F Y E O

Security Code Review of the Natgold Frontend

Natgold

February 2026
Version 1.0

Presented by:
FYEO Inc.
PO Box 147044
Lakewood CO 80214
United States

Security Level
Strictly Confidential

Table of Contents

Executive Summary

Overview

Natgold engaged FYEO Inc. to perform a Security Code Review of the Natgold Frontend.

The assessment was conducted remotely by the FYEO Security Team. Testing took place on January 26 - January 30, 2026, and focused on the following objectives:

- To provide the customer with an assessment of their overall security posture and any risks that were discovered within the environment during the engagement.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
- To identify potential issues and include improvement recommendations based on the results of our tests.

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the FYEO Security Team took to identify and validate each issue, as well as any applicable recommendations for remediation.

Key Findings

The following issues have been identified during the testing period. These should be prioritized for remediation to reduce the risk they pose:

- FYEO-NATF-01 – Missing Implementation Address Verification in Upgrade Flow
- FYEO-NATF-02 – Unpinned Frontend Dependencies Enable Supply Chain Attacks
- FYEO-NATF-03 – Missing Content Security Policy (CSP) Settings
- FYEO-NATF-04 – Missing noopener noreferrer on External Links
- FYEO-NATF-05 – Sensitive Details Exposed in Environment Configuration
- FYEO-NATF-06 – Missing Frontend Input Validation for Custodian Wallets

Based on our review process, we conclude that the reviewed code implements the documented functionality.

Scope and Rules of Engagement

The FYEO Review Team performed a Security Code Review of the Natgold Frontend. The following table documents the targets in scope for the engagement. No additional systems or resources were in scope for this assessment.

The source code was supplied through a private repository at https://github.com/fpco/natgold with the commit hash 1093ca36b547a6b28a8afedcabc48a83ef368440.

Remediations were submitted with the commit hash 66822660ea27b1f1dde1f45040be68ba9766659e.

Files included in the code review

```
natgold/
├── contract/
│   ├── script/
│   │   ├── helpers/
│   │   │   └── proposeSafeTx.js
│   │   ├── Deploy.s.sol
│   │   ├── ProposeUpgrade.s.sol
│   │   └── Upgrade.s.sol
│   └── src/
│       ├── NatGoldQueueOrchestrator.sol
│       └── NatGoldToken.sol
├── frontend/
│   ├── src/
│   │   ├── api/
│   │   │   ├── mutations/
│   │   │   │   ├── approveTx.ts
│   │   │   │   ├── createProject.ts
│   │   │   │   ├── executeTx.ts
│   │   │   │   ├── mintProject.ts
│   │   │   │   ├── rejectProject.ts
│   │   │   │   └── rejectTx.ts
│   │   │   └── queries/
│   │   │       ├── contractInfo.ts
│   │   │       ├── mintedProjects.ts
│   │   │       ├── pendingMints.ts
│   │   │       ├── pendingProjects.ts
│   │   │       ├── pendingRejections.ts
│   │   │       ├── pendingTxs.ts
│   │   │       ├── pendingUpgrades.ts
│   │   │       ├── queuedProjects.ts
│   │   │       ├── safe.ts
│   │   │       ├── signer.ts
```

Files included in the code review

```
│   │   │       ├── stats.ts
│   │   │       ├── txApprovers.ts
│   │   │       └── txRejections.ts
│   │   ├── utils/
│   │   │   ├── api.ts
│   │   │   ├── contract.ts
│   │   │   ├── environment.ts
│   │   │   ├── error.ts
│   │   │   ├── interaction.ts
│   │   │   ├── number.ts
│   │   │   ├── string.ts
│   │   │   ├── styles.ts
│   │   │   ├── txs.ts
│   │   │   ├── types.ts
│   │   │   └── version.ts
│   │   ├── routeTree.gen.ts
│   │   └── vite-env.d.ts
│   └── vite.config.ts
└── test-cli/
    ├── config.js
    ├── contracts.js
    ├── index.js
    ├── safe-service.js
    ├── services.js
    ├── test-config.js
    ├── test-propose.js
    └── test-safe-api.js
```

Table 1: Scope

Technical Analyses and Findings

During the Security Code Review of the Natgold Frontend, we discovered:

- 1 finding with HIGH severity rating.
- 1 finding with MEDIUM severity rating.
- 3 findings with LOW severity rating.
- 1 finding with INFORMATIONAL severity rating.

