



Lossless
Wrapped ERC20
SMART CONTRACT AUDIT
20.04.2023

Made in Germany by Chainsulting.de



Table of contents

1. Disclaimer.....	4
2. About the Project and Company	5
2.1 Project Overview.....	6
3. Vulnerability & Risk Level	7
4. Auditing Strategy and Techniques Applied.....	8
4.1 Methodology	8
5. Metrics	9
5.1 Tested Contract Files	9
5.2 Used Code from other Frameworks/Smart Contracts	10
5.3 CallGraph.....	11
5.4 Inheritance Graph	12
5.5 Source Lines & Risk.....	13
5.6 Capabilities	14
5.7 Source Unites in Scope	15
6. Scope of Work.....	16
6.1 Findings Overview	17
6.2 Manual and Automated Vulnerability Test.....	18
6.2.1 Overwriting Pending Withdrawal Requests.....	18
6.2.2 Missing Reentrancy Guard	19
6.2.3 Optimization of Blacklisted Funds Handling.....	21
6.2.4 Insecure Default Admin Role.....	24
6.2.5 Front-running.....	25



6.2.6 Missing Withdrawal Request Cancellation Function.....	27
6.2.7 Floating Pragma Set	30
6.2.8 Spelling Mistakes in Comments and Error Messages.....	31
6.3 SWC Attacks	32
6.4 Verify Claims	36
7. Executive Summary.....	37
8. About the Auditor	38



1. Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of Digital Assets Security Solutions OÜ. If you are not the intended receptor of this document, remember that any disclosure, copying or dissemination of it is forbidden.

Major Versions / Date	Description
0.1 (09.04.2022)	Layout
0.4 (10.04.2022)	Automated Security Testing Manual Security Testing
0.5 (11.04.2023)	Verify Claims and Test Deployment
0.6 (12.04.2023)	Testing SWC Checks
0.9 (13.04.2023)	Summary and Recommendation
1.0 (13.04.2023)	Final document
1.1 (20.04.2023)	Re-check 77b58cbc0603e3fd5776548f24ab72842768a9cd

2. About the Project and Company

Company address:

Digital Assets Security Solutions OÜ
Jõe 3-301
10151 Tallinn Estonia



Website: <https://lossless.io>

Twitter: <https://twitter.com/losslessdefi>

Telegram: https://t.me/lossless_io

Discord: <https://discord.com/invite/4CUqdMkvgN>

Medium: <https://losslessdefi.medium.com>

2.1 Project Overview

Lossless is a groundbreaking project in the DeFi (Decentralized Finance) ecosystem, which introduces the first hack mitigation tool for digital assets. The platform is designed to protect users' investments by implementing a sophisticated set of algorithms that detect and reverse fraudulent transactions in real-time, ultimately minimizing the impact of hacks and scams.

Key Features:

1. **Hack Detection and Mitigation:** Lossless leverages a proprietary algorithm to identify unauthorized transactions and suspicious activities. Upon detection, the protocol automatically freezes the affected assets and reverts the malicious transactions to secure users' funds.
2. **LSS Token:** Lossless utilizes its native utility token, LSS, which plays a vital role in the ecosystem. LSS token holders can participate in platform governance, propose updates to the protocol, and vote on essential decisions. Additionally, LSS tokens can be staked to earn rewards and gain access to premium features.
3. **Community-Driven Governance:** The Lossless platform is built on a decentralized governance model, empowering its users to have a say in the platform's future development and direction. This ensures that the ecosystem remains transparent, fair, and aligned with the interests of its users.
4. **Code Monitoring and Auditing:** The platform offers continuous code monitoring and third-party auditing services to ensure the highest security standards for its smart contracts. This further strengthens the protection of users' assets from vulnerabilities and exploits.
5. **Interoperability:** Lossless is designed to be compatible with multiple blockchain networks, allowing seamless integration and interaction with various DeFi protocols and applications.
6. **Open-Source Initiative:** Lossless maintains an open-source approach, encouraging developers and community members to contribute to the project. This fosters innovation and continuous improvement, making the platform more robust and secure over time.

In summary, Lossless is a pioneering project in the DeFi space, providing users with an innovative solution to mitigate the risks associated with hacks and malicious transactions. By combining advanced algorithms, community-driven governance, and robust security features, Lossless is poised to become an essential tool in the DeFi landscape, ensuring a safer and more secure environment for digital asset investments.

3. Vulnerability & Risk Level

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. Risk Level is computed based on CVSS version 3.0.

Level	Value	Vulnerability	Risk (Required Action)
Critical	9 – 10	A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.	Immediate action to reduce risk level.
High	7 – 8.9	A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.	Implementation of corrective actions as soon as possible.
Medium	4 – 6.9	A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.	Implementation of corrective actions in a certain period.
Low	2 – 3.9	A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.	Implementation of certain corrective actions or accepting the risk.
Informational	0 – 1.9	A vulnerability that have informational character but is not effecting any of the code.	An observation that does not determine a level of risk

4. Auditing Strategy and Techniques Applied

Throughout the review process, care was taken to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices. To do so, reviewed line-by-line by our team of expert pentesters and smart contract developers, documenting any issues as there were discovered.

4.1 Methodology

The auditing process follows a routine series of steps:

1. Code review that includes the following:
 - i. Review of the specifications, sources, and instructions provided to Chainsulting to make sure we understand the size, scope, and functionality of the smart contract.
 - ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
 - iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Chainsulting describe.
2. Testing and automated analysis that includes the following:
 - i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
 - ii. Symbolic execution, which is analysing a program to determine what inputs causes each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, actionable recommendations to help you take steps to secure your smart contracts.

