



Kommunitas Staking V3

Smart Contract Security Audit

Prepared by ShellBoxes

Jan 2nd, 2023 - Jan 7th, 2023

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	Kommunitas
Version	1.0
Classification	Public

Scope

Repository	Commit Hash
https://github.com/Kommunitas-net/staking-v3	f6bdf3df8ee71645e8863bc394bc18dfe57d4b6f

Re-Audit

Repository	Commit Hash
https://github.com/Kommunitas-net/staking-v3	ab2aadb8fc93a743df2e01a838e6ef32d8712be2

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1	Introduction	5
1.1	About Kommunitas	5
1.2	Approach & Methodology	5
1.2.1	Risk Methodology	6
2	Findings Overview	7
2.1	Summary	7
2.2	Key Findings	7
3	Finding Details	9
SHB.1	A User Can Get More Than A Single <code>KomV</code> Token	9
SHB.2	Any Worker Can Unstake to Any User Staker	11
SHB.3	Savior has Unrestricted Power to Withdraw Tokens with <code>emergencyWithdraw</code> Function	13
SHB.4	Owner Can Change Any Compound Type	15
SHB.5	Lack of Two-Factor Verification for Updating Admin Proxy Address	16
SHB.6	Centralization Risk	18
SHB.7	Owner Can Renounce Ownership	21
SHB.8	Race Condition	22
SHB.9	Missing Token Address Verification	24
SHB.10	Missing Value Verification	25
SHB.11	Missing Address Verification	28
SHB.12	Disynchronization between the <code>workerNumber</code> and the actual number of workers	31
4	Best Practices	33
BP.1	Using a Solidity Modifier to Encapsulate <code>onlySavior</code> Checks	33
BP.2	Optimizing Code Quality and Readability with Separate Pause/Unpause Functions	34
BP.3	Optimize Event Emission by Combining Functions	35
5	Tests	37
6	Conclusion	40

7	Scope Files	41
7.1	Audit	41
7.2	Re-Audit	41
8	Disclaimer	42

1 Introduction

Kommunitas engaged ShellBoxes to conduct a security assessment on the Kommunitas Staking V3 beginning on Jan 2nd, 2023 and ending Jan 7th, 2023. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Kommunitas

Kommunitas is a decentralized and tier-less Launchpad on Polygon. They are bridging the world to the biggest project in the most economical chain on cryptocurrency space. Kommunitas platform's goal is to allow project teams to focus on their project development and building their products, while the community handle the marketing, exposure and initial user base. They are looking for strong team with a unique and innovative vision in the cryptocurrency industry.

Issuer	Kommunitas
Website	https://kommunitas.net
Type	Solidity Smart Contract
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the *Kommunitas Staking V3* implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include , **3** high-severity, **4** medium-severity, **5** low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. A User Can Get More Than A Single <i>KomV</i> Token	HIGH	Fixed
SHB.2. Any Worker Can Unstake to Any User Staker	HIGH	Mitigated
SHB.3. Savior has Unrestricted Power to Withdraw Tokens with <i>emergencyWithdraw</i> Function	HIGH	Acknowledged
SHB.4. Owner Can Change Any Compound Type	MEDIUM	Fixed
SHB.5. Lack of Two-Factor Verification for Updating Admin Proxy Address	MEDIUM	Fixed
SHB.6. Centralization Risk	MEDIUM	Acknowledged
SHB.7. Owner Can Renounce Ownership	MEDIUM	Fixed
SHB.8. Race Condition	LOW	Fixed
SHB.9. Missing Token Address Verification	LOW	Fixed
SHB.10. Missing Value Verification	LOW	Fixed

SHB.11. Missing Address Verification	LOW	Fixed
SHB.12. Desynchronization between the <code>workerNumber</code> and the actual number of workers	LOW	Fixed

3 Finding Details

SHB.1 A User Can Get More Than A Single KomV Token

- Severity : **HIGH**
- Likelihood : 3
- Status : Fixed
- Impact : 2

Description:

The KomV token is an ERC20 which is minted to a staker if their staked amount has reached the minGetKomV value. It is used for voting, which means a user should not be able to vote more than once, this implies that the user should not have more than 1 token. However, the contract's logic allows a user to get more than a single voting token minted. This issue is caused by the contract checking the user's current balance, instead of verifying if they have already been minted a voting token.

Exploit Scenario:

1. The attacker stakes minGetKomV amount.
2. The `_stake` function checks if the attacker already has a KomV token by checking their balance.
3. Since the attacker's balance is 0, a KomV token is minted and given to the attacker.
4. The attacker transfers the KomV token to an account they own.
5. The attacker unstakes the whole amount.
6. The `_unstake` function checks if the attacker still has the KomV token by checking their balance.
7. Since the balance is 0 (because the KomV token was transferred to another account.)
8. The attacker repeats this process as many times as they want to obtain more KomV tokens.

