

Git Guardian: GitHub Repository Security Scanner

Dr. A.P Srivastava¹, Priya Tyagi², Deepansh Tyagi³, Sanskriti Sharma⁴, Anuj Singh⁵

¹Supervisor, Dept. of Computer Science and Engineering, NITRA Technical Campus, Ghaziabad, India

^{2,3,4,5}B.Tech Students, Dept. of Computer Science and Engineering, NITRA Technical Campus, Ghaziabad, India

Abstract - Git Guardian is a full-stack web application designed to enhance software security by automatically scanning public GitHub repositories for sensitive data exposures and code quality issues. In the modern software development landscape, inadvertent commits of API keys, JWT tokens, private keys, and AWS credentials represent one of the most critical and common security vulnerabilities. This project delivers an end-to-end security platform that allows users to submit any public GitHub repository URL for deep scanning. The system clones the repository, traverses all source files, and applies a suite of regex-based detection patterns to identify security threats across severity levels — critical, high, medium, and low. Scan results are persisted in a MongoDB database and presented to the user via an interactive React dashboard. The application also features a role-based access control (RBAC) system distinguishing regular users from administrators, an automated scheduled scanning feature powered by BullMQ and Redis, and full activity logging. The backend is built with Node.js and Express.js, while the frontend is built using React.js, Vite, and Tailwind CSS. The system is deployable to Vercel (frontend) and Render (backend) for cloud production use.

Key Words: GitHub Security, Repository Scanner, Secret Detection, RBAC, Node.js, React.js, BullMQ, MongoDB, JWT, Static Analysis

1. INTRODUCTION

The proliferation of open-source development and cloud-based version control platforms like GitHub has dramatically increased the risk of sensitive information being accidentally committed to public repositories. Developers, especially those new to software engineering, frequently commit configuration files, environment variables, and credential tokens that can be exploited by malicious actors.

Git Guardian addresses this problem head-on by providing an automated, user-friendly scanning tool. A user simply pastes a GitHub repository URL, and the system performs a complete security audit — checking every file for hard-coded secrets, exposed keys, and poor code practices. The results are presented in a structured, color-coded report with file names, line numbers, severity levels, and issue descriptions.

The project was developed as a demonstration of modern full-stack web development principles, integrating REST API design, asynchronous job processing, database modeling, JWT authentication, and cloud deployment into a cohesive and production-ready application.

2. PROBLEM STATEMENT AND MOTIVATION

According to multiple security research reports, thousands of valid credentials are exposed on GitHub every day. Developers often encounter the following security pitfalls:

- Accidentally commit .env files containing database URIs and API keys
- Hard-code JWT tokens or private keys inside source code for testing
- Leave TODO comments revealing incomplete security implementations
- Forget to add sensitive files to .gitignore before their first push

Existing enterprise-grade solutions like GitGuardian (commercial) or GitHub's native secret scanning require paid plans or are limited to repository owners. There is a clear gap for a free, accessible tool that any developer can use to audit any public repository. This project fills that gap with a lightweight, deployable web application.

3. OBJECTIVES

The primary objectives of this project are:

- To design and implement a full-stack web application for GitHub repository security scanning.
- To develop a regex-based code analysis engine capable of detecting critical security vulnerabilities.

- To implement a user authentication system with JWT-based access control and role-based permissions.
- To create an automated scanning scheduler that periodically audits a developer's repositories.
- To build an intuitive React.js dashboard to visualize scan results, history, and statistics.
- To deploy the application to cloud infrastructure (Vercel + Render) for real-world accessibility.
- To implement an activity logging system for admin oversight and audit trails.

4. SYSTEM ARCHITECTURE

Git Guardian follows a three-tier client-server architecture with an additional asynchronous job processing layer. The system is composed of the following major components:

4.1 Architecture Overview

Table -1: Architecture Layers

Layer	Technology	Responsibility
Presentation Layer	React.js + Vite + Tailwind CSS	User Interface, Dashboard, Reports
Application Layer	Node.js + Express.js	REST API, Business Logic, Auth
Data Layer	MongoDB + Mongoose	User Data, Scan Records, Activity Logs
Job Queue Layer	BullMQ + Redis	Async Scan Scheduling & Processing
External API Layer	GitHub REST API (Octokit)	Repository Listing for Auto-Scan

4.2 Request Flow

The data flow for a manual scan request follows this sequence:

- User submits a GitHub repository URL through the React dashboard.
- The frontend sends a POST request to /api/scans with a JWT Authorization header.
- The Express backend validates the JWT token via the auth middleware.
- A new Scan document is created in MongoDB with status: 'queued'.
- The processScan() function is invoked asynchronously as a background job.
- The service clones the repository using git clone --depth 1 into a temporary directory.
- All source files are enumerated and filtered by extension (JS, TS, PY, GO, etc.).
- Each file is read and passed through the regex pattern engine. Issues are collected.
- The temporary directory is deleted and the Scan document is updated with results.
- The frontend polls or the user refreshes to view the completed scan results.

5. TECHNOLOGY STACK

5.1 Backend Technologies

Table -2: Backend Technology Stack

Package	Version	Purpose
Node.js	v18+	JavaScript runtime for server-side execution
Express.js	v5.2.1	Web framework for building REST APIs
Mongoose	v9.1.1	MongoDB object modelling and schema definition

jsonwebtoken	v9.0.3	JWT generation and verification
bcryptjs	v3.0.3	Password hashing using the bcrypt algorithm
BullMQ	v5.66.4	Redis-backed message queue for background jobs
ioredis	v5.8.2	Redis client for Node.js
Octokit	v5.0.5	Official GitHub REST API client library
helmet	v8.1.0	HTTP security header middleware
axios	v1.13.2	HTTP client for making API requests

5.2 Frontend Technologies

Table -3: Frontend Technology Stack

Package	Version	Purpose
React.js	v18.2.0	Component-based UI library
Vite	v7.3.1	Next-generation frontend build tool
Tailwind CSS	v3.3.5	Utility-first CSS framework for styling
React Router DOM	v6.18.0	Client-side routing and navigation
Recharts	v3.6.0	Chart library for scan statistics
Framer Motion	v10.16.4	Animation library for React
Lucide React	v0.292.0	Icon set for UI components
React Toastify	v9.1.3	Toast notification system
Axios	v1.6.0	HTTP client for API communication

6. MODULES AND FEATURES

6.1 User Authentication Module

The authentication system uses JSON Web Tokens (JWT) for stateless session management. Passwords are hashed using bcryptjs with a salt factor of 10 before storage in MongoDB. The token has a 30-day expiration. All protected routes pass through the AUTH Middleware, which verifies the token and attaches the user object to the request context. Available authentication endpoints include: POST /api/auth/register to create a new user account; POST /api/auth/login to authenticate and receive a JWT token; GET /api/auth/me to retrieve the authenticated user’s profile; and PUT /api/auth/profile to update GitHub username, token, and auto-scan settings.

6.2 Scan Management Module

This is the core module of the application. It exposes endpoints to create, retrieve, and cancel scans. Each scan is associated with a user, has a status lifecycle (queued → scanning → completed / failed), and stores structured results including issue type, severity, affected file, and line number.

6.3 Admin Module

Administrators have access to a separate dashboard providing a platform-wide view of all users and their activities. Admin privileges are granted via the isAdmin flag in the User model and enforced through a dedicated adminMiddleware. The admin panel includes user management, activity log viewer with IP address tracking, and platform-wide scan statistics.

6.4 Automated Scanning Module

Users can enable the Auto-Scan feature in their profile settings, specifying a scan interval (default: 7 days). The scheduler runs as a Node.js setInterval loop that checks every hour for users whose auto-scan interval has elapsed. When due, the user's top 5 most recently updated GitHub repositories are fetched via the Octokit API and queued for scanning using BullMQ with a Redis backend.

6.5 Activity Logging Module

Every significant user action is recorded in the ActivityLog collection in MongoDB. Logged events include: REGISTER, LOGIN, SCAN_CREATED, and SCAN_CANCELLED. Each log entry captures the user ID, action type, additional details (such as the repository URL), and the client's IP address. This provides a full audit trail for admin review.

6.6 GitHub Repository Browser

When a user links their GitHub username to their profile, the dashboard provides a New Scan page that allows them to browse their public repositories via the GitHub API proxy route (/api/scans/github/:username). They can then select a repository directly to populate the scan URL field, making it easier to submit scans without manually copying URLs.

