Competitive Security Assessment

Mybucks_online

Sep 11th, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

• Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.

• Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.

• Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.

• Verify the code base is compliant with the most up-to-date industry standards and security best practices.

• Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

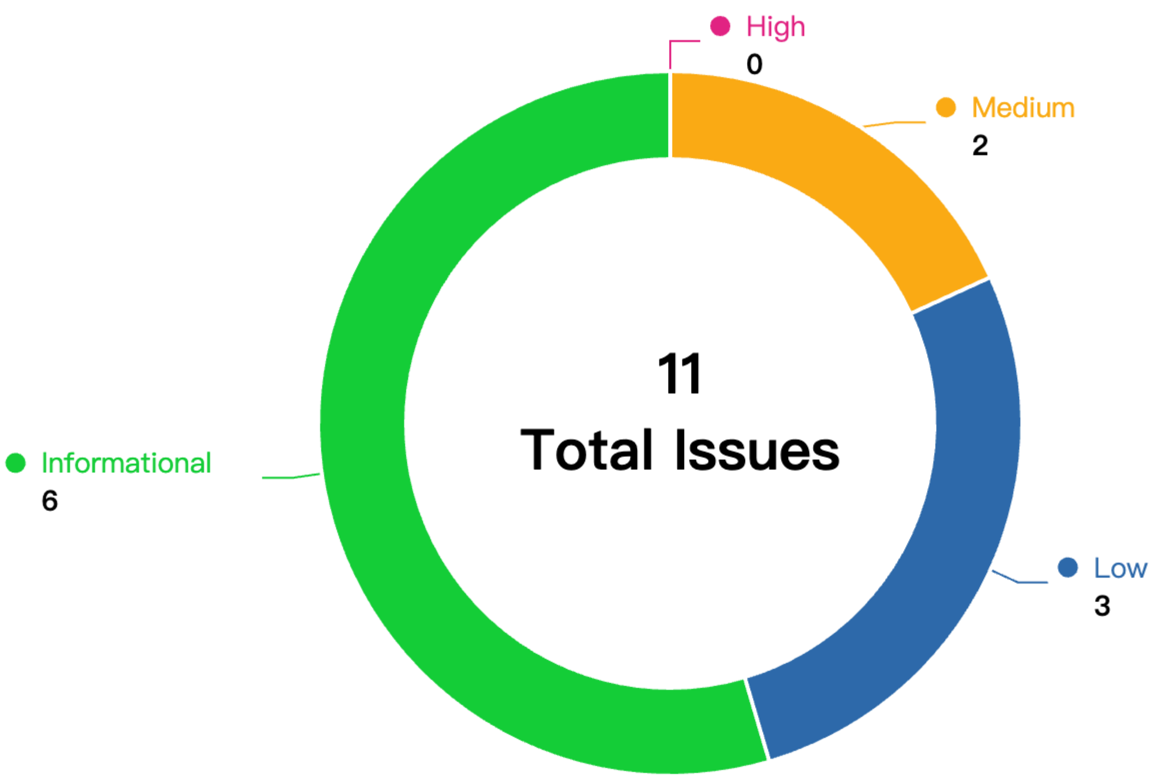
Overview

Project Name	Mybucks_online
Language	Javascript
Codebase	• https://github.com/mybucks-online/app.git • audit version-b7a2ef7beaef06be10a1381b47d8ef584093294f • final version-1255bf74c60737013001585e6d868691b11ebaa2

Audit Scope

File	SHA256 Hash
src/pages/Signin/index.jsx	e5cb3989cbf54169b19af9bccf55be8d2160d4338f43f128ae817e3ab69a54d6
src/pages/Token/index.jsx	e6a86d9098ee3c24323f705c99b9211606ac8874bb35ecfed873e78f44526048
src/pages/ConfirmTransaction/index.jsx	c4a685a132e3e8c450013f334c1a75f23c99d66648e7984e498c3532df8c440b
src/contexts/Store.jsx	815a50d608d3d5dfa6c1f9b221e1a7e44ba8ada742a36a735ef4e1a2880d8b92
src/lib/conf.js	bbe38c6afdad9865b67be3b6f6461e344472d8c53b704702a644871bc9c8439c
src/lib/account.js	289e155b960fb2979c6c5c79104e7724fa58f1849f2eade92ad0908a1ba16768