5. Metrics

The metrics section should give the reader an overview on the size, quality, flows and capabilities of the codebase, without the knowledge to understand the actual code.

5.1 Tested Contract Files

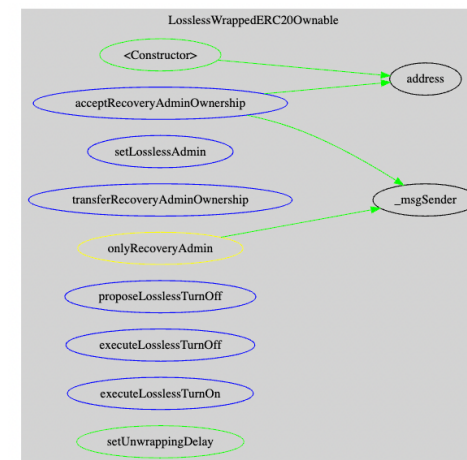
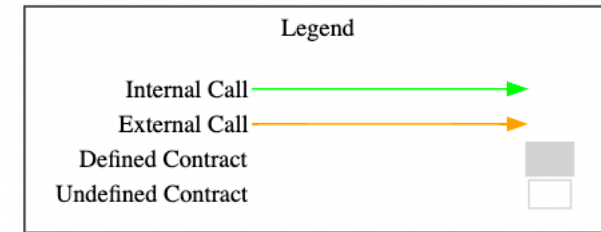
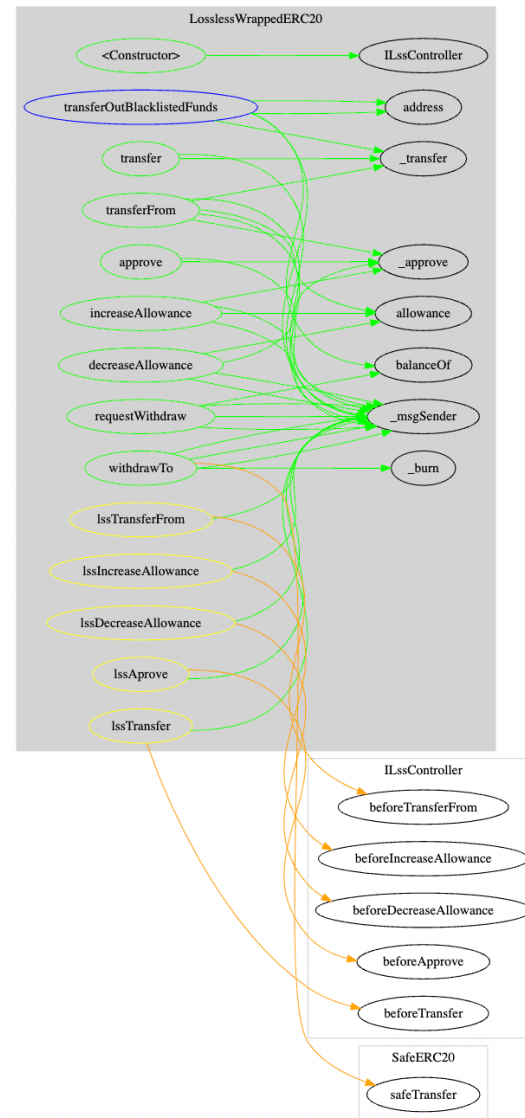
The following are the MD5 hashes of the reviewed files. A file with a different MD5 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different MD5 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review

File	Fingerprint (MD5)
./src/LosslessWrappedERC20.sol	6afb24b6df1f585152891d577c3ab1de
./src/LosslessWrappedERC20Ownable.sol	d1ea4e725fa79b51b513863700b7ad71
./src/Interfaces/ILosslessEvents.sol	f8ecbf55290f559ed2247c9ff7d21998

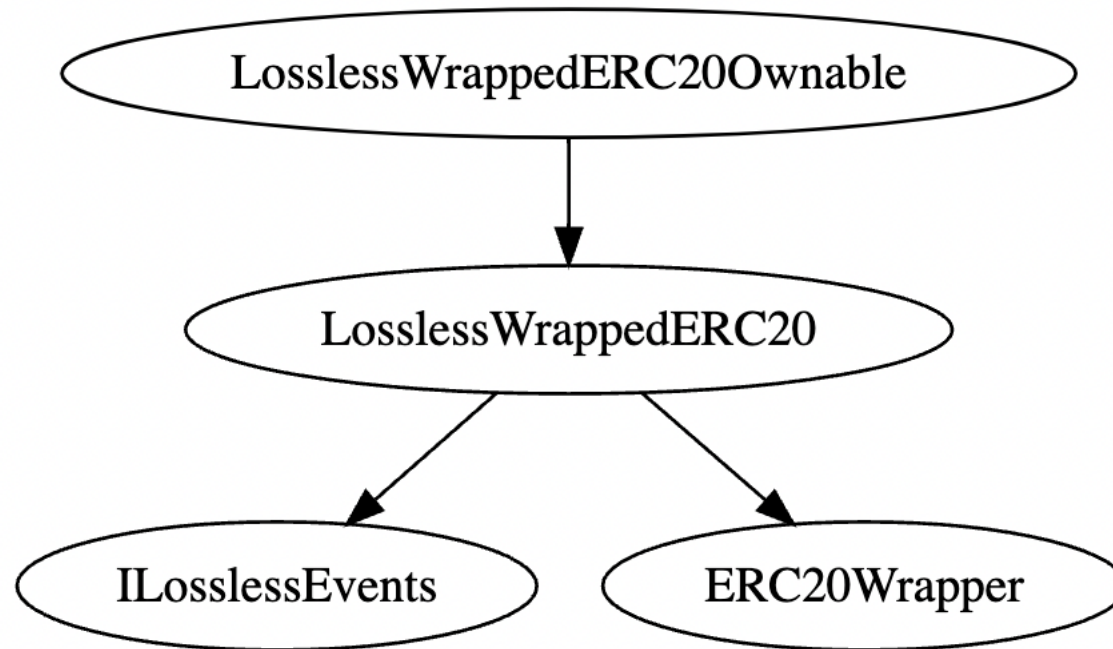
5.2 Used Code from other Frameworks/Smart Contracts (direct imports)

