(rewardToken).balanceOf(address(this));
106
107         // return if there's no amount available to distribute
108         if (balance == 0) {
109             return;
110         }
111
112         // get pid for the pendle market from bribe manager
113         uint256 _pid = IBribeManager(bribeManager).marketToPid(pendleMarket);
114
115         // transfer bribe
116         uint256 currentAllowance = IERC20(rewardToken).allowance(address(this), bribeManager);
117         if (currentAllowance < balance) {
118             IERC20(rewardToken).safeApprove(bribeManager, type(uint256).max);
119         }
120         IBribeManager(bribeManager).addBribeERC20(1, _pid, rewardToken, balance);
121
122         emit RewardsDistributed(pendleMarket, _pid, rewardToken, balance);
123     }
```

Listing 2.3: contracts/penpie_adapter.sol

Impact The project can't update the gauge rewards.

Suggestion Update the `IBribeManager` interface to the latest version.

2.1.3 Incorrect `voteUsed` for event `GaugeVoted`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `voteForGaugeWeight` function of the `GaugeController` contract, the `voteUsed` for event `GaugeVoted` is calculated as

$$newVoteData.slope * (newVoteData.end - newVoteData.voteTime)$$

where `voteTime` equals `block.timestamp`. However, the actual voted weight for the gauge is calculated as

$$newVoteData.slope * (newVoteData.end - nextTime)$$

where `nextTime` represents the time of the next week. Therefore, the `voteUsed` will be greater than the actual vote used.

```
265   function voteForGaugeWeight(address _gAddr, uint256 _userWeight)
266   external
267   nonReentrant
268{
269   require(
270       _userWeight >= 0 && _userWeight <= PREC,
271       "Invalid voting power provided"
272   );
273
274   // Get user's latest veToken stats
275   (, int128 slope,) = IVotingEscrow(votingEscrow).getLastUserPoint(msg.sender);
276
277   require(slope > 0, "no voting power available");
278
279   uint256 lockEnd = IVotingEscrow(votingEscrow).lockEnd(msg.sender);
280
281   uint256 nextTime = _getWeek(block.timestamp + WEEK);
282
283   require(lockEnd > nextTime, "Lock expires before next cycle");
284
285   // Prepare slopes and biases in memory
286   VoteData memory oldVoteData = userVoteData[msg.sender][_gAddr];
287   require(
288       block.timestamp >= oldVoteData.voteTime + WEIGHT_VOTE_DELAY,
289       "Can't vote so often"
290   );
291
292   VoteData memory newVoteData = VoteData({
293       slope: (SafeCast.toUint256(slope) * _userWeight) / PREC,
294       end: lockEnd,
295       power: _userWeight,
296       voteTime: block.timestamp
297   });
298   // Check and update powers (weights) used
299   _updateUserPower(oldVoteData.power, newVoteData.power);
```

```
300
301     _updateScheduledChanges(
302         oldVoteData,
303         newVoteData,
304         nextTime,
305         lockEnd,
306         _gAddr
307     );
308
309     _getTotal();
310     userVoteData[msg.sender][_gAddr] = newVoteData;
311     uint256 voteUsed = newVoteData.slope * (newVoteData.end - newVoteData.voteTime);
312
313     emit GaugeVoted(block.timestamp, msg.sender, _gAddr, _userWeight, voteUsed);
314 }
```

Listing 2.4: contracts/gauge_controller.sol

Impact Incorrect values of `voteUsed` may lead to unexpected results.

Suggestion Use `nextTime` instead of `newVoteData.voteTime` in the calculation of `voteUsed`.

2.1.4 Unnecessary unlimited approval

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `PenpieAdapter` contract, the `updateReward` function is callable by anyone, and it sets the `bribeManager`'s allowance of the `rewardToken` to `type(uint256).max`. This is unnecessary and potentially risky [1].