Files Affected:

SHB.1.1: KommunitasStakingV3.sol

```
436 if(  
437     stakerDetail[_sender].stakedAmount >= minGetKomV &&  
438     IERC20MintableBurnableUpgradeable(komVToken).balanceOf(_sender) == 0  
439 ){  
440     IERC20MintableBurnableUpgradeable(komVToken).mint(_sender, 1);  
441 }
```

SHB.1.2: KommunitasStakingV3.sol

```
512 if(  
513     stakerDetail[_sender].stakedAmount < minGetKomV &&  
514     IERC20MintableBurnableUpgradeable(komVToken).balanceOf(_sender) > 0  
515 ){  
516     IERC20MintableBurnableUpgradeable(komVToken).burn(_sender, 1);  
517 }
```

Recommendation:

Consider using a mapping of users who own the **KomV** token so that a staker should be the only one owning and responsible for that **KomV** token. The mapping will look as follows : `mapping(address => bool) public hasKomV;`

Updates

The Kommunitas team resolved the issue by adding the **hasKomV** mapping to identify the users who already minted their **KomV** token.

SHB.1.3: KommunitasStakingV3.sol

```
444 if(  
445     stakerDetail[_sender].stakedAmount >= minGetKomV &&  
446     IERC20MintableBurnableUpgradeable(komVToken).balanceOf(_sender) ==  
         ↪ 0 &&  
447     !hasKomV[_sender]
```

```

448     ){
449         IERC20MintableBurnableUpgradeable(komVToken).mint(_sender, 1);
450         hasKomV[_sender] = true;
451     }

```

SHB.1.4: KommunitasStakingV3.sol

```

523     stakerDetail[_sender].stakedAmount < minGetKomV &&
524     IERC20MintableBurnableUpgradeable(komVToken).balanceOf(_sender) >
        ↪ 0 &&
525     hasKomV[_sender]
526 ){
527     IERC20MintableBurnableUpgradeable(komVToken).burn(_sender, 1);
528     hasKomV[_sender] = false;
529 }

```

SHB.2 Any Worker Can Unstake to Any User Staker

- Severity: **HIGH**
- Likelihood: 2
- Status: Mitigated
- Impact: 3

Description:

The contract's **unstake** function allows any worker to unstake tokens from any staker, regardless of whether the **worker** has a permission or authorization to do so. This can potentially allow a worker to unstake tokens from a staker without their consent or knowledge, potentially leading to loss of funds or other negative consequences.

Files Affected:

SHB.2.1: KommunitasStakingV3.sol

```

322 function unstake(

```

```

323     uint232 _userStakedIndex,
324     uint256 _amount,
325     address _staker
326 ) external virtual whenNotPaused {
327     // set staker
328     if(!isWorker[_msgSender()]) _staker = _msgSender();

```

Recommendation:

To ensure that only the caller is able to unstake their own tokens in the `unstake` function, you can modify the function as follows:

SHB.2.2: KommunitasStakingV3.sol

```

function unstake(uint232 _userStakedIndex, uint256 _amount) external
    ↪ virtual whenNotPaused {
    address _staker = _msgSender();
    .....
}

```

This ensures that only the caller is able to unstake their own tokens, rather than allowing a worker to unstake tokens on behalf of someone else.

Updates

The Kommunitas team mitigated the risk by preventing the worker from unstaking before the due date.

SHB.2.3: KommunitasStakingV3.sol

```

326     function unstake(
327         uint232 _userStakedIndex,
328         uint256 _amount,
329         address _staker
330     ) external virtual whenNotPaused {
331         // get stake data
332         Stake memory stakeDetail = staked[_staker][_userStakedIndex];
333

```

```

334     // worker check
335     if(isWorker[_msgSender()]){
336         require(block.timestamp > stakeDetail.endedAt, "premature");
337     } else {
338         _staker = _msgSender();
339     }

```

SHB.3 Savior has Unrestricted Power to Withdraw Tokens with `emergencyWithdraw` Function

- Severity: **HIGH**
- Likelihood: 2
- Status: Acknowledged
- Impact: 3

Description:

The `emergencyWithdraw` function in the contract allows the `savior` address to withdraw any amount of tokens from the contract, without any restrictions or limitations. This gives the `savior` address excessive power and could potentially be abused.

Exploit Scenario:

The `savior` address could potentially use the `emergencyWithdraw` function to withdraw a large amount of tokens from the contract, potentially causing financial harm to the contract or its users. Also, the savior can set his address as a `receiver` and will be able to get the tokens.