The following chart displays the findings by severity.

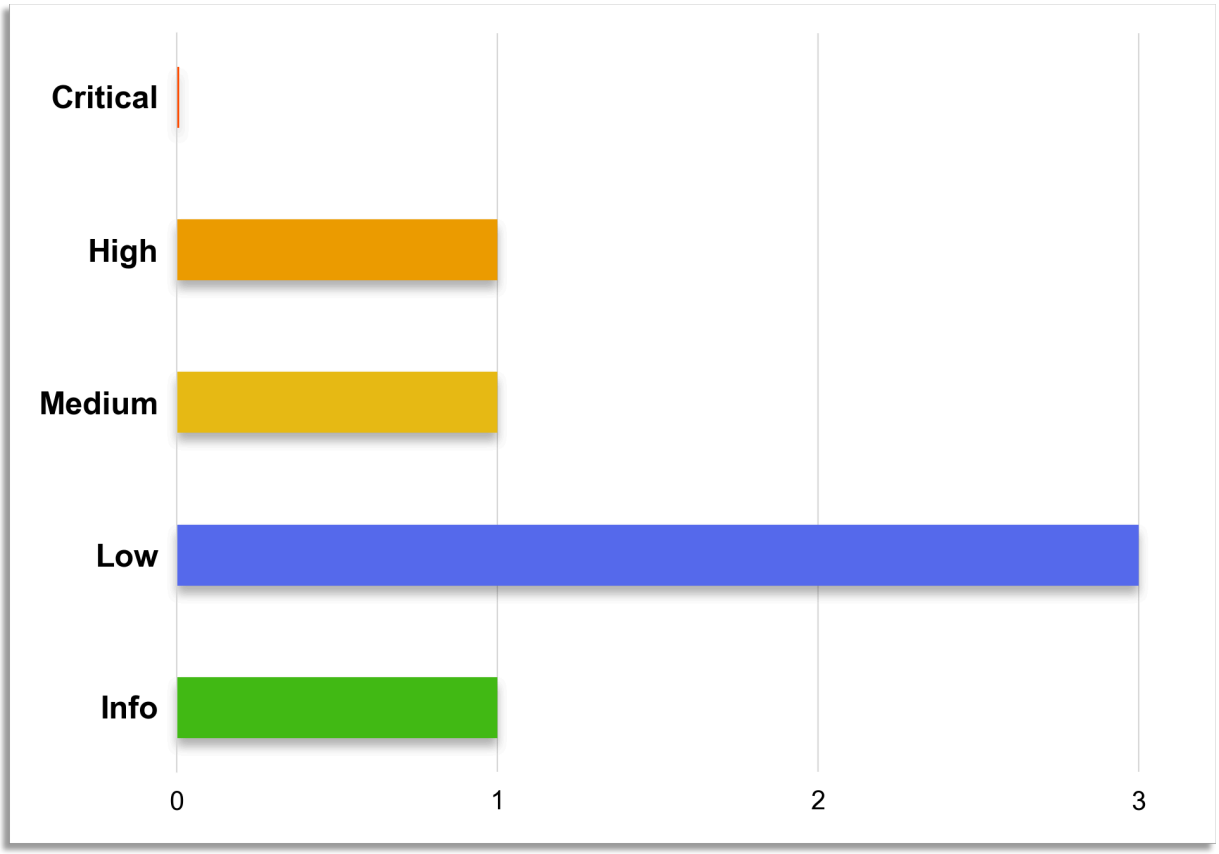

Figure 1: Findings by Severity

Findings

The *Findings* section provides detailed information on each of the findings, including methods of discovery, explanation of severity determination, recommendations, and applicable references.

The following table provides an overview of the findings.

