Kommunitas Bridge

Smart Contract Security Audit

Prepared by ShellBoxes

June 24th, 2025 - June 27th, 2025

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	Kommunitas
Version	1.0
Classification	Public

Scope

Repository	Commit Hash
https://github.com/Kommunitas-net/core-contract	80d62b358371b9e1ef5a0769b426b7d06d929111

Re-Audit

Repository	Commit Hash
https://github.com/Kommunitas-net/core-contract	0c36201b5e0a75619566b096926e61cf2a06ea50

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

Kommunitas engaged ShellBoxes to conduct a security assessment on the Kommunitas Bridge beginning on June 24th, 2025 and ending June 27th, 2025. 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. Kommunitas is the solution for Multi Chain oriented projects. Kommunitas welcomes project from various blockchain like Polygon, BSC, Ethereum, Avalance, Solana, etc...

Issuer	Kommunitas
Website	`https://www.kommunitas.net`
Type	Solidity Smart Contract
Documentation	Kommunitas Docs
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 Bridge 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 , 1 high-severity, 5 medium-severity, 2 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Self-Call to `releaseKom` Locks Funds or Enables Mint-Without-Proof	HIGH	Fixed
SHB.2. Unchecked Token `burn` Return Value	MEDIUM	Acknowledged
SHB.3. Re-Entrancy in `bridge()` via Token Callback	MEDIUM	Fixed
SHB.4. Same-Chain Replay – Duplicate Mint	MEDIUM	Acknowledged
SHB.5. No Executors Initialised – Bridge Inoperable	MEDIUM	Acknowledged
SHB.6. Owner Can Swap Source / Destination Tokens	MEDIUM	Acknowledged
SHB.7. Unbounded `srcTxHash` Size Enables Gas Griefing	LOW	Fixed
SHB.8. Bridge Transfers 100 % of User Balance Only	LOW	Acknowledged

3 Finding Details

SHB.1 Self-Call to `releaseKom` Locks Funds or Enables Mint-Without-Proof

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

Description:

On the BNB Chain branch of `bridge()` (L 83–98) the contract calls `this.releaseKom(…)`.

- If `address(this)` is not whitelisted as an executor (default), the call reverts, burning the user's KOM but minting nothing on BSC — funds are permanently lost.
- If the owner does add `address(this)` as an executor to "fix" the revert, the bridge can later invoke `releaseKom` with arbitrary parameters and mint KOM on BSC without any cross-chain proof.

Files Affected:

SHB.1.1: KommunitasBridge.sol

```
83  } else if (srcChainID == 56) { // BSC branch
84      (success, ) = _srcToken.call(
85          abi.encodeWithSignature(
86              "burn(address,uint256)", address(this), komBalance
87          )
88      );
89      require(success, "!burnToken");
90
91      // self-call, requires contract to be executor
92      this.releaseKom(srcChainID, new bytes(0),
```

```
93                     komBalance, sender);
94 }
```

Recommendation:

Remove the self-call entirely and rely on the normal relayer path, or enforce `srcChainID != block.chainid` inside `releaseKom`. If same-chain minting is required, track a unique per-user nonce or burn-hash before minting.

Updates

The team resolved the issue by adding a new internal function named _releaseKom. When the bridge function is called from chain ID 56 (BSC), it will redirect the call to _releaseKom.

SHB.2 Unchecked Token `burn` Return Value

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

Description:

Both burn paths (L 107–131) perform a low-level call and only test `success`. A malicious or misconfigured KOM implementation can return `false`, keep the user's tokens, yet make the bridge believe the burn succeeded.

Files Affected:

SHB.2.1: KommunitasBridge.sol

```
108 (success, ) = _srcToken.call(
109     abi.encodeWithSignature("burn(uint256)", komBalance)
110 );
111 require(success, "!burnToken"); //  only checks low-level success
```

Recommendation:

Use a typed interface `IKommunitasToken(address).burn(komBalance)` and require the boolean return value, or switch to OpenZeppelin's `IERC20Burnable`.

Updates

The Kommunitas team acknowledged the issue since their token behaves differently in different chains.

SHB.3 Re-Entrancy in `bridge()` via Token Callback

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

Description:

`bridge()` is not `nonReentrant`, yet it externally invokes an upgradable KOM proxy before finishing state changes and emitting the event. A hostile burn implementation can reenter `bridge()` while the user's allowance is still in place and generate multiple `KomBridged` events or inconsistent internal accounting.

Files Affected:

SHB.3.1: KommunitasBridge.sol

```
100 IERC20Metadata(_srcToken).safeTransferFrom(
101     sender, address(this), komBalance
102 );
103 (success, ) = _srcToken.call(...); // external callback into KOM
```

Recommendation:

Add `nonReentrant` to `bridge()` and, if possible, move the burn after local state is finalised.

Updates

The Kommunitas team fixed the issue by adding the nonReentrant guard.

SHB.4 Same-Chain Replay – Duplicate Mint

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

Description:

`releaseKom` (L 149–166) skips the `_isBridgeExecuted` mapping when `_srcChainID == block.chainid`. On BSC this lets an authorised executor mint unlimited KOM simply by re-sending the call with an empty or new `_srcTxHash`.

Files Affected:

SHB.4.1: KommunitasBridge.sol

```
149 if (block.chainid != _srcChainID) {
150     require(!_isBridgeExecuted[_srcChainID][_srcTxHash], "executed");
151     _isBridgeExecuted[_srcChainID][_srcTxHash] = true;
152 }
```

Recommendation:

Always mark a transfer as executed: hash the tuple $(\_srcChainID, \_srcTxHash, \_amount, \_receiver)$ or maintain a per chain incremental nonce.