Files Affected:

SHB.3.1: `KommunitasStakingV3.sol`

```

690     function emergencyWithdraw(
691         address _token,

```

```

692     uint256 _amount,
693     address _receiver
694 ) external virtual {
695     onlySavior();
696
697     // adjust amount to wd
698     uint256 balance = IERC20Upgradeable(_token).balanceOf(address(this))
699         ↔ ;
700     if(_amount > balance) _amount = balance;
701
702     IERC20MintableBurnableUpgradeable(_token).safeTransfer(
703         _receiver,
704         _amount
705     );
706 }

```

Recommendation:

It is recommended to implement restrictions or limitations on the [emergencyWithdraw](#) function to prevent the savior address from having unrestricted power to withdraw tokens from the contract. This could include implementing a maximum withdrawal limit or requiring additional approvals or checks before allowing the savior address to withdraw tokens.

Updates

The Kommunitas team acknowledged the risk, stating that the [emergencyWithdraw](#) is a part of the business logic for safety, and they are utilizing a multisig contract as a savior, which needs 2-of-3 approvals to perform the transaction.

SHB.4 Owner Can Change Any Compound Type

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Fixed
- Impact: 3

Description:

The contract's `changeCompoundType` function allows the owner to change the compound type of any staked tokens, regardless of whether they have permission or authorization to do so. This can potentially allow the owner to change the compound type of user's staked tokens without their consent, potentially leading to unexpected or unintended consequences for the affected user.

Files Affected:

SHB.4.1: `KommunitasStakingV3.sol`

```
375     function changeCompoundType(  
376         address _staker,  
377         uint232 _userStakedIndex,  
378         CompoundTypes _newCompoundType  
379     ) external virtual whenNotPaused {  
380         // owner validation  
381         if(_msgSender() != owner()) _staker = _msgSender();
```

Recommendation:

Consider removing the owner's power to change the compound type for a staker and set the `_staker` variable to the caller's address (`msg.sender`). This ensures that only the caller is able to change the compound type for their stake, rather than the owner having the power to do so for any staker.

Updates

The Kommunitas team resolved the issue by preventing the owner from changing the compound type for the users.

SHB.5 Lack of Two-Factor Verification for Updating Admin Proxy Address

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Fixed
- Impact: 3

Description:

The `transferProxyAdmin` function, in the `AdminProxyManager` contract, allows the current `proxyAdmin()` to update the admin proxy address without any additional verification or authentication. This can lead to permanently giving the admin role to a wrong admin, which cannot be revoked again.

Exploit Scenario:

When this function is called with a mistaken address as parameter by the existing `adminProxy`, the `AdminProxyManager` privileges are immediately transferred to this unknown address. The original admin will lose the contract and will be unable to retrieve their control.

Files Affected:

SHB.5.1: AdminProxyManager.sol

```
27 function transferProxyAdmin(address _newProxyAdmin) external virtual
    ↪ proxied {
28     require(_newProxyAdmin != address(0) && _newProxyAdmin !=
        ↪ _proxyAdmin(), "bad");
```



```

29
30     assembly {
31         sstore(0
           ↪ xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b
32         5d6103, _newProxyAdmin)
33     }

```

Recommendation:

Consider adding an extra function to permit the newly assigned admin to claim `proxyAdmin` control. This will stop the issue of automatic assignment of a mistaken address. When an address is set as the admin using the `transferProxyAdmin` function, the contract can have an additional function. For instance, `updateProxyAdmin` will be called by the assigned person. The later becomes the new `ProxyAdmin` and then the former admin no longer has the proxy admin privileges.

Updates

The Kommunitas team resolved the issue by implementing two-factor verification, this was done by adding the `_pendingProxyAdmin` variable and the `acceptProxyAdmin` function that allows the new admins to claim their ownership.

SHB.5.2: AdminProxyManager.sol

```

39     function acceptProxyAdmin() external {
40         address sender = msg.sender;
41         require(_pendingProxyAdmin == msg.sender, "bad");
42         _transferProxyAdmin(sender);
43     }

```

SHB.6 Centralization Risk

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Acknowledged
- Impact: 3

Description:

The functions `changeCompoundType`, `setMin`, `setPeriodInDays`, `setPenaltyFee` and `setAPY` are functions that modify values related to each lock index. However, the owner is the only one able to change these values at any time, without the consent of the stakers.

Exploit Scenario:

1. An owner can set the `minStaking` value to a low value and set the `APY` to a high value to motivate users to lock their funds.
2. The owner has enough stakers. The owner immediately sets the `penaltyFee` to a really high value, and changes all the staker's `compoundTypes`, and the `APY` to 0. Hence, the stakers are unable to unstake their funds.

