in collaboration with

Softwarica | Coventry University

SECURITY ASSESSMENT REPORT
ST6005CEM SECURITY CW2

IMAGE PURCHASED AND DOWNLOADED

IMAGE UPLOADED AND WATERMARKED

SUBMITTED BY:-
KARAN BOHARA
COVENTRY ID:-
13703532
COLLEGE ID:-
220404

SUBMITTED TO:-
ARYA
POKHAREL

Abstract

"Digital Vault" was engineered to serve as a comprehensive and highly secure e-commerce platform for the sale of digital assets. The project addresses the critical need for a marketplace that protects both creator-owned intellectual property and customer financial data by integrating a multi-layered, defence-in-depth security architecture. Key protective measures include robust Multi-Factor Authentication (MFA) with one-time-use recovery codes, strict Role-Based Access Control (RBAC), and secure, PCI-compliant transaction processing via Stripe. The platform leverages the MERN stack (MongoDB, Express.js, React, Node.js) to deliver a seamless user experience, with a backend that also enforces advanced password policies, rate limiting, and content protection through automated watermarking. This report details the security-first principles that guided the project's development, analyses the implementation of each security control, and demonstrates the platform's readiness as a trustworthy and scalable solution for digital commerce.

Table of abbreviation

Abbreviation	Definition
API	Application Programming Interface
CORS	Cross-Origin Resource Sharing
CSRF	Cross-Site Request Forgery
EXIF	Exchangeable Image File Format
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JWT	JSON Web Token
MFA	Multi-Factor Authentication
MIME	Multipurpose Internet Mail Extensions
NoSQL	Not only SQL
PCI DSS	Payment Card Industry Data Security Standard
PoC	Proof of Concept
RBAC	Role-Based Access Control
SSL/TLS	Secure Sockets Layer / Transport Layer Security
TOTP	Time-based One-Time Password
XSS	Cross-Site Scripting

Figure 1: Table of Abbreviation

Table of Contents

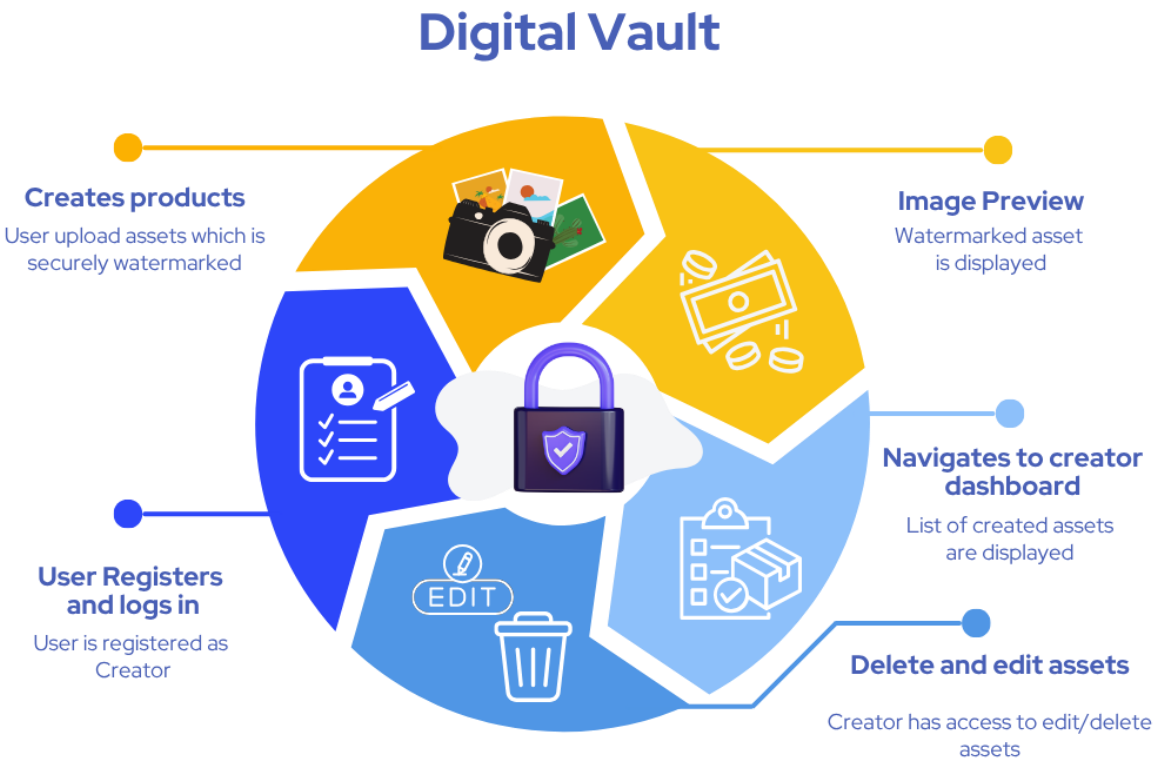
List of Figures

Introduction

"Digital Vault" was developed to provide a secure platform for creators to monetize digital assets and for customers to purchase them confidently. It addresses key e-commerce security challenges by protecting intellectual property and securing user data. The platform features multi-layered security, including Multi-Factor Authentication (MFA), email verification, secure session handling with JWT, and password hashing with bcrypt.js.

Figure 2: Introduction to Digital Vault

A core security feature of "Digital Vault" is protecting digital assets through automated watermarking of previews and a secure download system for verified buyers only. It also prioritizes a smooth user experience, with a React-based frontend offering a responsive interface creators get a full-featured dashboard, and customers can easily browse and purchase assets. The backend, built with Node.js and Express, manages APIs, security, and database operations. Combining strong security with user-friendly design, "Digital Vault" offers a complete solution for today’s digital marketplace.



System Design

The system architecture of "Digital Vault" is meticulously engineered to provide a high-security environment for e-commerce while maintaining a seamless and intuitive user experience. The platform is built on a modern full-stack model, which cleanly separates the client-side and server-side components. This separation not only enhances security by creating a clear boundary between the user interface and the core business logic but also improves maintainability and scalability.

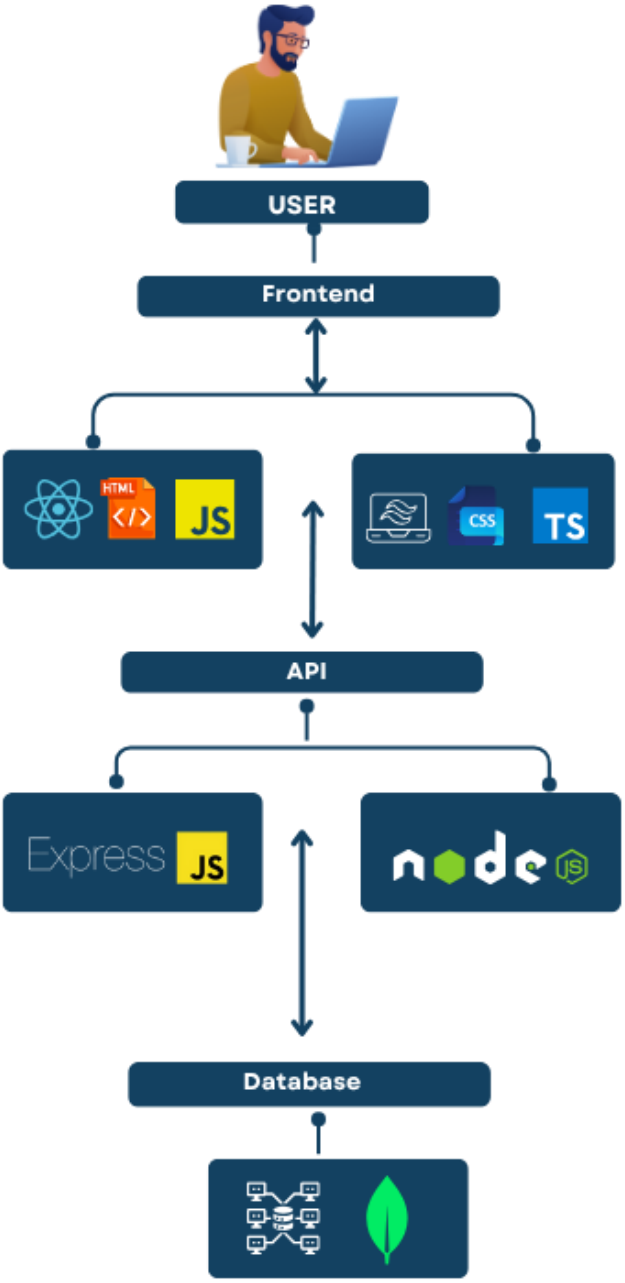


Figure 3: System Design

The frontend is a dynamic Single-Page Application (SPA) developed with React. It is responsible for all user-facing interactions, providing a responsive and interactive interface where creators can manage their products and customers can browse, purchase, and download digital assets. The backend is a powerful REST API powered by Node.js and Express, which handles all core operations, including user authentication, secure payment processing, product management, and asset protection.

Software Details

"Digital Vault" was developed using the MERN stack, chosen for its performance, scalability, and unified JavaScript environment. MongoDB stores user data, product details, and activity logs, while Node.js and Express power the backend API, managing business logic and security. Important security features rely on third-party libraries for HTTPS setup, JWT session management, and bcryptjs password hashing. The frontend was built with React, using custom CSS to create a unique, responsive, and user-friendly interface. Features include real-time validation on the registration form and a multi-tab dashboard for users. Both the frontend and backend run locally over HTTPS with a self-signed SSL certificate to encrypt all data exchanged and protect against interception. Tracking third-party dependencies is essential for security, and the following tables provide a detailed list of the packages used in both the frontend and backend of "Digital Vault."

S.N.	Package Name	Purpose
1	**react**	The core JavaScript library for building the component-based user interface.
2	**react-dom**	Provides the specific methods for interacting with the browser's DOM.
3	**react-router-dom**	Handles all client-side routing and navigation between pages (e.g., `/login`, `/dashboard`).
4	**react-scripts**	Includes the scripts and configurations needed to run the Create React App project (`npm start`, `npm run build`).
5	**styled-components**	Allows writing CSS styles directly within your JavaScript components for better organization.
6	**framer-motion**	A powerful library for creating fluid animations and transitions in the user interface.
7	**lucide-react & react-icons**	Provide a comprehensive set of high-quality icons to improve the UI design.
8	**react-confetti & react-use**	Utility libraries that can be used for special effects (like confetti on success) and other common React tasks.
9	**@testing-library/...**	A suite of packages included by default for testing your React components.
10	**web-vitals**	A library for measuring and reporting on your application's performance.