Finding #	Severity	Description	Status
FYEO-NATF-01	High	Missing Implementation Address Verification in Upgrade Flow	Remediated
FYEO-NATF-02	Medium	Unpinned Frontend Dependencies Enable Supply Chain Attacks	Remediated
FYEO-NATF-03	Low	Missing Content Security Policy (CSP) Settings	Acknowledged
FYEO-NATF-04	Low	Missing noopener noreferrer on External Links	Remediated
FYEO-NATF-05	Low	Sensitive Details Exposed in Environment Configuration	Remediated
FYEO-NATF-06	Informational	Missing Frontend Input Validation for Custodian Wallets	Remediated

Table 2: Findings Overview

The Classification of vulnerabilities

Security vulnerabilities and areas for improvement are weighted into one of several categories using, but is not limited to, the criteria listed below:

Critical – vulnerability will lead to a loss of protected assets

- This is a vulnerability that would lead to immediate loss of protected assets
- The complexity to exploit is low
- The probability of exploit is high

High - vulnerability has potential to lead to a loss of protected assets

- All discrepancies found where there is a security claim made in the documentation that cannot be found in the code
- All mismatches from the stated and actual functionality
- Unprotected key material
- Weak encryption of keys
- Badly generated key materials
- Txn signatures not verified
- Spending of funds through logic errors
- Calculation errors overflows and underflows

Medium - vulnerability hampers the uptime of the system or can lead to other problems

- Insecure calls to third party libraries
- Use of untested or nonstandard or non-peer-reviewed crypto functions
- Program crashes, leaves core dumps or writes sensitive data to log files

Low – vulnerability has a security impact but does not directly affect the protected assets

- Overly complex functions
- Unchecked return values from 3rd party libraries that could alter the execution flow

Informational

- General recommendations

Technical Analysis

The source code has been manually validated to the extent that the state of the repository allowed. The validation includes confirming that the code correctly implements the intended functionality.

Conclusion

Based on our review process, we conclude that the code implements the documented functionality to the extent of the reviewed code.

Technical Findings

General Observations

NatGold is a blockchain-based system designed to tokenize gold-backed mining projects on the Base blockchain. It converts approved mining projects into ERC-20 tokens through a structured smart contract process. The system uses two core contracts: `NatGoldToken.sol`, which manages token minting and distribution, and `NatGoldQueueOrchestrator.sol`, which manages project intake, approval, and lifecycle state. Projects move through a defined state machine (queued, minted, or rejected) and are processed using a First-In-First-Out queue to ensure deterministic ordering. Role-based access control limits administrative actions such as approvals and minting to authorized roles, and all project state changes are recorded on-chain.

Administrative operations are governed through a Gnosis Safe multisignature setup, requiring multiple approvals for actions such as project acceptance, mint execution, and contract upgrades. The system supports percentage-based token distribution across multiple wallets to reflect ownership structures. Both core contracts use the UUPS upgradeable proxy pattern to allow contract logic updates while preserving state. A React and TypeScript frontend provides functionality for project submission, queue visibility, and transaction tracking. Overall, the system provides a structured and auditable mechanism for managing the lifecycle and tokenization of gold mining projects on-chain.

Missing Implementation Address Verification in Upgrade Flow

Finding ID: FYEO-NATF-01
Severity: High
Status: Remediated

Description

The frontend displays upgrade metadata (bytecode hash, implementation address) but does not verify that the implementation address in the transaction calldata matches the displayed metadata. A compromised proposer could deploy a malicious implementation, propose an upgrade transaction pointing to it, but include a different (legitimate) implementation address in the metadata JSON. Co-signers would see the legitimate address and approved bytecode hash in the UI, but unknowingly sign a transaction that upgrades to the malicious implementation.

