Cyberscope

Static Analysis Report

Pocketcoin

February 2025

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

Commit 3e0523ac155019702b8a1c417bfc628b3539d5f2

Audited by © cyberscope

Table of Contents

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	3e0523ac155019702b8a1c417bfc628b3539d5f2

Audit Updates

Initial Audit	04 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.

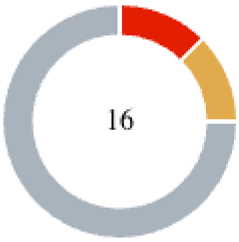
As an open-source project, Bastyon aims to provide a transparent and community-driven ecosystem. The desktop application, which is the focus of this audit, enables users to access the Bastyon network across multiple operating systems, including Windows, macOS, and Linux.

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	2
●	Medium	2
●	Minor / Informative	12

Severity	Unresolved	Acknowledged	Resolved	Other
● Critical	2	0	0	0
● Medium	2	0	0	0
● Minor / Informative	12	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
● (Critical)	EHS	Exposed Hardcoded Secrets	Unresolved
● (Critical)	ITC	Insecure TLS Configuration	Unresolved
● (Medium)	ITC	Incorrect typeof Comparison	Unresolved
● (Medium)	PCS(V	Potential Cross-Site Scripting (XSS) Vulnerability	Unresolved
● (Minor / Informative)	DCE	Dead Code Elimination	Unresolved
● (Minor / Informative)	DOM	Delete Operator Misuse	Unresolved
● (Minor / Informative)	DPK	Duplicate Property Keys	Unresolved
● (Minor / Informative)	EBS	Empty Block Statements	Unresolved
● (Minor / Informative)	MEC	Missing ESLint Configuration	Unresolved
● (Minor / Informative)	PEAF	Promise Executors Async Functions	Unresolved
● (Minor / Informative)	PBU	Prototype Builtins Usage	Unresolved
● (Minor / Informative)	RBC	Redundant Boolean Casting	Unresolved
● (Minor / Informative)	RVD	Redundant Variable Declaration	Unresolved
● (Minor / Informative)	SIE	Server Information Exposure	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.

```
self.imgur = {
    clientId: '61175058f8e21f4',
    secret: 'ea4020d8024dfb78d372d1cd21c2f3215c72ead4'
};

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.

ITC - Insecure TLS Configuration

Criticality	Critical
Status	Unresolved

Description

Disabling TLS certificate verification by setting `NODE_TLS_REJECT_UNAUTHORIZED` to 0 makes HTTPS requests insecure by allowing connections to untrusted or potentially malicious servers. This bypasses crucial security checks that verify the authenticity of SSL/TLS certificates, exposing the application to Man-in-the-Middle (MitM) attacks, data interception, and spoofing attacks. Attackers could intercept sensitive information, such as API keys, authentication tokens, and user credentials, by presenting a fraudulent TLS certificate, which the application would accept without validation.

```
process.env.NODE_TLS_REJECT_UNAUTHORIZED = '0';
```

Recommendation

TLS certificate verification should always remain enabled in production environments to ensure secure HTTPS connections. If a certificate validation issue arises, it should be resolved by properly configuring the system's certificate authority (CA) bundle rather than disabling security checks. Instead of disabling certificate verification, ensure that the system uses a valid CA store or explicitly specify trusted certificates in the `https.Agent` configuration. By keeping TLS verification enabled and handling certificates correctly, the application remains protected against MitM attacks and ensures secure communication with external services.

ITC - Incorrect typeof Comparison

Criticality	Medium
Status	Unresolved

Description

The `typeof` operator always returns a string, and its result should be compared to a valid string literal representing a JavaScript type. The codebase contains typeof comparisons with incorrect types. This can lead to unexpected behavior, potentially causing runtime errors.

For instance, the following code segment treats `undefined` as an identifier instead of a string literal.

```
if (block && typeof ckeys[key].block != undefined) {
    ckeys[key].block = block;
}
```

Recommendation

The team is strongly advised to always compare `typeof` results to string literals matching valid JavaScript types: "undefined", "object", "boolean", "number", "string", "function", "symbol", "bigint". Additionally, the usage of strict equality (`===`) instead of loose equality (`==`) is highly encouraged to avoid implicit type coercion issues. By ensuring proper `typeof` comparisons, the team could prevent logical errors and improve code reliability.

PCSV - Potential Cross-Site Scripting (XSS) Vulnerability

Criticality	Medium
Status	Unresolved

Description

The provided code directly injects unsanitized user input from window.location.search into the HTML document, leading to a DOM-based Cross-Site Scripting (XSS) vulnerability. This occurs when user-controlled data is dynamically inserted into the page without proper validation or sanitization. An attacker can craft a malicious URL containing a script payload, which executes when a user visits the page. For example, visiting https://example.com/page.html? would cause the script to execute within the victim's browser. Since the html() function interprets and renders HTML, this vulnerability allows an attacker to run arbitrary JavaScript, potentially leading to session hijacking, credential theft, or malware distribution.