Dependency / Import Path	Source
@openzeppelin/contracts/token/ERC20/IERC20.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v4.8.0/contracts/token/ERC20/IERC20.sol
@openzeppelin/contracts/token/ERC20/extensions/ERC20Wrapper.sol	https://github.com/OpenZeppelin/openzeppelin-contracts/tree/v4.8.0/contracts/token/ERC20/extensions/ERC20Wrapper.sol
lossless-v3/Interfaces/ILosslessController.sol	https://github.com/Lossless-Cash/lossless-v3/tree/master/contracts/Interfaces/ILosslessController.sol

5.3 CallGraph

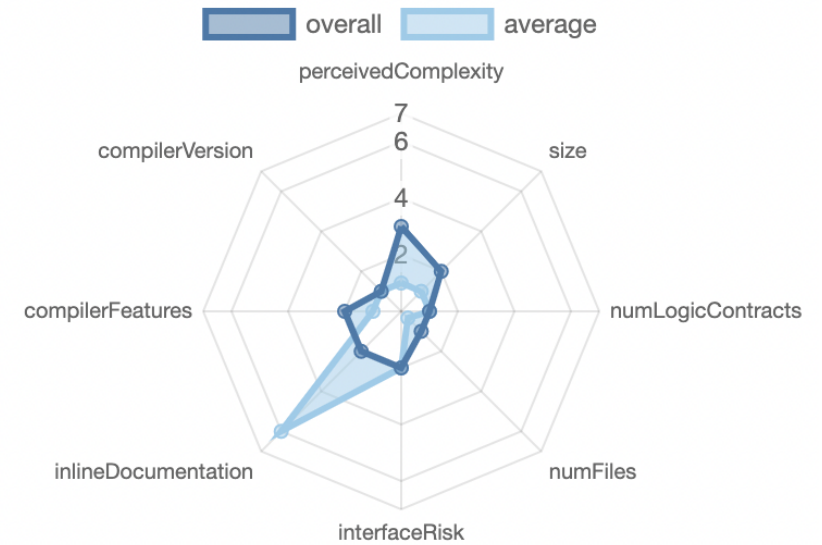
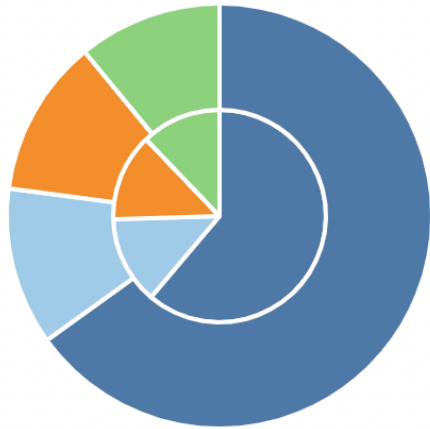


5.4 Inheritance Graph



5.5 Source Lines & Risk

source comment single block mixed
empty todo blockEmpty





5.6 Capabilities


Solidity Versions observed	 Experimental Features	 Can Receive Funds	 Uses Assembly	 Has Destroyable Contracts	
<div><div>^0.8.0</div><div>^0.8.4</div></div>		<div></div>	<div></div>	<div></div>	
 Transfers ETH	 Low-Level Calls	 DelegateCall	 Uses Hash Functions	 ECTrecover	 New/Create/Create2
<div></div>	<div></div>	<div></div>	<div>yes</div>	<div></div>	

Exposed Functions







This section lists functions that are explicitly declared public or payable. Please note that getter methods for public stateVars are not included.

 Public	 Payable				
15	0				
External	Internal	Private	Pure	View	
7	17	0	0	0	

StateVariables

Total	 Public
10	8

5.7 Source Unites in Scope

Type	File	Logic Contracts	Interfaces	Lines	nLines	nSLOC	Comment Lines	Complex. Score	Capabilities
	src/Interfaces/ILosslessEvents.sol	_____	1	18	18	16	1	1	_____
	src/LosslessWrappedERC20Ownable.sol	1	_____	127	124	91	21	51	
	src/LosslessWrappedERC20.sol	1	_____	258	215	145	33	93	Σ
	Totals	2	1	403	357	252	55	145	 Σ

- **Lines:** total lines of the source unit
- **nLines:** normalized lines of the source unit (e.g. normalizes functions spanning multiple lines)
- **nSLOC:** normalized source lines of code (only source-code lines; no comments, no blank lines)
- **Comment Lines:** lines containing single or block comments
- **Complexity Score:** a custom complexity score derived from code statements that are known to introduce code complexity (branches, loops, calls, external interfaces, ...)