Figure 4: Packages and Libraries Used in Frontend

SN	Package Name	Purpose
1	**express**	The core web framework for building your Node.js API.
2	**mongoose**	An Object Data Modeler (ODM) to create schemas and interact with MongoDB.
3	**dotenv**	Loads environment variables from your `.env` file to keep secrets safe.
4	**jsonwebtoken**	Creates and verifies JSON Web Tokens (JWT) for secure user authentication.
5	**bcryptjs**	Securely hashes and salts user passwords before storing them.
6	**cors**	Enables Cross-Origin Resource Sharing for frontend-backend communication.
7	**helmet**	Sets secure HTTP headers as a first line of defense against attacks.
8	**express-rate-limit**	Protects against brute-force attacks by limiting requests.
9	**express-mongo-sanitize**	Sanitizes user input to prevent NoSQL injection attacks.
10	**stripe**	Integrates with Stripe for payments and handles webhooks.
11	**nodemailer**	Sends transactional emails (e.g., verification, password resets).
12	**speakeasy & qrcode**	Generates TOTP secrets and QR codes for MFA.
13	**multer**	Handles file uploads with validation.
14	**sharp**	Processes images (watermarks, EXIF stripping).
15	**nodemon**	Auto-restarts the server during development.
16	**https**	Encrypts all communication between browsers and servers using SSL/TLS

Figure 5: Packages and Libraries Used in Backend

The project followed a "backend-first" development approach. The API and database schemas were built and rigorously tested with Postman first to ensure all business logic and security controls were sound before the frontend was developed. This methodology proved effective in creating a stable and secure foundation for the application.

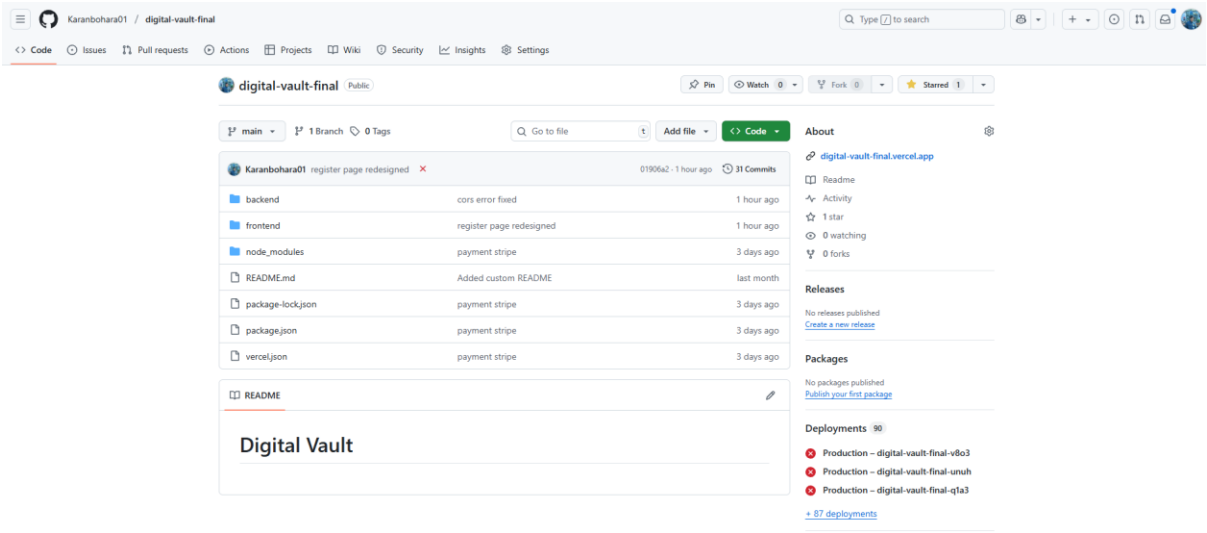


Figure 6: Commits on the GitHub Repo

Security by Design Principles

The architecture of "Digital Vault" follows modern security standards, especially the OWASP Top 10, which highlights common web application risks like injection attacks and broken access controls. By adopting a "Security by Design" approach, security was built into every stage of development through risk analysis, threat modelling, and secure design patterns. To defend against real-time threats, key security measures were built into the app's middleware, simulating a Web Application Firewall (WAF) to filter and monitor traffic. A strict CORS policy adds another layer of protection, allowing requests only from the trusted frontend URL (e.g., https://localhost:3000) and blocking all others even similar-looking ones like http://localhost:3000 to prevent unauthorized access.

```
app.use(cors({ origin: process.env.FRONTEND_URL, credentials: true }));
app.set('trust proxy', 1);
```

To prevent unauthorized embedding and bandwidth theft, "Digital Vault" uses server-side hotlink protection. A middleware checks the Referer header of image requests and compares it to the trusted frontend URL. If the request comes from another domain, it returns a 403 Forbidden error, blocking the image from being displayed elsewhere.

```
backend > middleware > JS hotlinkProtection.js > ...
1   // backend/middleware/hotlinkProtection.js (Updated)
    Tabnine | Edit | Explain
2   const hotlinkProtect = (req, res, next) => {
3       const referer = req.headers.referer;
4       const allowedReferer = process.env.FRONTEND_URL;
5
6       // Allow if the request is from our own frontend OR has no referer (direct access)
7       if (!referer || referer.startsWith(allowedReferer)) {
8           next();
9       } else {
10          console.log(`[HOTLINK BLOCKED] Request from: ${referer}`);
11          res.status(403).send('Hotlinking is not permitted.');
12      }
13  };
14  module.exports = hotlinkProtect;
```

Finally, each API route on the backend was designed to only accept specific HTTP methods (e.g., GET, POST, PUT). This is a simple but effective security measure that ensures an endpoint designed to fetch data cannot be used to submit or delete it, reinforcing the intended logic of the application.

```
const express = require('express');
const router = express.Router();
const { createProduct, getProducts, getProductById, getMyProducts, downloadProductFile, updateProduct, deleteProduct } = require('../controllers/
const { protect, authorize } = require('../middleware/authMiddleware');
const upload = require('../middleware/uploadMiddleware');


router.get('/', getProducts);
router.get('/my-products', protect, authorize('creator'), getMyProducts);
router.get('/:id', getProductById);


// 2. Add the upload.single('image') middleware to this route
router.post('/', protect, authorize('creator'), upload.single('image'), createProduct);
router.get('/:id/download', protect, authorize('customer'), downloadProductFile);
router.put('/:id', protect, authorize('creator'), updateProduct);
router.delete('/:id', protect, authorize('creator'), deleteProduct);


module.exports = router;
```

A major consideration in any project is the security of third-party dependencies. If an unfound vulnerability were discovered in a package used by "Digital Vault," it could lead to a zero-day exploit. To mitigate this risk, continuous maintenance, regular dependency updates (using tools like npm audit), and monitoring is essential.

Checklist

"Digital Vault" was designed with a strong focus on security to create a trustworthy marketplace for digital assets. While perfect security is an ongoing goal, a wide range of measures were implemented to make the platform as secure as possible. These include strong password policies, asset watermarking, secure payment handling, and strict access controls. Every feature was built with a security-first mindset to protect both creators and users. Documenting these

efforts is essential to understand the platform's overall security posture. The following checklist outlines all key features that contribute to making "Digital Vault" a safe and reliable environment for digital commerce.

Application's Features

S.N.	Feature	Requirements	Implemented	How it was achieved	Why it was done
1	**User-Centric Design**	Responsive, modern, and intuitive interface for all user roles.	✅	• Custom CSS • Component-based React architecture	Enhance user experience.
2	**User Registration & Authentication**	Secure registration, email verification, and a robust login process.	✅	• React & Node.js • Nodemailer	Register new users, verify email authenticity, provide secure access.
3	**Product Listings**	A marketplace for creators to list and sell unique digital products.	✅	• MERN Stack (CRUD)	Provide the core business logic of the platform.
4	**Customizable User Profiles & Dashboard**	A personalized dashboard for each role (Customer, Creator, Admin).	✅	• React & Node.js	Provide role-specific functionality (view orders, manage products, view logs).
5	**Secure E-commerce Flow**	Secure payment processing and delivery of digital goods.	✅	• Stripe Integration	Process payments securely and fulfill orders.
6	**Detailed Activity Logging**	Log all critical user and system actions for auditing.	✅	• Custom logService • Mongoose	Track all security-relevant activity.

Figure 7: Application's Features

Password Requirements

S.N.	Feature (Client + Server Side)	Requirements	Implemented	How it was achieved	Why it was done
1	**Password Length**	Minimum and maximum password length.	✅	`minlength: 8` in Mongoose schema	Improve security.
2	**Password Complexity**	Include uppercase, lowercase, numbers, and special characters.	✅	Real-time validation in React	Improve security.
3	**Password History**	Users cannot reuse their last 5 recent passwords.	✅	By storing previous password hashes	Prevent recycling of compromised passwords.
4	**Password Expiry**	A policy that prompts users to change their password periodically (90 days).	✅	`passwordChangedAt` timestamp checked by middleware	Mitigate the risk of compromised passwords.
5	**Account Lockout**	A mechanism that locks user accounts after 10 failed login attempts.	✅	`failedLoginAttempts` counter in loginUser controller	Prevent brute-force attacks.
6	**Password Strength Assessment**	Provide users with real-time feedback on password strength.	✅	Custom React component with a checklist	Improve user experience and encourage strong passwords.

Figure 8: Password Requirements

Audit Trial

S.N.	Feature	Requirements	Implemented	How it was achieved	Why it was done
1	**Audit Trail Record**	An admin-friendly audit trail that logs all user activities.	✅	`Log` model and a dedicated Admin Dashboard tab	Track user activity for security and forensics.
2	**User Access Level**	3 levels of user access (Customer, Creator, Admin).	✅	Custom `authorize()` middleware and protected routes	Give overall control to admin and enforce least privilege.

Figure 9: Audit Trial

Session Management

S.N.	Feature	Requirements	Implemented	How it was achieved	Why it was done
1	**Session Creation**	A secure session is created when a user logs in.	✅	Using JSON Web Tokens (`JWT`)	Provide secure login info once per session.
2	**Session Handling**	Proper headers for session handling and protection against hijacking.	✅	`protect` middleware verifying the JWT on every request	Improve Security.
3	**Session Expiration**	Set session expiration to enhance security.	✅	JWT `expiresIn: '30d'` policy	User does not have to write credentials on every visit.
4	**Secure User Information**	All critical information such as passwords should be stored in an encrypted form in the database.	✅	Using `bcryptjs` with a salt	Improve Security.

Figure 10: Session Management

Extra Security Measures

S.N.	Feature	Requirements	Implemented	How it was achieved	Why it was done
1	**MFA (Multi-Factor Authentication)**	Implement a second factor of authentication.	✅	Using `speakeasy` for TOTP with recovery codes	Drastically improve account security.
2	**CAPTCHA**	Prevent automated bots from registering accounts.	✅	Google reCAPTCHA v2 on the registration form	Prevent spam and automated attacks.
3	**Hide Source Files**	When a user tries to view the source, nothing is seen.	✅	`GENERATE_SOURCEMAP=false` in .env	Improve Security.
4	**Prevent XSS attacks**	Take measures against Cross-Site Scripting.	✅	Using `helmet` to set secure headers	Improve Security.
5	**Sanitize Data**	Sanitize user data against NoSQL injection.	✅	Custom `sanitizeRequest` middleware	Improve Security.
6	**Secure Frontend & Backend**	Host React app and Node.js API in HTTPS.	✅	Using self-signed SSL certificates with `mkcert`	Improve Security.
7	**Asset Protection**	Protect creator assets from theft.	✅	Watermarking with `sharp` and Hotlink Prevention	Protect intellectual property.

