



Cyberscope

Static Analysis Report

Pocketcoin

March 2025

Repository <https://github.com/pocketnetteam/pocketnet.gui>

Commit [a81ad26890c5e10e28a0de8a691340b72300d67a](#)

Audited by © cyberscope

Table of Contents

| | |
|--|-----------|
| Table of Contents | 1 |
| Risk Classification | 2 |
| Review | 3 |
| Audit Updates | 3 |
| Overview | 4 |
| Audit Scope | 4 |
| Findings Breakdown | 5 |
| Diagnostics | 6 |
| EHS - Exposed Hardcoded Secrets | 7 |
| Description | 7 |
| Recommendation | 7 |
| DCE - Dead Code Elimination | 8 |
| Description | 8 |
| Recommendation | 8 |
| EBS - Empty Block Statements | 9 |
| Description | 9 |
| Recommendation | 9 |
| MEC - Missing ESLint Configuration | 10 |
| Description | 10 |
| Recommendation | 10 |
| PEAF - Promise Executors Async Functions | 11 |
| Description | 11 |
| Recommendation | 12 |
| PBU - Prototype Builtins Usage | 13 |
| Description | 13 |
| Recommendation | 13 |
| RVD - Redundant Variable Declaration | 14 |
| Description | 14 |
| Recommendation | 14 |
| UV - Undefined Variables | 15 |
| Description | 15 |
| Recommendation | 15 |
| ULD - Unexpected Lexical Declarations | 16 |
| Description | 16 |
| Recommendation | 17 |
| Summary | 18 |
| Disclaimer | 19 |
| About Cyberscope | 20 |

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation:** This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation:** This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical:** Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium:** Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor:** Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative:** Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

| Severity | Likelihood / Impact of Exploitation |
|-----------------------|--|
| ● Critical | Highly Likely / High Impact |
| ● Medium | Less Likely / High Impact or Highly Likely/ Lower Impact |
| ● Minor / Informative | Unlikely / Low to no Impact |

Review

| | |
|------------|---|
| Repository | https://github.com/pocketnetteam/pocketnet.gui |
| Commit | a81ad26890c5e10e28a0de8a691340b72300d67a |

Audit Updates

| | |
|-------------------|-------------|
| Initial Audit | 04 Mar 2025 |
| Corrected Phase 2 | 11 Mar 2025 |

Overview

The Bastyon Desktop Browser is a decentralized social network client built on blockchain technology. It provides users with a censorship-resistant platform for content creation, interaction, and cryptocurrency-based rewards without relying on centralized entities. Unlike traditional social media platforms, Bastyon prioritizes privacy, allowing users to register and interact without requiring an email or phone number. Authentication is secured through a private key mechanism, ensuring full user control over accounts.

As an open-source project, Bastyon aims to provide a transparent and community-driven ecosystem. The desktop application, which is the focus of this audit, enables users to access the Bastyon network across multiple operating systems, including Windows, macOS, and Linux.

Audit Scope

The primary objective of this audit was to perform a static security analysis of the Bastyon Desktop Browser's codebase. The audit focused on identifying security vulnerabilities, performance bottlenecks, maintainability concerns, and deviations from best practices through static code analysis. The assessment did not involve dynamic testing, runtime analysis, or penetration testing of live environments.

The audit specifically examined areas including security vulnerabilities, code maintainability, performance optimizations, and best practices compliance. By identifying and addressing issues in these areas, the audit aims to enhance the security, reliability, and efficiency of the Bastyon Desktop Browser, ensuring a safer and more robust user experience.

Findings Breakdown



| | |
|---------------------|---|
| Critical | 1 |
| Medium | 0 |
| Minor / Informative | 8 |

| Severity | Unresolved | Acknowledged | Resolved | Other |
|---------------------|------------|--------------|----------|-------|
| Critical | 1 | 0 | 0 | 0 |
| Medium | 0 | 0 | 0 | 0 |
| Minor / Informative | 8 | 0 | 0 | 0 |

Diagnostics

● Critical ● Medium ● Minor / Informative

| Severity | Code | Description | Status |
|----------|------|-----------------------------------|------------|
| ● | EHS | Exposed Hardcoded Secrets | Unresolved |
| ● | DCE | Dead Code Elimination | Unresolved |
| ● | EBS | Empty Block Statements | Unresolved |
| ● | MEC | Missing ESLint Configuration | Unresolved |
| ● | PEAF | Promise Executors Async Functions | Unresolved |
| ● | PBU | Prototype Builtins Usage | Unresolved |
| ● | RVD | Redundant Variable Declaration | Unresolved |
| ● | UV | Undefined Variables | Unresolved |
| ● | ULD | Unexpected Lexical Declarations | Unresolved |