6. Scope of Work

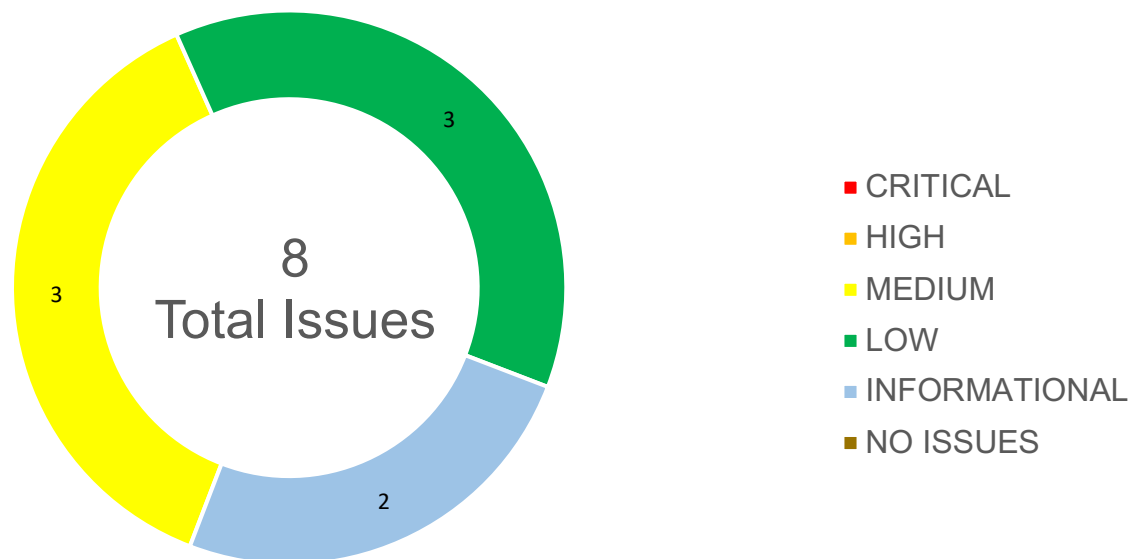
The Lossless Team provided us with the files that needs to be tested. The scope of the audit is the wrapped ERC20 contract.

The team put forward the following assumptions regarding the security, usage of the contracts:

- Ensure that the deposit, withdrawal, and wrap/unwrap of the underlying token is working correctly and securely.
- Verify that users can stake and unstake their wrapped tokens correctly, and the contract handles these actions securely and efficiently.
- Assess whether the fees collected during the deposit, withdrawal, and wrap/unwrap processes are accurately calculated and fairly distributed to the appropriate parties.
- Test the functionality of the pause and unpause mechanisms, ensuring that only authorized users can control these features and that they function as intended.
- Validate that the owner and manager roles have the proper authorization and control over the contract's functions, and that these roles cannot be abused or exploited.
- Analyze the contract for potential vulnerabilities, such as re-entrancy attacks, integer overflow/underflow, or any other issues that may compromise the security and functionality of the contract.
- Confirm that the contract adheres to the latest Ethereum standards and best practices for smart contract development. The smart contract is coded according to the newest standards and in a secure way.

The main goal of this audit was to verify these claims. The auditors can provide additional feedback on the code upon the client's request.

6.1 Findings Overview



No	Title	Severity	Status
6.2.1	Overwriting Pending Withdrawal Requests	MEDIUM	FIXED
6.2.2	Missing Reentrancy Guard	MEDIUM	FIXED
6.2.3	Optimization of Blacklisted Funds Handling	MEDIUM	ACKNOWLEDGED
6.2.4	Insecure Default Admin Role	LOW	ACKNOWLEDGED
6.2.5	Front-running	LOW	ACKNOWLEDGED
6.2.6	Missing Withdrawal Request Cancellation Function	LOW	FIXED
6.2.7	Floating Pragma Set	INFORMATIONAL	FIXED
6.2.8	Spelling Mistakes in Comments and Error Messages	INFORMATIONAL	FIXED

6.2 Manual and Automated Vulnerability Test

CRITICAL ISSUES

During the audit, Chainsulting's experts found **no Critical issues** in the code of the smart contract.

HIGH ISSUES

During the audit, Chainsulting's experts found **no High issues** in the code of the smart contract.

MEDIUM ISSUES

During the audit, Chainsulting's experts found **3 Medium issues** in the code of the smart contract

6.2.1 Overwriting Pending Withdrawal Requests

Severity: MEDIUM

Status: FIXED

Code: CWE-362

File(s) affected: LosslessWrappedERC20.sol

Update: [commit:66984d9a529c57a6ee5207f9b40bbac7d3cc14e9](https://github.com/Chainsulting/lossless-wrapped-erc20/commit/66984d9a529c57a6ee5207f9b40bbac7d3cc14e9)

Attack / Description	In the LosslessWrappedERC20 contract, when users request a withdrawal using the requestWithdraw function, there is no check for an existing pending withdrawal request. As a result, users can overwrite a previous request, potentially leading to unexpected behavior and loss of funds.
Code	Line 218 - 229 (LosslessWrappedERC20.sol) <pre>function requestWithdraw(uint256 amount) public { require(amount <= balanceOf(_msgSender()),</pre>

