QuillAudits

AUDIT REPORT

October 2025

For

MNEE

Table of Content

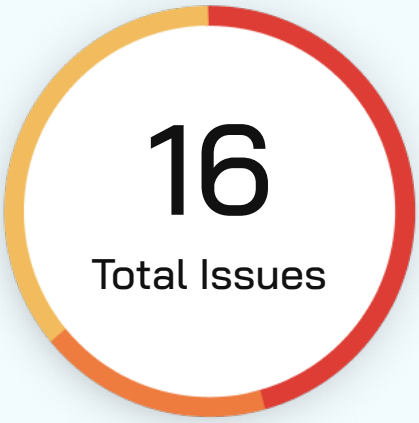
Executive Summary

Project Name	MNEE
Project URL	https://github.com/rockwalletcode/mnee-cosigner https://github.com/rockwalletcode/mnee-portal
Overview	MNEE (pronounced “money”) is a fast, USD-backed stablecoin designed for digital payments, gaming, remittances, and more. Fully collateralized 1:1 with U.S. Treasury bills and cash equivalents, MNEE ensures stability and transparency through regular audits. It operates on the high-speed **1Sat Ordinals protocol**, offering near-zero fees and instant finality. MNEE is regulated in Antigua with full AML/KYC compliance.
Review 1	11th February 2025 - 25th march 2025
Updated Code Received	7th October 2025
Review 2	28th October 2025 - 29th October 2025

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical	0(0.0%)
High	7 (43.8%)
Medium	3 (18.7%)
Low	6 (37.5%)
Informational	0(0.0%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	1	1	0	0
Partially Resolved	0	0	0	0	0
Resolved	0	6	2	6	0

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Authentication Issues	High	Resolved
2	Broken Access Control	High	Resolved
3	Broken Access Control	High	Acknowledged
4	Amount Value is Not Validated During Burn	High	Resolved
5	Unvalidated Blacklisted Request Cancellation	High	Resolved
6	Broken Access Control	High	Resolved
7	Unauthorised Refund	High	Resolved
8	Usage of weak hashing library (MD5)	Medium	Resolved
9	Insecure Input Validation	Medium	Resolved
10	No Rate Limit	Medium	Acknowledged
11	Leakage of sensitive information in logger message	Low	Resolved

Issue No.	Issue Title	Severity	Status
12	Middleware Does not Handle Token Expiration	Low	Resolved
13	User Enumeration	Low	Resolved
14	Weak Password Check	Low	Resolved
15	Improper Error Handling	Low	Resolved
16	Outdated Server Components	Low	Resolved

Checked Vulnerabilities

- Improper Authentication
- Improper Resource Usage
- Improper Authorization
- Insecure File Uploads
- Insecure Direct Object References
- Client-Side Validation Issues
- Rate Limit
- Input Validation
- Injection Attacks
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery
- Security Misconfiguration
- Broken Access Controls
- Insecure Cryptographic Storage
- Insufficient Cryptography
- Insufficient Session Expiration
- Insufficient Transport Layer Protection
- Unvalidated Redirects and Forwards
- Information Leakage
- Broken Authentication and Session Management
- Denial of Service (DoS) Attacks
- Malware
- Third-Party Components

And More..

Techniques and Methods

Throughout the pentest of application, care was taken to ensure:

- Information gathering – Using OSINT tools information concerning the web architecture, information leakage, web service integration, and gathering other associated information related to web server & web services.
- Using Automated tools approach for Pentest like Nessus, Acunetix etc.
- Platform testing and configuration
- Error handling and data validation testing
- Encryption-related protection testing
- Client-side and business logic testing

Tools and Platforms used for Pentest:

Burp Suite	Acunetix	Nmap
DNSenum	Neucli	Metasploit
Dirbuster	Nabbu	Horusec
SQLMap	Turbo Intruder	Postman
Netcat	Nessus	And Many more..

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

High Severity Issues

Authentication Issues

Resolved

Description

The application has multiple authentication based issues.

1. Lack of Rate Limiting & Brute Force Protection: The code does not implement rate limiting or account lockout mechanisms, which makes it susceptible to brute-force attacks.
2. No MFA (Multi-Factor Authentication) Support: The authentication process only relies on email and password, which is not sufficient against credential stuffing attacks.
3. Authentication and Authorization Cookie: The application has not implemented any Authorization cookies or token. This can allow unauthorized users to interact with APIs by directly sending the API a
4. request to carry out an action. Since there is no authorization token, all requests are expected to be accepted by API without verification.
5. CSRF Protection: The application has not implemented CSRF token. An anti-CSRF token should be implemented to block users from attempting (1) brute-force attacks and also perform a CSRF attack by crafting special requests.

Vulnerable File

mnee-portal-master/src/utils/auth.ts
mnee-portal-master/src/utils/api.ts

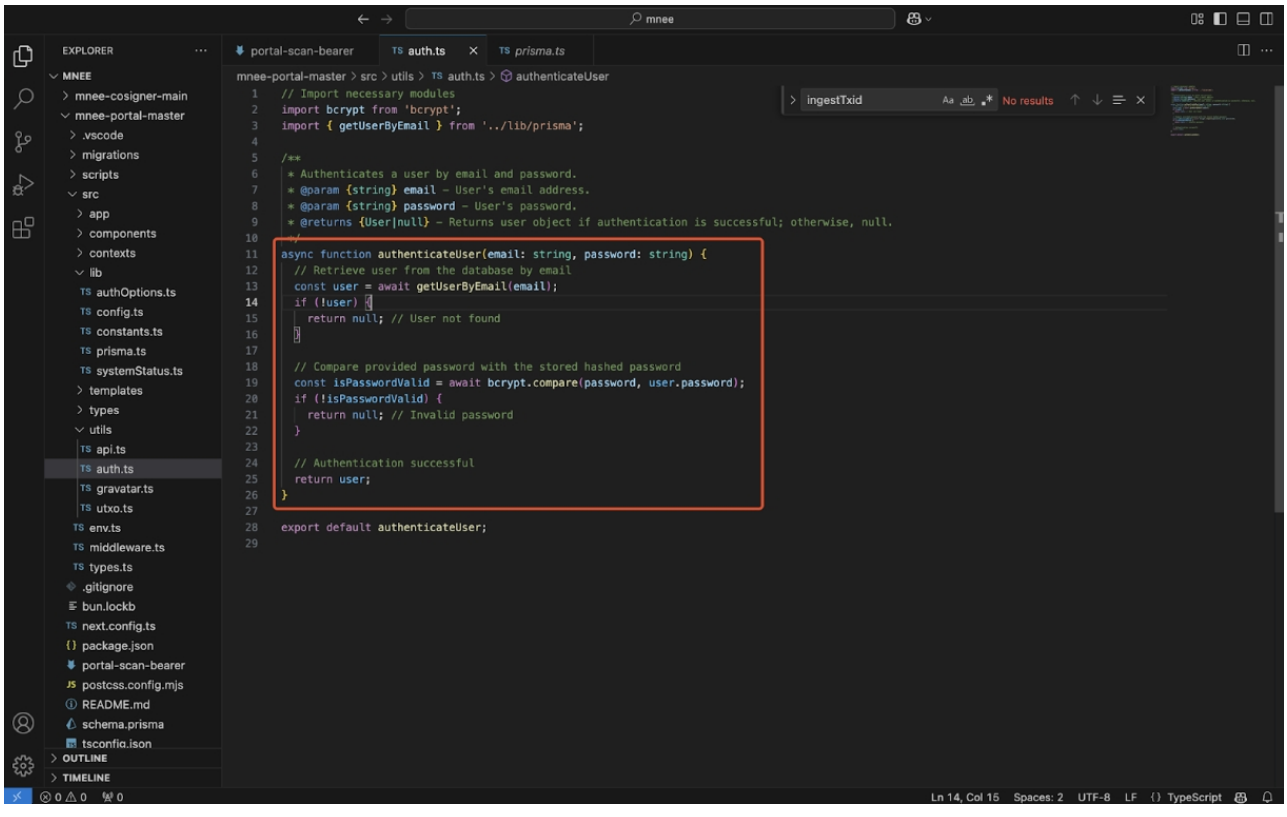
Impact

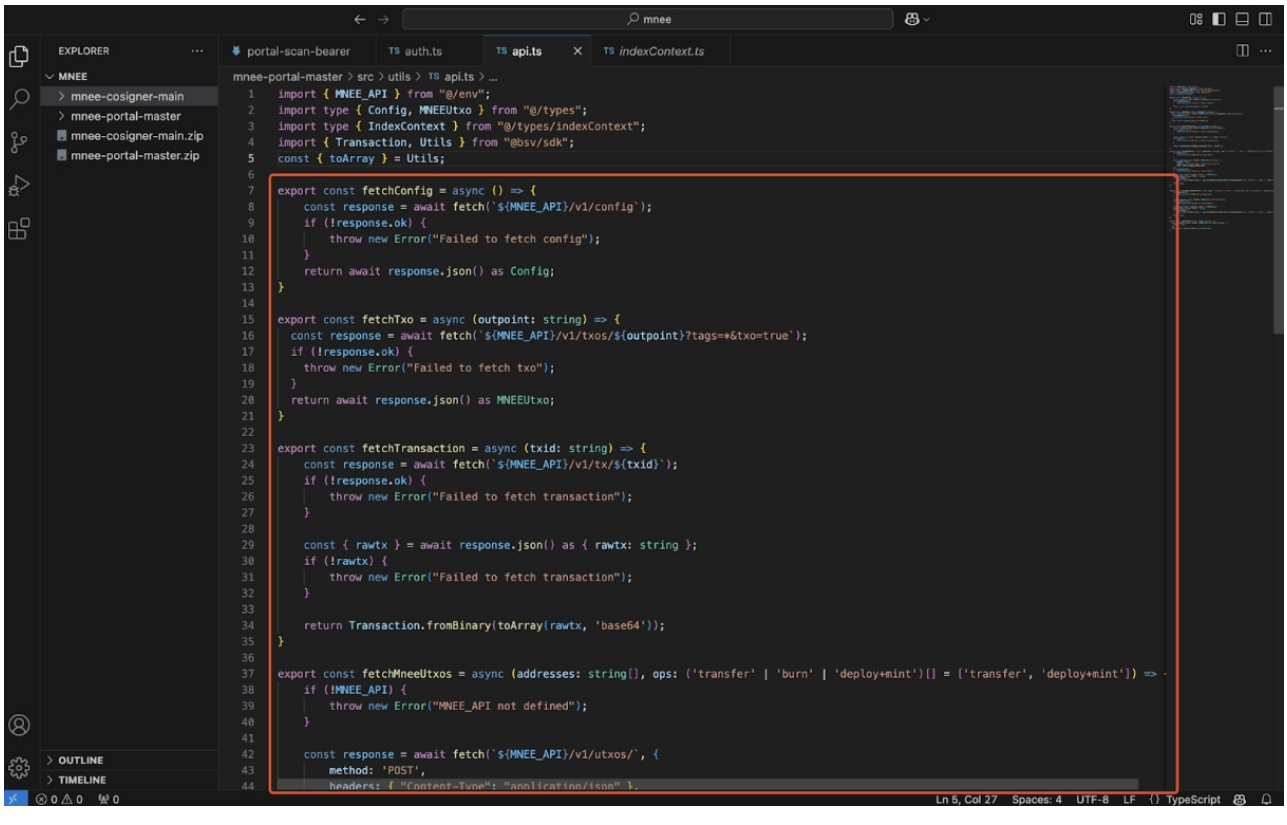
1. Attackers can brute-force username and password values