Proof of Issue

File name: frontend/src/components/common/PendingUpgradeCard/index.tsx
Line number: 158

```
<div>
  <p className={cn("text-dark-500 font-medium")}>Implementation</p>
  <code className={cn("text-dark-900 text-xs break-all")}>
    {upgrade.implementation}
  </code>
</div>
```

File name: frontend/src/api/queries/pendingUpgrades.ts
Line number: 40

```
const getUpgradeMetadata = (tx: ProposalTransaction) => {
  if (tx.origin) {
    try {
      const origin = JSON.parse(tx.origin)
      if (origin.t === "upg") {
        return {
          contractName: origin.c,
          newVersion: origin.v,
          implementation: origin.i,  // From metadata, not verified
          bytecodeHash: origin.h,
        }
      }
    } catch (e) {
      console.error("Error parsing upgrade metadata from origin:", e)
    }
  }
  // Fallback extracts from calldata
  return {
    implementation: `0x${tx.txData.slice(34, 74)}`,  // Only used as fallback
  }
}
```

Severity and Impact Summary

Co-signers could unknowingly approve an upgrade to a malicious implementation while believing they are approving a legitimate one. The bytecode hash verification becomes meaningless if the implementation address doesn't match. This breaks the security model where FYEO verifies source code and emails approved hash to co-signers.

Recommendation

Before displaying upgrade metadata, decode the `upgradeToAndCall` calldata to extract the actual implementation address parameter. Compare it against the metadata implementation address. If they don't match, display a critical warning and prevent signing. This ensures the displayed bytecode hash corresponds to the implementation that will actually be deployed.

Remediation Notes

The frontend now implements cryptographic verification of upgrade transactions. The system extracts the implementation address from the actual transaction calldata and compares it against the metadata, displaying a security validation status and disabling signing if they don't match. Additionally, the frontend fetches the deployed bytecode from the implementation address on-chain, normalizes it (replacing embedded addresses with zeros and stripping Solidity metadata), computes its SHA-256 hash, and compares it against the metadata bytecode hash. The UI displays verification status and prevents signing if either check fails.

Build reproducibility has been ensured by setting `bytecode_hash = "none"` in `foundry.toml` and using identical bytecode normalization logic in both the deployment script and frontend. The system now provides defense-in-depth with three verification layers: automated address verification ensures metadata matches transaction calldata, automated bytecode verification cryptographically proves deployed code matches the approved hash, and manual FYEO audit verifies source code legitimacy. Co-signers can now rely on automated cryptographic verification instead of manual bytecode hash comparison.

Unpinned Frontend Dependencies Enable Supply Chain Attacks

Finding ID: FYEO-NATF-02
Severity: Medium
Status: Remediated

Description

The frontend package.json uses caret (^) and tilde (~) version ranges for dependencies, allowing automatic minor and patch updates. This creates supply chain attack risk where compromised dependency updates could inject malicious code without explicit approval.

Proof of Issue

File name: frontend/package.json

```
"dependencies": {
  "@ariakit/react": "^0.4.20",
  "@fontsource/oxygen": "^5.2.8",
  "@fontsource/poppins": "^5.2.7",
  "@reown/appkit": "^1.8.15",
  "@reown/appkit-adapter-ethers": "^1.8.15",
  "@safe-global/api-kit": "^4.0.1",
  "@safe-global/protocol-kit": "^6.1.2",
  "@safe-global/types-kit": "^3.0.0",
  "@tailwindcss/vite": "^4.1.17",
  "@tanstack/react-query": "^5.90.11",
  etc
}
```

Severity and Impact Summary

Compromised dependency updates could inject malicious code to steal private keys, manipulate transaction data, or exfiltrate sensitive information. The Safe multisig integration makes this particularly critical as malicious code could manipulate what users sign.

Recommendation

Pin all dependencies to exact versions (remove ^ and ~). Use FYEO's 3rd party library scanning when upgrading. Consider using lock file verification in CI/CD.

Missing Content Security Policy (CSP) Settings

Finding ID: FYEO-NATF-03
Severity: Low
Status: Acknowledged

Description

The frontend application lacks Content Security Policy headers, which are important for preventing XSS attacks and controlling resource loading. The index.html file and Vite configuration do not include CSP settings.