EHS - Exposed Hardcoded Secrets

| | |
|-------------|------------|
| Criticality | Critical |
| Status | Unresolved |

Description

Hardcoding sensitive values like API keys, client secrets, and credentials directly into the source code is a serious security risk. Exposing secrets in a public or shared repository makes them vulnerable to leaks, potentially allowing attackers to misuse the credentials. If this code is committed to a public repository, attackers could gain unauthorized access to the API using the exposed credentials.

```
firebase.initializeApp({  
  messagingSenderId: "1020521924918",  
  projectId: 'pocketnet',  
  apiKey: 'AIzaSyC_Jeet2gpKRZp44iATwlFFA7iGNysabkk',  
  appId: '1:1020521924918:ios:ab35cc84f0d10d86aacb97',  
});
```

Recommendation

The team is strongly advised to store sensitive values in environment variables and access them dynamically. Additionally, the team could use cloud-based secret managers like AWS Secrets Manager, HashiCorp Vault, or environment secrets in CI/CD pipelines. If a secret is leaked, it should be immediately revoked and a new one should be generated. By implementing these best practices, the team could significantly reduce the risk of credential leaks and unauthorized access.

DCE - Dead Code Elimination

| | |
|-------------|---------------------|
| Criticality | Minor / Informative |
| Status | Unresolved |

Description

In JavaScript, dead code is code that is written in the codebase, but is never executed or reached during normal execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a codebase more difficult to understand and maintain, and can also increase the size of the app and the cost of deploying and interacting with it. The following code segment is an example of unreachable code.

```
if : function(){  
    return false  
    if(isTablet()) return true  
}
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

EBS - Empty Block Statements

| | |
|--------------------|---------------------|
| Criticality | Minor / Informative |
| Status | Unresolved |

Description

Empty block statements, such as an unimplemented if statement, often appear in code due to incomplete refactoring or as placeholders for future logic. While these empty blocks do not cause runtime errors, they can create confusion for anyone reading the code, as they suggest that logic was intended to be included but was either forgotten or left unfinished. This can lead to misunderstandings about the purpose of the code and may cause issues during future maintenance or development.

Recommendation

To mitigate this issue, the team is advised to rewrite such code segments accordingly:

- If a block is not needed, the team could remove it entirely. This will keep the codebase clean and reduce unnecessary complexity.
- If a block is intended to be implemented later, the team could add a TODO comment explaining what needs to be added. This provides clear guidance for future development.
- If a block was left empty by mistake, the team should complete the necessary logic to handle the required cases.

Ensuring that all block statements are purposeful and complete helps maintain a clean, understandable, and functional codebase.

MEC - Missing ESLint Configuration

| | |
|--------------------|---------------------|
| Criticality | Minor / Informative |
| Status | Unresolved |

Description

The codebase lacks an ESLint configuration, which is a valuable tool for identifying and fixing issues in JavaScript code. ESLint not only helps catch errors but also enforces a consistent code style and promotes best practices. It can significantly enhance code quality and maintainability by providing a standardized approach to coding conventions and identifying potential problems.

Recommendation

The team is strongly advised to integrate ESLint into the project by creating an ESLint configuration file (e.g., `.eslintrc.js` or `.eslintrc.json`) and defining rules that align with the team's coding standards. Consider using popular ESLint configurations, such as Airbnb, Standard, or your own customized set of rules. By incorporating ESLint into the project, the team ensures consistent code quality, catches potential problems early in the development process, and establishes a foundation for collaborative and maintainable code.

PEAF - Promise Executors Async Functions

| | |
|-------------|---------------------|
| Criticality | Minor / Informative |
| Status | Unresolved |

Description

Using an async function as the executor inside a new Promise constructor is problematic because:

1. If the async function throws an error before explicitly calling resolve or reject, the error is lost, and the Promise does not reject as expected. This makes debugging difficult.
2. The async function already returns a Promise, making the explicit new Promise wrapper unnecessary in most cases.
3. Awaiting inside the Promise constructor is often unnecessary and can lead to unpredictable execution flow.