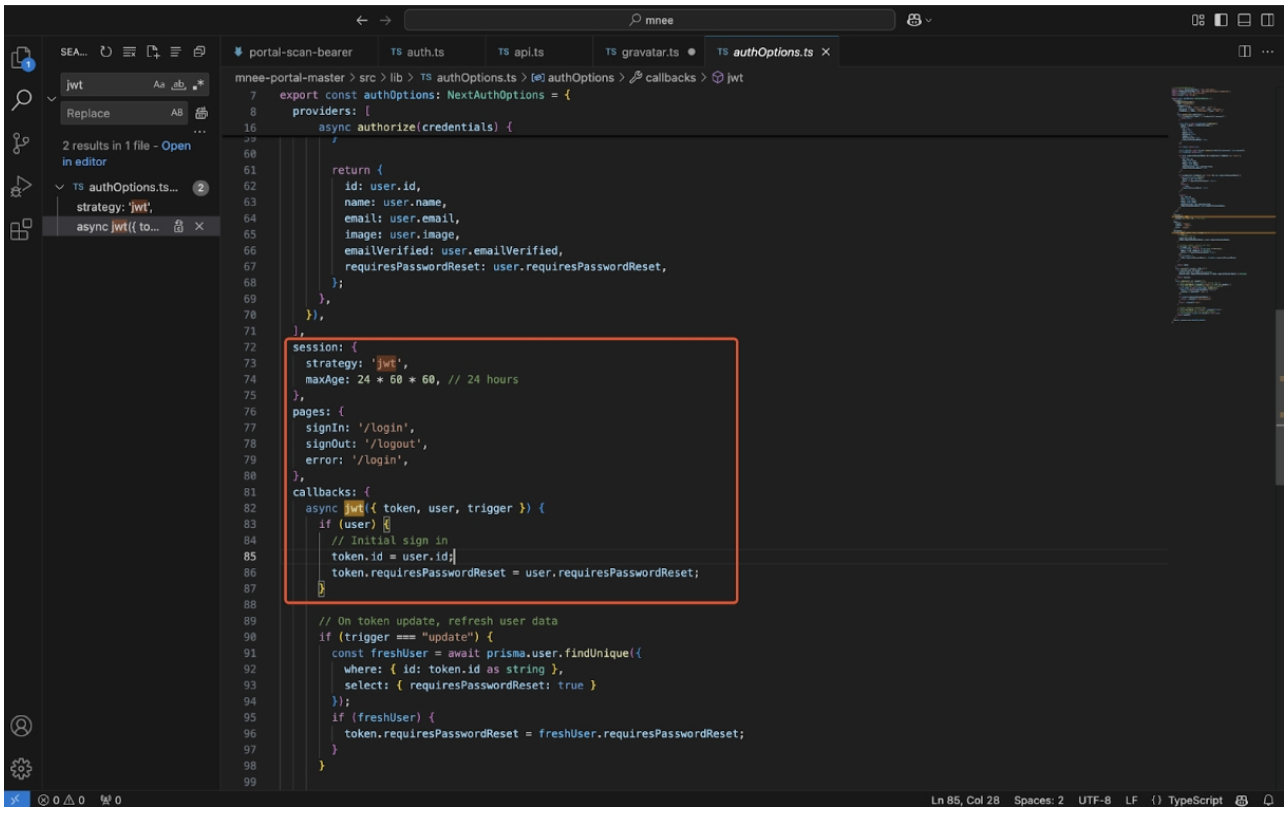
Recommendation

1. Implement rate limiting to prevent brute-force attacks on authentication mechanisms
2. Implement rate limiting on all API endpoints. You can also implement an anti-CSRF token to prevent brute-force attacks as a add-on security mechanism
3. Implement authentication/authorization cookies to prevent Broken Access Control attacks
4. Implement a check against IDOR vulnerabilities (application wide). The function should check if the corresponding value passed in the request body belongs to the person who is currently authenticated and if he is authorized to execute the request with the data parsed.

POC


mnee
EXPLORER
MNEE
mnee-cosigner-main
mnee-portal-master
.vscode
migrations
scripts
src
app
components
contexts
lib
authOptions.ts
config.ts
constants.ts
prisma.ts
systemStatus.ts
templates
types
utils
api.ts
auth.ts
gravatar.ts
utxo.ts
env.ts
middleware.ts
types.ts
.gitignore
bun.lockb
next.config.ts
package.json
portal-scan-bearer
postcss.config.mjs
README.md
schema.prisma
tsconfig.json
OUTLINE
TIMELINE
portal-scan-bearer
auth.ts
prisma.ts
mnee-portal-master > src > utils > auth.ts > authenticateUser
ingestTxid
No results
// Import necessary modules
import bcrypt from 'bcrypt';
import { getUserByEmail } from '../lib/prisma';
/**
* Authenticates a user by email and password.
* @param {string} email - User's email address.
* @param {string} password - User's password.
* @returns {User|null} - Returns user object if authentication is successful; otherwise, null.
*/
async function authenticateUser(email: string, password: string) {
// Retrieve user from the database by email
const user = await getUserByEmail(email);
if (!user) {
return null; // User not found
}
// Compare provided password with the stored hashed password
const isPasswordValid = await bcrypt.compare(password, user.password);
if (!isPasswordValid) {
return null; // Invalid password
}
// Authentication successful
return user;
}
export default authenticateUser;
Ln 14, Col 15 Spaces: 2 UTF-8 LF TypeScript


mnee
EXPLORER
MNEE
mnee-cosigner-main
mnee-portal-master
mnee-cosigner-main.zip
mnee-portal-master.zip
OUTLINE
TIMELINE
portal-scan-bearer
auth.ts
api.ts
indexContext.ts
mnee-portal-master > src > utils > api.ts > ...
import { MNEE_API } from "@/env";
import type { Config, MNEEUtxo } from "@/types";
import type { IndexContext } from "@/types/indexContext";
import { Transaction, Utils } from "@bsv/sdk";
const { toArray } = Utils;
export const fetchConfig = async () => {
const response = await fetch(`${MNEE_API}/v1/config`);
if (!response.ok) {
throw new Error("Failed to fetch config");
}
return await response.json() as Config;
}
export const fetchTxo = async (outpoint: string) => {
const response = await fetch(`${MNEE_API}/v1/txos/${outpoint}?tags=*&txo=true`);
if (!response.ok) {
throw new Error("Failed to fetch txo");
}
return await response.json() as MNEEUtxo;
}
export const fetchTransaction = async (txid: string) => {
const response = await fetch(`${MNEE_API}/v1/tx/${txid}`);
if (!response.ok) {
throw new Error("Failed to fetch transaction");
}
const { rawtx } = await response.json() as { rawtx: string };
if (!rawtx) {
throw new Error("Failed to fetch transaction");
}
return Transaction.fromBinary(toArray(rawtx, 'base64'));
}
export const fetchMneeUtxos = async (addresses: string[], ops: ('transfer' | 'burn' | 'deploy+mint')[] = ['transfer', 'deploy+mint']) =>
if (!MNEE_API) {
throw new Error("MNEE_API not defined");
}
const response = await fetch(`${MNEE_API}/v1/utxos/`, {
method: 'POST',
Ln 5, Col 27 Spaces: 4 UTF-8 LF TypeScript


mnee
portal-scan-bearer
auth.ts
api.ts
gravatar.ts
authOptions.ts
mnee-portal-master > src > lib > authOptions.ts > authOptions > callbacks > jwt
export const authOptions: NextAuthOptions = {
providers: [
async authorize(credentials) {
return {
id: user.id,
name: user.name,
email: user.email,
image: user.image,
emailVerified: user.emailVerified,
requiresPasswordReset: user.requiresPasswordReset,
};
},
}),
],
session: {
strategy: 'jwt',
maxAge: 24 * 60 * 60, // 24 hours
},
pages: {
signIn: '/login',
signOut: '/logout',
error: '/login',
},
callbacks: {
async jwt({ token, user, trigger }) {
if (user) {
// Initial sign in
token.id = user.id;
token.requiresPasswordReset = user.requiresPasswordReset;
}
// On token update, refresh user data
if (trigger === "update") {
const freshUser = await prisma.user.findUnique({
where: { id: token.id as string },
select: { requiresPasswordReset: true }
});
if (freshUser) {
token.requiresPasswordReset = freshUser.requiresPasswordReset;
}
}
Ln 85, Col 28 Spaces: 2 UTF-8 LF TypeScript

Broken Access Control

Resolved

Description

The application lacks controls for access control possibly allowing -

a. Any user to approve a mint request by providing a userId.
b. Any user can delete a mint request by providing Mint request id. Additionally, any user can delete a mint request of himself as well as of any other user.
c. Identify pending requests
d. List Approvers
e. Set password resets for any email address by providing a valid email address

Vulnerable File

mnee-portal-master/scripts/approve.ts
mnee-portal-master/scripts/delete-mint-request.ts
mnee-portal-master/scripts/find-pending.ts
mnee-portal-master/scripts/list-approvers.ts