Files Affected:

SHB.6.1: KommunitasStakingV3.sol

```
381 function changeCompoundType(  
382     address _staker,  
383     uint232 _userStakedIndex,  
384     CompoundTypes _newCompoundType  
385 ) external virtual whenNotPaused {  
386     // owner validation  
387     if(_msgSender() != owner()) _staker = _msgSender();  
388  
389     // get stake data  
390     Stake memory stakeDetail = staked[_staker][_userStakedIndex];
```

```

391
392     require(
393         staked[_staker].length > _userStakedIndex && // user staked index
            ↪ validation
394         stakeDetail.compoundType != _newCompoundType, // compound type
            ↪ validation
395         "bad"
396     );
397
398     // assign new compound type
399     staked[_staker][_userStakedIndex].compoundType = _newCompoundType;
400 }

```

SHB.6.2: KommunitasStakingV3.sol

```

635 function setMin(
636     uint64 _minStaking,
637     uint64 _minPrivatePartner,
638     uint64 _minGetKomV,
639     uint16 _minLockIndexGetGiveaway
640 ) external virtual onlyOwner {
641     if(_minStaking > 0) minStaking = _minStaking;
642     if(_minPrivatePartner > 0){
643         minPrivatePartner = _minPrivatePartner;
644         privatePartnerStakedAmount = 0; // reset private partner total
            ↪ staked amount
645     }
646     if(_minGetKomV > 0) minGetKomV = _minGetKomV;
647     if(_minLockIndexGetGiveaway > 0){
648         minLockIndexGetGiveaway = _minLockIndexGetGiveaway;
649         giveawayStakedAmount = 0; // reset giveaway total staked amount
650     }
651 }

```

SHB.6.3: KommunitasStakingV3.sol

```

653 function setPeriodInDays(
654     uint16 _lockIndex,
655     uint128 _newLockPeriodInDays
656 ) external virtual onlyOwner {
657     require(lockNumber > _lockIndex, "bad");
658     lock[_lockIndex].lockPeriodInSeconds = _newLockPeriodInDays * 86400;
659 }

```

SHB.6.4: KommunitasStakingV3.sol

```

661 function setPenaltyFee(
662     uint16 _lockIndex,
663     uint64 _feeInPercent_d2
664 ) external virtual onlyOwner {
665     require(lockNumber > _lockIndex, "bad");
666     lock[_lockIndex].feeInPercent_d2 = _feeInPercent_d2;
667 }

```

SHB.6.5: KommunitasStakingV3.sol

```

669 function setAPY(
670     uint16 _lockIndex,
671     uint64 _apy_d2
672 ) external virtual onlyOwner {
673     require(lockNumber > _lockIndex, "bad");
674     lock[_lockIndex].apy_d2 = _apy_d2;
675 }

```

SHB.6.6: KommunitasStakingV3.sol

```

682 function togglePause() external onlyOwner virtual {
683     if(paused()){
684         _unpause();
685     } else {
686         _pause();
687     }
688 }

```

Recommendation:

Since these functions modify state variables related to the stakers, such changes should be proposed to the stakers, and the majority should either accept or deny these proposals.

Updates

The Kommunitas team acknowledged the risk, stating that they are using a multisig wallet, and they are planning to implement a governance system to enable stakers to vote on new proposals.

SHB.7 Owner Can Renounce Ownership

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Fixed
- Impact: 3

Description:

Typically, the account that deploys the contract is also its owner. Consequently, the owner is able to engage in certain privileged activities in his own name. In smart contracts, the `renounceOwnership` function is used to renounce ownership, which means that if the contract's ownership has never been transferred, it will never have an Owner, rendering some owner-exclusive functionality unavailable.

Files Affected:

SHB.7.1: KommunitasStakingV3.sol

```
15 contract KommunitasStakingV3 is
16     Initializable,
17     UUPSUpgradeable,
18     OwnableUpgradeable,
19     PausableUpgradeable,
20     AdminProxyManager,
```

Recommendation:

We recommend that you prevent the owner from calling `renounceOwnership` without first transferring ownership to a different address. Additionally, if you decide to use a multi-signature wallet, then the execution of the `renounceOwnership` will require for at least two or more users to be confirmed. Alternatively, you can disable Renounce Ownership functionality by overriding it.

Updates

The Kommunitas team resolved the issue by removing the `renounceOwnership` function from the `OwnableUpgradeable` contract.

SHB.8 Race Condition

- Severity: **LOW**
- Likelihood: 1
- Status: Fixed
- Impact: 2

Description:

The `setMin` function in the contract allows the owner to update the `minStaking` variable, which is used to validate the minimum amount required for staking in the `stake` function. However, the `setMin` function does not use any synchronization mechanism to prevent concurrent access, which could lead to a race condition. This could lead to unpredictable behavior in the `stake` function, since the `minStaking` variable may be updated concurrently with a `stake` transaction.