Figure 11: Security Measures

Making Digital Vault Secure

The security features mentioned in the previous checklist were implemented to ensure that "Digital Vault" is as secure as possible. As achieving a perfectly secure application is an endless game of cat and mouse, the following features make "Digital Vault" a robust and trustworthy marketplace for digital assets. From advanced password policies to extra measures like asset watermarking and secure payment handling, every component was built with a security-first mindset.

User Authentication & Verification

The platform's security relies on a strong multi-layered authentication system. It uses Multi-Factor Authentication (MFA) with the TOTP algorithm, powered by the speakeasy library. Users scan a QR code to link an authenticator app, and future logins require both a password and a 6-digit time-based code for added protection.

```
const mfaSetup = async (req, res) => {
    try {
        const user = await User.findById(req.user.id);

        const secret = speakeasy.generateSecret({
            name: `DigitalVault (${user.email})`,
        });

        user.mfaTempSecret = secret.base32;
        await user.save();

        qrcode.toDataURL(secret.otpauth_url, async (err, data_url) => {
            if (err) {
                await logActivity(user._id, 'MFA_QR_GENERATION_FAIL', 'error', {
                    error: err.message,
                    method: req.method,
                    ipAddress: req.ip,
                });
                throw new Error('Could not generate QR code');
            }

            await logActivity(user._id, 'MFA_SETUP_INITIATED', 'info', {
                method: req.method,
                ipAddress: req.ip,
            });

            res.json({ qrCodeUrl: data_url });
        });
```

Figure 12: MFA Setup

```
const mfaVerify = async (req, res) => {
    try {
        const { token } = req.body;
        const user = await User.findById(req.user.id);

        if (!user.mfaTempSecret) {
            await logActivity(user._id, 'MFA_VERIFY_FAIL', 'warn', {
                method: req.method,
                ipAddress: req.ip,
                reason: 'Temp secret not found',
            });
            return res.status(400).json({ message: 'MFA setup has not been initiated.' });
        }

        const isVerified = speakeasy.totp.verify({
            secret: user.mfaTempSecret,
            encoding: 'base32',
            token: token,
            window: 1
        });
```

Figure 13: MFA Verify

To prevent spam and ensure valid users, email verification is required. New accounts start as inactive, and a one-time verification link is sent by email. Users must verify their account before they can log in.

```
const verifyEmail = async (req, res) => {
    try {
        const verificationToken = req.params.token;
        const hashedToken = crypto
            .createHash('sha256')
            .update(verificationToken)
            .digest('hex');

        const user = await User.findOne({ emailVerificationToken: hashedToken });

        if (!user) {
            await logActivity(null, 'EMAIL_VERIFICATION_FAIL', 'warn', {
                method: req.method,
                ipAddress: req.ip,
                reason: 'Invalid or expired token',
            });

            return res.status(400).send('<h1>Error</h1><p>Invalid verification token.</p>');
        }

        user.isVerified = true;
        user.emailVerificationToken = undefined;
        await user.save();

        await logActivity(user._id, 'USER_EMAIL_VERIFIED', 'info', {
            email: user.email,
            method: req.method,
            ipAddress: req.ip,
        });
```

Figure 14: Email Verification

To block bots, the registration form uses Google reCAPTCHA v2. The backend verifies each response with Google's API to ensure only humans can create accounts.

```
const registerUser = async (req, res) => {
    try {
        const { name, email, password, role, captchaToken } = req.body;
        const ipAddress = req.ip || req.connection.remoteAddress;

        // 1. CAPTCHA Validation
        if (!captchaToken) {···
        }
        const verifyUrl = `https://www.google.com/recaptcha/api/siteverify?secret=${process.env.RECAPTCHA_SECRET_KEY}&response=${captchaToken}`;
        const recaptchaResponse = await axios.post(verifyUrl);

        if (!recaptchaResponse.data.success) {
            await logActivity(null, 'USER_REGISTER_FAIL', 'fatal', { email, ipAddress, reason: 'CAPTCHA verification failed.' });
            return res.status(400).json({ message: 'CAPTCHA verification failed. Please try again.' });
        }
```

Figure 15: Captcha Verification

Password Management & Policies

User accounts are protected with a strong password policy. During registration, the React form checks for complexity minimum 8 characters, including uppercase, lowercase, numbers, and symbols and gives real-time feedback. Account creation is blocked until all criteria are met.

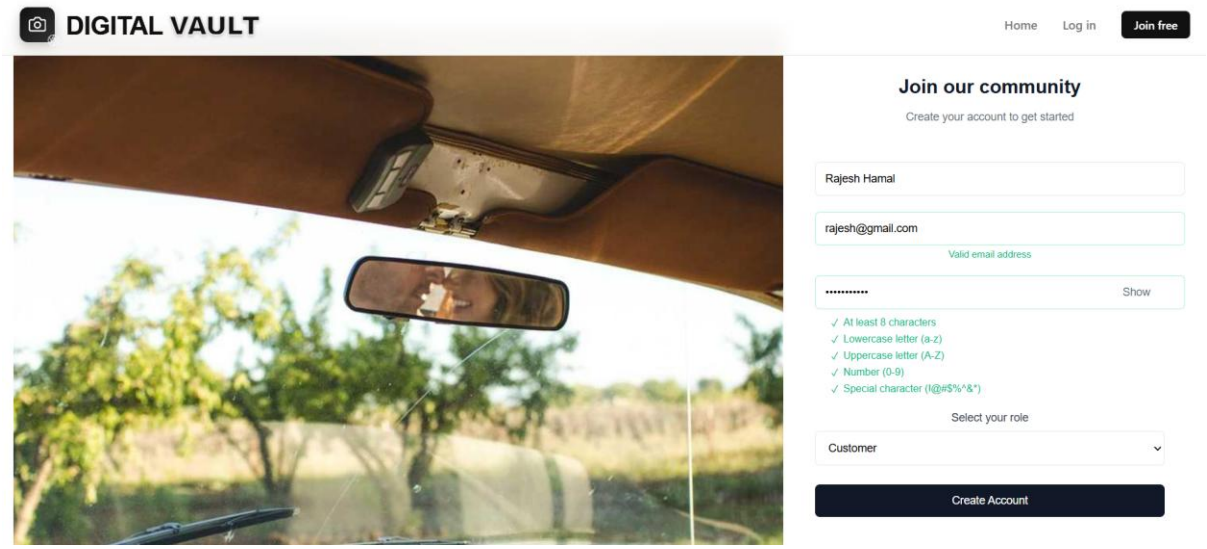


Figure 16: Real-time Password Checklist

The backend stores hashes of the last five passwords to prevent users from reusing recent ones during updates.

```
password: {
    type: String,
    required: [true, 'Please provide a password'],
    minlength: 8, // Enforce a minimum password length
},
// --- NEW FIELDS ---
passwordChangedAt: Date,
passwordHistory: [String], // An array to store old password hashes
```

Furthermore, 90-day password expiry is enforced by middleware that blocks access if the password is outdated, prompting users to update it.

```
const protect = async (req, res, next) => {
    let token;

    // Check if the request headers contain the authorization token
    if (req.headers.authorization && req.headers.authorization.startsWith('Bearer')) {
        try {
            // 1. Get token from header (it comes in the format 'Bearer <token>')
            token = req.headers.authorization.split(' ')[1];

            // 2. Verify the token using our secret key
            const decoded = jwt.verify(token, process.env.JWT_SECRET);


            req.user = await User.findById(decoded.id).select('-password');

            // --- CHECK FOR PASSWORD EXPIRY ---
            if (req.user && req.user.passwordChangedAt) {
                // Define your policy: password expires after 90 days
                const passwordAgeInDays = (Date.now() - req.user.passwordChangedAt.getTime()) / (1000 * 60 * 60 * 24);
                if (passwordAgeInDays > 90) {
                    return res.status(401).json({ message: 'Your password has expired. Please update it.' });
                }
            }

            // 4. Move on to the next function (the actual route controller)
            next();
        } catch (error) {
            console.error(error);
            res.status(401).json({ message: 'Not authorized, token failed' });
        }
    }
```

Figure 17: Password Expiration Date

To prevent brute-force attacks, accounts lock after 10 failed logins, blocking access for a set time and logging the event as "fatal."

```
if (user.lockUntil && user.lockUntil > Date.now()) {
    await logActivity(user._id, 'USER_LOGIN_FAIL', 'fatal', {
        ipAddress,
        method: req.method,
        reason: 'Account locked',
    });
    const timeLeft = Math.ceil((user.lockUntil - Date.now()) / 60000);
    return res.status(403).json({ message: `Account is locked. Try again in ${timeLeft} minutes.` });
}
```

Figure 18: Account Lockout

Audit Trail (Activity Logging)

"Digital Vault" includes a centralized audit logging system via a reusable logService, enabling easy recording of key security events throughout the app for monitoring and forensic analysis.

```
// logService.js
const logActivity = async (userId, action, level = 'info', details = {}) => {
    try {
        const cleanDetails = sanitizeDetails(details);

        const logEntry = new Log({
            user: userId,
            action,
            level,
            details: cleanDetails,
            method: details?.req?.method || details?.method || '[UNKNOWN]', // ✅ Extract method
        });

        await logEntry.save();
    } catch (error) {
        console.error('🚨 Failed to write log entry:', error.message);
    }
};


module.exports = { logActivity };
```

Figure 19: Log Activity

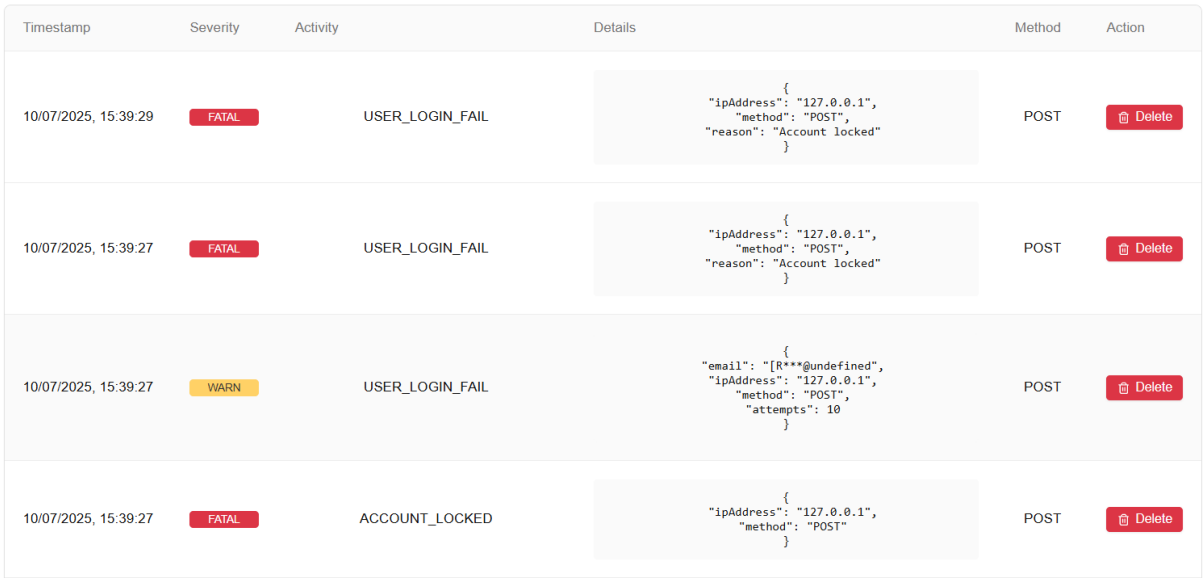
Logs are stored in a MongoDB collection, capturing timestamp, severity (info, warn, fatal), action (e.g., USER_LOGIN_FAIL), user ID, and IP address. This creates a reliable audit trail for monitoring and investigating security incidents.


```
_id: ObjectId('68665d91083faf97f910dcab')
user : ObjectId('68665653766764 61fb883b6d')
action : "USER_LOGIN_SUCCESS"
level : "info"
▾ details : Object
    ipAddress : "127.0.0.1"
    method : "POST"
createdAt : 2025-07-03T10:38:09.636+00:00
__v : 0
```

```
_id: ObjectId('68665d99083faf97f910dcaf')
user : ObjectId('68665be05cae556ff405c833')
action : "USER_LOGIN_SUCCESS"
level : "info"
▾ details : Object
    ipAddress : "127.0.0.1"
    method : "POST"
createdAt : 2025-07-03T10:38:17.664+00:00
__v : 0
```


Figure 20: Logs in the Database

The "Digital Vault" dashboard offers a secure admin-only "Activity Logs" tab that displays all system events in a clear table. Sensitive data like passwords and tokens are automatically redacted, giving admins a complete, up-to-date view of security events.