	<pre> "LSS: Request exceeds balance"); Unwrapping storage unwrapping = unwrappingRequests[_msgSender()]; unwrapping.unwrappingAmount = amount; unwrapping.unwrappingTimestamp = block.timestamp + unwrappingDelay; emit UnwrapRequested(_msgSender(), amount); } </pre>
Result/Recommendation	<p>Consider adding a check in the requestWithdraw function to prevent users from overwriting their existing withdrawal requests. Alternatively, you could implement a mechanism to handle multiple withdrawal requests for a single user, such as by using an array of pending withdrawal requests or by allowing users to add to an existing request. This will ensure that users' funds are handled safely and that pending withdrawal requests are not accidentally overwritten.</p>

6.2.2 Missing Reentrancy Guard

Severity: MEDIUM

Status: FIXED

Code: CWE-835

File(s) affected: LosslessWrappedERC20.sol

Update: [commit:82d8ea062a10594ed8f20e13b5466274d39a801b](https://github.com/0xSantitas/lossless-wrapped-erc20/commit/82d8ea062a10594ed8f20e13b5466274d39a801b)

Attack / Description	<p>The withdrawTo function in the LosslessWrappedERC20 contract is vulnerable to reentrancy attacks as it does not have any reentrancy protection mechanisms. An attacker could potentially exploit this vulnerability to withdraw more tokens than they should be able to.</p>
Code	<p>Line 234 – 257 (LosslessWrappedERC20.sol)</p>

	<pre> function withdrawTo(address to, uint256 amount) public override returns (bool) { Unwrapping storage unwrapping = unwrappingRequests[_msgSender()]; require(block.timestamp >= unwrapping.unwrappingTimestamp, "LSS: Unwrapping not ready yet"); require(amount <= unwrapping.unwrappingAmount, "LSS: Amount exceeds requested amount"); unwrapping.unwrappingAmount -= amount; _burn(_msgSender(), amount); SafeERC20.safeTransfer(underlying, to, amount); emit UnwrapCompleted(_msgSender(), to, amount); return true; } </pre>
Result/Recommendation	<p>To fix this issue, we recommend to use a reentrancy guard to prevent recursive calls. You can use the ReentrancyGuard provided by the OpenZeppelin Contracts library to implement this protection mechanism:</p> <p>Example:</p> <pre>import "@openzeppelin/contracts/security/ReentrancyGuard.sol";</pre>

	<pre> contract LosslessWrappedERC20 is ERC20Wrapper, ILosslessEvents, ReentrancyGuard { ... function withdrawTo(address to, uint256 amount) public override nonReentrant returns (bool) { ... _burn(_msgSender(), amount); SafeERC20.safeTransfer(underlying, to, amount); ... } } </pre>
--	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

6.2.3 Optimization of Blacklisted Funds Handling

Severity: MEDIUM

Status: ACKNOWLEDGED

Code: CWE-400

File(s) affected: LosslessWrappedERC20.sol

Update: As only a max of two blacklisted addresses would be processed at a time (originating from report(...) and secondReport(...)) it has been decided to avoid adding complex logic to this function.

Attack / Description	The LosslessWrappedERC20 contract has a function, transferOutBlacklistedFunds, that may become expensive to execute as the number of blacklisted addresses increases. This could lead to gas-related issues and potentially hinder the contract's functionality when dealing with a large number of blacklisted addresses.
Code	<p>Line 92 - 108 (LosslessWrappedERC20.sol)</p> <pre> function transferOutBlacklistedFunds(address[] calldata from) external { require(isLosslessOn, "LSS: Lossless not active"); require(</pre>

	<pre> _msgSender() == address(lossless), "LERC20: Only lossless contract"); uint256 fromLength = from.length; for (uint256 i = 0; i < fromLength;) { uint256 fromBalance = balanceOf(from[i]); _transfer(from[i], address(lossless), fromBalance); unchecked { i++; } } } </pre>
Result/Recommendation	<p>Optimize the transferOutBlacklistedFunds function to minimize gas costs and improve the contract's efficiency. Consider implementing a more efficient data structure or mechanism to handle blacklisted funds, such as a Merkle tree, to reduce the complexity of the function. Additionally, evaluate the possibility of breaking down the process of transferring blacklisted funds into smaller, more manageable transactions to prevent exceeding the block gas limit.</p> <p>Example:</p> <ol style="list-style-type: none"> 1. Modify the LosslessWrappedERC20 contract to include a new Merkle tree data structure: <pre> import "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol"; //... contract LosslessWrappedERC20 is ERC20Wrapper, ILosslessEvents { </pre>

```
//...

bytes32 public blacklistedRoot;

//...
}

2. Add a function to update the Merkle tree root:

function updateBlacklistedRoot(bytes32 newRoot) external onlyOwner {
    blacklistedRoot = newRoot;
    emit BlacklistedRootUpdated(newRoot);
}

event BlacklistedRootUpdated(bytes32 newRoot);

3. Replace the transferOutBlacklistedFunds function with a more efficient approach using the
Merkle tree:

function transferOutBlacklistedFunds(
    address[] calldata from,
    bytes32[] calldata proof
) external onlyOwner {
    bytes32 node = keccak256(abi
```

LOW ISSUES

During the audit, Chainsulting's experts found **3 Low issues** in the code of the smart contract

6.2.4 Insecure Default Admin Role

Severity: LOW

Status: ACKNOWLEDGED

Code: CWE-862

File(s) affected: LosslessWrappedERC20Ownable.sol

Update: The decision to use a multi-sig is left to the user, this is recommended multiple times in the documentation, as for smaller protocols might not be entirely feasible. For these reasons it was decided to not add the recommended logic.