```
return new Promise(async (resolve, reject) => {
  let req = request(meta[key].url);

  progress(req, { throttle: 500 })
    .on('progress', function (state) {
      if (progressState) {
        let st = progressState(state);
        if (st && st.break) req.abort();
      }
    })
    .on('error', function (err) {
      if (err.message === 'aborted') return resolve();
      console.log(err);
      return reject(err);
    })
    .on('end', function () {
      return resolve();
    })
    .pipe(fs.createWriteStream(endFile, { mode: 0o755 }));
}).then(r => {
  return Promise.resolve(endFile);
});
```

Recommendation

To mitigate this issue, it is recommended to remove the `async` keyword from the executor function and handle asynchronous operations using callbacks or `.then() / .catch()`.

If `await` is needed, an `async` function without manually creating a `Promise` would be more suitable. Additionally, all errors should be properly handled using explicit `reject` calls. By making these changes, the promise executors become more predictable, easier to debug, and adheres to best practices in asynchronous JavaScript.

PBU - Prototype Builtins Usage

| | |
|-------------|---------------------|
| Criticality | Minor / Informative |
| Status | Unresolved |

Description

Calling `hasOwnProperty` directly on an object can be unsafe because the object may have a property named `hasOwnProperty`, which would shadow the built-in method. This can lead to unexpected behavior, including crashes or security vulnerabilities, particularly when dealing with user-supplied data such as JSON inputs.

For instance, an object created using `Object.create(null)` won't inherit from `Object.prototype`, leading to a `TypeError` when calling `hasOwnProperty` directly.

```
if (overrides.hasOwnProperty(key))  
{  
    defaults[key] = overrides[key];  
}
```

Recommendation

To avoid potential issues, always call `hasOwnProperty` from `Object.prototype` instead of directly invoking it on an object. This ensures that the correct method is used, even if the object has a conflicting property, preventing unintended errors or security issues.

RVD - Redundant Variable Declaration

| | |
|--------------------|---------------------|
| Criticality | Minor / Informative |
| Status | Unresolved |

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The codebase declares certain variables that are not used in a meaningful way. As a result, these variables are redundant.

Recommendation

The team is advised to remove any unnecessary variables to clean up the code. If they are meant for future usage, the team could prefix them with `_` (e.g., `_actions`) to indicate intentional non-use. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

UV - Undefined Variables

| | |
|--------------------|---------------------|
| Criticality | Minor / Informative |
| Status | Unresolved |

Description

Several variables are used in the codebase but are not defined before use. This leads to potential `ReferenceError` exceptions at runtime and is indicative of missing imports, typos, or undeclared dependencies.

Recommendation

The team is advised to verify if these variables should be imported or declared before usage. If they belong to a third-party library, ensure the dependency is properly installed and imported.

ULD - Unexpected Lexical Declarations

| | |
|-------------|---------------------|
| Criticality | Minor / Informative |
| Status | Unresolved |

Description

The codebase uses lexical declarations such as `let`, `const`, and `function` inside `case` or `default` clauses within a switch statement. Lexical declarations are hoisted to the top of the switch block, meaning they are in scope for the entire switch but remain uninitialized until the case where they are declared is executed. If a different case is executed first, accessing such variables will throw a `ReferenceError` before the variable is initialized.

This can lead to unexpected behavior and difficult-to-debug issues since the variable appears to be in scope but isn't initialized unless its associated case is executed. The following segment is a sample of all the occurrences.

```
case 413:
  const err413 = Error('Error 413: Video was rejected by Peertube server');

  err413.video = {
    size: parameters.video.size,
    type: parameters.video.type,
  };

  return Promise.reject(err413);

case 415:
  const err415 = Error('Error 415: Unsupported video type');

  err415.video = {
    type: parameters.video.type,
  };

  return Promise.reject(err415);
...
```

Recommendation

To mitigate this issue, the team is advised to wrap each case clause in a block `{}` to ensure that lexical declarations are scoped to their respective cases, ensuring that variables are only accessible and initialized within the correct case block. By properly scoping lexical declarations within switch statements, the code will become more stable and less prone to errors related to variable hoisting and initialization.

Summary

Pocketcoin implements a desktop app mechanism. This audit investigates security issues, business logic concerns, and potential improvements. The analysis focused on identifying vulnerabilities in the Bastyon Desktop Browser, through static code analysis. Additionally, the audit examined maintainability, performance optimizations, and adherence to best coding practices in JavaScript.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security. Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.



The Cyberscope team

cyberscope.io