Proof of Issue

File name: frontend/index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <link rel="icon" type="image/svg+xml" href="/favicon.png" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>NatGold</title>
  <!-- Missing CSP headers -->
</head>
```

File name: frontend/vite.config.ts

```
export default defineConfig({
  server: {
    port: 3000,
  },
  plugins: [
    // No security headers configuration
  ],
})
```

Severity and Impact Summary

Without CSP headers, the application is vulnerable to XSS attacks, unauthorized script injection, and malicious resource loading. This could lead to user data theft, session hijacking, or malicious actions performed on behalf of users.

Recommendation

Implement Content Security Policy headers through Vite configuration or server-side headers. Configure appropriate directives for script sources, style sources, and other resource types to prevent unauthorized content execution.

Missing noopener noreferrer on External Links

Finding ID: FYEO-NATF-04
Severity: **Low**
Status: **Remediated**

Description

One external link using `target="_blank"` is missing the `rel="noopener noreferrer"` attribute. This creates a security vulnerability where the opened page can access the opener window via `window.opener` and potentially redirect users to phishing sites.

Proof of Issue

File name: frontend/src/components/common/Project/Documents/index.tsx
Line number: 27

```
<a href={project.documentStorageUrl} target="_blank">
  ... {project.documentStorageUrl}
</a>
```

Severity and Impact Summary

Malicious document storage URLs could use `window.opener` to redirect the NatGold UI to a phishing site while the user views documents. This is a reverse tabnapping attack vector. Users might not notice the redirect and enter credentials on the fake site.

Recommendation

Add `rel="noopener noreferrer"` to all `target="_blank"` links to prevent opened pages from accessing the opener window.

Sensitive Details Exposed in Environment Configuration

Finding ID: FYEO-NATF-05
Severity: Low
Status: Remediated

Description

The frontend environment configuration file contains hardcoded sensitive information such as API keys, that should not be exposed in client-side code.

Proof of Issue

File name: frontend/src/utils/environment.ts
Line number: 7-17

```
export const SAFE_API_KEY = "...JWT..."
```

Severity and Impact Summary

Hardcoded API keys and sensitive identifiers are exposed to anyone who can access the frontend code. This could lead to unauthorized API usage or service abuse. API keys could be used to impersonate the application.

Recommendation

Move sensitive configuration to environment variables and implement proper secrets management. Use build-time environment variable injection for client-side configuration while keeping sensitive keys on the server side.

Missing Frontend Input Validation for Custodian Wallets

Finding ID: FYEO-NATF-06
Severity: **Informational**
Status: **Remediated**

Description

The project creation form has three validation gaps that can cause transaction failures after multisig approval: floating-point precision issues in distribution sum validation, no Ethereum address format validation, and no duplicate wallet detection.

Proof of Issue

File name: frontend/src/routes/projects/create.tsx
Line number: 68

```
onFormValidate={(formState, form) => {
  const sum = formState.values.custodianDistributions.reduce(
    (acc, distr) => acc + Number(distr),
    0,
  )

  if (sum !== 100) {  // Floating-point comparison can fail
    formState.values.custodianDistributions.forEach((_, i) => {
      form.setError(
        form.names.custodianDistributions[i],
        "Total distribution must equal 100%.",
      )
    })
  }
  // No address validation or duplicate checking
}
```

Floating-point precision: Frontend validates `sum !== 100` but uses floating-point arithmetic. The mutation converts percentages to basis points using `BigInt(d * 100)`. Due to floating-point precision, values like 33.34 produce `33.34 * 100 = 3334.0000000000005`, causing `BigInt()` to throw "cannot be converted to a BigInt because it is not an integer". Frontend validation passes (33.33 + 33.33 + 33.34 = 100) but transaction creation crashes.

No address validation: Form accepts any string as wallet address without checking Ethereum address format (0x + 40 hex chars).

No duplicate detection: Same address can be added multiple times with different percentages, causing confusion.

File name: contract/src/NatGoldQueueOrchestrator.sol
Line number: 254