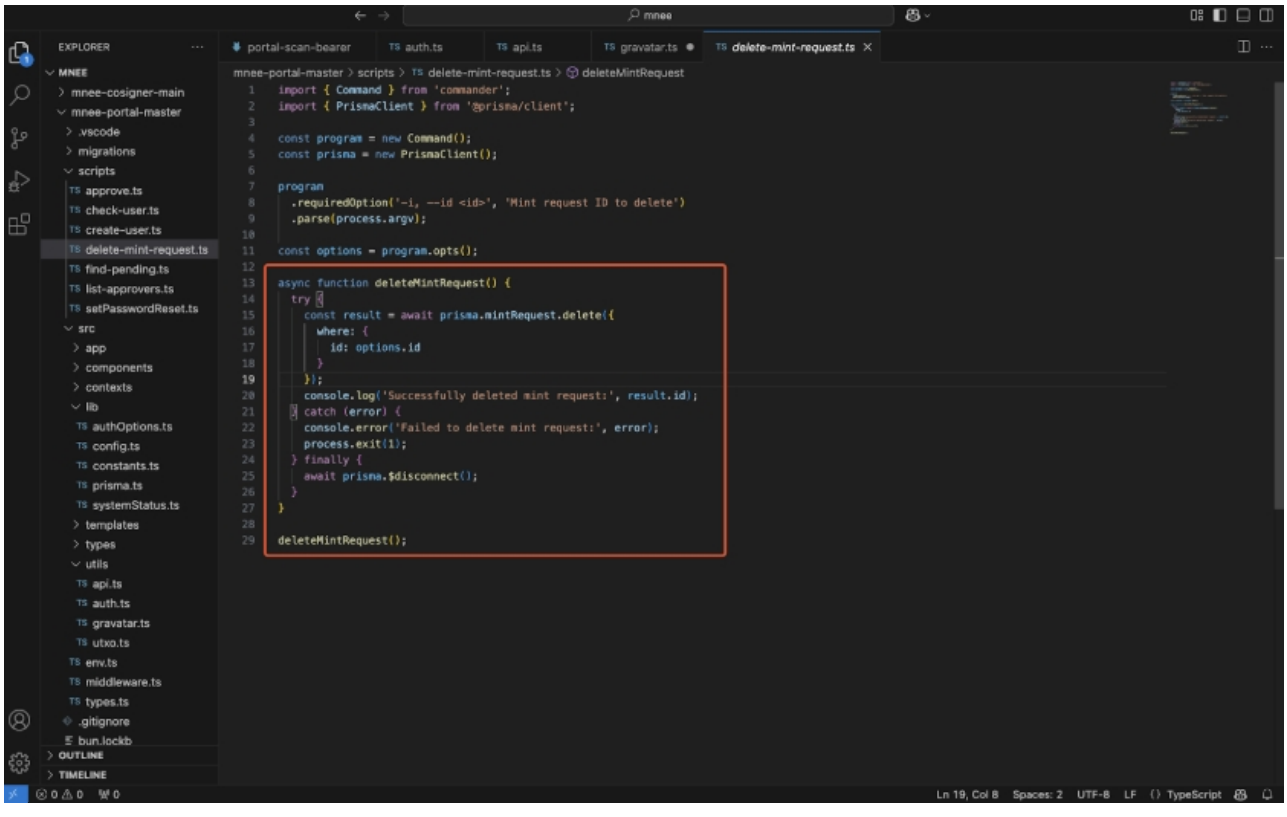
Impact

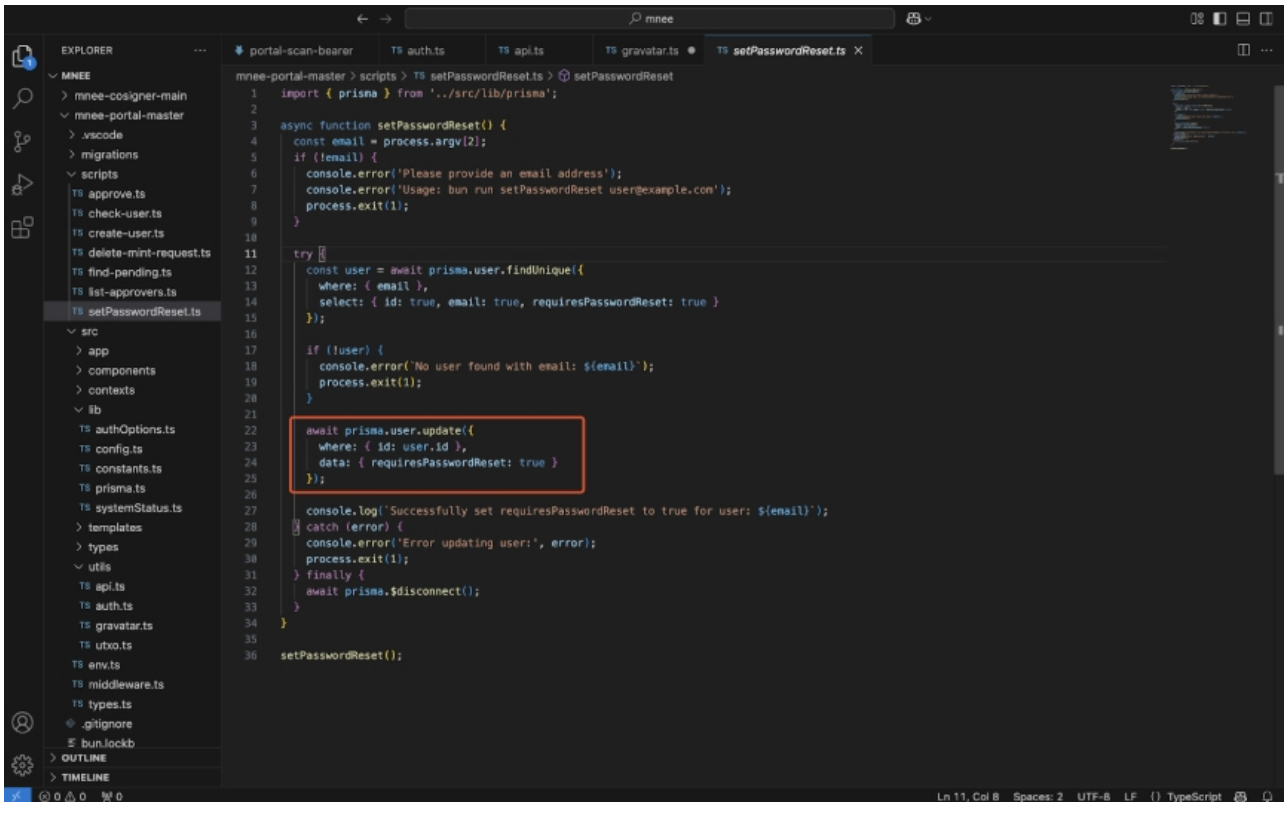
Sybil Attacks – A malicious actor can create multiple fake accounts to approve their own requests.
Collusion – Groups can approve each other's mints to manipulate token supply.
Spamming – Random users could approve unrelated mint requests, cluttering the approval system.
Attackers can delete any mint request, causing financial and operational disruptions.
Gain unauthorised access to data

Recommendation

Before approving, check if userId has approval rights

POC


mnee
EXPLORER
MNEE
mnee-cosigner-main
mnee-portal-master
.vscode
migrations
scripts
approve.ts
check-user.ts
create-user.ts
delete-mint-request.ts
find-pending.ts
list-approvers.ts
setPasswordReset.ts
src
app
components
contexts
lib
authOptions.ts
config.ts
constants.ts
prisma.ts
systemStatus.ts
templates
types
utils
api.ts
auth.ts
gravatar.ts
utxo.ts
env.ts
middleware.ts
types.ts
.gitignore
bun.lockb
OUTLINE
TIMELINE
portal-scan-bearer
auth.ts
api.ts
gravatar.ts
delete-mint-request.ts
mnee-portal-master > scripts > delete-mint-request.ts > deleteMintRequest
import { Command } from 'commander';
import { PrismaClient } from '@prisma/client';
const program = new Command();
const prisma = new PrismaClient();
program
.requiredOption('-i, --id <id>', 'Mint request ID to delete')
.parse(process.argv);
const options = program.opts();
async function deleteMintRequest() {
try {
const result = await prisma.mintRequest.delete({
where: {
id: options.id
}
});
console.log('Successfully deleted mint request:', result.id);
} catch (error) {
console.error('Failed to delete mint request:', error);
process.exit(1);
} finally {
await prisma.$disconnect();
}
}
deleteMintRequest();
Ln 19, Col 8 Spaces: 2 UTF-8 LF TypeScript


mnee
EXPLORER
MNEE
mnee-cosigner-main
mnee-portal-master
.vscode
migrations
scripts
approve.ts
check-user.ts
create-user.ts
delete-mint-request.ts
find-pending.ts
list-approvers.ts
setPasswordReset.ts
src
app
components
contexts
lib
authOptions.ts
config.ts
constants.ts
prisma.ts
systemStatus.ts
templates
types
utils
api.ts
auth.ts
gravatar.ts
utxo.ts
env.ts
middleware.ts
types.ts
.gitignore
bun.lockb
OUTLINE
TIMELINE
portal-scan-bearer
auth.ts
api.ts
gravatar.ts
setPasswordReset.ts
mnee-portal-master > scripts > setPasswordReset.ts > setPasswordReset
import { prisma } from '../src/lib/prisma';
async function setPasswordReset() {
const email = process.argv[2];
if (!email) {
console.error('Please provide an email address');
console.error('Usage: bun run setPasswordReset user@example.com');
process.exit(1);
}
try {
const user = await prisma.user.findUnique({
where: { email },
select: { id: true, email: true, requiresPasswordReset: true }
});
if (!user) {
console.error(`No user found with email: ${email}`);
process.exit(1);
}
await prisma.user.update({
where: { id: user.id },
data: { requiresPasswordReset: true }
});
console.log(`Successfully set requiresPasswordReset to true for user: ${email}`);
} catch (error) {
console.error('Error updating user:', error);
process.exit(1);
} finally {
await prisma.$disconnect();
}
}
setPasswordReset();
Ln 11, Col 8 Spaces: 2 UTF-8 LF TypeScript

Broken Access Control

Acknowledged

Description

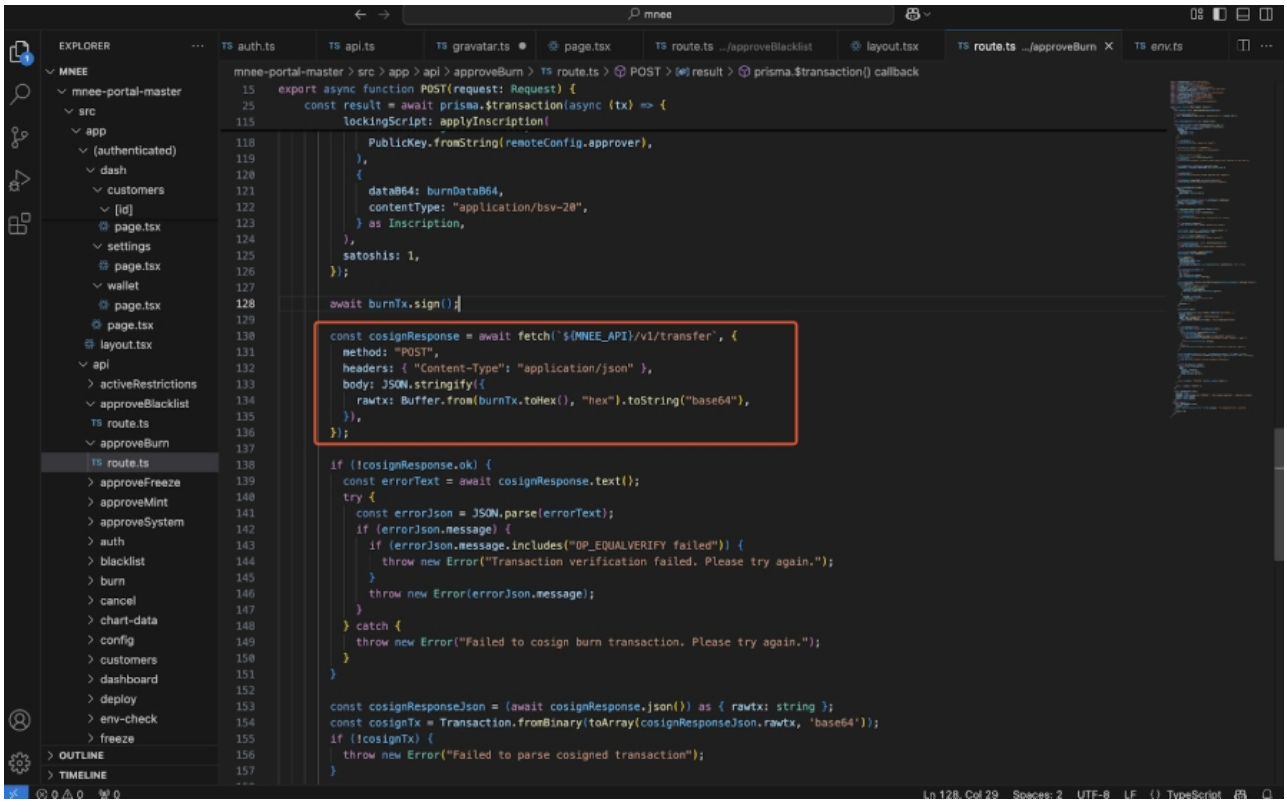
The code is not adding the session token or authentication cookie in the fetch request to MNEE_API/v1/transfer.

Vulnerable File

mnee-portal-master/src/app/api/approveBurn/route.ts
mnee-portal-master/src/app/api/approveMint/route.ts
mnee-portal-master/src/app/api/deploy/route.ts
mnee-portal-master/src/contexts/CustomerContext.tsx

Impact

The request to MNEE_API/v1/transfer lacks authentication, potentially allowing unauthorized transactions or exposing sensitive operations to attackers.
This issue is observed application wide and should be fixed at all vulnerable instances.



Amount Value is Not Validated During Burn

Resolved

Description

It is possible that an attacker can supply a negative value for the amount while trying to execute a Burn request. The code currently does not check if the value of the amount parameter is greater than 0.