Files Affected:

SHB.8.1: KommunitasStakingV3.sol

```
635     function setMin(  
636         uint64 _minStaking,  
637         uint64 _minPrivatePartner,  
638         uint64 _minGetKomV,  
639         uint16 _minLockIndexGetGiveaway  
640     ) external virtual onlyOwner {  
641         if(_minStaking > 0) minStaking = _minStaking;  
642         if(_minPrivatePartner > 0){  
643             minPrivatePartner = _minPrivatePartner;  
644             privatePartnerStakedAmount = 0; // reset private partner total  
                ↪ staked amount  
645         }  
646         if(_minGetKomV > 0) minGetKomV = _minGetKomV;  
647         if(_minLockIndexGetGiveaway > 0){  
648             minLockIndexGetGiveaway = _minLockIndexGetGiveaway;  
649             giveawayStakedAmount = 0; // reset giveaway total staked amount  
650         }  
651     }
```

Recommendation:

To fix the race condition issue in the `setMin` function, you can use a synchronization mechanism such as a `require` statement to ensure that the `minStaking` variable is not updated concurrently with a stake transaction.

Updates

The Kommunitas team resolved the issue by requiring the contract to be paused before executing the `setMin` function.

SHB.8.2: KommunitasStakingV3.sol

```
660     function setMin(  
        
```

```

661     uint64 _minStaking,
662     uint64 _minPrivatePartner,
663     uint64 _minGetKomV,
664     uint16 _minLockIndexGetGiveaway
665 ) external virtual whenPaused onlyOwner {
666     if(_minStaking > 0) minStaking = _minStaking;
667     if(_minPrivatePartner > 0){
668         minPrivatePartner = _minPrivatePartner;
669         privatePartnerStakedAmount = 0; // reset private partner total
670         ↪ staked amount
671     }
672     if(_minGetKomV > 0) minGetKomV = _minGetKomV;
673     if(_minLockIndexGetGiveaway > 0){
674         minLockIndexGetGiveaway = _minLockIndexGetGiveaway;
675         giveawayStakedAmount = 0; // reset giveaway total staked amount
676     }
677     // unpause
678     _unpause();
679 }

```

SHB.9 Missing Token Address Verification

- Severity: **LOW**
- Likelihood: 1
- Status: Fixed
- Impact: 2

Description:

The contract's `init` function allows the setting of `komToken` and `komVToken` token addresses without verifying that the addresses are contract addresses. This can potentially allow an attacker to set these token addresses to non-contract addresses or `address(0)`.

Files Affected:

SHB.9.1: KommunitasStakingV3.sol

```
124     komToken = _komToken;  
125     komVToken = _komVToken;
```

Recommendation:

It is recommended to verify that the addresses being set as the `komToken` and `komVToken` token addresses are indeed contract addresses. This can be done by calling the `isContract` function on the addresses in question. This function is provided by the Ethereum Contract Address Validation library, which can be found here: [Address.sol](#)

Updates

The Kommunitas team resolved the issue by using the `isContract` function to make sure the `_komToken` and the `_komVToken` addresses refer to smart contracts.

SHB.9.2: KommunitasStakingV3.sol

```
119     require(  
120         _lockPeriodInDays.length == _apy_d2.length &&  
121         _lockPeriodInDays.length == _feeInPercent_d2.length &&  
122         AddressUpgradeable.isContract(_komToken) &&  
123         AddressUpgradeable.isContract(_komVToken) &&  
124         _savior != address(0),  
125         "mislength"  
126     );
```

SHB.10 Missing Value Verification

- Severity: **LOW**
- Status: Fixed
- Likelihood: 1
- Impact: 2

Description:

There are three functions `setPeriodInDays`, `setPenaltyFee`, and `setAPY` that are used to update state variables related to the staking process. These functions can only be accessed by the `owner`. However, there are no checks in place to ensure that the values of these state variables are not set to unreasonable values. For example, there is no check to prevent the `APY` from being set to 0 or the `penaltyFee` from being set to a large number.

Files Affected:

SHB.10.1: KommunitasStakingV3.sol

```
653 function setPeriodInDays(  
654     uint16 _lockIndex,  
655     uint128 _newLockPeriodInDays  
656 ) external virtual onlyOwner {  
657     require(lockNumber > _lockIndex, "bad");  
658     lock[_lockIndex].lockPeriodInSeconds = _newLockPeriodInDays * 86400;  
659 }
```

SHB.10.2: KommunitasStakingV3.sol

```
661 function setPenaltyFee(  
662     uint16 _lockIndex,  
663     uint64 _feeInPercent_d2  
664 ) external virtual onlyOwner {  
665     require(lockNumber > _lockIndex, "bad");  
666     lock[_lockIndex].feeInPercent_d2 = _feeInPercent_d2;  
667 }
```

SHB.10.3: KommunitasStakingV3.sol

```
669 function setAPY(  
670     uint16 _lockIndex,  
671     uint64 _apy_d2  
672 ) external virtual onlyOwner {  
673     require(lockNumber > _lockIndex, "bad");
```

```

674     lock[_lockIndex].apy_d2 = _apy_d2;
675 }

```

Recommendation:

For the `setPeriodInDays` function, it is recommended to use a list of predefined options (such as 30, 90, 120,..) rather than allowing the input of any number.