Attack / Description	Many important functions can be only called by the admin, which could create a single point of failure if the admins private key is compromised.
Code	Line 54 (LosslessWrappedERC20Ownable.sol) <pre>function setLosslessAdmin(address newAdmin) external onlyRecoveryAdmin</pre>
Result/Recommendation	<p>Consider implementing a more secure multi-signature governance mechanism for administrative actions or make it mandatory to use multi-sig wallets such as Gnosis Safe. By doing so, you will distribute the power among multiple parties, reducing the risk of a single point of failure. OpenZeppelin's AccessControl contract can be used to set up a multisig admin role.</p> <p>Example:</p> <pre>import "@openzeppelin/contracts/access/AccessControl.sol"; contract LosslessController is AccessControl { bytes32 public constant MULTISIG_ADMIN_ROLE = keccak256("MULTISIG_ADMIN_ROLE"); constructor(address[] memory adminAddresses, uint256 requiredSignatures) { require(adminAddresses.length >= requiredSignatures, "Invalid parameters"); for (uint256 i = 0; i < adminAddresses.length; i++) {</pre>

	<pre> _setupRole(MULTISIG_ADMIN_ROLE, adminAddresses[i]); } } </pre>
--	----------------------------------------------------------------------------------

6.2.5 Front-running

Severity: LOW

Status: ACKNOWLEDGED

Code: CWE-453

File(s) affected: LosslessWrappedERC20.sol

Update: It has been decided to not include changes for this point as the logic of these contracts are not in scope of providing financial protection to the protocols. The users can decide to add it if needed.

Attack / Description	The LosslessWrappedERC20 contract may be vulnerable to front-running attacks, where attackers can observe pending withdrawal requests, specifically in the requestWithdraw function, and manipulate transactions before they are executed.
Code	<p>Line 218 - 229 (LosslessWrappedERC20.sol)</p> <pre> function requestWithdraw(uint256 amount) public { require(amount <= balanceOf(_msgSender()), "LSS: Request exceeds balance"); Unwrapping storage unwrapping = unwrappingRequests[_msgSender()]; unwrapping.unwrappingAmount = amount; unwrapping.unwrappingTimestamp = block.timestamp + unwrappingDelay; emit UnwrapRequested(_msgSender(), amount); } </pre>

Result/Recommendation	<p>Consider implementing a commit-reveal scheme to avoid revealing users' intent until the transaction is executed, thus minimizing the risk of front-running attacks.</p> <p>Example:</p> <ol style="list-style-type: none"> 1. Add a new mapping for storing commit hashes and a commit duration: <pre>mapping(address => bytes32) private commitHashes; uint256 public commitDuration;</pre> 2. Add a function to commit a withdrawal request: <pre>function commitWithdraw(uint256 amount, bytes32 secret) public { require(amount <= balanceOf(_msgSender()), "LSS: Commit exceeds balance"); bytes32 hash = keccak256(abi.encodePacked(_msgSender(), amount, secret)); commitHashes[_msgSender()] = hash; emit WithdrawCommitted(_msgSender(), hash); }</pre> <pre>event WithdrawCommitted(address indexed user, bytes32 commitHash);</pre> 3. Modify the requestWithdraw function to reveal the withdrawal request: <pre>function requestWithdraw(uint256 amount, bytes32 secret) public { bytes32 hash = keccak256(abi.encodePacked(_msgSender(), amount, secret));</pre>

	<pre> require(hash == commitHashes[_msgSender()], "LSS: Commit hash does not match"); // Clear the commit hash to prevent reuse commitHashes[_msgSender()] = bytes32(0); Unwrapping storage unwrapping = unwrappingRequests[_msgSender()]; unwrapping.unwrappingAmount = amount; unwrapping.unwrappingTimestamp = block.timestamp + unwrappingDelay; emit UnwrapRequested(_msgSender(), amount); } </pre> <p>By implementing this commit-reveal scheme, users will first commit their withdrawal request, hiding their intent. Then, they can reveal their request after the commit duration has passed, minimizing the risk of front-running attacks.</p>
--	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

6.2.6 Missing Withdrawal Request Cancellation Function

Severity: LOW

Status: FIXED

Code: CWE-710

File(s) affected: LosslessWrappedERC20.sol

Update: commit:ffe062ec7a772553d9979173853eb3ca2ccf502b

Attack / Description	In the LosslessWrappedERC20 contract, there is no function to cancel a withdrawal request. Users may want to cancel a withdrawal request in certain situations, such as if they made a mistake or changed their mind. The absence of a cancellation function may lead to inconvenience for the users.
Code	Line 234 - 257 (LosslessWrappedERC20.sol)

```

function withdrawTo(
    address to,
    uint256 amount
) public override returns (bool) {
    Unwrapping storage unwrapping = unwrappingRequests[_msgSender()];

    require(
        block.timestamp >= unwrapping.unwrappingTimestamp,
        "LSS: Unwrapping not ready yet"
    );

    require(
        amount <= unwrapping.unwrappingAmount,
        "LSS: Amount exceeds requested amount"
    );

    unwrapping.unwrappingAmount -= amount;
    _burn(_msgSender(), amount);
    SafeERC20.safeTransfer(underlying, to, amount);

    emit UnwrapCompleted(_msgSender(), to, amount);

    return true;
}

```

Result/Recommendation

Consider adding a function to allow users to cancel their pending withdrawal requests. This could be achieved by implementing a `cancelWithdrawRequest` function that resets the withdrawal request data for the user. By providing this functionality, you can improve the user experience and give users more control over their funds.