Updates

The Kommunitas team acknowledged this issue by stating that When bridging occurs on BSC, they do not require the txHash since the minting happens on the same smart contract at a single transaction (burn-mint). they also added a validation to ensure _srcChainID is not BSC.

SHB.5 No Executors Initialised – Bridge Inoperable

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

Description:

After `init` (L 64–96) the contract starts with `_executorNumber == 0`. Without at least one executor, `releaseKom` can never be called, permanently locking all cross-chain transfers.

Recommendation:

Inside `init` add an initial executor, e.g. the owner or a multisig:
`_isExecutor[sender] = true; _executorNumber = 1;`

Updates

The Kommunitas team acknowledged this issue and will ensure it is addressed before the bridge starts.

SHB.6 Owner Can Swap Source / Destination Tokens

- Severity : MEDIUM
- Status : Acknowledged
- Likelihood : 2
- Impact : 2

Description:

`setSrcToken` and `setDstToken` are unrestricted `onlyOwner` calls. A compromised or impatient owner can redirect burns to a fake token or mint unlimited KOM on the destination chain.

Files Affected:

SHB.6.1: KommunitasBridge.sol

```
119 function setDstToken(address dstToken_) external onlyOwner {
120     _dstToken = dstToken_;
121 }
```

Recommendation:

Emit explicit events, add a time-lock / multisig, and verify that the new token implements the expected interface & decimals.

Updates

The Kommunitas team acknowledged this issue and ensured the safety of the contract owner by using a multisignature wallet.

SHB.7 Unbounded `srcTxHash` Size Enables Gas Griefing

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

Description:

`srcTxHash` is an arbitrary-length `bytes`. An attacker may pass a 32 kB value, forcing the executor to pay >500 k gas for a single `SSTORE`.

Recommendation:

Cap the length:

```
require(_srcTxHash.length == 32, "bad tx-hash");
```

Updates

The Kommunitas team has resolved this issue by adding the verification on the length of the transaction hash.

SHB.8 Bridge Transfers 100 % of User Balance Only

- Severity: LOW
- Status: Acknowledged
- Likelihood: 1
- Impact: 1

Description:

`bridge()` always fetches `balanceOf(sender)`; users cannot choose an amount. Mistakes or custody wallets with mixed funds risk sending all KOM unintentionally.

Recommendation:

Add an `_amount` parameter, require $0 < \_amount \leq$ balanceOf(sender), and leave the remaining balance untouched.

Updates

The Kommunitas team acknowledged this issue and stated that this is a business decision to ensure that no KOM tokens circulate outside BSC in user wallets.

4 Best Practices

BP.1 Cache `_srcToken` Storage Pointer Once

Description:

Inside `bridge()` the contract dereferences `_srcToken` three times. Each read incurs an `SLOAD` (2100 gas). Caching the address in a local variable reduces the cost by two `SLOAD` about 400 gas per user transaction.

Files Affected:

BP.1.1: KommunitasBridge.sol

```
98  function bridge() external whenNotPaused {
99      address sender = _msgSender();
100     uint256 komBalance = IERC20Metadata(_srcToken).balanceOf(sender);
101
102     IERC20Metadata(_srcToken).safeTransferFrom( // 2nd SLOAD
103         sender, address(this), komBalance
104     );
105     ...
106 }
```

Recommendation:

```
1  function bridge() external whenNotPaused nonReentrant {
2      address sender = _msgSender();
3      IERC20Metadata src = IERC20Metadata(_srcToken); // cache once
4      uint256 komBalance = src.balanceOf(sender);
5
6      src.safeTransferFrom(sender, address(this), komBalance);
7      ...
8  }
```

Status - Fixed

The Kommunitas team has resolved the issue by caching the address in a local variable.

BP.2 Replace String Revert Messages with Custom Errors

Description:

Literal revert strings like `"!burnToken"` and `"!chainID"` are embedded in bytecode (64 gas/byte) and cost an extra 260 gas on each failure. Custom errors move the data to calldata and refund 1 800 gas per revert.

Files Affected:

BP.2.1: KommunitasBridge.sol

```
185 require(success, "!burnToken");
186 ...
187 else { revert("!chainID"); }
```

Recommendation:

```
1 error BurnFailed();
2 error UnsupportedChain();
3
4 if (!success) revert BurnFailed();
5 ...
6 else {
7     // no valid branch →abort
8     revert UnsupportedChain();
9 }
```

Status - Acknowledged

The team acknowledged the issue, as they want a clearer message for revert purposes in UX.

BP.3 Micro-Optimise `togglePause()` Branch

Description:

`togglePause()` contains two branches with identical cost; however a conditional operator combined with an `unchecked{}` block shaves 200 gas by avoiding one redundant `JUMPDEST`.

Files Affected:

BP.3.1: KommunitasBridge.sol

```
181 function togglePause() external onlyOwner {
182     if (paused()) {
183         _unpause();
184     } else {
185         _pause();
186     }
187 }
```

Recommendation:

```
1 function togglePause() external onlyOwner {
2     unchecked {
3         paused() ? _unpause() : _pause();
4     }
5 }
```

Status - Fixed

The team fixed the issue by using a conditional operator.

5 Conclusion

In this audit, we examined the design and implementation of Kommunitas Bridge 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.

6 Scope Files

6.1 Audit

Files	MD5 Hash
bridge/KommunitasBridge.sol	2845b59f8394aa2777c28f05081e25e3

6.2 Re-Audit

Files	MD5 Hash
bridge/KommunitasBridge.sol	79981b35919e6f9c20744daf28974813

7 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