For the `setPenaltyFee` function, it is recommended to set a maximum limit for the penalty fee that cannot be exceeded. Similarly, for the `setAPY` function, it is recommended to set a minimum value that must be met.

Updates

The Kommunitas team mitigated the risk by verifying the arguments of the `setPeriodInDays` and the `setPenaltyFee`, and verifying the upper limit in the `setAPY` function.

SHB.10.4: KommunitasStakingV3.sol

```

681     function setPeriodInDays(
682         uint16 _lockIndex,
683         uint128 _newLockPeriodInDays
684     ) external virtual onlyOwner {
685         require(
686             lockNumber > _lockIndex &&
687             _newLockPeriodInDays >= 86400 &&
688             _newLockPeriodInDays <= (5 * yearInSeconds),
689             "bad"
690         );
691         lock[_lockIndex].lockPeriodInSeconds = _newLockPeriodInDays * 86400;
692     }

```

SHB.10.5: KommunitasStakingV3.sol

```

694     function setPenaltyFee(
695         uint16 _lockIndex,
696         uint64 _feeInPercent_d2
697     ) external virtual onlyOwner {

```

```

698     require(
699         lockNumber > _lockIndex &&
700         _feeInPercent_d2 >= 100 &&
701         _feeInPercent_d2 < 10000,
702         "bad"
703     );
704     lock[_lockIndex].feeInPercent_d2 = _feeInPercent_d2;
705 }

```

SHB.10.6: KommunitasStakingV3.sol

```

707     function setAPY(
708         uint16 _lockIndex,
709         uint64 _apy_d2
710     ) external virtual onlyOwner {
711         require(
712             lockNumber > _lockIndex &&
713             _apy_d2 < 10000,
714             "bad"
715         );
716         lock[_lockIndex].apy_d2 = _apy_d2;
717     }

```

SHB.11 Missing Address Verification

- Severity: **LOW**
- Likelihood: 1
- Status: Fixed
- Impact: 2

Description:

Certain functions lack a safety check in the address, the address-type arguments should include a zero-address test, otherwise, the contract's functionality may become inaccessible.

Files Affected:

SHB.11.1: KommunitasStakingV3.sol

```
677 function setSavior(address _savior) external virtual {
678     onlySavior();
679     savior = _savior;
680 }
```

SHB.11.2: KommunitasStakingV3.sol

```
105 function init(
106     address _komToken,
107     address _komVToken,
108     uint128[] calldata _lockPeriodInDays,
109     uint64[] calldata _apy_d2,
110     uint64[] calldata _feeInPercent_d2,
111     address _savior
112 ) external initializer proxied {
113     __UUPSUpgradeable_init();
114     __Pausable_init();
115     __Ownable_init();
116     __AdminProxyManager_init(_msgSender());
117
118     require(
119         _lockPeriodInDays.length == _apy_d2.length &&
120         _lockPeriodInDays.length == _feeInPercent_d2.length,
121         "misslength"
122     );
123
124     komToken = _komToken;
125     komVToken = _komVToken;
126     lockNumber = uint16(_lockPeriodInDays.length);
127     savior = _savior;
```

SHB.11.3: KommunitasStakingV3.sol

```
613 function addWorker(address _worker) external virtual onlyOwner {
614     isWorker[_worker] = true;
615     ++workerNumber;
616 }
```

SHB.11.4: KommunitasStakingV3.sol

```
618 function removeWorker(address _worker) external virtual onlyOwner {
619     isWorker[_worker] = false;
620     --workerNumber;
621 }
```

SHB.11.5: KommunitasStakingV3.sol

```
623 function changeWorker(
624     address _oldWorker,
625     address _newWorker
626 ) external virtual onlyOwner {
627     isWorker[_oldWorker] = false;
628     isWorker[_newWorker] = true;
629 }
```

SHB.11.6: KommunitasStakingV3.sol

```
631 function toggleTrustedForwarder(address _forwarder) external virtual
    ↪ onlyOwner {
632     isTrustedForwarder[_forwarder] = !isTrustedForwarder[_forwarder];
633 }
```

Recommendation:

We recommend that you make sure the addresses provided in the arguments are different from the `address(0)`.