Vulnerable File

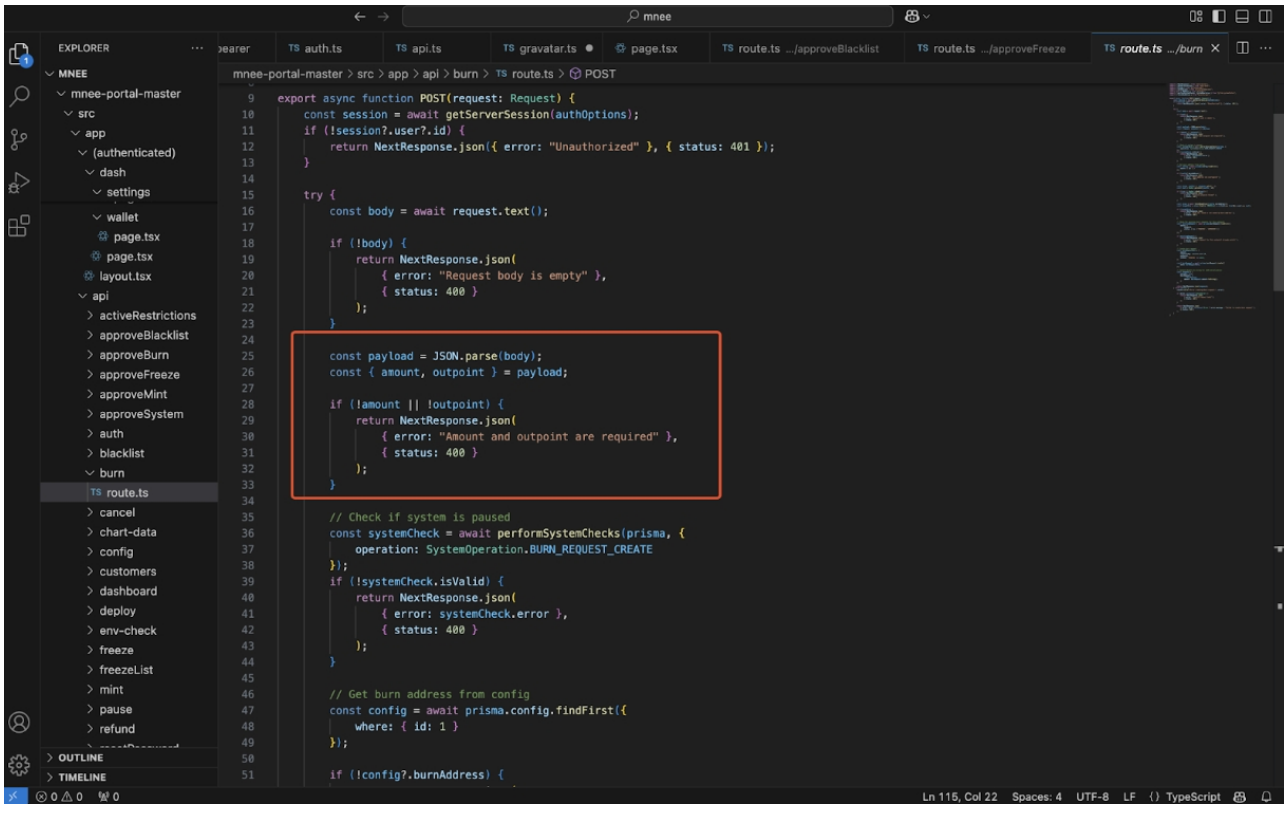
mnee-portal-master/src/app/api/burn/route.ts

Impact

Attackers can submit a negative value or 0 as an amount. This may cause unintended behaviour and possible loss of funds.

Recommendation

Implement checks for the value of the supplied amount to be greater than 0.
Additionally the code should also check if the user trying to execute the request is blacklisted or not.



Unvalidated Blacklisted Request Cancellation

Resolved

Description

Unlike other request types, blacklistRequestId should check if session.user.id matches requestedBy. If a match is found, the request should be denied. Currently, the code is missing this match.

Vulnerable File

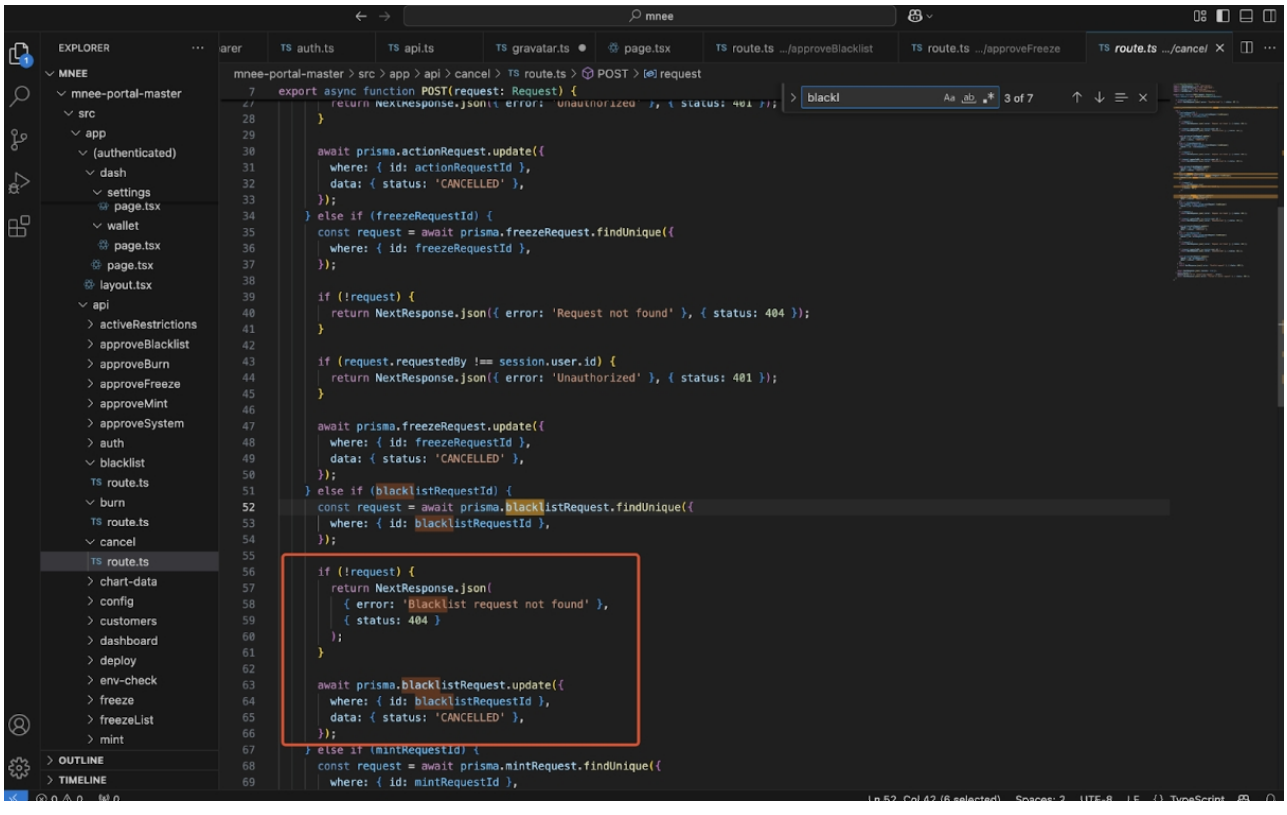
mnee-portal-master/src/app/api/cancel/route.ts

Impact

This might allow the same user who is blacklisted to cancel a request.

Recommendation

Implement a check if session.user.id matches requestedBy. If a match is found, the request should be denied. Currently, the code is missing this match.



Broken Access Control

Resolved

Description

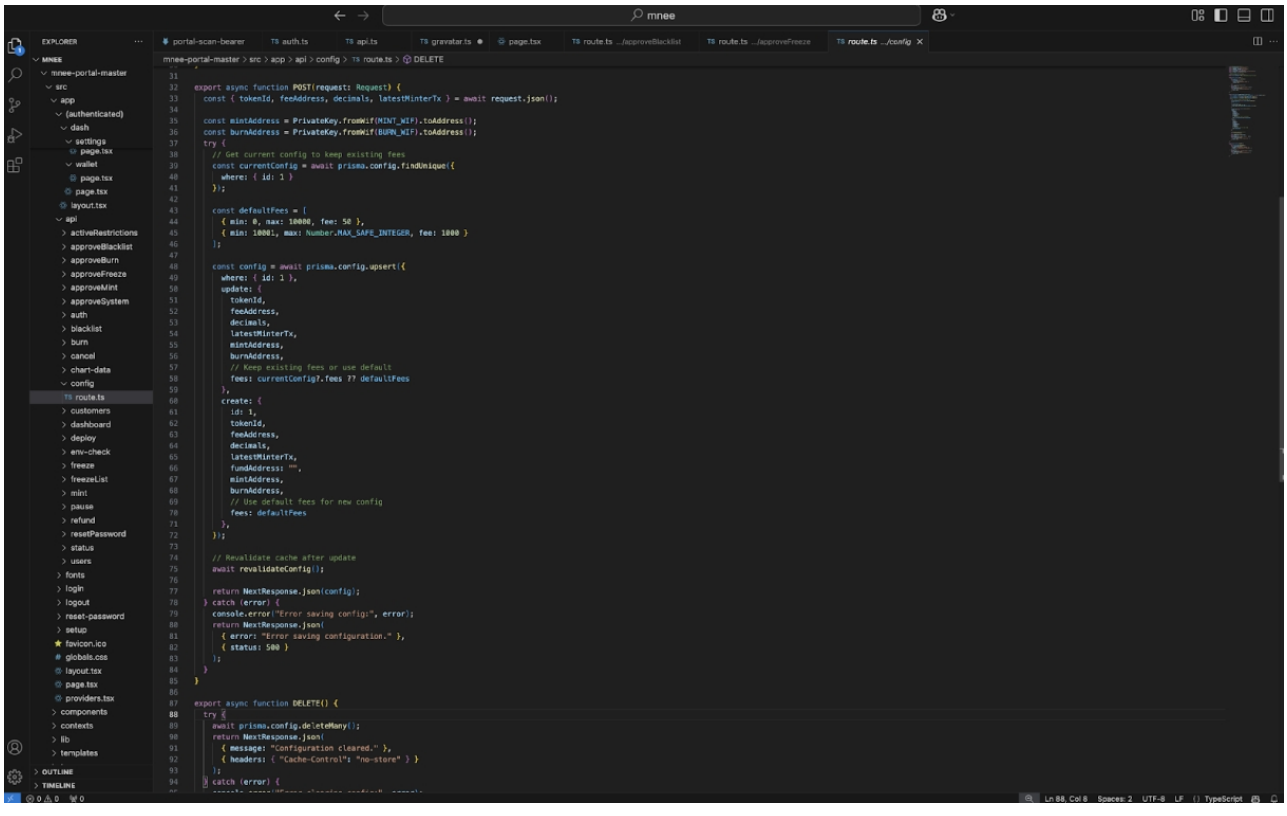
The POST and DELETE routes allow updates and deletions of the configuration without authentication. This means anyone who has access to the API could modify or delete critical configuration settings, including the minting and burning process.

Vulnerable File

mnee-portal-master/src/app/api/config/route.ts

Impact

Anyone who has access to the API could modify or delete critical configuration settings.



```
export async function POST(request: Request) {
  const { tokenId, feeAddress, decimals, latestMinterTx } = await request.json();

  const mintAddress = PrivateKey.fromWif(MINT_WIF).toAddress();
  const burnAddress = PrivateKey.fromWif(BURN_WIF).toAddress();
  try {
    // Get current config to keep existing fees
    const currentConfig = await prisma.config.findUnique({
      where: { id: 1 }
    });

    const defaultFees = [
      { min: 0, max: 10000, fee: 50 },
      { min: 10001, max: Number.MAX_SAFE_INTEGER, fee: 1000 }
    ];

    const config = await prisma.config.upsert({
      where: { id: 1 },
      update: {
        tokenId,
        feeAddress,
        decimals,
        latestMinterTx,
        mintAddress,
        burnAddress,
        // Keep existing fees or use default
        fees: currentConfig?.fees ?? defaultFees
      },
      create: {
        id: 1,
        tokenId,
        feeAddress,
        decimals,
        latestMinterTx,
        fundAddress: "",
        mintAddress,
        burnAddress,
        // Use default fees for new config
        fees: defaultFees
      },
    });

    // Revalidate cache after update
    await revalidateConfig();

    return NextResponse.json(config);
  } catch (error) {
    console.error("Error saving config:", error);
    return NextResponse.json(
      { error: "Error saving configuration." },
      { status: 500 }
    );
  }
}

export async function DELETE() {
  try {
    await prisma.config.deleteMany();
    return NextResponse.json(
      { message: "Configuration cleared." },
      { headers: { "Cache-Control": "no-store" } }
    );
  } catch (error) {
```

Unauthorised Refund

Resolved

Description

The refund logic in the POST request handler allows users to specify any refundAddress, without verifying whether the address belongs to the original sender. This introduces a security vulnerability, as attackers can exploit this mechanism to redirect refunds to their own wallets, stealing funds from legitimate users.