```
require(totalPercentage == 10_000, "Orchestrator: percentages must sum to 10000
(100%)");
```

Contract requires exact sum of 10000 basis points.

Severity and Impact Summary

Certain valid percentage combinations pass frontend validation but crash during transaction creation with BigInt conversion error. This blocks project creation for affected percentage splits. Invalid addresses and duplicates cause transaction failures after multisig approval, wasting gas. Typos in addresses could result in permanent token loss.

Recommendation

Use integer basis points throughout: validate sum equals 10000 basis points (not 100 percent), store as integers, and convert to BigInt without multiplication. Add Ethereum address format validation. Check for duplicate addresses in custodian wallet array.

Our Process

Methodology

FYEO Inc. uses the following high-level methodology when approaching engagements. They are broken up into the following phases.

Figure 2: Methodology Flow

Kickoff

The project is kicked off as the sales process has concluded. We typically set up a kickoff meeting where project stakeholders are gathered to discuss the project as well as the responsibilities of participants. During this meeting we verify the scope of the engagement and discuss the project activities. It's an opportunity for both sides to ask questions and get to know each other. By the end of the kickoff there is an understanding of the following:

- Designated points of contact
- Communication methods and frequency
- Shared documentation
- Code and/or any other artifacts necessary for project success
- Follow-up meeting schedule, such as a technical walkthrough
- Understanding of timeline and duration

Ramp-up

Ramp-up consists of the activities necessary to gain proficiency on the project. This can include the steps needed for familiarity with the codebase or technological innovation utilized. This may include, but is not limited to:

- Reviewing previous work in the area including academic papers
- Reviewing programming language constructs for specific languages
- Researching common flaws and recent technological advancements

Review

The review phase is where most of the work on the engagement is completed. This is the phase where we analyze the project for flaws and issues that impact the security posture. Depending on the project this may include an analysis of the architecture, a review of the code, and a specification matching to match the architecture to the implemented code.

In this code audit, we performed the following tasks:

1. Security analysis and architecture review of the original protocol
2. Review of the code written for the project
3. Compliance of the code with the provided technical documentation

The review for this project was performed using manual methods and utilizing the experience of the reviewer. No dynamic testing was performed, only the use of custom-built scripts and tools were used to assist the reviewer during the testing. We discuss our methodology in more detail in the following sections.

Code Safety

We analyzed the provided code, checking for issues related to the following categories:

- General code safety and susceptibility to known issues
- Poor coding practices and unsafe behavior
- Leakage of secrets or other sensitive data through memory mismanagement
- Susceptibility to misuse and system errors
- Error management and logging

This list is general and not comprehensive, meant only to give an understanding of the issues we are looking for.

Technical Specification Matching

We analyzed the provided documentation and checked that the code matches the specification. We checked for things such as:

- Proper implementation of the documented protocol phases
- Proper error handling
- Adherence to the protocol logical description

Reporting

FYEO Inc. delivers a draft report that contains an executive summary, technical details, and observations about the project.

The executive summary contains an overview of the engagement including the number of findings as well as a statement about our general risk assessment of the project. We may conclude that the overall risk is low but depending on what was assessed we may conclude that more scrutiny of the project is needed.

We report security issues identified, as well as informational findings for improvement, categorized by the following labels:

- Critical
- High
- Medium
- Low
- Informational

The technical details are aimed more at developers, describing the issues, the severity ranking and recommendations for mitigation.

As we perform the audit, we may identify issues that aren't security related, but are general best practices and steps that can be taken to lower the attack surface of the project. We will call those out as we encounter them and as time permits.

As an optional step, we can agree on the creation of a public report that can be shared and distributed with a larger audience.

Verify

After the preliminary findings have been delivered, this could be in the form of the approved communication channel or delivery of the draft report, we will verify any fixes within a window of time specified in the project. After the fixes have been verified, we will change the status of the finding in the report from open to remediated.

The output of this phase will be a final report with any mitigated findings noted.

Additional Note

It is important to note that, although we did our best in our analysis, no code audit or assessment is a guarantee of the absence of flaws. Our effort was constrained by resource and time limits along with the scope of the agreement.

While assessing the severity of the findings, we considered the impact, ease of exploitability, and the probability of attack. This is a solid baseline for severity determination.