Updates

The Kommunitas team resolved the issue by verifying all the address arguments to be different from the `address(0)`.

SHB.12 Disynchronization between the `workerNumber` and the actual number of workers

- Severity: **LOW**
- Likelihood: 1
- Status: Fixed
- Impact: 1

Description:

The `addWorker` and `removeWorker` functions, add the worker's address to a mapping and then increment or decrement the `workerNumber` variable to keep track of the number of workers currently active in the contract. However, these functions do not check if a worker already exists (in case of `addWorker`) or not (in case of `removeWorker`), and still increments the `workerNumber` which causes a mismatch between the `workerNumber` and the actual number of workers.

Files Affected:

SHB.12.1: KommunitasStakingV3.sol

```
615 function addWorker(address _worker) external virtual onlyOwner {  
616     isWorker[_worker] = true;  
617     ++workerNumber;  
618 }
```

SHB.12.2: KommunitasStakingV3.sol

```
620 function removeWorker(address _worker) external virtual onlyOwner {  
621     isWorker[_worker] = false;  
622     --workerNumber;
```

```
623 }
```

Recommendation:

Consider adding a `require` statement that checks that the entered address is not `address(0)` and that it does not already exist in the `addWorker`. This can be done as follows :

SHB.12.3: KommunitasStakingV3.sol

```
613 function addWorker(address _worker) external virtual onlyOwner {
614     require(_worker != address(0) && !isWorker[_worker], "worker already
        ↪ exists");
615     isWorker[_worker] = true;
616     ++workerNumber;
617 }
```

and for the `removeWorker` the code will look something like this :

SHB.12.4: KommunitasStakingV3.sol

```
618 function removeWorker(address _worker) external virtual onlyOwner {
619     require(isWorker[_worker], "worker does not exist");
620     isWorker[_worker] = false;
621     --workerNumber;
622 }
```

Updates

The Kommunitas team resolved the issue by preventing the addition of the `address(0)` and the modification of the `workerNumber` if the address already exists.

4 Best Practices

BP.1 Using a Solidity Modifier to Encapsulate `onlySavior` Checks

Description:

It is generally a good practice to use Solidity modifiers to encapsulate and reuse common checks or functionality in a contract. Modifiers allow you to define a set of conditions or requirements that must be met in order to execute the code in a function or method. In the case of the `onlySavior` function, it appears to be used to enforce that only the savior address is allowed to execute certain functions or methods in the contract. However, using a separate function to perform this check can be somewhat inefficient and can potentially lead to code duplication if the same check is needed in multiple functions.

1. Define a modifier named `onlySavior` that contains the check for the savior address. For example:

BP.1.1: `KommunitasStakingV3.sol`

```
modifier onlySavior() {  
    require(_msgSender() == savior, "!savior");  
    _;  
}
```

2. Apply the `onlySavior` modifier to any functions or methods that should only be accessible to the savior address. For example:

BP.1.2: `KommunitasStakingV3.sol`

```
function someFunction() public onlySavior {  
    // Function body  
}
```

BP.2 Optimizing Code Quality and Readability with Separate Pause/Unpause Functions

Description:

In the `togglePause` function, the pause/unpause logic is currently encapsulated in a single function. This can potentially make the code more difficult to read and understand, as the purpose and behavior of the function may not be immediately clear. One potential solution to improve the readability and clarity of this code is to separate the pause/unpause logic into two separate functions. For example:

BP.2.1: KommunitasStakingV3.sol

```
function pause() external onlyOwner virtual {
    _pause();
}

function unpause() external onlyOwner virtual {
    _unpause();
}
```

This approach allows you to clearly distinguish the pause and unpause functionality and make it more explicit in the contract code. It also allows you to give the functions descriptive names that reflect their purpose, which can make the code easier to understand and maintain.

Status - Not Fixed

BP.3 Optimize Event Emission by Combining Functions

Description:

The contract includes the `emitUnstaked` function that only emits an event and does not perform any other actions. This can potentially lead to unnecessary gas costs and code complexity. It is recommended to optimize the contract by removing the function that only emits an event, and adding the event emission directly to the core function that performs additional actions. This can help reduce gas costs and code complexity by reducing the number of function calls and events that are emitted.