```
80     function updateReward() external { _updateReward(); }
```

Listing 2.5: contracts/penpie_adapter.sol

```
103     function _updateReward() internal {
104         // get current balance for the reward token
105         uint256 balance = IERC20(rewardToken).balanceOf(address(this));
106
107         // return if there's no amount available to distribute
108         if (balance == 0) {
109             return;
110         }
111
112         // get pid for the pendle market from bribe manager
113         uint256 _pid = IBribeManager(bribeManager).marketToPid(pendleMarket);
114
115         // transfer bribe
116         uint256 currentAllowance = IERC20(rewardToken).allowance(address(this), bribeManager);
```

[1] https://blocksec.com/blog/exploring-the-tradeoff-between-convenience-and-security-in-unlimited-approval-erc-20-tokens

```
117         if (currentAllowance < balance) {
118             IERC20(rewardToken).safeApprove(bribeManager, type(uint256).max);
119         }
120         IBribeManager(bribeManager).addBribeERC20(1, _pid, rewardToken, balance);
121
122         emit RewardsDistributed(pendleMarket, _pid, rewardToken, balance);
123     }
```

Listing 2.6: contracts/penpie_adapter.sol

Impact Unlimited approval may lead to unexpected financial loss.

Suggestion Approve only `balance` instead of `type(uint256).max`.

2.2 Additional Recommendation

2.2.1 Ensure consistency between `rewardToken` and `VotingEscrow.assetToken`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description It is recommended to assign `rewardToken` with `VotingEscrow.assetToken` in the `initialize` function of the `VeRewards` contract, rather than assigning it with an argument. This ensures consistency between `rewardToken` and `VotingEscrow.assetToken`. Otherwise, the following code snippet may not work properly.

```
55  function initialize(
56      address _votingEscrow,
57      address _rewardToken
58  ) initializer public {
59      __Pausable_init();
60      __Ownable_init();
61      __ReentrancyGuard_init();
62
63      require(_votingEscrow != address(0x0), "_votingEscrow nil");
64      require(_rewardToken != address(0x0), "_rewardToken nil");
65
66      votingEscrow = _votingEscrow;
67      rewardToken = _rewardToken;
68
69      genesisWeek = _getWeek(block.timestamp);
70      lastProfitsUpdate = genesisWeek;
71  }
```

Listing 2.7: contracts/ve_rewards.sol

```
103 IERC20(rewardToken).safeApprove(votingEscrow, profits);
104 IVotingEscrow(votingEscrow).depositFor(msg.sender, uint128(profits));
```

Listing 2.8: contracts/ve_rewards.sol

Suggestion Assign `rewardToken` with `VotingEscrow.assetToken` in the `VeRewards` contract.

2.2.2 Maintain consistency in the style of code implementation and usage

Status Fixed in `Version 2`

Introduced by `Version 1`

Description It is recommended to maintain consistency in the style of code implementation and usage throughout the contracts. For instance, the `VeRewards`, `PenpieAdapter`, and `Cashier` contracts inherit `OwnableUpgradeable`, while others inherit `AccessControl` or `AccessControlUpgradeable`. Additionally, some contracts rely on the `_floorToWeek` or `_getWeek` function for calculations, while others use the expression `block.timestamp / WEEK * WEEK` directly.

Suggestion Revise the code accordingly.

2.2.3 Remove redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `VotingEscrow` contract inherits the `AccessControlUpgradeable` contract, which provides an external function `grantRole` to grant a specific role to an address. Therefore, the `assignRewardsManager` function is redundant and can be removed.

```
147   function assignRewardsManager(address rewardsContract) public onlyRole(DEFAULT_ADMIN_ROLE) {
148       _grantRole(REWARDS_MANAGER_ROLE, rewardsContract);
149   }
```

Listing 2.9: contracts/voting_escrow.sol

Suggestion Remove the redundant `assignRewardsManager` function.

2.2.4 Optimize gas usage

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `VotingEscrow` contract, the `depositFor` function accepts a uint128 parameter `_value`, which is then cast to uint256. Changing the parameter type to uint256 could optimize gas usage.

```
175   function depositFor(address _addr, uint128 _value)
176       external
177       nonReentrant
178       whenNotPaused
179       onlyRole(REWARDS_MANAGER_ROLE)
180   {
181       LockedBalance memory locked_ = LockedBalance({
182           amount: locked[_addr].amount,
183           end: locked[_addr].end
184       });
185
186       require(_value > 0, "Must stake non zero amount");
187       require(locked_.amount > 0, "No existing lock found");
```

```
188         require(locked_.end > block.timestamp, "Cannot add to expired lock. Withdraw");
189
190         _depositFor(_addr, _value, 0, locked_, LockAction.DEPOSIT_FOR);
191     }
```

Listing 2.10: contracts/voting_escrow.sol

Suggestion Change the parameter type to uint256.

2.2.5 Add a non-zero check for important addresses

Status Fixed in `Version 2`

Introduced by `Version 1`

Description It is recommended to add a check to ensure that `votingEscrow` is a non-zero address in the `initialize` function of the `GaugeController` contract.