Vulnerable File

mnee-portal-master/src/app/api/refund/route.ts

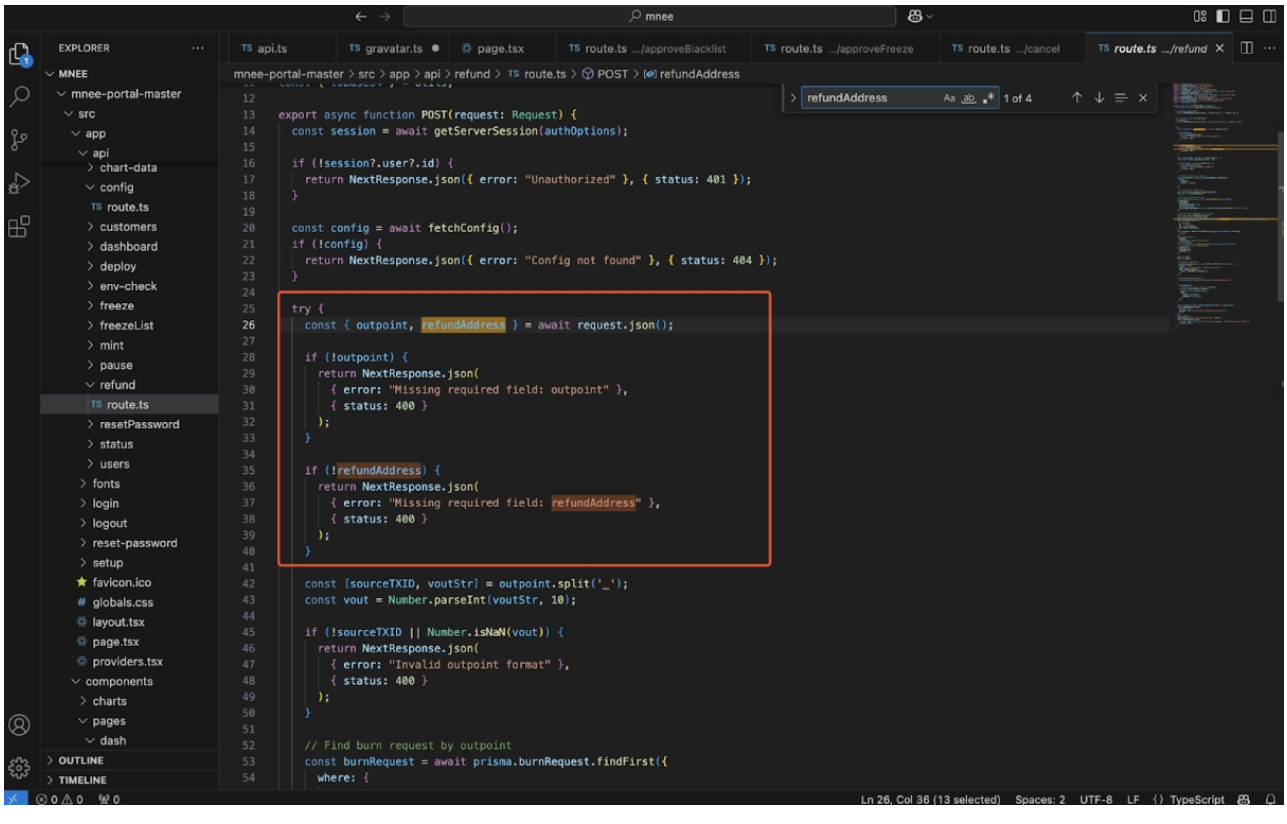
Impact

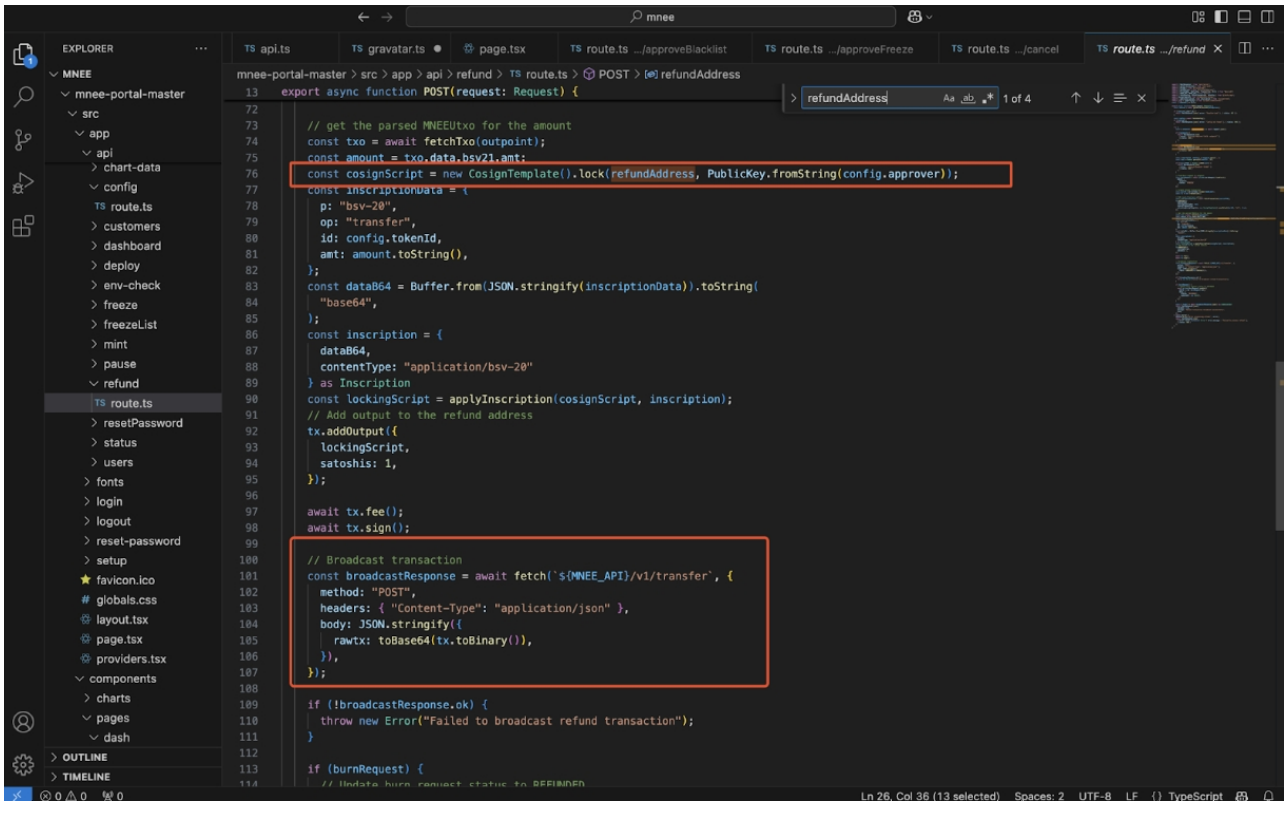
Attackers can request refunds for someone else's UTXO but provide their own address, effectively stealing the funds.

Recommendation

The refund address should be fetched from the original transaction sender, not from user input.

POC




mnee
EXPLORER
MNEE
mnee-portal-master
src
app
api
chart-data
config
route.ts
customers
dashboard
deploy
env-check
freeze
freezeList
mint
pause
refund
route.ts
resetPassword
status
users
fonts
login
logout
reset-password
setup
favicon.ico
globals.css
layout.tsx
page.tsx
providers.tsx
components
charts
pages
dash
OUTLINE
TIMELINE
api.ts
gravatar.ts
page.tsx
route.ts .../approveBlacklist
route.ts .../approveFreeze
route.ts .../cancel
route.ts .../refund
mnee-portal-master > src > app > api > refund > route.ts > POST > refundAddress
refundAddress
1 of 4
export async function POST(request: Request) {
// get the parsed MNEEUtxo for the amount
const txo = await fetchTxo(outpoint);
const amount = txo.data.bsv21.amt;
const cosignScript = new CosignTemplate().lock(refundAddress, PublicKey.fromString(config.approver));
const inscriptionData = {
p: "bsv-20",
op: "transfer",
id: config.tokenId,
amt: amount.toString(),
};
const dataB64 = Buffer.from(JSON.stringify(inscriptionData)).toString(
"base64",
);
const inscription = {
dataB64,
contentType: "application/bsv-20"
} as Inscription
const lockingScript = applyInscription(cosignScript, inscription);
// Add output to the refund address
tx.addOutput({
lockingScript,
satoshis: 1,
});
await tx.fee();
await tx.sign();
// Broadcast transaction
const broadcastResponse = await fetch(`${MNEE_API}/v1/transfer`, {
method: "POST",
headers: { "Content-Type": "application/json" },
body: JSON.stringify({
rawtx: toBase64(tx.toBinary()),
}),
});
if (!broadcastResponse.ok) {
throw new Error("Failed to broadcast refund transaction");
}
if (burnRequest) {
Ln 26, Col 36 (13 selected) Spaces: 2 UTF-8 LF {} TypeScript

Medium Severity Issues

Usage of weak hashing library (MD5)

Resolved

Description

Using a weak hashing library like MD5 increases the risk of data breaches. MD5 is vulnerable to collision attacks, where two different inputs produce the same output, compromising data integrity and security.

Vulnerable File

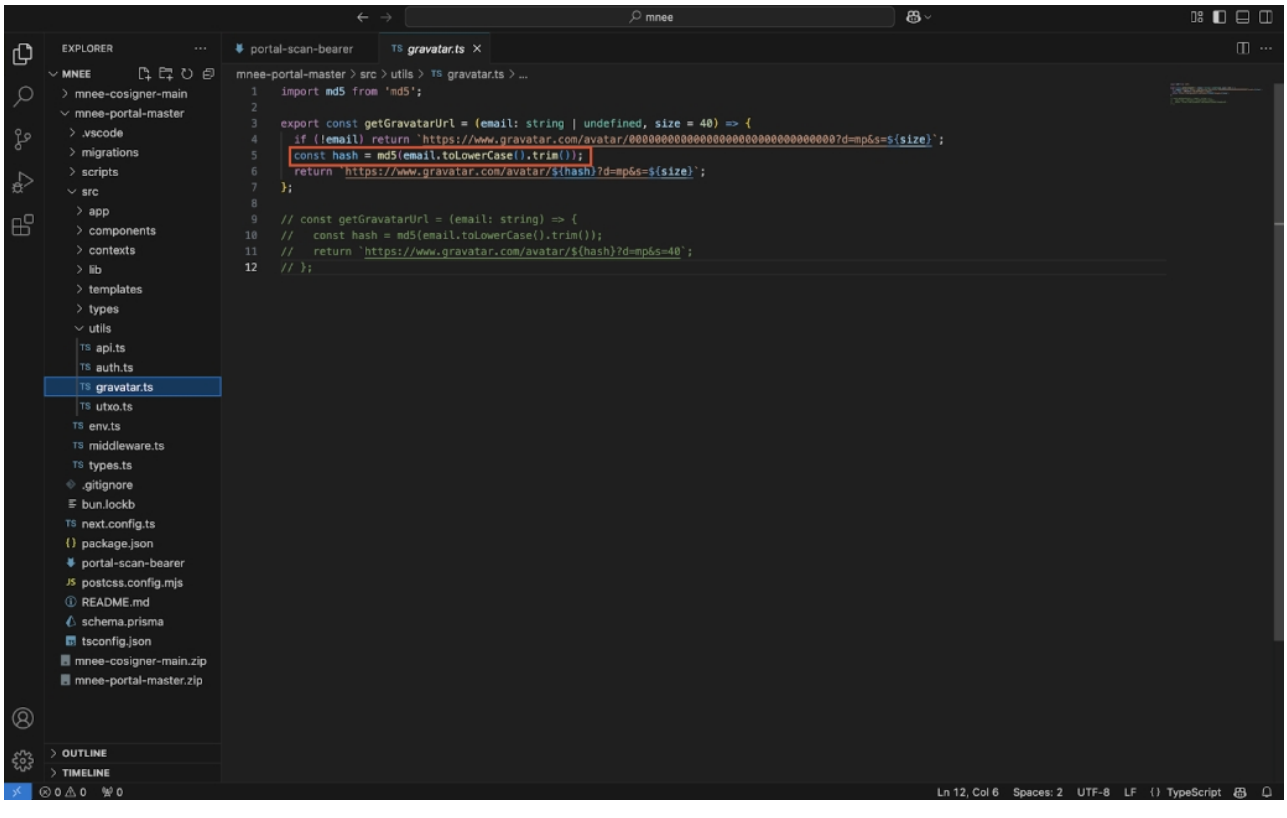
mnee-portal/src/utils/gravatar.ts:5

Impact

1. MD5 is vulnerable to collision attacks, where two different inputs produce the same output, compromising data integrity and security.