Example:

```

function cancelWithdrawRequest() public {
    Unwrapping storage unwrapping = unwrappingRequests[_msgSender()];

    // Check if there is an active withdrawal request
    require(
        unwrapping.unwrappingAmount > 0,
        "LSS: No active withdrawal request"
    );

    // Check if the withdrawal request is still pending
    require(
        unwrapping.unwrappingTimestamp > block.timestamp,
        "LSS: Withdrawal request already executable"
    );

    // Reset the withdrawal request data
    unwrapping.unwrappingAmount = 0;
    unwrapping.unwrappingTimestamp = 0;

    emit WithdrawRequestCanceled(_msgSender());
}

event WithdrawRequestCanceled(address indexed user);

```

This function checks if the user has an active withdrawal request and if the request is still pending (i.e., the unwrapping timestamp is in the future). If both conditions are met, it resets the withdrawal request data, effectively canceling the request. Finally, it emits an event to notify that the withdrawal request has been canceled. Remember to add the `WithdrawRequestCanceled` event to the contract:

	event WithdrawRequestCanceled(address indexed user);
--	------------------------------------------------------

INFORMATIONAL ISSUES

During the audit, Chainsulting's experts found **2 Informational issues** in the code of the smart contract

6.2.7 Floating Pragma Set

Severity: INFORMATIONAL

Status: FIXED

Code: SWC-103

File(s) affected: All

Update: [commit:2d1459aba585148b623eb50d5de376dd8a806437](https://github.com/Chainsulting/lossless-wrapped-erc20/commit/2d1459aba585148b623eb50d5de376dd8a806437)

Attack / Description	The current pragma Solidity directive is "^0.8.0". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.
Code	Line 2 (LosslessWrappedERC20.sol) <pre>pragma solidity ^0.8.0;</pre> Line 2 (LosslessWrappedERC20Ownable.sol) <pre>pragma solidity ^0.8.4;</pre>
Result/Recommendation	It is recommended to follow the latter example, as future compiler versions may handle certain language constructions in a way the developer did not foresee. i.e. Pragma solidity 0.8.0

	Reference: https://swcregistry.io/docs/SWC-103
--	--------------------------------------------------------------------------------------------------

6.2.8 Spelling Mistakes in Comments and Error Messages

Severity: INFORMATIONAL

Status: FIXED

Code: NA

File(s) affected: LosslessWrappedERC20.sol, LosslessWrappedERC20Ownable.sol

Update: [commit:b18aa9383e0d305339fba57865cd6765827156d](https://github.com/ChainSecurity/ChainSecurity/commit/b18aa9383e0d305339fba57865cd6765827156d)

Attack / Description	There are a few spelling mistakes found in the comments and error messages of the smart contract. These mistakes do not impact the functionality of the contract but could lead to confusion or misunderstanding when reading the contract. It is essential to have clear and accurate documentation for better readability and maintainability.
Code	<p>Line 73 (LosslessWrappedERC20Ownable.sol)</p> <pre>/// @notice This function is for accepting the revoery admin ownership transfer</pre> <p>Line 79 (LosslessWrappedERC20Ownable.sol)</p> <pre>"LERC20: Must be canditate"</pre> <p>Line 39 (LosslessWrappedERC20.sol)</p> <pre>modifier lssAprove(address spender, uint256 amount)</pre>
Result/Recommendation	<p>Correct the spelling mistakes in the comments and error messages to improve readability and maintainability. Here are the suggested corrections:</p> <pre>/// @notice This function is for accepting the recovery admin ownership transfer</pre> <pre>"LERC20: Must be candidate"</pre>

	modifier lssApprove(address spender, uint256 amount)
--	------------------------------------------------------

6.3 SWC Attacks

ID	Title	Relationships	Test Result
SWC-131	Presence of unused variables	CWE-1164: Irrelevant Code	✓
SWC-130	Right-To-Left-Override control character (U+202E)	CWE-451: User Interface (UI) Misrepresentation of Critical Information	✓
SWC-129	Typographical Error	CWE-480: Use of Incorrect Operator	✓
SWC-128	DoS With Block Gas Limit	CWE-400: Uncontrolled Resource Consumption	✓
SWC-127	Arbitrary Jump with Function Type Variable	CWE-695: Use of Low-Level Functionality	✓
SWC-125	Incorrect Inheritance Order	CWE-696: Incorrect Behavior Order	✓
SWC-124	Write to Arbitrary Storage Location	CWE-123: Write-what-where Condition	✓
SWC-123	Requirement Violation	CWE-573: Improper Following of Specification by Caller	✓

ID	Title	Relationships	Test Result
SWC-122	Lack of Proper Signature Verification	CWE-345: Insufficient Verification of Data Authenticity	✓
SWC-121	Missing Protection against Signature Replay Attacks	CWE-347: Improper Verification of Cryptographic Signature	✓
SWC-120	Weak Sources of Randomness from Chain Attributes	CWE-330: Use of Insufficiently Random Values	✓
SWC-119	Shadowing State Variables	CWE-710: Improper Adherence to Coding Standards	✓
SWC-118	Incorrect Constructor Name	CWE-665: Improper Initialization	✓
SWC-117	Signature Malleability	CWE-347: Improper Verification of Cryptographic Signature	✓
SWC-116	Timestamp Dependence	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	✓
SWC-115	Authorization through tx.origin	CWE-477: Use of Obsolete Function	✓
SWC-114	Transaction Order Dependence	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	✓