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
MOL-1	Private Key Generation Has Weak Randomness	Logical	Medium	Fixed	***
MOL-2	Ensure Raw Password has been not used	Logical	Medium	Fixed	***
MOL-3	Using libs with known vulnerabilities	Code Style	Low	Fixed	***
MOL-4	Unused Password Backup Functionality	Logical	Low	Fixed	***
MOL-5	Unrestricted Password Length: Potential Security and Performance Vulnerability	Logical	Low	Fixed	***
MOL-6	The gasMultiplier lacks custom function	Logical	Informational	Acknowledged	***
MOL-7	Should not allow copying into raw password and confirm password	Logical	Informational	Fixed	***
MOL-8	Lazy loading opt	Code Style	Informational	Acknowledged	***

MOL-9	Lack of check the return value of the function `getTokenBalancesForWalletAddress`	Logical	Informational	Fixed	***
MOL-10	Infura API key exposed directly	DOS	Informational	Fixed	***
MOL-11	Code redundancy	Code Style	Informational	Fixed	***

MOL-1:Private Key Generation Has Weak Randomness

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	***

Code Reference

- code/src/lib/conf.js#L18-L27

```
export const splitPasswordAndSalt = (rawPassword) => [
  rawPassword
    .split("")
    .filter((v, id) => id % 2 === 0)
    .join(""),
  rawPassword
    .split("")
    .filter((v, id) => id % 2 === 1)
    .join(""),
];
```

Description

***:
A critical vulnerability has been identified in the password handling mechanism, specifically in the salt generation process. The current implementation uses a deterministic method to split the user's input into a password and a salt, which significantly undermines the security benefits typically provided by salting.
The vulnerable code is as follows:

```
export const splitPasswordAndSalt = (rawPassword) => [
  rawPassword
    .split("")
    .filter((v, id) => id % 2 === 0)
    .join(""),
  rawPassword
    .split("")
    .filter((v, id) => id % 2 === 1)
    .join(""),
];
```

This function takes the raw password input and splits it into two parts:

1. The password: consisting of characters at even indices (0, 2, 4, ...)
2. The salt: consisting of characters at odd indices (1, 3, 5, ...)

This approach introduces a security flaw:

1. Lack of Randomness: The salt is entirely dependent on the password, violating the fundamental principle that a salt should be random and unique for each password.

Exploitation Example:
Consider a user's raw password input: "P@ssw0rd!"

Using the current `splitPasswordAndSalt` function:

- Password becomes: "Pswr!"
- Salt becomes: "@so0d"

An attacker aware of this pattern could exploit it in several ways:

1. Reduced Search Space: The attacker knows that for any N-character password, they only need to guess N/2 characters (rounded up) correctly.
2. Character Position Inference: The attacker can infer that special characters and numbers are more likely to appear in the salt (odd positions) rather than the password.
3. Custom Dictionary Attacks: The attacker can create a custom dictionary that only tries combinations where the complexity is concentrated in odd-numbered positions, significantly reducing the search space.
4. Length Inference: The length of the actual password is always roughly half of the raw input, allowing attackers to make educated guesses about password length policies.

Impact:
This vulnerability severely weakens the password security model. This could lead to faster and more efficient password cracking attempts, potentially compromising user accounts and sensitive data.

Recommendation

***:
To address this vulnerability and significantly enhance password security, the following changes are recommended:

1. Implement a cryptographically secure random salt generation:

```
import crypto from 'crypto';

const generateSalt = () => crypto.randomBytes(16).toString('hex');
```

Client Response

client response : Fixed in PR.
Thank you for your detailed explanation.
I totally agree with this, and would like to remove the splitting mechanism, and improve the key generation like the following:

```
import { Buffer } from "buffer";
import { ethers } from "ethers";
import { scrypt } from "scrypt-js";

const HASH_OPTIONS = {
  N: 32768, // CPU/memory cost parameter, 2^15
  r: 8, // block size parameter
  p: 5, // parallelization parameter
  keyLen: 64,
};

// rawPassword: at least 12 characters user input, lowercase, uppercase, digits, and special chara
cters
// passcode: at least 6 characters
async function generatePrivateKey(rawPassword, passcode) {
  const password = rawPassword
  const salt = rawPassword.slice(-4) + passcode

  const passwordBuffer = Buffer.from(password);
  const saltBuffer = Buffer.from(salt);

  const hashBuffer = await scrypt(
    passwordBuffer,
    saltBuffer,
    HASH_OPTIONS.N,
    HASH_OPTIONS.r,
    HASH_OPTIONS.p,
    HASH_OPTIONS.keyLen,
    (p) => console.log(Math.floor(p * 100))
  );
  const hashHex = Buffer.from(hashBuffer).toString("hex");
  const privateKey = ethers.keccak256(abi.encode(["string"], [hashHex]));

  return privateKey;
}
```

As it is fully decentralized, we don't have any storage for salt or hash results.
Users should remember and input both password and passcode.
I hope utilizing additional passcode will improve the randomness and security issues.
Please let me know your thoughts.

MOL-2:Ensure Raw Password has been not used

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	***

Code Reference

- code/README.md#L71-L92

```
71: async function generatePrivateKey(rawPassword) {
72:   // separate even, odds characters from the rawPassword
73:   const password = rawPassword.split().filter((, index) => index % 2 === 0).join("")
74:   const salt = rawPassword.split().filter((, index) => index % 2 === 1).join("")
75:
76:   const passwordBuffer = Buffer.from(password);
77:   const saltBuffer = Buffer.from(salt);
78:
79:   const hashBuffer = await scrypt(
80:     passwordBuffer,
81:     saltBuffer,
82:     HASH_OPTIONS.N,
83:     HASH_OPTIONS.r,
84:     HASH_OPTIONS.p,
85:     HASH_OPTIONS.keyLen,
86:     (p) => console.log(Math.floor(p * 100))
87:   );
88:   const hashHex = Buffer.from(hashBuffer).toString("hex");
89:   const privateKey = ethers.keccak256(abi.encode(["string"], [hashHex]));
90:
```

```
91:   return privateKey;
92: }
```

Description

***: We can find the `generatePrivateKey` function, which use raw password (divided into password and salt) and other fixed parameters to generate private key. It means that if 2 users create a same raw password, will have same private key. However, there has not any method to avoid this case.
That is, this application assumes that different users will create different passwords, which is unrealistic.

Recommendation

***: Avoid the raw password has been used before creating. For example, adding user ID, which can be a part for generating private key.

Client Response

client response : Fixed in PR.
Thank you for your detailed explanation.
So I would like to add an additional input field, just passcode with original single password.

```
import { Buffer } from "buffer";
import { ethers } from "ethers";
import { scrypt } from "scrypt-js";

const HASH_OPTIONS = {
  N: 32768, // CPU/memory cost parameter, 2^15
  r: 8, // block size parameter
  p: 5, // parallelization parameter
  keyLen: 64,
};

// rawPassword: at least 12 characters user input, lowercase, uppercase, digits, and special chara
cters
// passcode: at least 6 characters
async function generatePrivateKey(rawPassword, passcode) {
  const password = rawPassword
  const salt = rawPassword.slice(-4) + passcode

  const passwordBuffer = Buffer.from(password);
  const saltBuffer = Buffer.from(salt);

  const hashBuffer = await scrypt(
    passwordBuffer,
    saltBuffer,
    HASH_OPTIONS.N,
    HASH_OPTIONS.r,
    HASH_OPTIONS.p,
    HASH_OPTIONS.keyLen,
    (p) => console.log(Math.floor(p * 100))
  );
  const hashHex = Buffer.from(hashBuffer).toString("hex");
  const privateKey = ethers.keccak256(abi.encode(["string"], [hashHex]));

  return privateKey;
}
```

Adding passcode will reduce the probability of duplicated passwords, and improve the security performance. Please let me know your thoughts.

MOL-3:Using libs with known vulnerabilities

Category	Severity	Client Response	Contributor
Code Style	Low	Fixed	***

Code Reference

- code/src/pages/Token/index.jsx#L5

```
5: import { ethers } from "ethers";
```

Description

***: The token page is using `ethers` lib:

```
import { ethers } from "ethers";
```

The result of the "npm audit" command shows this lib depends on vulnerable versions of ws and will be affected by a DoS when handling a request with many HTTP headers:

```
# npm audit report

ws  8.0.0 - 8.17.0
Severity: high
ws affected by a DoS when handling a request with many HTTP headers - https://github.com/advisories/GHSA-3h5v-q93c-6h6q
fix available via `npm audit fix`
node_modules/ws
  ethers  6.0.0-beta.1 - 6.13.0
  Depends on vulnerable versions of ws
  node_modules/ethers

2 high severity vulnerabilities
```

Recommendation

***: Consider using latest version of the `ethers` lib.

Client Response

client response : Fixed in PR. I will upgrade ethers package into latest version.

MOL-4:Unused Password Backup Functionality

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/src/pages/Home/index.jsx#L181-L184

```
181: const backupPassword = () => {
182:     copy(joinPasswordAndSalt(password, salt));
183:     toast("Password copied into clipboard.");
184:   };
```

Description

***:
A significant user experience issue has been identified in the application's password management system. The code includes a `backupPassword` function designed to allow users to securely copy their password and salt combination to the clipboard. However, this function is not being utilized anywhere in the application, rendering a critical security feature inaccessible to users.
The unused function is as follows:

```
const backupPassword = () => {
  copy(joinPasswordAndSalt(password, salt));
  toast("Password copied into clipboard.");
};
```

This oversight has several important implications:

1. Security Risk: Users are unable to easily create secure backups of their passwords.
2. Reduced Functionality: A key feature for user account management is effectively non-existent, despite being implemented in the codebase.

Impact:
The lack of an accessible password backup feature significantly impairs the user's ability to manage their account securely. This could lead to increased support requests, potential account lockouts, and in worst-case scenarios, permanent loss of user accounts and associated data.

Recommendation

***:
To address this issue and improve user account security, the following actions are recommended:
Implement a user interface element (e.g., a button) that calls the `backupPassword` function:

```
<Button onClick={backupPassword}>Backup Password</Button>
```

Place this button in a logical location within the user's account management or settings page.

Client Response

client response : Fixed. I added password backup feature in a new PR.

MOL-5:Unrestricted Password Length: Potential Security and Performance Vulnerability

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/src/pages/Signin/index.jsx#L135-L173

```
const [disabled, setDisabled] = useState(false);
  const [progress, setProgress] = useState(0);

  const [password, salt] = useMemo(
    () => splitPasswordAndSalt(rawPassword),
    [rawPassword]
  );
  const hasMinLength = useMemo(
    () => rawPassword.length >= RAW_PASSWORD_MIN_LENGTH,
    [rawPassword]
  );
  const hasLowercase = useMemo(() => /[a-z]/.test(rawPassword), [rawPassword]);
  const hasUppercase = useMemo(() => /[A-Z]/.test(rawPassword), [rawPassword]);
  const hasNumbers = useMemo(() => /\d/.test(rawPassword), [rawPassword]);
  const hasSpecialChars = useMemo(
    () => /[ !"#$%&'()*+,\-./:;<=>?@[\\\]^_`{|}~]/.test(rawPassword),
    [rawPassword]
  );
  const hasMatchedPassword = useMemo(
    () => rawPassword && rawPassword === rawPasswordConfirm,
    [rawPassword, rawPasswordConfirm]
  );
  const hasEmpty = useMemo(
    () => !rawPassword || !password || !salt,
    [rawPassword, rawPasswordConfirm]
  );

  const hasInvalidInput = useMemo(
    () =>
      disabled ||
      hasEmpty ||
      !hasMinLength ||
      !hasUppercase ||
      !hasLowercase ||
      !hasNumbers ||
      !hasSpecialChars ||
      !hasMatchedPassword,
    [[rawPassword, rawPasswordConfirm, disabled]]
  );
```

Description

***:

A significant security and performance vulnerability has been identified in the password validation mechanism of the application. The current implementation enforces various password strength criteria but fails to impose an upper limit on password length. This oversight can lead to several potential issues:

1. User Interface Manipulation: Malicious users could input extremely long strings, potentially breaking the UI layout or causing client-side performance issues.
2. Database Overflow: If not properly handled, excessively long passwords could cause database issues, including potential overflow errors or increased storage costs.
3. Increased Attack Surface: Longer passwords provide more data for potential cryptanalysis, which could theoretically weaken the overall security of the password storage system.

The relevant code snippet is as follows:

```
const hasMinLength = useMemo(
  () => rawPassword.length >= RAW_PASSWORD_MIN_LENGTH,
  [rawPassword]
);
// ... other password criteria checks ...

const hasInvalidInput = useMemo(
  () =>
    disabled ||
    hasEmpty ||
    !hasMinLength ||
    !hasUppercase ||
    !hasLowercase ||
    !hasNumbers ||
    !hasSpecialChars ||
    !hasMatchedPassword,
  [[rawPassword, rawPasswordConfirm, disabled]]
);
```

As evident, while there is a check for minimum length (`hasMinLength`), there is no corresponding check for maximum length.
A malicious user could input an extremely long string as a password. This would pass all current validation checks but could potentially cause system instability or performance degradation.

Recommendation

***:
To address this vulnerability and enhance overall system security and performance, the following modifications are recommended:

1. Implement a maximum password length check:

```
const MAX_PASSWORD_LENGTH = 128; // Example value, adjust as needed

const hasValidLength = useMemo(
  () => rawPassword.length >= RAW_PASSWORD_MIN_LENGTH && rawPassword.length <= MAX_PASSWORD_LENGT
H,
  [rawPassword]
);
```

2. Update the `hasInvalidInput` check to include the new length validation:

```
const hasInvalidInput = useMemo(
  () =>
    disabled ||
    hasEmpty ||
    !hasValidLength ||
    !hasUppercase ||
    !hasLowercase ||
    !hasNumbers ||
    !hasSpecialChars ||
    !hasMatchedPassword,
  [[rawPassword, rawPasswordConfirm, disabled]]
);
```

3. Add client-side input restriction to prevent excessively long inputs:

```
<input
  type="password"
  value={rawPassword}
  onChange={(e) => setRawPassword(e.target.value.slice(0, MAX_PASSWORD_LENGTH))}
  maxLength={MAX_PASSWORD_LENGTH}
/>
```

Client Response

client response : Fixed in PR.
Considering the last two reasons, we do not have any database as our system is fully decentralized. Additionally, I believe a longer password provides higher safety.
However, it is impractical to use passwords that are hundreds of characters long. Extremely long passwords can break the UI and are unnecessary. Therefore, I will set an upper limit of 128 characters for password input and 16 characters for new passcode input.

MOL-6:The gasMultiplier lacks custom function

Category	Severity	Client Response	Contributor
Logical	Informational	Acknowledged	***

Code Reference

- code/src/lib/conf.js#L108-L109

```
export const gasMultiplier = (option) =>
  option === "high" ? 175n : option === "average" ? 150n : 100n;
```

Description

***: The gas fee calculation of the current wallet only has three cases: basic gas fee, 1.5 times gas fee, and 1.75 times gas fee. There is a lack of a design for users to set their own gas fees, which may cause inconvenience to users. For example, when a user urgently needs to transfer funds quickly, 1.75 times the gas fee may not be able to provide a fast enough speed.

Recommendation

***: Add a design that allows users to set their own gas fees.

Client Response

client response : Acknowledged. I would like to keep current version for a while.
Thank you.

MOL-7:Should not allow copying into raw password and confirm password

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/src/pages/Signin/index.jsx#L220C9-L244C17

```
<div>
            <Label htmlFor="password">Password</Label>
            <Input
              id="password"
              type="password"
              placeholder="Password"
              disabled={disabled}
              value={rawPassword}
              onChange={(e) => setRawPassword(e.target.value)}
              onKeyDown={onKeyDown}
            />
          </div>

          <div>
            <Label htmlFor="password-confirm">Confirm Password</Label>
            <Input
              id="password-confirm"
              type="password"
              placeholder="Confirm password"
              disabled={disabled}
              value={rawPasswordConfirm}
              onChange={(e) => setRawPasswordConfirm(e.target.value)}
              onKeyDown={onKeyDown}
            />
          </div>
```

Description

***: When inputing password into the text boxes, there are two inputs that must be filled which is the password and confirm password. Both have to match, this precursion is to reduce human error. But the precursion is not taken to its complete processes, as it allows the user to simply copy the input into both feilds.
This should not be the case, the user should not be able to copy text into password or confirm password, as it reduces the validation that and beats the confirm password sanity check. This especially important for few reasons.

1. The input is a password type, and the user is not given an option to view the typed password, the password is always masked as the user inputs the keys.
2. The end product is an blockchain address generated through a one way hashing function, which can never be reversed or recovered. So there should be more scrutiny.
3. This is the standard for most financial platforms like banks which disable copy.
4. The combination of masked password and ability to copy password into confirmation box is not a good one and is bound to cause user errors which would have unredeemable consequences.

Recommendation

*****:** Disable paste into confirmation box and password box.

Client Response

client response : Fixed in PR. I will disallow copy/paste into password fields.

MOL-8:Lazy loading opt

Category	Severity	Client Response	Contributor
Code Style	Informational	Acknowledged	***

Code Reference

- code/src/pages/Token/index.jsx#L10-L14

```
10: import BackIcon from "@mybucks/assets/icons/back.svg";
11: import RefreshIcon from "@mybucks/assets/icons/refresh.svg";
12: import ArrowUpRightIcon from "@mybucks/assets/icons/arrow-up-right.svg";
13: import InfoRedIcon from "@mybucks/assets/icons/info-red.svg";
14: import InfoGreenIcon from "@mybucks/assets/icons/info-green.svg";
```

Description

***: Lazy Loading and On-Demand Loading: To enhance performance, you can consider using dynamic import() or other lazy loading techniques for icons, style files, and similar resources.

Recommendation

***: Examples:

```
const BackIcon = React.lazy(() => import("@mybucks/assets/icons/back.svg"));
const RefreshIcon = React.lazy(() => import("@mybucks/assets/icons/refresh.svg"));
```

Client Response

client response : Acknowledged. All assets are very small size. so right now I would like to ignore it.
thank you.

MOL-9:Lack of check the return value of the function `getTokenBalancesForWalletAddress`

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/src/contexts/Store.jsx#L141-L145

```
const { data } =
        await client.BalanceService.getTokenBalancesForWalletAddress(
          chainId,
          account.address
        );
```

Description

***: As per the doc of CovalentClient, the `getTokenBalancesForWalletAddress` function will return `error`. However, in `Store.jsx` page it does not check the error:

```
const { data } =
        await client.BalanceService.getTokenBalancesForWalletAddress(
          chainId,
          account.address
        );
```

Recommendation

***: Consider following fix:

```
const { data, error } =
        await client.BalanceService.getTokenBalancesForWalletAddress(
          chainId,
          account.address
        );
if (error) {
        setConnectivity(false);
        console.log(error);
        return;
    }
```

Client Response

client response : Fixed in PR.