Recommendation

- Do not use MD5 for hashing or security purposes. It is no longer considered secure for cryptographic applications.
- Do opt for stronger hashing algorithms like SHA-256 for enhanced security.
  const crypto = require("crypto");
  const key = "secret key";
  const hash = crypto.createHmac("sha256",key).update(user.password).digest('hex');



Insecure Input Validation

Resolved

Description

The application does not validate and sanitize input values. User input should always be validated and sanitized before processing through APIs or functions. The txid is directly fetched from the URL and processed in the function without verification if it is a valid txid.

Vulnerable File

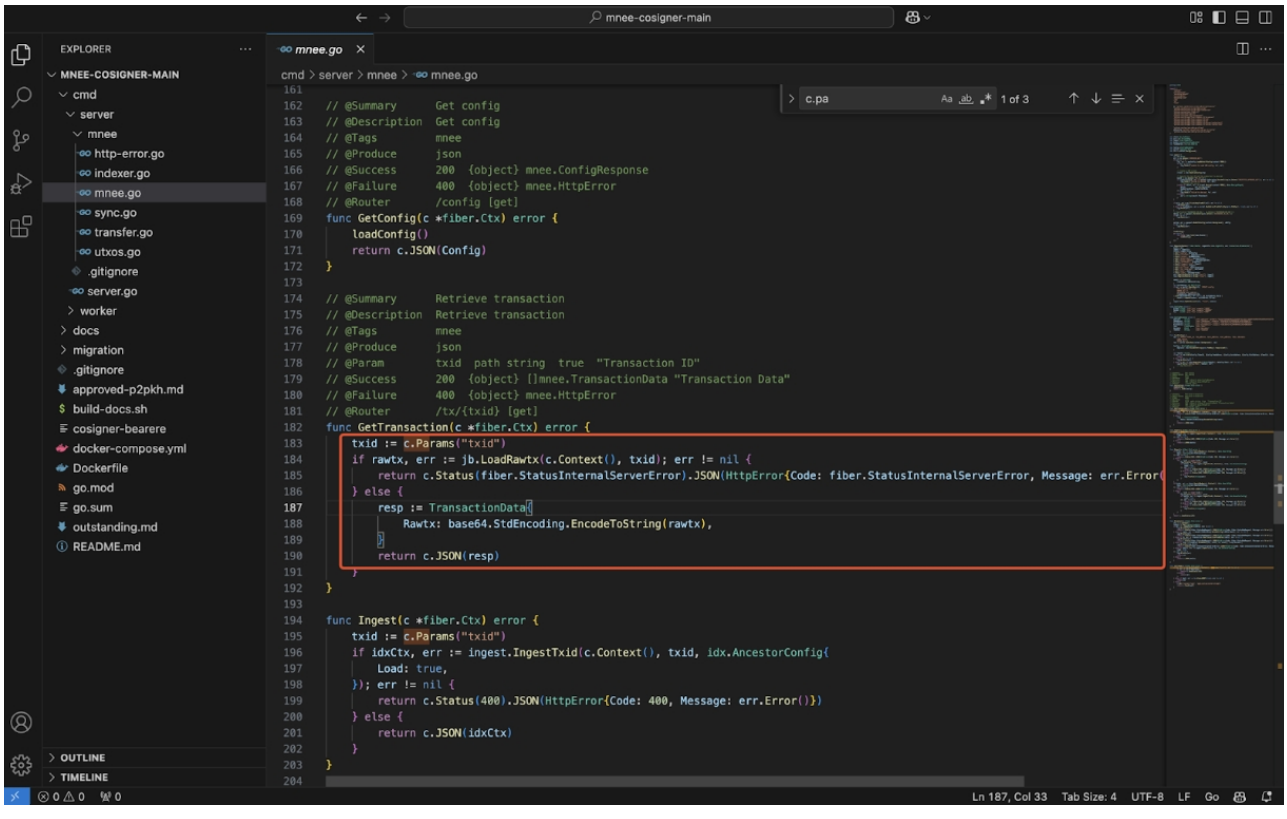
mnee-cosigner/cmd/server/mnee/mnee.go

Impact

It is possible to inject malicious values in the API leading to unexpected behaviour in the application.

Recommendation

Implement a regex of function that checks if input parsed is valid for the specific function and no insecure or malicious value is being parsed.



No Rate Limit

Acknowledged

Description

The application has not implemented a rate limiter to limit users from executing multiple malicious requests on the application server leading to increased bandwidth and server load. All API functions should have a rate limiter defined to limit max 10 requests per minute as a normal user

Vulnerable File

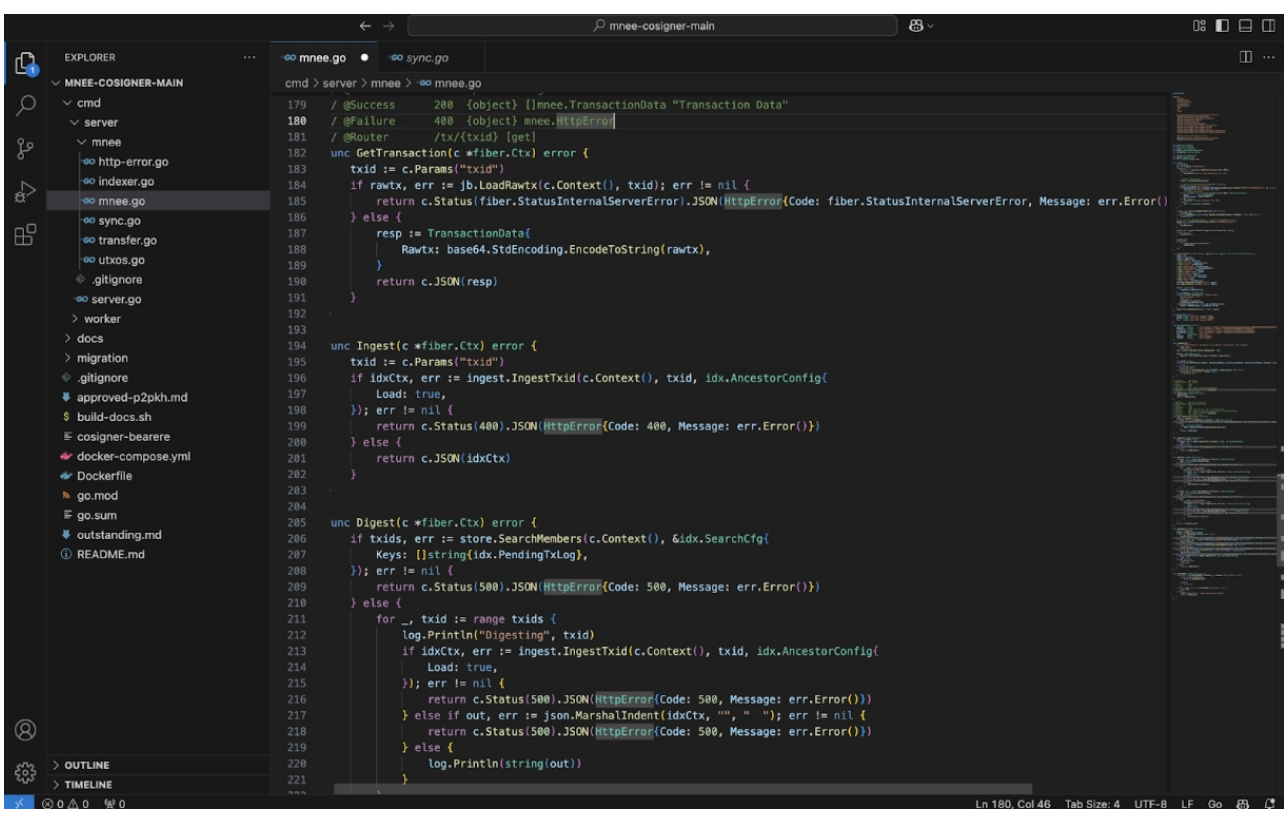
mnee-cosigner/cmd/server/mnee/mnee.go

Impact

Attackers can send bulk data to API Endpoints. Due to lack of limitation, APIs would process this data and at one stage, the exhaustive pressure on the server will increase and very high bandwidth will be consumed leading to a DOS attack.

Recommendation

Implement a rate limiter to limit max up to 10 requests per minute.



MNEE Team's Comment

Rate Limit is handled through our proxy server for now.

Low Severity Issues

Leakage of sensitive information in logger message

Resolved

Description

The application logs sensitive information unwantedly in the user's browser. Sensitive information leakage through logger messages can compromise user privacy and security. This vulnerability occurs when sensitive data, such as personal identifiable information (PII), is included in log messages, making it accessible to unauthorized individuals.

Vulnerable File

mnee-portal-master/

src/app/api/resetPassword/route.ts:18
scripts/check-user.ts:23
scripts/create-user.ts:37
scripts/find-pending.ts:25
scripts/list-approvers.ts:49
scripts/list-approvers.ts:52
scripts/list-approvers.ts:61
src/app/api/approveBlacklist/route.ts:50
src/app/api/approveFreeze/route.ts:44
src/app/api/approveFreeze/route.ts:106
src/app/api/approveFreeze/route.ts:141
src/app/api/resetPassword/route.ts:18
src/components/pages/dash/content/customers/view.tsx:82