```
116     function initialize(address _votingEscrow) initializer public {
117         __AccessControl_init();
118         __ReentrancyGuard_init();
119
120         votingEscrow = _votingEscrow;
121         timeTotal = block.timestamp / WEEK * WEEK;
122
123         _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
124         _grantRole(AUTHORIZED_OPERATOR, msg.sender);
125     }
```

Listing 2.11: contracts/gauge_controller.sol

Suggestion Add a non-zero check accordingly.

2.2.6 Remove the `resetAllowance` function

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `PenpieAdapter` contract, the privileged `resetAllowance` function is used to reset the bribeManager's allowance to zero, as specified in the design document provided by Bedrock: *Strictly, the PenpieAdapter contract (gauge) owner is permitted to reset the allowance granted to the BribeManager contract.*

However, the `updateReward` function is callable by anyone, and it increases the `bribeManager`'s allowance to `type(uint256).max`. Therefore, after the owner calls the `resetAllowance` function to reset the allowance, a malicious user can send one token to the `PenpieAdapter` contract and invoke the `updateReward` function to increase the allowance, making the owner's reset operation useless. Hence, as the unlimited approval issue can be solved in Section 2.1.4, the `resetAllowance` function can be removed.

```
85     function resetAllowance() external onlyOwner {
86         uint256 currentAllowance = IERC20(rewardToken).allowance(address(this), bribeManager);
87         if (currentAllowance != 0) {
```

```
88          IERC20(rewardToken).safeApprove(bribeManager, 0);
89          emit ResetAllowanceForBribeManager(block.timestamp);
90      }
91  }
```

Listing 2.12: contracts/penpie_adapter.sol

Suggestion Remove the `resetAllowance` function.

2.3 Note

2.3.1 Potential centralization risks

Description Multiple privileged functions are used to execute important operations, which can lead to centralization risks. For example, only the `BedrockDAO` contract owner can mint `BR` tokens, which are utility tokens for the voting and rewarding process.

2.3.2 Concerns regarding reward distribution

Description There may raise some concerns relate to the reward distribution:

- **Relying on backend service**. Both the VE and gauge reward distribution rely on the weekly calls from the *BedrockDAO Rewarder Backend service*. If this service goes down, users will receive incorrect rewards.
- **Locking dust VE rewards**. VE rewards are calculated based on `weeklyProfits` and the proportion of user's `veBR` for the week. Upon reward calculation, the rewards will be rounded down to prevent overflowing the `accountedBalance`. Consequently, a portion of `weeklyProfits` may remain unclaimed due to precision loss. These residual dust rewards will be locked in the `VeRewards` contract, as the contract does not provide an interface to process these rewards.
- **Updating** `globalWeekEmission`. In the `_distributeRewards` function of the `Cashier` contract, the gauge's rewards are calculated as `globalWeekEmission * gaugeRelativeWt`. If the contract owner updates `globalWeekEmission` when there exist unpaid gauges, then gauges with the same weight will receive different rewards. Therefore, `globalWeekEmission` should only be updated after all rewards have been distributed.

2.3.3 Lack of pause/unpause mechanisms

Description As specified in the design document, all smart contracts should implement a pause/unpause mechanism, as follows: *All smart contracts of the BedrockDAO project, including VotingEscrow, VeRewards, BedrockDAO, PenpieAdapter, GaugeController, and Cashier, should indeed support pause and unpause operations.*

Most of them inherit the `PausableUpgradeable` contract. However, the implementation does not strictly adhere to the document. For example, the `GaugeController` contract does not inherit this mechanism, and the `PenpieAdapter` contract does not utilize it to restrict the `updateReward` function.

Note that in `Version 2`, the pause/unpause mechanism is introduced to the `GaugeController` contract. Specifically, the privileged functions to modify gauges, gauge types, and weights are prohibited when the contract is paused. However, the `voteForGaugeWeight` function is not implemented with such a restriction, allowing voting at all times. To align with the protocol design, the `whenNotPaused` modifier is removed from several functions, enabling the withdrawals of expired locks and the distribution of VE and gauge rewards even when the contracts are paused.

Feedback from the Project

- `PenpieAdapter`, `Cashier`, `VeRewards`: The project allows rewards to be distributed as usual during other contracts are paused.
- `GaugeController`: Users should be able to utilize their already locked voting power even if the `VotingEscrow` contract is paused.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS