SECUREWEI

SECURITY AUDIT REPORT

Prepared by
SECUREWEI

Prepared for
CHAINSEAL

Don't trust, verify.

Table of Contents

Summary

About ChainSeal

This report documents the results of a comprehensive security audit of the ChainSeal smart contract. ChainSeal is a decentralized file authenticity verification platform built on the Polygon blockchain. The contract allows developers to publish cryptographic hashes of their software, enabling users to verify downloaded files without reliance on centralized servers. The audit evaluates the security, correctness, and overall design of the ChainSeal smart contract to ensure the safety and reliability of its operations.

Security Assessment

The security audit of the ChainSeal smart contract was conducted over a six-day period from May 21 to May 26, 2025. During this time, our team performed a structured evaluation of the contract using both manual review and automated tooling. Emphasis was placed on critical components such as access control, data immutability, blacklist enforcement, and ENS integration.

Over the course of the audit, we identified a total of nine security issues:

- 6 **Major** severity vulnerabilities
- 1 **Low** severity vulnerability
- 2 **Info** severity vulnerabilities

Scope

The audit focused exclusively on the ChainSeal smart contract located at https://polygonscan.com/address/0x03c4f7d5cf73559ae3db5f11bad068189c9c3723#code. Only the on-chain contract code was assessed; no frontend, server infrastructure, or off-chain components were in scope.
The following files were in scope:

- ChainSeal.sol

Type: Solidity

Platform: Polygon

Disclaimer

The findings and conclusions set forth in this audit report do not constitute, and should not be construed as, a warranty or guarantee of security. Our assessment reflects the professional judgment of our security experts based on the scope, resources, and documentation made available to us during the engagement. While we have employed industry-recognized methodologies and rigorous testing procedures to identify potential vulnerabilities, no audit can ever be exhaustive or completely eliminate all risks. Security is an ongoing process: new threats may emerge, environments may change, and additional weaknesses may arise after the completion of our work. Accordingly, this report should not be relied upon as a substitute for your own continuous security monitoring, due diligence, or additional reviews.

Severity Criteria

Severity	Description
High	Critical vulnerabilities where core functionalities of the protocol or funds are at significant risk.
Major	Vulnerabilities that may affect certain financial aspects or cause specific functionalities to malfunction.
Low	Issues without direct financial impact but could affect user experience or minor protocol operations.
Info	Suggestions for improvements such as gas efficiencies, adherence to best practices, or optimizations.
Likelihood	**Description**
High	The vulnerability can be exploited with minimal or no specific conditions.
Major	Exploitation requires certain conditions to be met or may not present sufficient incentives to be exploited.
Low	Exploitation demands rare conditions or incurs financial costs for the attacker.
N/A	Not applicable.

Findings Summary

Severity	Likelihood	Title
Major	Major	Unbounded Loops in *getByPublisher* Leads to Potential Gas Exhaustion
Major	Major	Unbounded Array Return in *get* function
Major	Major	Unbounded Loop in *getPublishingIndex* Risks Gas Exhaustion
Major	Major	Limited Control for Multiple Entries by Same Publisher Under Same Hash
Major	Low	Unbounded Array Growth in *blacklistedAddresses* Leading to *getBlacklist* DoS
Major	Low	Inconsistent State Management in ENS Registration
Low	Major	Incorrect sizing in *getByPublisher* leads to oversized return array
Info	N/A	Loop condition should use cached variable
Info	N/A	Duplicate Key Storage in *allKeys*

Detailed Findings

Unbounded Loops in *getByPublisher* Leads to Potential Gas Exhaustion

Severity: Major
Likelihood: Major
Target: ChainSeal.sol
Status: Acknowledged
ID: 1

Description

The ***getByPublisher*** function retrieves all software entries for a given publisher. It first iterates through all keys published by the user to count total entries, then allocates a memory array of that size, and iterates again to populate this array. If a publisher has published a large number of software entries, or if the keys they published are associated with many entries from other publishers (as ***records[key]*** stores entries from all publishers for that key), these loops can consume gas exceeding the block gas limit.

Impact

This can render ***getByPublisher*** unusable for prolific publishers or for publishers whose software hashes are also used by many others. Services or users relying on this function would be unable to fetch complete software lists, leading to a Denial of Service for this specific functionality.

Recommendation

Consider storing the ***Software*** structure instead of the key in ***publishedBy***, this way ***getByPublisher*** can simply return ***publishedBy[publisher]***.
On publish, push the new ***Software*** entry to both ***records[key]*** and ***publishedBy[msg.sender]***. Then ***getByPublisher*** becomes a single array read.

Unbounded Array Return in *get* function

Severity: Major
Likelihood: Major
Target: ChainSeal.sol
Status: Acknowledged
ID: 2

Description

The get function unconditionally reads and returns *records[key]*. There is no bound on the number of *Software* entries that can accumulate under any one hash. A malicious publisher can continuously call *publish* with the same hash to bloat the array. Subsequent calls to *get* must allocate, copy and return every entry in that array leading to excessive gas costs.

Impact

By spamming the contract with zero-value publishes under a single hash, an attacker can force all future *get* calls to consume excessive amounts of gas. Legitimate users or integrators will be unable to retrieve software records for that hash, effectively a denial-of-service.

Recommendation

On-chain dynamic arrays of unbounded length cannot be fully mitigated under EVM gas and memory constraints.
Consider rely on off-chain indexing instead of *get* for full records retrieval.

Unbounded Loop in *getPublishingIndex* Risks Gas Exhaustion

Severity: Major
Likelihood: Major
Target: ChainSeal.sol
Status: Acknowledged
ID: 3

Description

The ***getPublishingIndex*** function performs a linear search over ***records[key]*** to locate a publisher's entry index. If ***records[key]*** contains a very large number of entries (either because the software hash is extremely popular or an attacker spams publishes for the same hash) the loop can become prohibitively expensive. In the worst case it may run out of gas.

Impact

The owner might be unable to use this helper function to find an index for ***flagWarning*** or ***clearWarning*** if the hash is associated with too many entries, hindering their ability to manage warnings.

Recommendation

On ***publish***, consider storing the index along with the key (or the ***Software*** structure for the mitigation of [1]) in ***publishedBy***, this way ***getPublishingIndex*** can simply access ***publishedBy[publisher]*** and return the index information.

Limited Control for Multiple Entries by Same Publisher Under Same Hash

Severity: Major
Likelihood: Major
Target: ChainSeal.sol
Status: Acknowledged
ID: 4

Description

The ***getPublishingIndex*** function returns the index of the *first* software entry found in ***records[key]*** that matches the given publisher. If a publisher calls publish multiple times with the same hash string (e.g., for software with identical content but different name or version metadata), multiple Software structs will be stored for that publisher under the same key. The owner, relying on ***getPublishingIndex***, can only easily obtain the index for the first such entry.

Impact

The owner may find it difficult to use ***flagWarning*** or ***clearWarning*** for subsequent entries from the same publisher under the same hash, as ***getPublishingIndex*** won't provide their indices.

Recommendation

On ***publish***, consider storing the index along with the key (or the ***Software*** structure for the mitigation of [1]) in ***publishedBy***, this way ***getPublishingIndex*** can simply access ***publishedBy[publisher]*** and return the index information.

Unbounded Array Growth in *blacklistedAddresses* Leading to *getBlacklist* DoS

Severity: Major
Likelihood: Low
Target: ChainSeal.sol
Status: Acknowledged
ID: 5

Description

The ***blacklistAddress*** function appends addresses to the ***blacklistedAddresses*** array. However, ***unblacklistAddress*** only updates the *blacklist* mapping status and does not remove the address from the ***blacklistedAddresses*** array. Consequently, this array grows indefinitely with each unique address ever blacklisted. The ***getBlacklist*** function iterates over this entire array (effectively twice: once to count, once to populate results) to return currently blacklisted addresses.

Impact

If many addresses are blacklisted over time, even if subsequently un-blacklisted, the ***blacklistedAddresses*** array can become extremely large. This will cause ***getBlacklist*** to consume excessive gas, potentially exceeding the block gas limit and rendering it unusable.

Recommendation

Modify ***unblacklistAddress*** to remove the address from ***blacklistedAddresses***. This will reduce gas costs, though scalability will remain limited due to the iteration of the array.
Additionally, consider refactoring the ***getBlacklist*** function to return data for a specific blacklisted address, rather than the entire array, or simply let off-chain clients read the array and mapping directly and handle the filtering logic themselves.

Inconsistent State Management in ENS Registration

Severity: Major
Likelihood: Low
Target: ChainSeal.sol
Status: Acknowledged
ID: 6

Description

The ***registerENS*** function allows the owner to associate an ENS-like name with an address and a verification status. However, it does not correctly handle cleanup of old mappings when a name is reassigned to a new address or an address is assigned a new name.

- If ***name1*** is registered to ***addrA***, then ***name1*** is registered to ***addrB***: ***ensNames[name1]*** will point to ***addrB***, and ***ensAddresses[addrB]*** will point to ***name1***. However, ***ensAddresses[addrA]*** will still (incorrectly) point to ***name1***.
- If ***addr1*** is registered with ***nameA***, then ***addr1*** is registered with ***nameB***: ***ensAddresses[addr1]*** will point to ***nameB***, and ***ensNames[nameB]*** will point to ***addr1***. However, ***ensNames[nameA]*** will still (incorrectly) point to ***addr1***.

Impact

These inconsistencies lead to a broken bi-directional link between names and addresses. Queries like ***getENSAddress(name1)*** might return ***addrB***, while ***getENSName(addrA)*** might return ***name1***. This can provide misleading information to users or dApps relying on the ENS feature.

Recommendation

Modify ***registerENS*** to explicitly clear or update stale reverse mappings when a name is reassigned, and stale forward mappings when an address is reassigned a new name.

Incorrect sizing in *getByPublisher* leads to oversized return array

Severity: Low
Likelihood: Major
Target: ChainSeal.sol
Status: Acknowledged
ID: 7

Description

In ***getByPublisher*** the contract first computes ***records[keys[i]].length*** but it is the total number of submissions under that hash by all publishers, not just the target publisher. As a result:

- ***count*** can be much larger than the actual number of records published by ***publisher***.
- The returned result array will contain trailing “empty” entries for slots where ***entries[j].publisher != publisher***.

Impact

- Clients relying on ***getByPublisher*** will receive arrays padded with meaningless zeroed elements.
- Excessive gas consumption.

Recommendation

Before you allocate the result array, count only those entries whose ***publisher*** field equals the target address.
Alternatively, let indexing off-chain rather than returning arbitrarily large arrays in a single call.

Loop condition should use cached variable

Severity: Info
Likelihood: N/A
Target: ChainSeal.sol
Status: Acknowledged
ID: 8

Description

The loop conditions at line 120 and line 130 reference ***blacklistedAddresses.length*** directly within the loop condition. Accessing the length member of a storage array repeatedly incurs unnecessary gas costs.

Cache the array length in a local uint variable before the loop begins and use that cached value in the loop condition. This will reduce repeated storage reads and optimize gas usage.

Duplicate Key Storage in *allKeys*

Severity: Info
Likelihood: N/A
Target: ChainSeal.sol
Status: Acknowledged
ID: 9

Description

The ***publish*** function allows users to submit software metadata identified by a hash regardless of whether the same hash was published before, this key is then appended to the ***allKeys*** array.
This results in multiple identical keys being stored in ***allKeys*** if the same hash is published more than once. The contract does not check for duplicates before appending to ***allKeys***.

Methodology

The audit was conducted using a comprehensive and multi-layered approach to identify security vulnerabilities, logical flaws, and deviations from best practices within the smart contract codebase. The following methodologies were employed:

Manual Code Review

An extensive manual analysis was performed on the smart contract source code. This involved a line-by-line review to assess logic correctness, access control, input validation, gas optimization, and adherence to industry standards. Manual auditing was also used to identify subtle logic bugs, business logic flaws, and edge cases that automated tools may overlook.

Automated Static Analysis

We employed a suite of state-of-the-art static analysis tools to detect known vulnerability patterns and unsafe programming constructs. These tools scanned the contract code for issues.
Findings from static analysis were carefully reviewed and validated to eliminate false positives and ensure contextual relevance.

Fuzz Testing

Fuzzing techniques were applied to the smart contracts to simulate a wide range of randomized inputs and edge cases. This dynamic analysis aimed to uncover unexpected behavior, such as assertion failures, state inconsistencies, or crashes that may occur under irregular transaction sequences or malicious inputs.

Threat Modeling and Risk Assessment

We analyzed the smart contracts within the broader context of their intended ecosystem and business logic to identify potential threat vectors. Scenarios such as front-running, oracle manipulation, and privilege escalation were considered to assess the system's robustness against known attack vectors.

About SecureWei

SecureWei is a blockchain security firm dedicated to safeguarding decentralized systems through rigorous smart contract auditing and protocol assessments. Our mission is to empower developers and projects with the confidence that their code is secure, reliable, and aligned with industry best practices.
At SecureWei, we combine deep domain expertise with cutting-edge analysis techniques to uncover hidden vulnerabilities and design flaws before they can be exploited. Our team brings together professionals with backgrounds in cybersecurity, formal verification, and decentralized application development, enabling us to deliver high-impact, actionable security insights.

Why Choose SecureWei?

- Thoroughness: Every line of code is reviewed by experienced auditors to uncover subtle bugs and complex attack vectors.
- Trust: Our reputation is built on integrity, transparency, and consistently high standards.
- Collaboration: We provide detailed reports and work hand-in-hand with teams to remediate issues effectively.
- Future-Focused: We stay ahead of the curve with evolving blockchain standards and security trends.

Contact

To learn more about our services or to request an audit, feel free to reach out:

- Twitter (X): @securewei
- Telegram: @securewei
- Website: securewei.com

SecureWei is committed to raising the security bar in Web3.