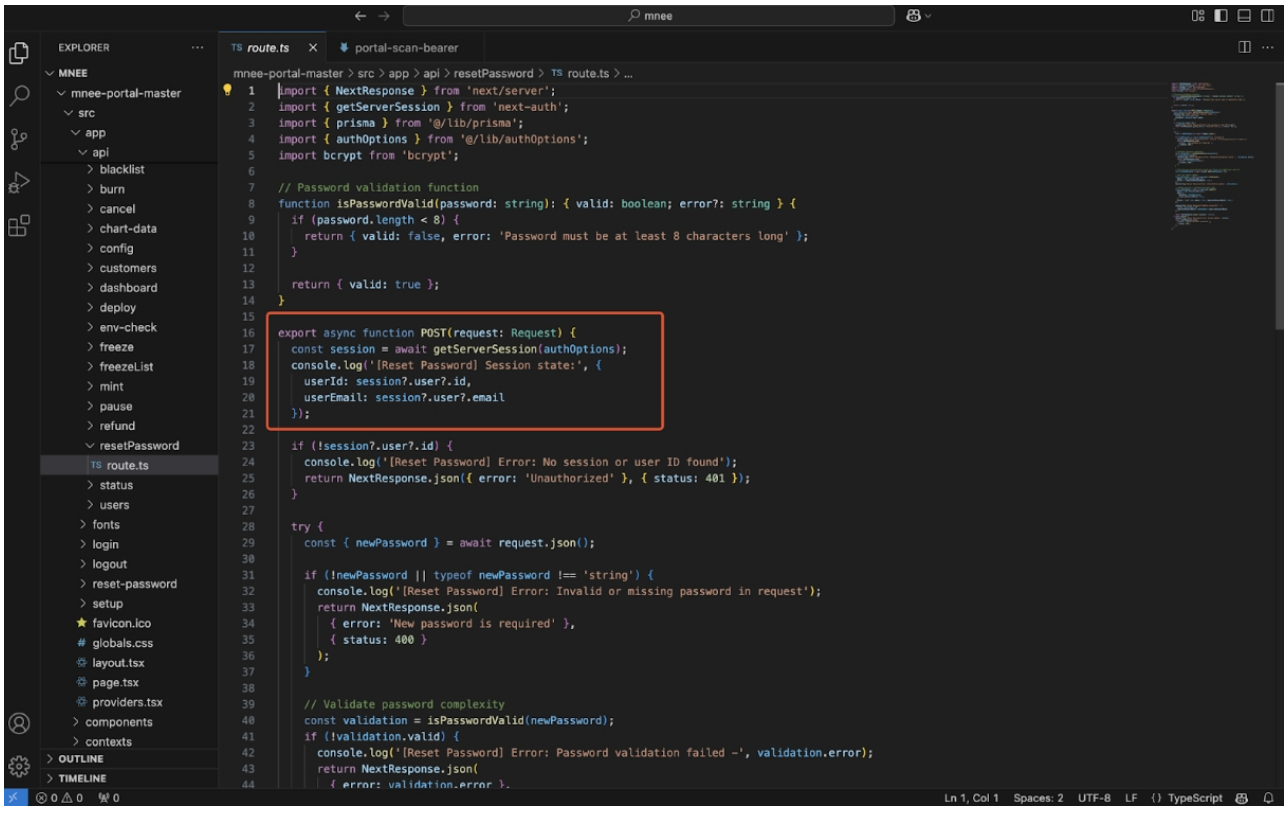
Recommendation

Do not include sensitive data in logger messages. This can lead to unintended exposure of private information.Do use non-sensitive, unique identifiers to reference users in log messages. This approach maintains user privacy while still allowing for effective logging. logger.info(`User is: ${user.uuid}`)

Impact

Logging sensitive information can allow attackers to steal logged information under certain conditions.

POC


mnee
EXPLORER
route.ts
portal-scan-bearer
mnee-portal-master > src > app > api > resetPassword > route.ts > ...
MNEE
mnee-portal-master
src
app
api
blacklist
burn
cancel
chart-data
config
customers
dashboard
deploy
env-check
freeze
freezeList
mint
pause
refund
resetPassword
route.ts
status
users
fonts
login
logout
reset-password
setup
favicon.ico
globals.css
layout.tsx
page.tsx
providers.tsx
components
contexts
OUTLINE
TIMELINE
import { NextResponse } from 'next/server';
import { getServerSession } from 'next-auth';
import { prisma } from '@/lib/prisma';
import { authOptions } from '@/lib/authOptions';
import bcrypt from 'bcrypt';
// Password validation function
function isPasswordValid(password: string): { valid: boolean; error?: string } {
if (password.length < 8) {
return { valid: false, error: 'Password must be at least 8 characters long' };
}
return { valid: true };
}
export async function POST(request: Request) {
const session = await getServerSession(authOptions);
console.log('[Reset Password] Session state:', {
userId: session?.user?.id,
userEmail: session?.user?.email
});
if (!session?.user?.id) {
console.log('[Reset Password] Error: No session or user ID found');
return NextResponse.json({ error: 'Unauthorized' }, { status: 401 });
}
try {
const { newPassword } = await request.json();
if (!newPassword || typeof newPassword !== 'string') {
console.log('[Reset Password] Error: Invalid or missing password in request');
return NextResponse.json(
{ error: 'New password is required' },
{ status: 400 }
);
}
// Validate password complexity
const validation = isPasswordValid(newPassword);
if (!validation.valid) {
console.log('[Reset Password] Error: Password validation failed -', validation.error);
return NextResponse.json(
Ln 1, Col 1
Spaces: 2
UTF-8
LF
TypeScript

Middleware Does not Handle Token Expiration

Resolved

Description

The middleware only checks if a token exists (!!token), but doesn't verify if it is expired.

Vulnerable File

mnee-portal-master/src/middleware.ts

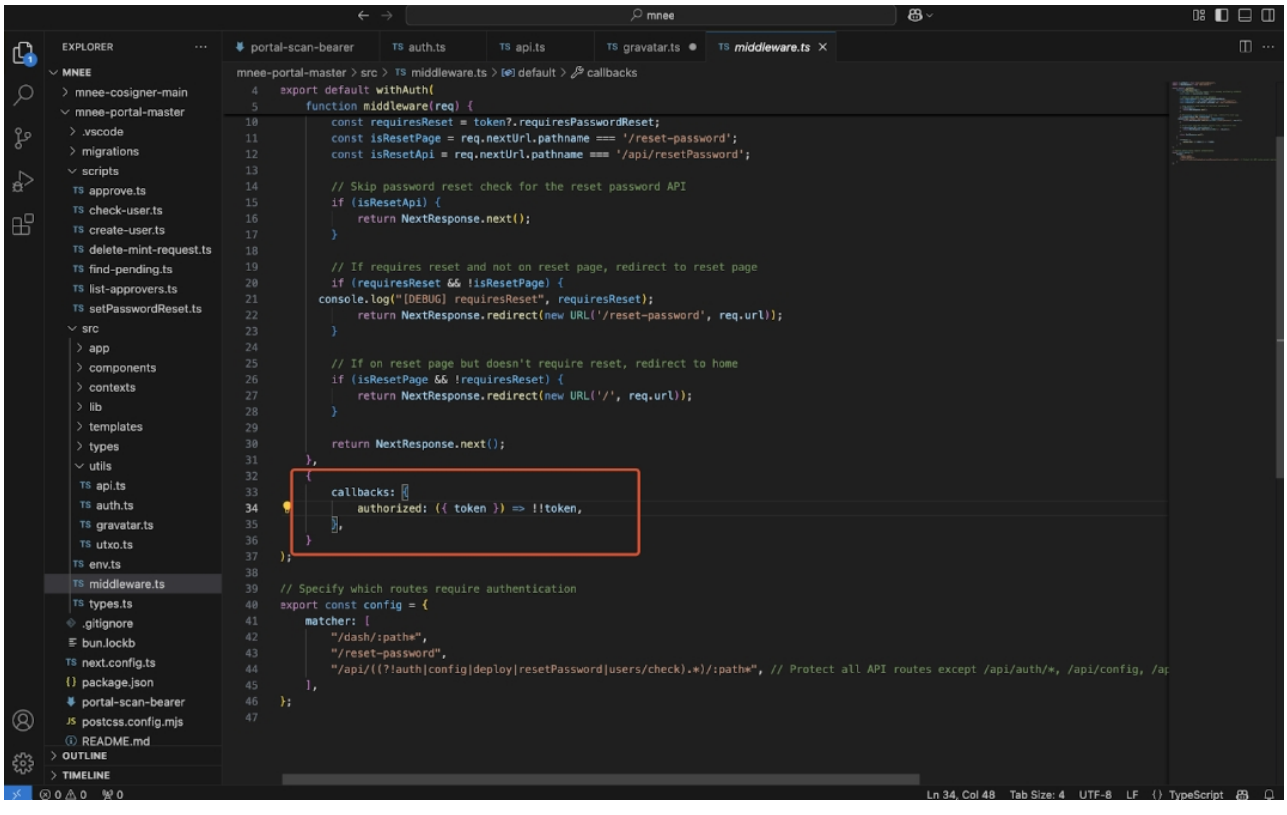
Impact

It is possible to use old generated tokens if tokens are not expired. Old password reset tokens if re-used by hackers could lead to severe security issues.

Recommendation

Modify the authorized callback to check token expiration

POC



User Enumeration

Resolved

Description

The application API prints generic error message if a supplied user email is not found in the database. Attackers can brute-force the email address field and identify valid registered email addresses.

Vulnerable File

mnee-portal-master/scripts/check-user.ts

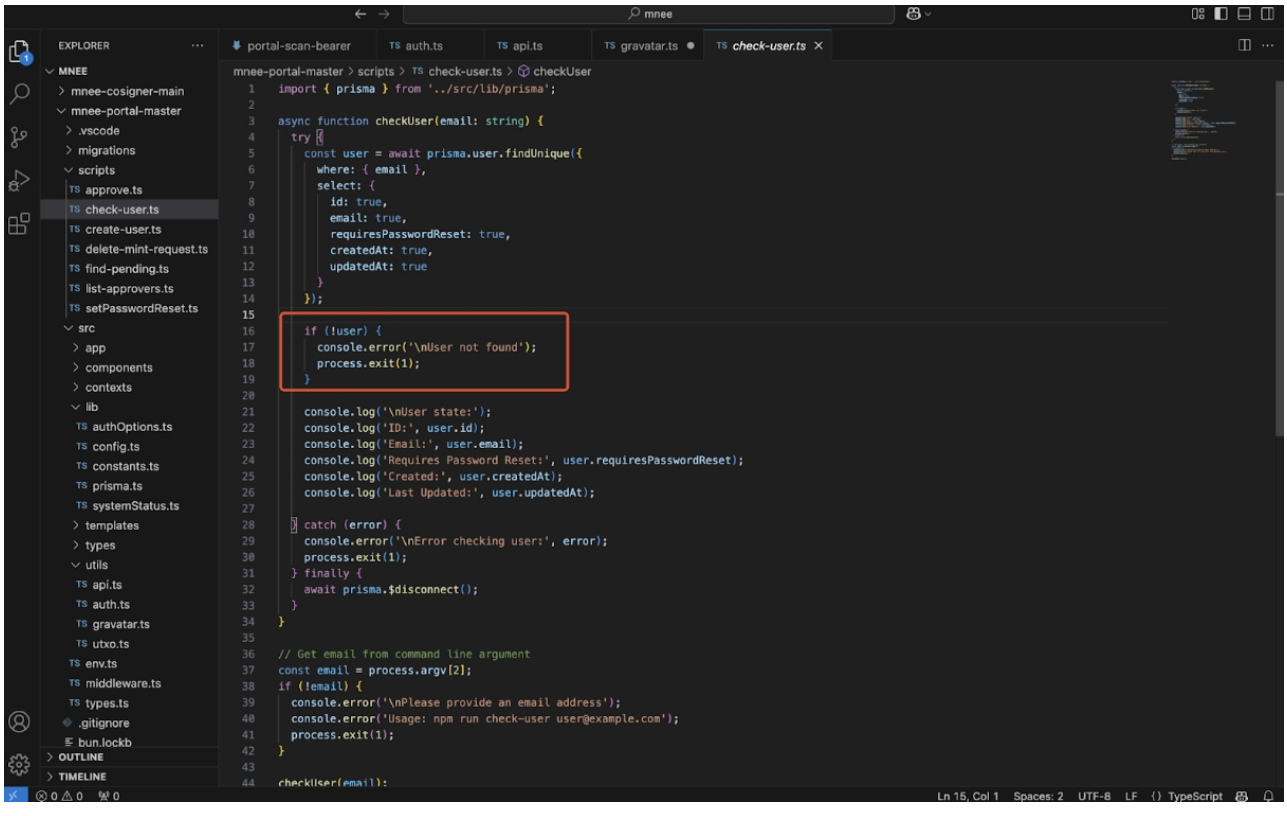
Impact

Generic error messages can allow attackers to identify valid user email addresses. Using this information attackers could further plan their attack path against individual users.