7. DATABASE DESIGN

The application uses MongoDB as its primary database, accessed via the Mongoose ODM. The schema design reflects the three core entities of the system:

7.1 User Schema

Table -4: User Schema

Field	Type	Description
name	String (required)	Full name of the user
email	String (unique, req.)	User email used as login identifier
password	String (required)	Bcrypt-hashed password
githubToken	String	Optional personal GitHub access token
githubUsername	String	GitHub username for repository browsing
isAdmin	Boolean (def: false)	Admin privilege flag
autoScanEnabled	Boolean (def: false)	Toggle for automated scanning
autoScanInterval	Number (def: 7)	Days between automated scans
lastAutoScan	Date	Timestamp of the last automated scan

7.2 Scan Schema

Table -5: Scan Schema

Field	Type	Description
user	ObjectId (ref: User)	Owner of the scan
repoUrl	String (required)	Full URL of the scanned GitHub repository
status	Enum	queued scanning completed failed
scanType	Enum	manual automated
results.issues	Array of Objects	Detected issues with type, severity, file, line
results.stats	Object	Counts per severity: critical, high, medium, low
error	String	Error message if scan failed

8. API REFERENCE

Table -6: API Endpoints

Method	Endpoint	Auth	Description
POST	/api/auth/register	Public	Register a new user account
POST	/api/auth/login	Public	Login and receive a JWT token
GET	/api/auth/me	Private	Get current authenticated user profile
PUT	/api/auth/profile	Private	Update user profile and scan preferences
POST	/api/scans	Private	Initiate a new repository scan
GET	/api/scans	Private	Get all scans for the authenticated user
GET	/api/scans/:id	Private	Get a specific scan by ID
POST	/api/scans/:id/cancel	Private	Cancel an active in-progress scan
GET	/api/scans/github/:username	Private	Fetch user's public GitHub repositories
GET	/api/admin/users	Admin	Get all registered users
GET	/api/admin/activity	Admin	Get platform-wide activity logs
GET	/api/health	Public	Health check endpoint

9. SECURITY ANALYSIS ENGINE

The core scanning logic resides in backend/services/scanService.js. It implements a pattern-matching engine using JavaScript regular expressions applied line-by-line across all source files in the cloned repository.

9.1 Detection Patterns

Table -7: Detection Pattern Matrix

Pattern ID	Severity	Detection Target	Example Pattern
aws_key	Critical	AWS Access Key ID	/[A-Z0-9]{20}/g
aws_secret	Critical	AWS Secret Access Key	/[A-Za-z0-9/+=]{40}/g
jwt	High	Hardcoded JWT Token	/eyJ[A-Za-z0-9-_.]+.../g
private_key	Critical	PEM Private Key Block	/-----BEGIN PRIVATE KEY-----/g
generic_api_key	High	Generic API Key	/api_key\s*[:=]\s*.../gi
.env file	Critical	Environment Config File	File path ends with .env
console_log	Low	Debug Print Statement	/console\.log\s*\(/g
todo	Low	TODO Comments	/\s*\s*TODO:/g

9.2 Scan Process Details

The scanning process follows a carefully designed pipeline: URL Normalization ensures the repository URL begins with https:// for git clone compatibility. Repository Cloning uses git clone --depth 1 to fetch only the latest commit. File Discovery recursively enumerates all files, excluding .git and node_modules directories. File Type Filtering scans only files with relevant extensions (.js, .ts, .env, .json, .py, .go, .rb, etc.). Chunk Processing handles files in batches of 25 using Promise.all()

for parallel execution. Issue Aggregation stores all detected issues with file path, line number, severity, and description. Finally, the Scan document is updated in MongoDB with complete results and the temporary clone directory is removed.

10. AUTO-SCAN SCHEDULER

The automated scanning feature provides developers with passive, continuous security monitoring of their GitHub repositories without manual intervention. This module is implemented in backend/jobs/autoScan.js.

10.1 Architecture

The scheduler uses two components from the BullMQ library: a Queue and a Worker, both backed by a Redis connection managed by ioredis. The system is designed to be fault-tolerant — if Redis is unavailable, the application continues to function normally for manual scans while logging a single warning.