Files Affected:

BP.3.1: KommunitasStakingV3.sol

```
568 function emitUnstaked(  
569     address _stakerAddress,  
570     uint128 _lockPeriodInDays,  
571     CompoundTypes _compoundType,  
572     uint256 _amount,  
573     uint256 _reward,  
574     uint256 _penaltyPremature,  
575     uint128 _stakedAt,  
576     uint128 _endedAt,  
577     bool _isPremature  
578 ) internal virtual {  
579     emit Unstaked(  
580         _stakerAddress,  
581         _lockPeriodInDays,  
582         _compoundType,  
583         _amount,  
584         _reward,
```

```
585     _penaltyPremature,  
586     _stakedAt,  
587     _endedAt,  
588     uint128(block.timestamp),  
589     _isPremature  
590 );  
591 }
```

Status - Not Fixed

5 Tests

→ [StakingV3](#) (27 passing (1m))

- ✓ Success: Stake 100 kom in no compounding type (415ms)
- ✓ Success: Unstake 100 kom in no compounding type (46ms)
- ✓ Success: Stake 100 kom in rewardOnly compounding type
- ✓ Success: Unstake 100 kom in rewardOnly compounding type (65ms)
- ✓ Success: Stake 100 kom in principalAndReward compounding type
- ✓ Success: Unstake 100 kom in principalAndReward compounding type (53ms)
- ✓ Success: Full premature unstake 100 kom in no compounding type (64ms)
- ✓ Success: Partial premature unstake 80 of 100 kom in no compounding type (68ms)
- ✓ Success: Full premature unstake 100 kom in rewardOnly compounding type (54ms)
- ✓ Success: Partial premature unstake 80 of 100 kom in rewardOnly compounding type (67ms)
- ✓ Success: Full premature unstake 100 kom in principalAndReward compounding type (54ms)
- ✓ Success: Partial premature unstake 80 of 100 kom in principalAndReward compounding type (67ms)
- ✓ Success: Stake 500k kom in no compounding type

- ✓ Success: Unstake 500k kom in no compounding type (59ms)
- ✓ Success: Stake 500k kom in rewardOnly compounding type
- ✓ Success: Unstake 500k kom in rewardOnly compounding type (65ms)
- ✓ Success: Stake 500k kom in principalAndReward compounding type
- ✓ Success: Unstake 500k kom in principalAndReward compounding type (62ms)
- ✓ Success: Full premature unstake 500k kom in no compounding type (64ms)
- ✓ Success: Partial premature unstake 600k of 800k kom in no compounding type (76ms)
- ✓ Success: Partial premature unstake 200k of 800k kom in no compounding type (76ms)
- ✓ Success: Full premature unstake 500k kom in rewardOnly compounding type (64ms)
- ✓ Success: Partial premature unstake 600k of 800k kom in rewardOnly compounding type (75ms)
- ✓ Success: Partial premature unstake 200k of 800k kom in rewardOnly compounding type (74ms)
- ✓ Success: Full premature unstake 500k kom in principalAndReward compounding type (54ms)
- ✓ Success: Partial premature unstake 600k of 800k kom in principalAndReward compounding type (74ms)

- ✓ Success: Partial premature unstake 200k of 800k kom in principalAndReward compounding type (73ms)

6 Conclusion

In this audit, we examined the design and implementation of Kommunitas Staking V3 contract and discovered several issues of varying severity. Kommunitas team addressed 9 issues raised in the initial report and implemented the necessary fixes, while classifying the rest as a risk with low-probability of occurrence. Shellboxes' auditors advised Kommunitas Team to maintain a high level of vigilance and to keep those findings in mind in order to avoid any future complications.

7 Scope Files

7.1 Audit

Files	MD5 Hash
contracts/KommunitasStakingV3.sol	a4a94d6910cce3457860ebac89e70fd0
contracts/interface/IERC20MintableBurnableUpgradeable.sol	9af652f839f640e7a7884ae356963f18
contracts/interface/IKommunitasStakingV3.sol	e6571d83f8da80268f88c72078f3f084
contracts/util/AdminProxyManager.sol	4b03425e63129be5e9c3c3744e760370
contracts/util/ERC1967.sol	51185e23ee344363c77388be75e0c0e1

7.2 Re-Audit

Files	MD5 Hash
contracts/KommunitasStakingV3.sol	fcdeac0b5c31867867ee004381db8d39
contracts/util/AdminProxyManager.sol	5faab490d0f406375288c3ed8d7068c5
contracts/util/OwnableUpgradeable.sol	919731340efa4bb64950ec96f37051e7
contracts/interface/IERC20MintableBurnableUpgradeable.sol	9af652f839f640e7a7884ae356963f18
contracts/interface/IKommunitasStakingV3.sol	e6571d83f8da80268f88c72078f3f084
contracts/util/ERC1967.sol	51185e23ee344363c77388be75e0c0e1

8 Disclaimer

Shellboxes reports should not be construed as "endorsements" or "disapprovals" of particular teams or projects. These reports do not reflect the economics or value of any "product" or "asset" produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology's proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don't offer any kind of investing advice and shouldn't be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.



For a Contract Audit, contact us at contact@shellboxes.com