Timestamp	Severity	Activity	Details	Method	Action
10/07/2025, 15:39:29	FATAL	USER_LOGIN_FAIL	{ "ipAddress": "127.0.0.1", "method": "POST", "reason": "Account locked" }	POST	Delete
10/07/2025, 15:39:27	FATAL	USER_LOGIN_FAIL	{ "ipAddress": "127.0.0.1", "method": "POST", "reason": "Account locked" }	POST	Delete
10/07/2025, 15:39:27	WARN	USER_LOGIN_FAIL	{ "email": "[R***@undefined", "ipAddress": "127.0.0.1", "method": "POST", "attempts": 10 }	POST	Delete
10/07/2025, 15:39:27	FATAL	ACCOUNT_LOCKED	{ "ipAddress": "127.0.0.1", "method": "POST" }	POST	Delete

Figure 21: Logs Activity in the Admin Dashboard

User Authentication and Authorization

"Digital Vault" uses a strong authentication and authorization system with three user roles: customer, creator, and admin. A custom authorize() middleware checks each protected request to ensure users only access functions allowed for their role.

```
// --- New RBAC Middleware ---
Tabnine | Edit | Explain
const authorize = (...roles) => {
    return (req, res, next) => {
        // 'req.user' is attached by our 'protect' middleware
        if (!req.user || !roles.includes(req.user.role)) {

            return res.status(403).json({ message: 'Your role is not authorized to access this route' });
        }
        next();
    };
};
```

Figure 22: RBAC

On the client side, Protected Routes check user roles via AuthContext and redirect unauthorized users for secure access.

```jsx
import React, { useContext } from 'react';
import { Navigate, useLocation } from 'react-router-dom';
import { AuthContext } from '../context/AuthContext';

Tabnine | Edit | Explain
const CreatorProtectedRoute = ({ children }) => {
    const { isAuthenticated, user } = useContext(AuthContext);
    const location = useLocation();

    if (!isAuthenticated) {
        // If not logged in, redirect to login page
        return <Navigate to="/login" state={{ from: location }} replace />;
    }

    if (user.role !== 'creator') {
        // If logged in but not a creator, redirect to home page
        // You could also redirect to an "Unauthorized" page
        alert('Access Denied: This page is for creators only.');
        return <Navigate to="/" replace />;
    }

    // If logged in AND a creator, render the page
    return children;
};

export default CreatorProtectedRoute;
```

Figure 23: Protected Routes in the client side

To prevent privilege escalation, users cannot change their role via the profile update API. Only admins can modify roles through a separate admin-only endpoint.

```jsx
const updateUserByAdmin = async (req, res) => {
    console.log(' updateUserByAdmin called');

    try {
        const user = await User.findById(req.params.id);

        if (!user) {
            return res.status(404).json({ message: 'User not found' });
        }


        user.name = req.body.name || user.name;
        user.email = req.body.email || user.email;
        user.role = req.body.role || user.role;

        const updatedUser = await user.save();

        await logActivity(req.user.id, 'ADMIN_UPDATE_USER_SUCCESS', 'warn', { updatedUserId: updatedUser._id });
        res.json({
            _id: updatedUser._id,
            name: updatedUser.name,
            email: updatedUser.email,
            role: updatedUser.role,
        });

    } catch (error) {
        await logActivity(req.user.id, 'ADMIN_UPDATE_USER_FAIL', 'error', { error: error.message });
        res.status(500).json({ message: 'Server Error' });
    }
};
```

Figure 24: Admin Only Endpoint

Secure Session Management

"Digital Vault" uses stateless session management with JWTs. After login, the server creates a signed token containing the user's ID, role, and a 30-day expiry, which is sent securely to the client.

```
        user.failedLoginAttempts = 0;
        user.lockUntil = undefined;
        await user.save({ validateBeforeSave: false });

        if (user.isMfaEnabled) {
            return res.json({ mfaRequired: true, userId: user._id });
        } else {
            await logActivity(user._id, 'USER_LOGIN_SUCCESS', 'info', {
                ipAddress,
                method: req.method,
            });
            return res.status(200).json({
                _id: user._id,
                name: user.name,
                email: user.email,
                role: user.role,
                token: generateToken(user._id),
            });
        }

    } catch (error) {
        await logActivity(null, 'LOGIN_CONTROLLER_ERROR', 'error', {
            error: error.message,
            method: req.method,
        });
        res.status(500).json({ message: 'Server Error' });
    }
};
```

Figure 25: Token generation

The token is stored in localStorage and sent with every request in the Authorization header. A server middleware verifies the token's validity and expiration, ensuring each request is authenticated and preventing session hijacking.

```
// --- New RBAC Middleware ---
Tabnine | Edit | Explain
const authorize = (...roles) => {
    return (req, res, next) => {
        // 'req.user' is attached by our 'protect' middleware
        if (!req.user || !roles.includes(req.user.role)) {

            return res.status(403).json({ message: 'Your role is not authorized to access this route' });
        }
        next();
    };
};
```

Encrypted Information & Asset Protection

"Digital Vault" employs a multi-faceted strategy to protect sensitive data and creator-owned assets, ensuring that information is secure both at rest and in transit.

Data at Rest: Password Hashing

User passwords and MFA recovery codes are securely hashed and salted with bcryptjs before storage. A Mongoose pre-save hook ensures plain-text passwords are always converted to irreversible hashes, protecting credentials even if the database is breached.

```
userSchema.pre('save', async function (next) {
    // Only run this function if password was actually modified
    if (!this.isModified('password')) return next();

    if (!this.isNew) {
        this.passwordHistory.unshift(this._previousPassword); // Store the old password
        // Keep the history to a maximum of 5 recent passwords
        if (this.passwordHistory.length > 5) {
            this.passwordHistory.pop();
        }
    }

    // Hash the new password
    const salt = await bcrypt.genSalt(12);
    this.password = await bcrypt.hash(this.password, salt);

    next();
});
```

Figure 26: Password Hashing

Digital Asset Protection

The app uses a two-folder system to protect creators' work. Original high-quality files are stored privately and aren’t directly accessible. Public previews are watermarked using the sharp library and stored separately, ensuring only verified buyers can access the original assets.

Figure 27: Creator Only Access Control

Frontend and Backend in HTTPS

"Digital Vault" secures all data in transit by running the frontend and backend exclusively over HTTPS. In development, this is done using a self-signed SSL certificate with mkcert and Node.js's https module, ensuring encryption of sensitive data against interception.

Figure 28: HTTPS Implementation

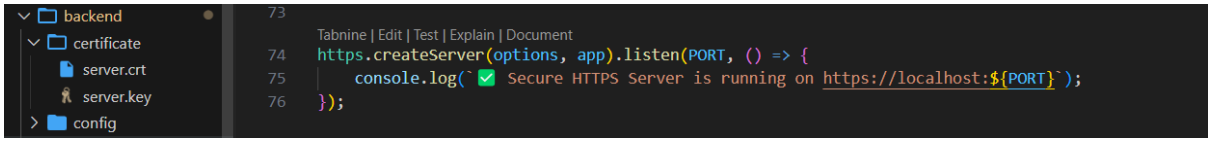


Figure 29: Certificates

```
// Load your SSL certificate files
const options = {
    key: fs.readFileSync(path.join(__dirname, 'certificate/server.key')),
    cert: fs.readFileSync(path.join(__dirname, 'certificate/server.crt')),
};
```

Input Sanitization

To prevent database injection attacks, a custom middleware sanitizes all incoming data by removing malicious characters like the $ operator. It scans requests before they reach the database, blocking injection attempts.

```
Tabnine | Edit | Explain
const sanitize = (obj) => {
    for (const key in obj) {
        if (/^\$/.test(key)) { // Check if the key starts with '$'
            delete obj[key];
        } else if (typeof obj[key] === 'object' && obj[key] !== null) {
            // If the value is another object, sanitize it recursively
            sanitize(obj[key]);
        }
    }
    return obj;
};

Tabnine | Edit | Explain
const sanitizeRequest = (req, res, next) => {
    if (req.body) {
        req.body = sanitize(req.body);
    }
    if (req.query) {
        req.query = sanitize(req.query);
    }
    if (req.params) {
        req.params = sanitize(req.params);
    }
    next();
};

module.exports = sanitizeRequest;
```

Figure 30: Input Sanitization

Bot and Spam Prevention (CAPTCHA)

To prevent bot signups, Google reCAPTCHA v2 is used on the registration form. Users complete the challenge to get a token, which the backend verifies with Google before allowing registration.

Figure 31: Captcha Implementation

Content & Asset Protection

To protect content and resources, Hotlink Prevention middleware blocks image requests from unauthorized domains by checking the Referer header, preventing theft and saving bandwidth.

Figure 32: Hotlink Prevention

Conclusion

In conclusion, the "Digital Vault" project successfully demonstrates the development of a secure and fully functional e-commerce platform for digital assets. By integrating a multi-layered, defence-in-depth security model, the application effectively addresses critical modern web vulnerabilities. Key security measures such as Multi-Factor Authentication, Role-Based Access Control, secure payment processing with Stripe, and robust asset protection through watermarking have been successfully implemented and tested. The final application stands as a comprehensive proof of concept, showcasing how a security-first mindset can be integrated into every stage of development to create a trustworthy and scalable digital marketplace.

References

Axis, T. (2023) *Why node JS is better than other languages*. Available at: https://techaxis.com.np/blog/blog_detail/why-node-js-is-better-than-other-languages (Accessed: 02 July 2024).