ID	Title	Relationships	Test Result
SWC-113	DoS with Failed Call	CWE-703: Improper Check or Handling of Exceptional Conditions	✓
SWC-112	Delegatecall to Untrusted Callee	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	✓
SWC-111	Use of Deprecated Solidity Functions	CWE-477: Use of Obsolete Function	✓
SWC-110	Assert Violation	CWE-670: Always-Incorrect Control Flow Implementation	✓
SWC-109	Uninitialized Storage Pointer	CWE-824: Access of Uninitialized Pointer	✓
SWC-108	State Variable Default Visibility	CWE-710: Improper Adherence to Coding Standards	✓
SWC-107	Reentrancy	CWE-841: Improper Enforcement of Behavioral Workflow	✗
SWC-106	Unprotected SELFDESTRUCT Instruction	CWE-284: Improper Access Control	✓
SWC-105	Unprotected Ether Withdrawal	CWE-284: Improper Access Control	✓
SWC-104	Unchecked Call Return Value	CWE-252: Unchecked Return Value	✓

ID	Title	Relationships	Test Result
SWC-103	Floating Pragma	CWE-664: Improper Control of a Resource Through its Lifetime	✗
SWC-102	Outdated Compiler Version	CWE-937: Using Components with Known Vulnerabilities	✓
SWC-101	Integer Overflow and Underflow	CWE-682: Incorrect Calculation	✓
SWC-100	Function Default Visibility	CWE-710: Improper Adherence to Coding Standards	✓

6.4 Verify Claims

- 6.4.1 Ensure that the deposit, withdrawal, and wrap/unwrap of the underlying token is working correctly and securely.
Status: tested and verified ✓
- 6.4.2 Verify that users can stake and unstake their wrapped tokens correctly, and the contract handles these actions securely and efficiently.
Status: tested and verified ✓
- 6.4.3 Ensure that the deposit, withdrawal, and wrap/unwrap of the underlying token is working correctly and securely.
Status: tested and verified ✓
- 6.4.4 Verify that users can stake and unstake their wrapped tokens correctly, and the contract handles these actions securely and efficiently.
Status: tested and verified ✓
- 6.4.5 Assess whether the fees collected during the deposit, withdrawal, and wrap/unwrap processes are accurately calculated and fairly distributed to the appropriate parties.
Status: tested and verified ✓
- 6.4.6 Test the functionality of the pause and unpause mechanisms, ensuring that only authorized users can control these features and that they function as intended.
Status: tested and verified ✓
- 6.4.7 Analyze the contract for potential vulnerabilities, such as reentrancy attacks, integer overflow/underflow, or any other issues that may compromise the security and functionality of the contract.
Status: tested and verified ✓
- 6.4.8 Confirm that the contract adheres to the latest Ethereum standards and best practices for smart contract development. The smart contract is coded according to the newest standards and in a secure way.
Status: tested and verified ✓

7. Executive Summary

Two (2) independent experts from Chainsulting have conducted a comprehensive audit of the LosslessWrappedERC20 and LosslessWrappedERC20Ownable smart contract codebase.

The primary objective of this audit was to validate the security and functionality claims of the smart contracts. Throughout the audit process, which included manual and automated security testing, no critical or high-severity issues were identified. However, three medium-severity, three low-severity, and two informational issues were discovered. These findings and their corresponding recommendations have been thoroughly documented in this report.

We strongly encourage the Lossless team to implement the suggested recommendations to further strengthen the security and enhance the readability of the smart contract codebase. By addressing these concerns, the team can ensure a more secure and transparent experience for users and other stakeholders interacting with the smart contracts.

Update (20.04.2023):

Following the comprehensive audit of the smart contract codebase, the identified issues have been addressed with satisfactory results. Out of the eight issues initially discovered, five have been fixed, and three have been acknowledged by the development team. The fixed issues include two medium-severity problems, one low-severity issue, and both informational concerns. The remaining acknowledged issues, consisting of one medium-severity and two low-severity findings, have been taken into consideration by the team. The successful resolution of these issues demonstrates the commitment to enhancing the security and functionality of the smart contract, ensuring a more robust and reliable system.

8. About the Auditor

Chainsulting, established in 2017 and headquartered in Germany, is a distinguished software development firm specializing in the Web3 domain. The company identifies pathways, opportunities, and risks while delivering all-encompassing Web3 solutions. Chainsulting's service offerings encompass Web3 development, security, and consulting.

Chainsulting performs code audits on prominent blockchains, including Solana, Tezos, Ethereum, Binance Smart Chain, and Polygon, in order to minimize risk and foster trust and transparency within the dynamic cryptocurrency community. Chainsulting has also assessed and secured smart contracts for numerous leading DeFi projects.

With over [\\$100 billion](#) in user funds safeguarded across various DeFi protocols, Chainsulting's team of experts leverages their extensive technical expertise in the Web3 sphere to provide bespoke smart contract audit solutions that cater to the ever-changing needs of their clientele.

Check our website for further information: <https://chainsulting.de>

How We Work



1 -----

PREPARATION

Supply our team with audit ready code and additional materials



2 -----

COMMUNICATION

We setup a real-time communication tool of your choice or communicate via e-mails.



3 -----

AUDIT

We conduct the audit, suggesting fixes to all vulnerabilities and help you to improve.



4 -----

FIXES

Your development team applies fixes while consulting with our auditors on their safety.



5 -----

REPORT

We check the applied fixes and deliver a full report on all steps done.