```
var html = window.location.search.substr(1);
$('html .cnt').html(decodeURIComponent(html))
...
```

Recommendation

User input should always be sanitized before being inserted into the DOM. A recommended approach is to use the DOMPurify library to strip out any potentially malicious content before rendering the input. The `DOMPurify.sanitize()` function ensures that the injected content does not contain harmful scripts while preserving safe HTML elements. If the input does not require any HTML formatting, using `text()` instead of `html()` provides additional protection by ensuring that the input is treated as plain text. By implementing proper sanitization techniques, the risk of DOM-based XSS attacks is significantly reduced. This ensures that user-generated input cannot introduce security vulnerabilities while maintaining the intended functionality of the application.

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.

DOM - Delete Operator Misuse

Criticality	Minor / Informative
Status	Unresolved

Description

The `delete` operator is meant to remove properties from objects, not to delete local variables declared with var, let, or const. Using `delete` on a variable has no effect and does not free up memory as intended. Instead, it can lead to confusion and unexpected behavior in the code.

The following code segments is an example of using the `delete` operator to delete variables.

```
delete pkbuffer
delete hash
delete hashtrue
```

Recommendation

The team is advised to remove the redundant `delete` statements, as they do not achieve the intended effect. If these variables need to be cleared for garbage collection, the team could either explicitly set them to `null` or `undefined` or use block-scoped declarations (let or const) to ensure variables are automatically garbage-collected when they go out of scope.

DPK - Duplicate Property Keys

Criticality	Minor / Informative
Status	Unresolved

Description

Defining multiple properties with the same key in an object literal can lead to unexpected behavior because the last defined property will overwrite previous ones. This makes the code harder to read and debug, as earlier definitions are silently ignored. This is particularly problematic when dealing with large codebases or dynamic object creation.

For instance, the following property keys are duplicated.

```
getstatisticcontentbydays : {
    time : 3600
},

getstatisticbydays : {
    time : 3600
},
```

Recommendation

To mitigate this issue, the team is recommended to remove duplicate keys to ensure each property is uniquely defined. By ensuring object keys are unique, the team would improve code readability, maintainability, and prevent subtle bugs caused by silent property overwrites.

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.

RBC - Redundant Boolean Casting

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 casts certain variables, that have already been coerced to a Booleans, to Booleans. The extra casting is redundant. The following expression is an example of such segments.

```
group.active = !!!group.active;
```

Recommendation

To mitigate this issue, it is recommended to remove all the unnecessary Boolean casts. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

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 unnessary 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.

SIE - Server Information Exposure

Criticality	Minor / Informative
Status	Unresolved

Description

By default, Express sets the `X-Powered-By` HTTP response header, revealing that the application is built using the Express framework. Exposing framework details to potential attackers increases the risk of targeted attacks, as they can exploit known vulnerabilities specific to that framework version. Attackers often use this information to tailor their exploits, making it easier to identify weaknesses in the application.

```
app = express();
app.use(express.json({limit: '5mb'}))
app.use(express.urlencoded({ extended: true, limit: '5mb' }))
app.use(compression({ filter: shouldCompress }))
```

Recommendation

The `X-Powered-By` header should be disabled to prevent unnecessary information disclosure. This can be done using the built-in Express method `app.disable('x-powered-by')`. Additionally, using security middleware such as Helmet further enhances protection by setting HTTP headers that mitigate various attacks. Helmet is a middleware that sets various security-related HTTP headers, protecting the application against common vulnerabilities such as clickjacking, XSS, and MIME-sniffing attacks. Using Helmet along with disabling X-Powered-By significantly reduces the risk of exposing sensitive information about the server’s underlying technology.

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.

```
case 'money':
    if (notification.outputs.length && !notification.outputs?.[0]?.addresshash){
        notification.cointype =
this.proxy.pocketnet.kit.getCoibaseType(notification.outputs[0])
    }

    let amount = notification?.outputs?.find(el => el.addresshash ===
address)?.value;

    notification.amount = amount ? amount / 100000000 : 0

    if(notification?.inputs?.find(el=>el.addresshash === address)){
        notification.ignore = true
    }

    break
...
```

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