Brad, D. (2022) *How to create your own SSL certificate authority for local HTTPS development*. Available at: https://deliciousbrains.com/ssl-certificate-authority-for-local-https-development/ (Accessed: 18 July 2024).

Helms, T. (2021) *Why tailwind is the best choice for custom CSS*. Available at: https://tracehelms.com/blog/why-tailwind-is-the-best-choice-for-custom-css (Accessed: 02 July 2024).

Hensley, J. (2024) *The 5 must-have principles of design strategy*. Available at: https://www.emergeagency.com/insights/detail/digital-product-strategy-framework-design-principles/ (Accessed: 25 July 2024).

Nerdifico, W. (2020) *Why mongodb and not SQL databases?* The freeCodeCamp Forum. Available at: https://forum.freecodecamp.org/t/why-mongodb-and-not-sql-databases/418134 (Accessed: 05 July 2024).

Pat, R. (2024) *System design: What is it and why it is important*. Available at: https://segwitz.com/what-is-system-design-and-why-it-is-necessary/ (Accessed: 20 July 2024).

Rodrigues, J. (2023) *The importance of secure file sharing in the Digital age*. Available at: https://www.titanfile.com/blog/the-importance-of-secure-file-sharing-in-the-digital-age/ (Accessed: 24 June 2024).

Ronne, D. (2024) *Nodemailer example: Learn how to send emails from nodejs app*. Available at: https://www.bacancytechnology.com/blog/send-email-using-nodemailer/ (Accessed: 01 July 2024).

Scottis, T. (2020) *Send emails with node.js: API and Nodemailer (SMTP)*. Available at: https://www.mailersend.com/blog/send-email-nodejs (Accessed: 14 July 2024).

Water, A. (2022) *Why is mern stack popular for web application development?* Available at: https://herovired.com/learning-hub/blogs/why-is-mern-stack-popular-for-web-application-development/ (Accessed: 26 June 2024).

Jones, M., Bradley, J., and Sakimura, N. (2015) *JSON Web Token (JWT) - RFC 7519*. Available at: https://www.rfc-editor.org/info/rfc7519 (Accessed: 11 July 2024).

Nielsen, J. (2021) *The UX of Security: Taming the Friction of Authentication*. Nielsen Norman Group. Available at: https://www.nngroup.com/articles/ux-security-friction/ (Accessed: 11 July 2024).

OWASP Foundation (2023) *OWASP Secure Coding Practices-Quick Reference Guide*. Available at: https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/ (Accessed: 11 July 2024).

Provos, N. and Mazières, D. (1999) 'A Future-Adaptable Password Scheme', in *Proceedings of the 1999 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association.

Stripe, Inc. (2024) *Stripe Docs: Verifying signatures from webhooks*. Available at: https://stripe.com/docs/webhooks/signatures (Accessed: 11 July 2024).

Vanderkam, D. (2021) *The Wrong Way to Store a Token in a Single-Page App*. Available at: https://pragmaticwebsecurity.com/articles/spasecurity/wrong-way-to-store-a-token.html (Accessed: 11 July 2024).

Appendix

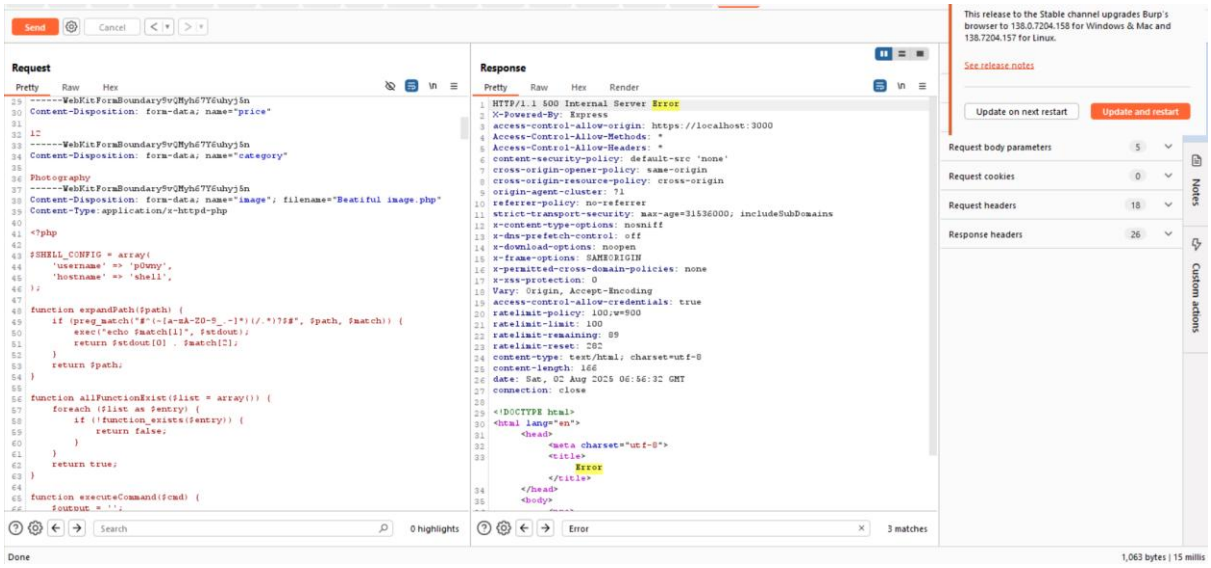
GitHub Link: https://github.com/Karanbohara01/digital-vault-final.git

Digital Vault Video: cw2 final video.mp4

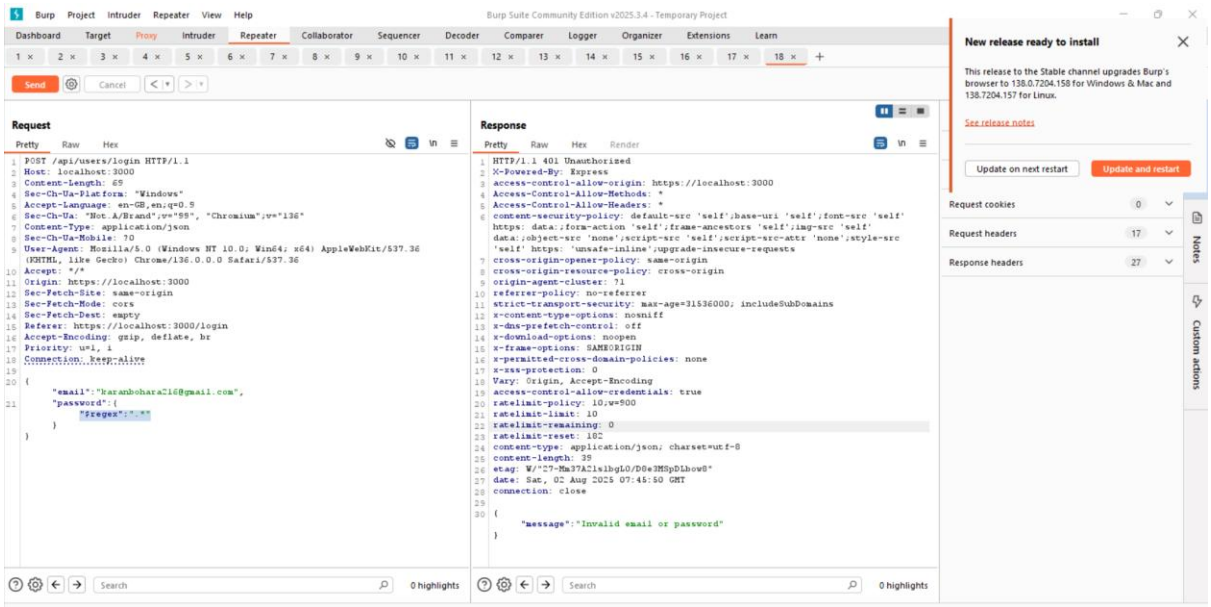
Proof of Concept of PenTesting Video: Cw2-Pentesting-Final Video.mp4

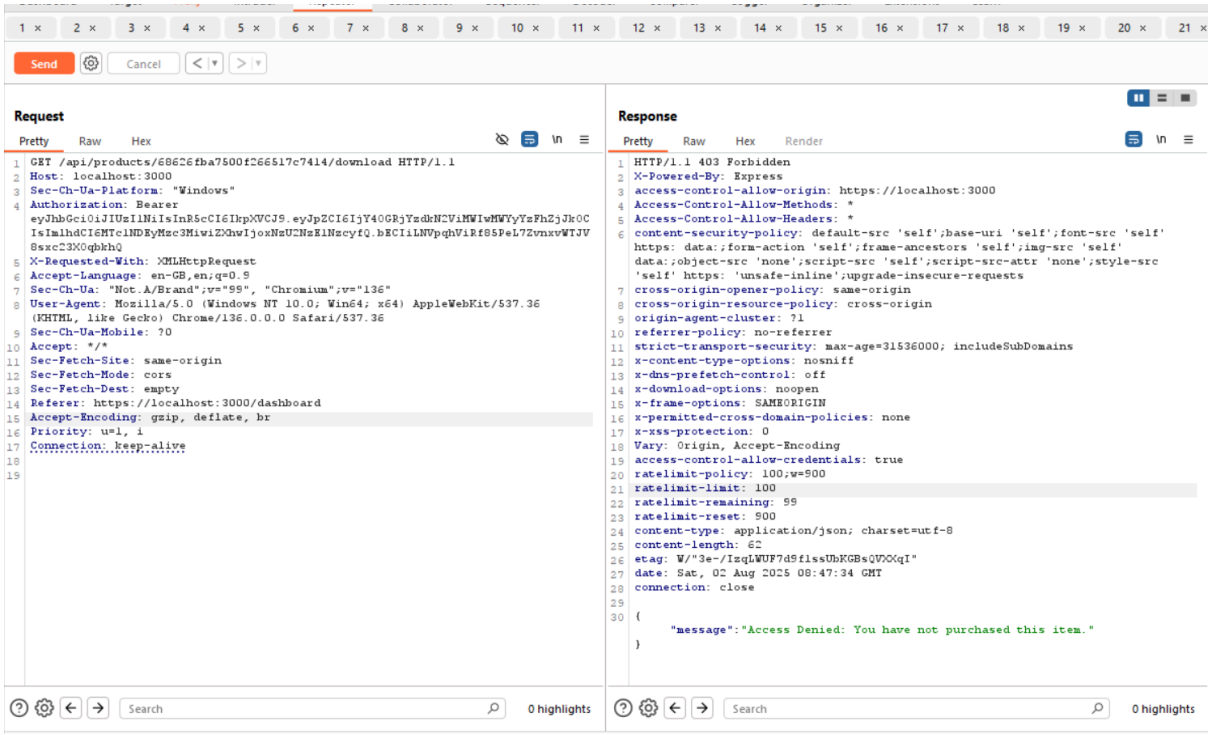
Proof of Concept for PenTesting

Security Testing for File Upload Functionality

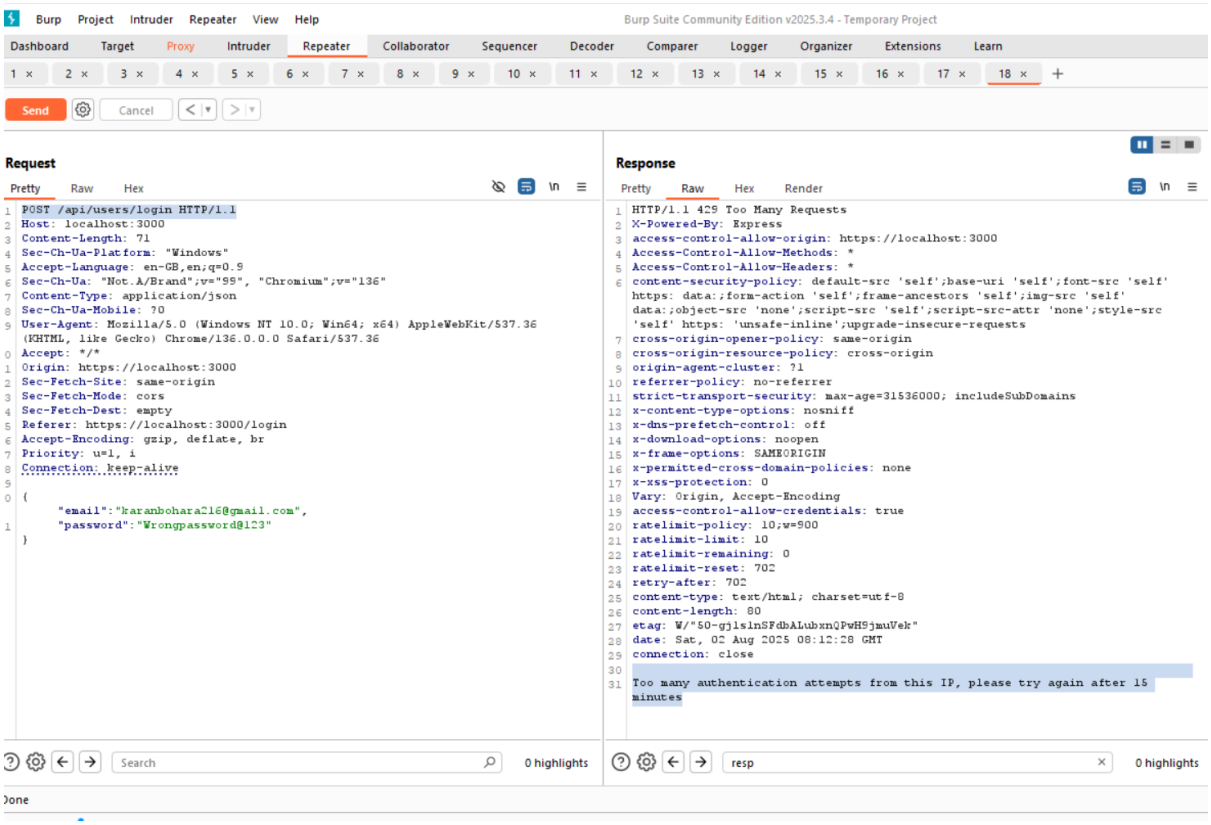


Security Testing for NoSQL Injection

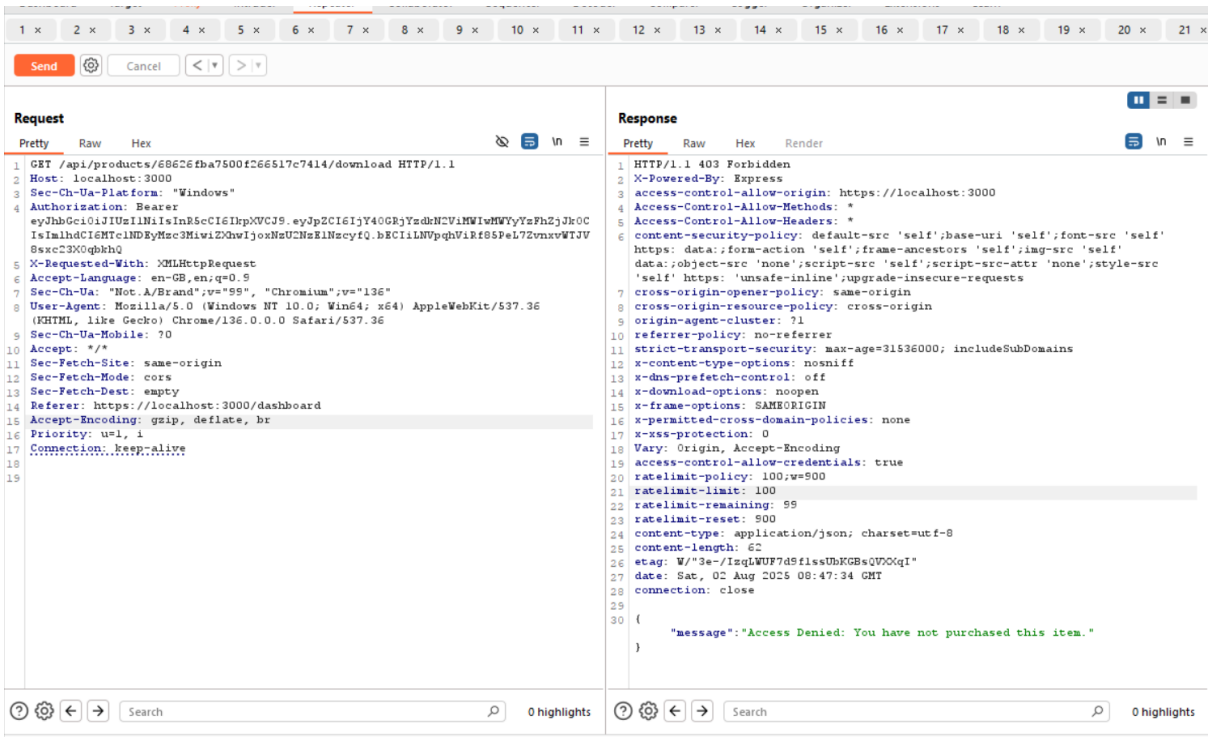




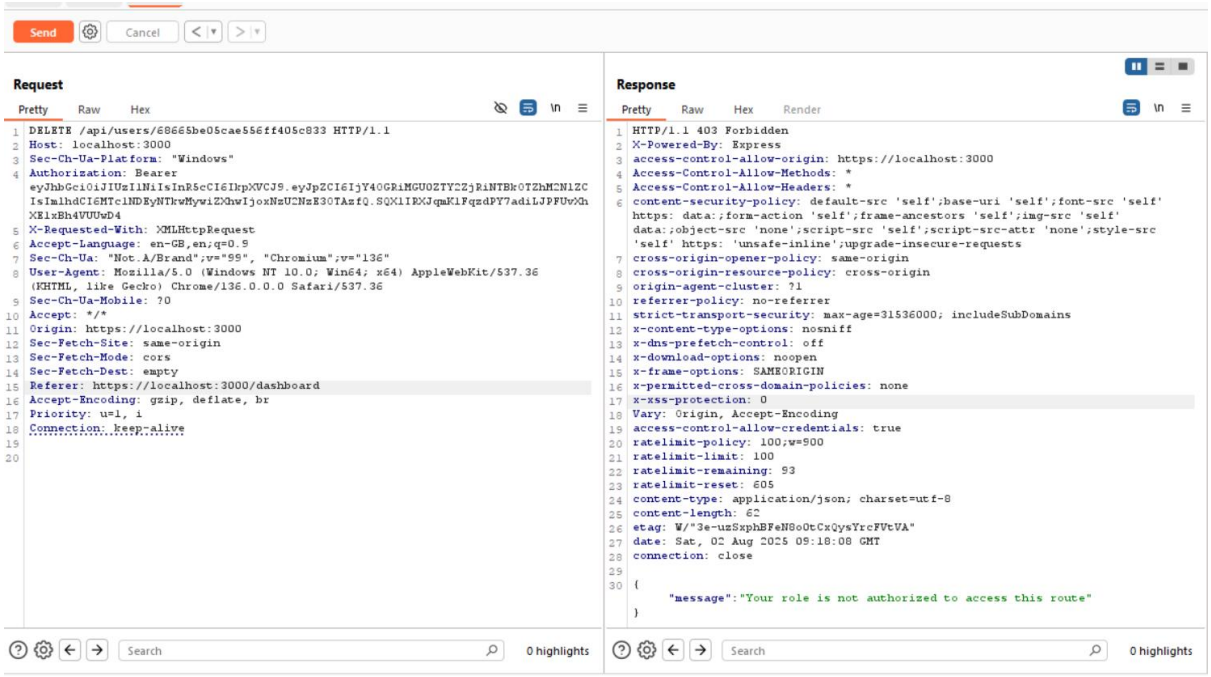