MOL-10:Infura API key exposed directly

Category	Severity	Client Response	Contributor
DOS	Informational	Fixed	***

Code Reference

- code/src/lib/conf.js#L48-L105

```
export const NETWORKS = {
  1: {
    chainId: 1,
    name: "ethereum",
    label: "Ethereum",
    provider: "https://mainnet.infura.io/v3/" + import.meta.env.VITE_INFURA_API_KEY,
    scanner: "https://etherscan.io",
  },
  56: {
    chainId: 56,
    name: "bsc",
    label: "BNB Chain",
    provider: "https://bsc-dataseed.binance.org/",
    scanner: "https://bscscan.com",
  },
  137: {
    chainId: 137,
    name: "polygon",
    label: "Polygon",
    provider: "https://polygon-mainnet.infura.io/v3/" + import.meta.env.VITE_INFURA_API_KEY,
    scanner: "https://polygonscan.com",
  },
  42161: {
    chainId: 42161,
    name: "arbitrum",
    label: "Arbitrum",
    provider: "https://arbitrum-mainnet.infura.io/v3/" + import.meta.env.VITE_INFURA_API_KEY,
    scanner: "https://arbiscan.io",
  },
  43114: {
    chainId: 43114,
    name: "avalanche",
    label: "Avalanche C-Chain",
    provider: "https://avalanche-mainnet.infura.io/v3/" + import.meta.env.VITE_INFURA_API_KEY,
    scanner: "https://snowtrace.io",
  },
  10: {
    chainId: 10,
    name: "optimism",
    label: "Optimism",
    provider: "https://optimism-mainnet.infura.io/v3/" + import.meta.env.VITE_INFURA_API_KEY,
    scanner: "https://optimistic.etherscan.io",
  },
  59144: {
    chainId: 59144,
    name: "linea",
    label: "Linea",
    provider: "https://linea-mainnet.infura.io/v3/" + import.meta.env.VITE_INFURA_API_KEY,
    scanner: "https://lineascan.build",
  },
  42220: {
    chainId: 42220,
    name: "celo",
    label: "Celo",
    provider: "https://celo-mainnet.infura.io/v3/" + import.meta.env.VITE_INFURA_API_KEY,
    scanner: "https://celoscan.io",
  },
};
```

Description

***: API key can be found directly in the variables of the compiled code or accessed through the browser's web inspector. Attackers could potentially deplete the API quota by launching a DDoS attack.

Recommendation

***: Forward through the backend server, set limits, and monitor login status.

Client Response

client response : Fixed in PR. I've added `app.mybucks.online` into `allow origin` for `Infura` and `Covalenthq` API. This will protect our API key from the attack.

MOL-11:Code redundancy

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	***

Code Reference

- code/src/lib/conf.js#L38

```
38: // btc | evm | solana | tron
```

Description

***:
In the configuration files, the constants which contained the name of chains
`// btc | evm | solana | tron export const NETWORK_BTC = "btc"; export const NETWORK_EVM = "evm"; export const NETWORK_TRON = "tron"; export const NETWORK_SOLANA = "sol";`
are defined and exported but never used.
Also to keep in mind that the current implementation just covers evm chains. and the address generation (hashing) mechanism used here is not compactable for chains like BTC. So it means its an unneccesary code.

Recommendation

***: Remove unused exports.

Client Response

client response : Fixed in PR.
I agree. Right now it is only available for EVM chains, and I would like to add Tron network asap. So I will remove BTC and Solana options here.

Disclaimer