Recommendation

Print generic error message

POC



Weak Password Check

Resolved

Description

The application has a isPasswordValid function which only checks if password length is greater than 8 characters. It should also check for special characters, numbers, common password patterns, and numeric characters.

Vulnerable File

mnee-portal-master/scripts/create-user.ts
mnee-portal-master/src/app/reset-password/page.tsx

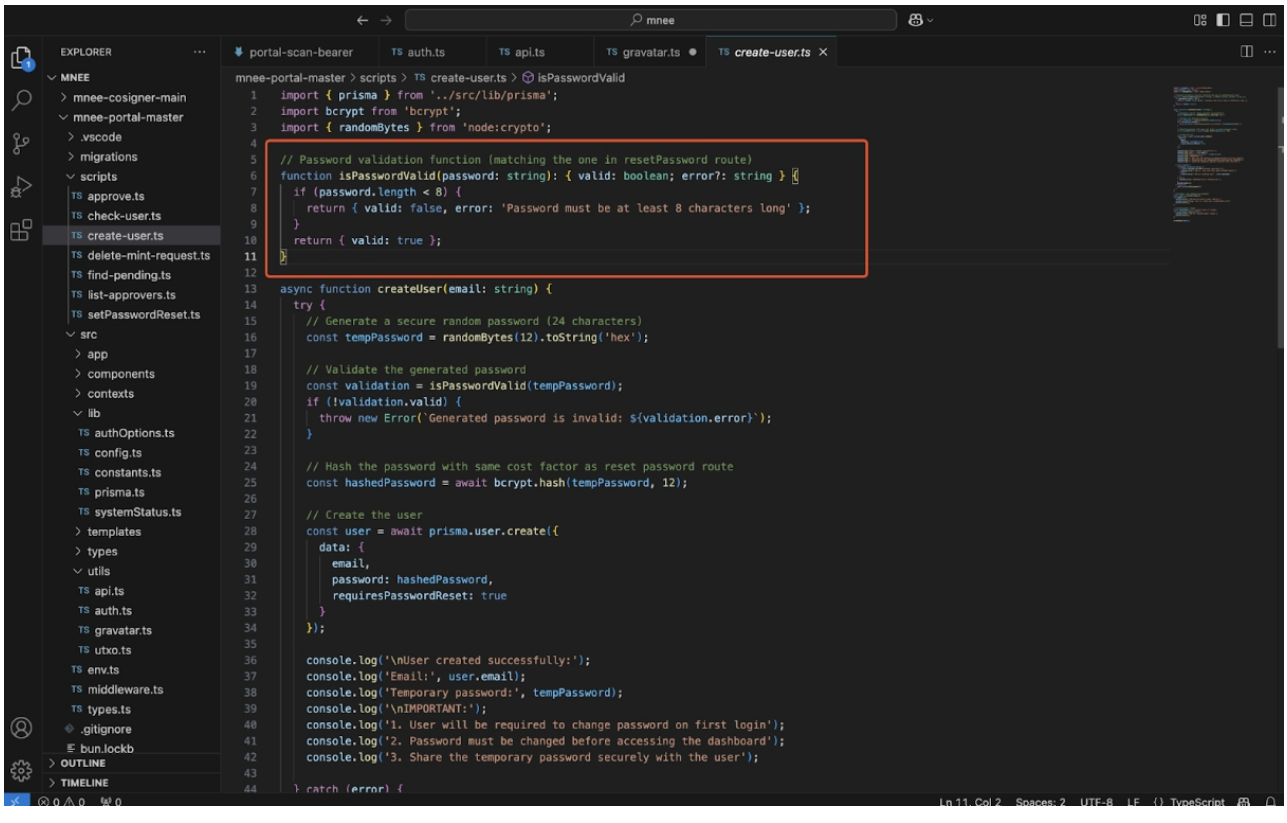
Impact

Users might create 8 length passwords (example: test@123 or john@123) but it would still be weak. It is important to implement checks for special characters, numbers, common password patterns, and numeric characters to create a strong password.

Recommendation

Implement checks for special characters, numbers, common password patterns, and numeric characters

POC



Improper Error Handling

Resolved

Description

The application currently logs API errors without sanitization. If an API request fails, the error message could expose internal details.

Vulnerable File

mnee-portal-master/src/contexts/CustomerContext.tsx

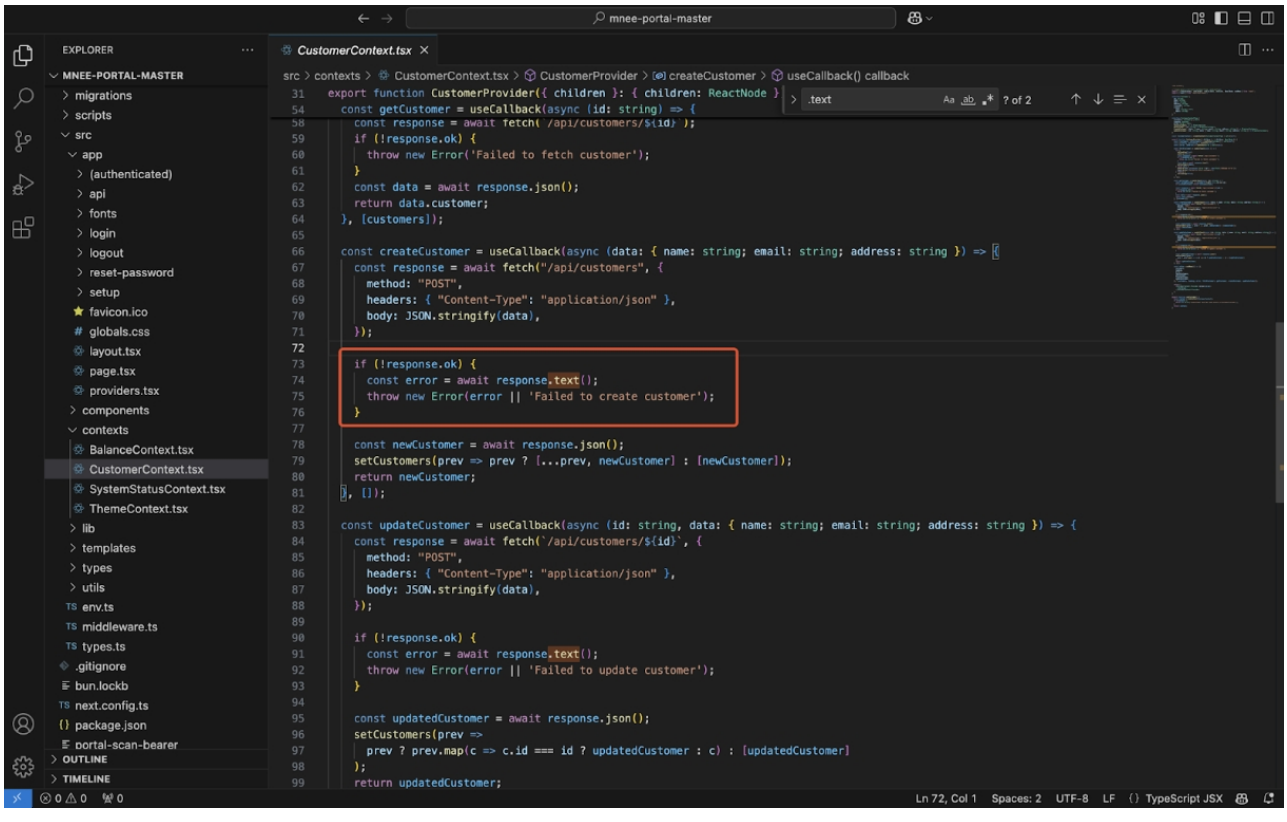
Impact

Attackers can gain access to sensitive information through API errors.

Recommendation

Never expose raw API errors to users. Instead, return a generic error message

POC



Outdated Server Components

Resolved

Description

The application is using github.com/redis/go-redis/v9 v9.7.0 which is outdated and vulnerable to CVE-2025-2992.

Vulnerable File

go.mod

Impact

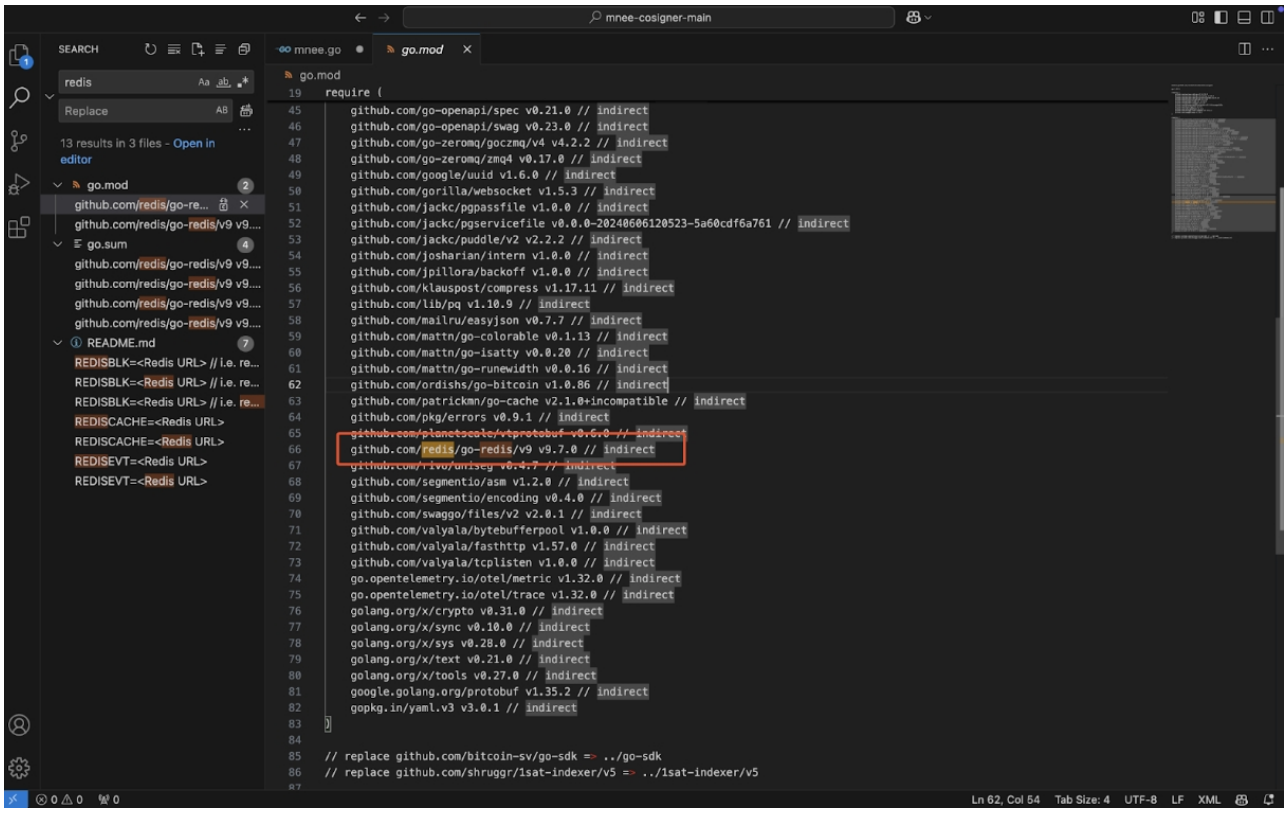
Update the dependency to the latest stable secure version.

Recommendation

The dependency has know exploit available. Attackers can exploit the known vulnerability.

POC

The dependency has know exploit available. Attackers can exploit the known vulnerability.



Closing Summary

In this report, we have considered the security of the MNEE. We performed our audit according to the procedure described above.

Some issues of High, medium, low, and Informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits Dapp audit is not a security warranty, investment advice, or an endorsement of the MNEE. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multi-step process. One audit cannot be considered enough.

We recommend that the MNEE Team put in place a bug bounty program to encourage further analysis of the source code by other third parties.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

October 2025

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com