Rate-Limited Authentication Testing

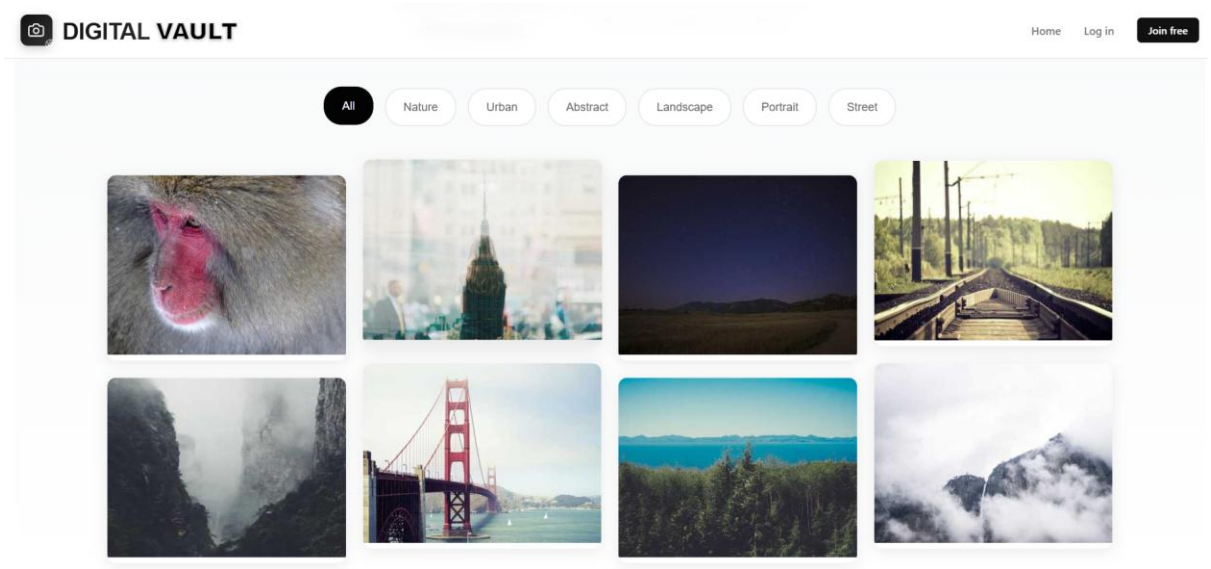


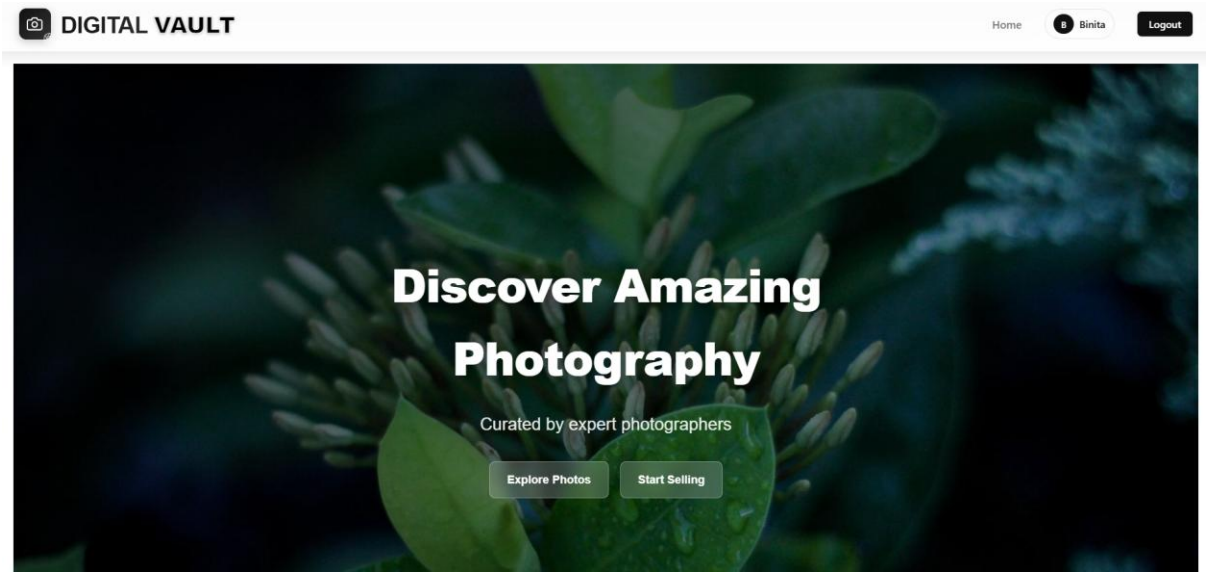
Authorization Bypass Testing for Secure Downloads

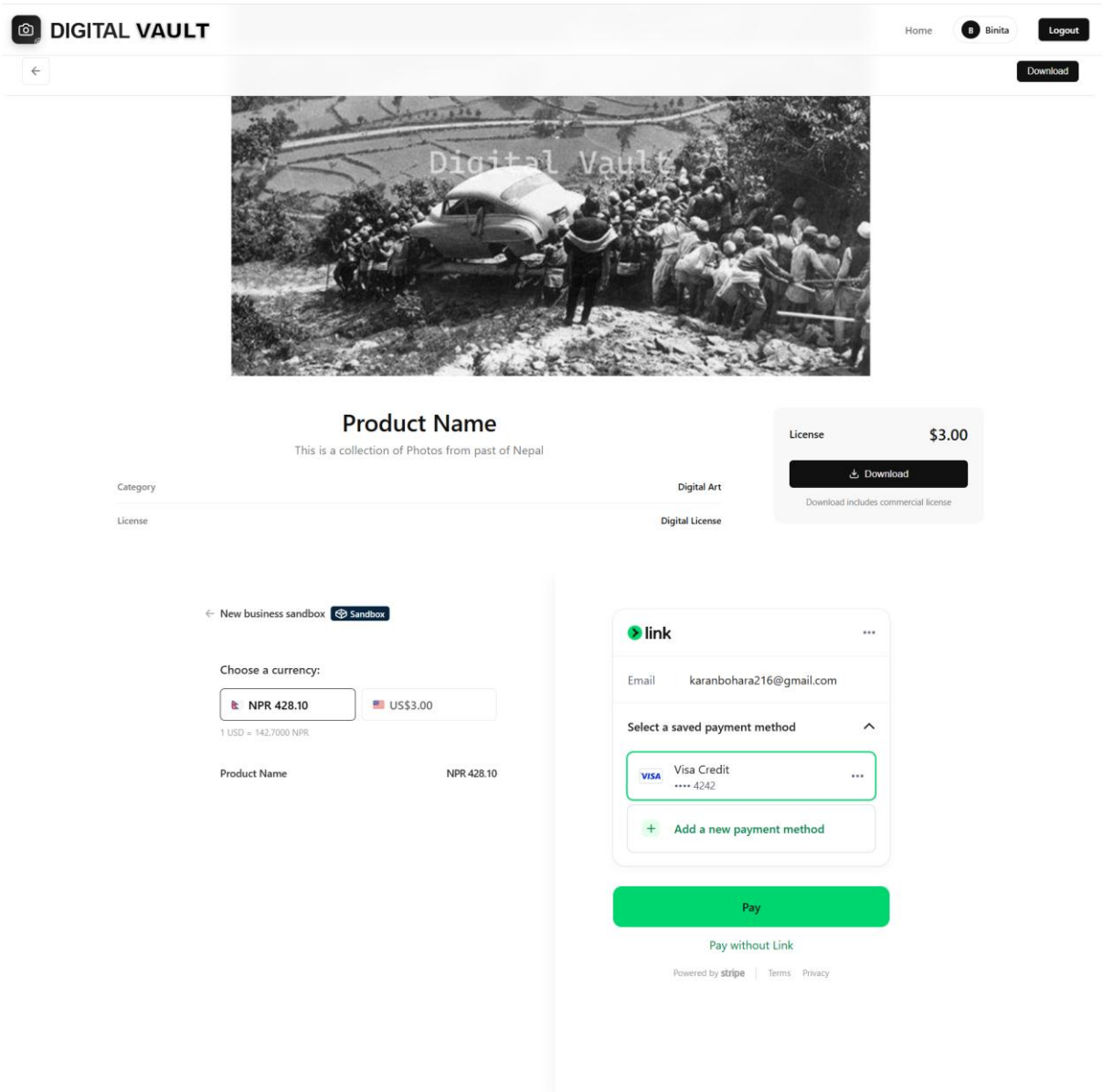


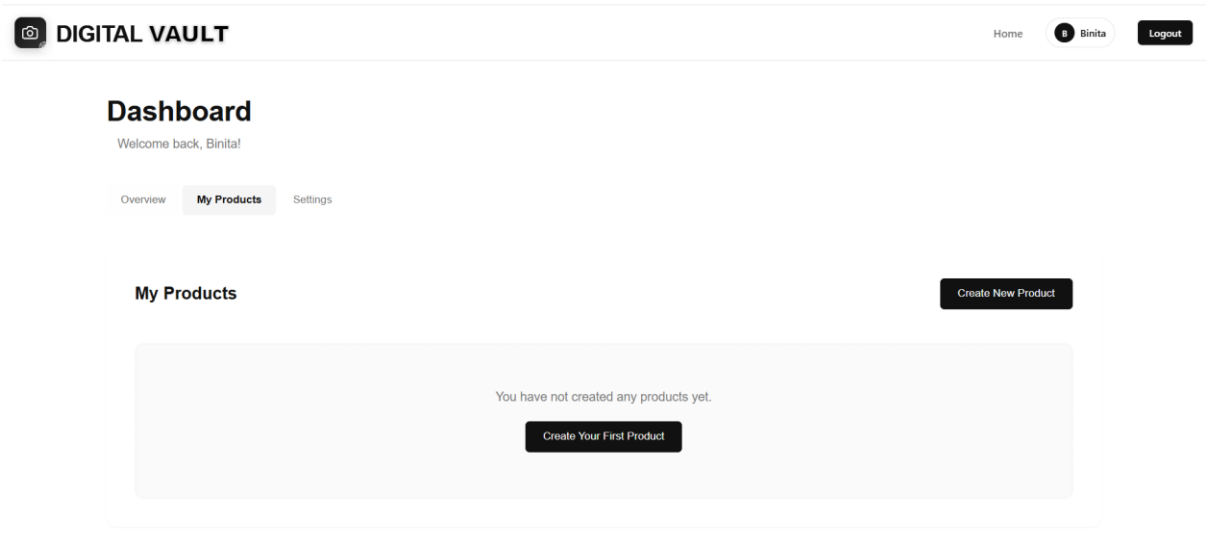
Authorization Bypass in Role-Based Access Systems

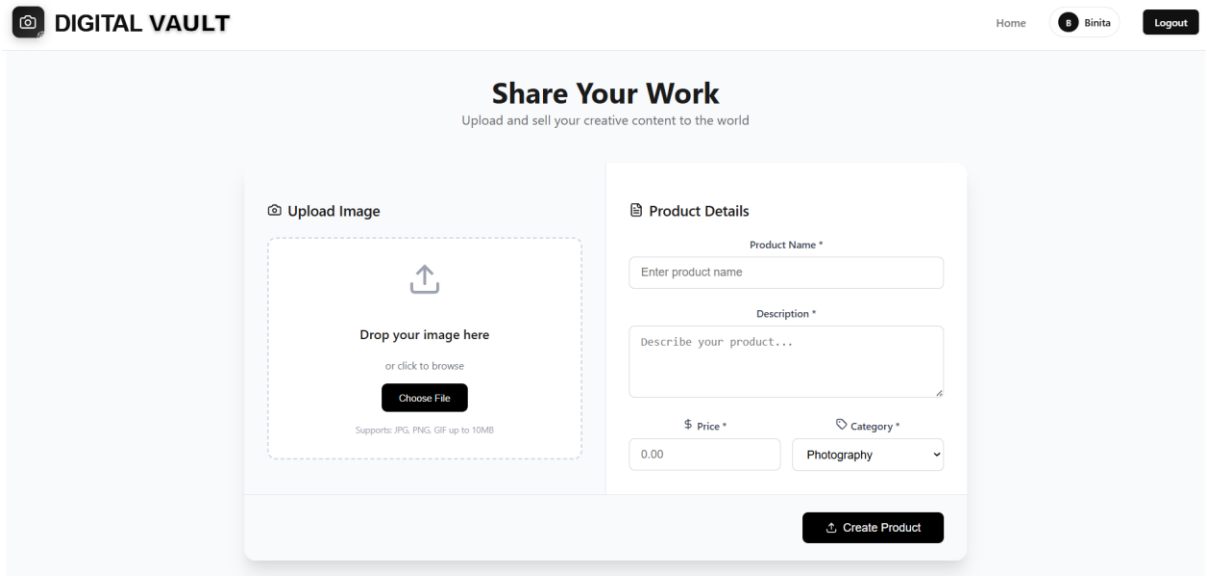

Send
Cancel
Request
Pretty
Raw
Hex
DELETE /api/users/68665be05cae556ff405c833 HTTP/1.1
Host: localhost:3000
Sec-Ch-Ua-Platform: "Windows"
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY4OGRiMGU0ZTY2ZjRiNTBk0TZhM2N1ZC
IsImlhdCI6MTc1NDEyNTkwMywiZXhwIjoxNzU2NzE30TAzfQ.SQX1IRXJqmKlFqzdPY7adiLJPFUvXh
XElxBh4VUUwD4
X-Requested-With: XMLHttpRequest
Accept-Language: en-GB,en;q=0.9
Sec-Ch-Ua: "Not.A/Brand";v="99", "Chromium";v="136"
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/136.0.0.0 Safari/537.36
Sec-Ch-Ua-Mobile: ?0
Accept: */*
Origin: https://localhost:3000
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: https://localhost:3000/dashboard
Accept-Encoding: gzip, deflate, br
Priority: u=1, i
Connection: keep-alive
Search
0 highlights
Response
Pretty
Raw
Hex
Render
HTTP/1.1 403 Forbidden
X-Powered-By: Express
access-control-allow-origin: https://localhost:3000
Access-Control-Allow-Methods: *
Access-Control-Allow-Headers: *
content-security-policy: default-src 'self';base-uri 'self';font-src 'self'
https: data:;form-action 'self';frame-ancestors 'self';img-src 'self'
data:;object-src 'none';script-src 'self';script-src-attr 'none';style-src
'self' https: 'unsafe-inline';upgrade-insecure-requests
cross-origin-opener-policy: same-origin
cross-origin-resource-policy: cross-origin
origin-agent-cluster: ?1
referrer-policy: no-referrer
strict-transport-security: max-age=31536000; includeSubDomains
x-content-type-options: nosniff
x-dns-prefetch-control: off
x-download-options: noopen
x-frame-options: SAMEORIGIN
x-permitted-cross-domain-policies: none
x-xss-protection: 0
Vary: Origin, Accept-Encoding
access-control-allow-credentials: true
ratelimit-policy: 100;w=900
ratelimit-limit: 100
ratelimit-remaining: 93
ratelimit-reset: 605
content-type: application/json; charset=utf-8
content-length: 62
etag: W/"3e-uzSxphBFeN8oOtCxQysYrcFVtVA"
date: Sat, 02 Aug 2025 09:18:08 GMT
connection: close
{
"message":"Your role is not authorized to access this route"
}
Search
0 highlights

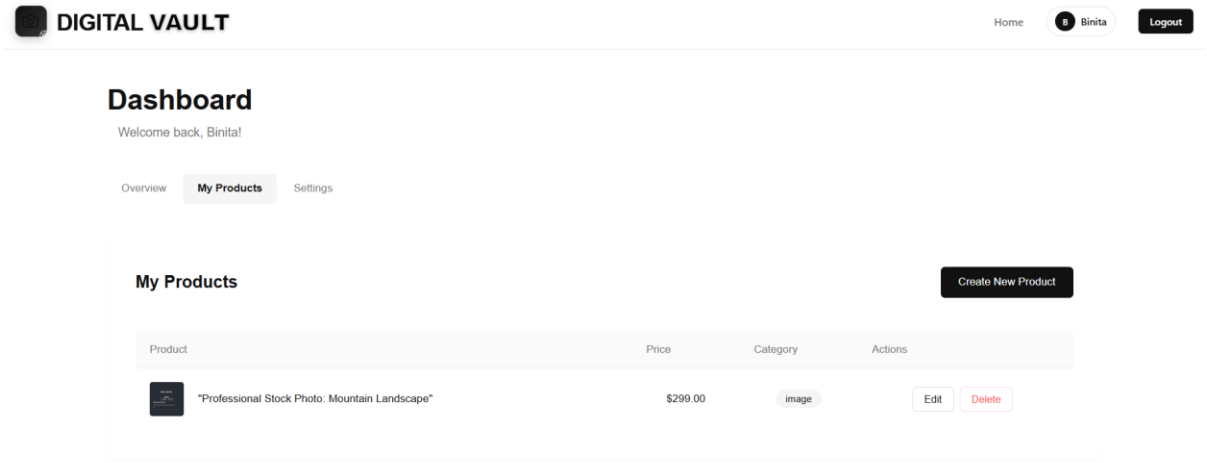

DIGITAL VAULT
Home
Log in
Join free
All
Nature
Urban
Abstract
Landscape
Portrait
Street

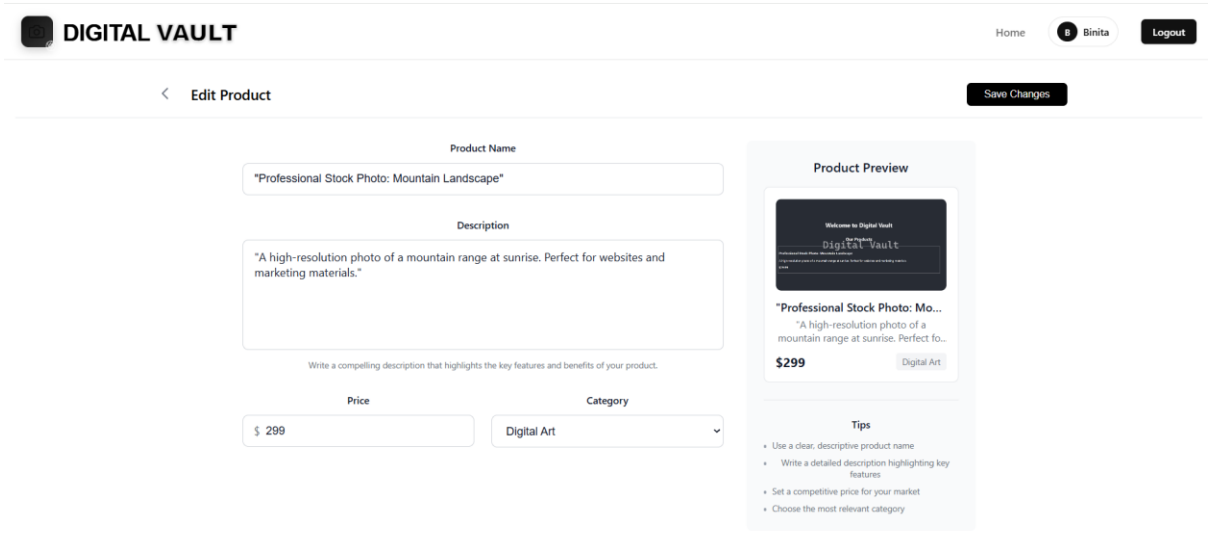

DIGITAL VAULT
Home
Binita
Logout
Discover Amazing Photography
Curated by expert photographers
Explore Photos
Start Selling

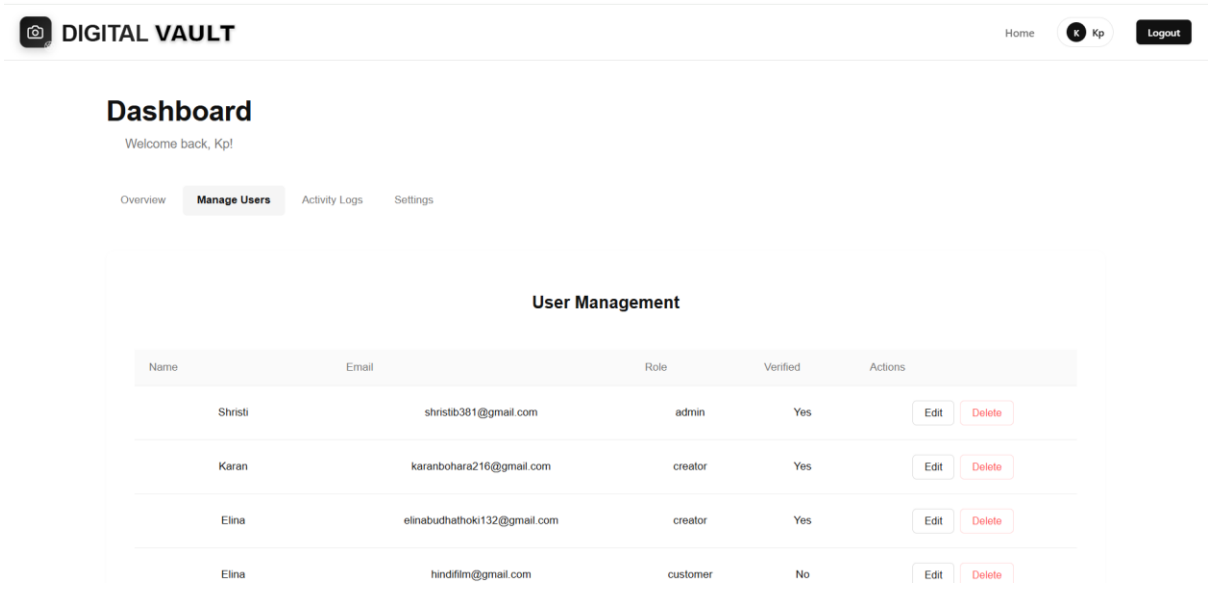

DIGITAL VAULT
Home
Binita
Logout
Download
Digital Vault
Product Name
This is a collection of Photos from past of Nepal
Category
Digital Art
License
Digital License
License
$3.00
Download
Download includes commercial license
New business sandbox
Sandbox
Choose a currency:
NPR 428.10
US$3.00
1 USD = 142.7000 NPR
Product Name
NPR 428.10
link
Email
karanbohara216@gmail.com
Select a saved payment method
Visa Credit
•••• 4242
Add a new payment method
Pay
Pay without Link
Powered by stripe
Terms
Privacy


DIGITAL VAULT
Home
Binita
Logout
Dashboard
Welcome back, Binita!
Overview
My Products
Settings
My Products
Create New Product
You have not created any products yet.
Create Your First Product


DIGITAL VAULT
Home
Binita
Logout
Share Your Work
Upload and sell your creative content to the world
Upload Image
Drop your image here
or click to browse
Choose File
Supports: JPG, PNG, GIF up to 10MB
Product Details
Product Name *
Enter product name
Description *
Describe your product...
Price *
0.00
Category *
Photography
Create Product


DIGITAL VAULT
Home
Binita
Logout
Dashboard
Welcome back, Binita!
Overview
My Products
Settings
My Products
Create New Product
Product
Price
Category
Actions
"Professional Stock Photo: Mountain Landscape"
$299.00
image
Edit
Delete


DIGITAL VAULT
Home
B Binita
Logout
Edit Product
Save Changes
Product Name
"Professional Stock Photo: Mountain Landscape"
Description
"A high-resolution photo of a mountain range at sunrise. Perfect for websites and marketing materials."
Write a compelling description that highlights the key features and benefits of your product.
Price
$ 299
Category
Digital Art
Product Preview
"Professional Stock Photo: Mo...
"A high-resolution photo of a mountain range at sunrise. Perfect fo...
$299
Digital Art
Tips
Use a clear, descriptive product name
Write a detailed description highlighting key features
Set a competitive price for your market
Choose the most relevant category


DIGITAL VAULT
Home
K Kp
Logout
Dashboard
Welcome back, Kp!
Overview
Manage Users
Activity Logs
Settings
User Management
Name
Email
Role
Verified
Actions
Shristi
shristib381@gmail.com
admin
Yes
Edit
Delete
Karan
karanbohara216@gmail.com
creator
Yes
Edit
Delete
Elina
elinabudhathoki132@gmail.com
creator
Yes
Edit
Delete
Elina
hindifilm@gmail.com
customer
No
Edit
Delete