10.2 Scheduling Logic

A setInterval loop executes every 3,600,000 milliseconds (1 hour). On each execution, all users with autoScanEnabled = true are fetched from the database. The current timestamp is compared against each user's lastAutoScan date plus their configured autoScanInterval in days. If the interval has elapsed, a new job is pushed to the BullMQ scan-queue. The Worker picks up the job, fetches the user's top 5 GitHub repositories via Octokit, and sequentially scans each using the same processScan() function. The user's lastAutoScan timestamp is updated after all scans complete.

10.3 User Configuration

Users can configure the auto-scan feature from their profile page in the dashboard. They must set their GitHub username to enable the feature. The scan interval can be customized in days (default: 7 days). Auto-scans create Scan records with scanType: 'automated' so they are distinguishable from manual scans in the history view.

11. FRONTEND DESIGN

The frontend is a Single Page Application (SPA) built with React.js and Vite. Navigation is handled by React Router DOM v6 with protected routes. The application is styled entirely with Tailwind CSS utility classes, with Framer Motion providing smooth page and component transitions.

11.1 Application Pages

Table -8: Application Pages

Page / Component	Route	Description
Login	/login	User authentication form with toast feedback
Register	/register	Account creation with validation
Dashboard	/dashboard	Main user workspace with tabbed navigation
New Scan Tab	/dashboard/new-scan	URL input form + GitHub repo browser
Scans History	/dashboard/scans	Paginated table of all user scans
Auto-Scan Tab	/dashboard/auto-scan	Enable/configure automated scanning
Scan Details	/scan/:id	Full report: issue list, severity chart
Admin Dashboard	/admin	Admin-only: user list, activity logs
Profile Page	/dashboard/profile	Update GitHub credentials and preferences

11.2 State Management and API Layer

Global authentication state is managed via React Context API (AuthContext.jsx). The context provides the current user object and JWT token to all components. A centralized api.js service file configures an Axios instance with the base URL and automatically attaches the Authorization header from localStorage.

11.3 Route Protection

The ProtectedLayout.jsx component wraps all authenticated routes. It reads the AuthContext to verify the presence of a valid user session. If the user is not authenticated, they are redirected to the /login page. Admin routes additionally check the isAdmin flag before rendering admin-specific components.

12. DEPLOYMENT

Git Guardian is configured for cloud deployment with separate hosting for the frontend and backend, following modern decoupled deployment practices.

12.1 Backend Deployment - Render

The Node.js/Express backend is deployed to Render as a Web Service. The server.js file includes environment detection logic to avoid running interval-based schedulers in serverless environments. A vercel.json configuration file is included for optional Vercel serverless deployment of the backend as well.

12.2 Frontend Deployment - Vercel

The React frontend built with Vite is deployed to Vercel. The frontend .env.production file contains the production API base URL pointing to the Render backend. Vercel's automatic GitHub integration enables continuous deployment on every push to the main branch.

12.3 Environment Variables

Table -9: Environment Variables

Variable	Location	Description
MONGO_URI	Backend .env	MongoDB connection string (Atlas or local)
JWT_SECRET	Backend .env	Secret key for signing JWT tokens
GITHUB_TOKEN	Backend .env	GitHub personal access token for API rate limits
REDIS_HOST	Backend .env	Redis server hostname (default: 127.0.0.1)
REDIS_PORT	Backend .env	Redis server port (default: 6379)
PORT	Backend .env	Express server port (default: 5000)
VITE_API_URL	Frontend .env	Backend API base URL for Axios

13. TESTING

13.1 Manual Testing

The application was tested manually across multiple scenarios including: Authentication (registration with existing email conflict check, login with wrong password 401, JWT expiry handling); Scanning (public repos with known secrets, repos with .env files, private repos expected failure, malformed URLs, rate-limited GitHub requests); Admin Panel (verifying admin-only access restriction, viewing all users, browsing activity logs); and Auto-Scan (enabling auto-scan, verifying BullMQ job creation, confirming scan records are created with scanType: 'automated').

13.2 Error Handling Scenarios

Table -10: Error Handling

Scenario	Expected Behavior
Invalid / private repository URL	Scan status set to 'failed' with descriptive error message
Git clone timeout (>60 seconds)	Process aborted after 3 retry attempts with error logged

Redis unavailable	Application continues; auto-scan disabled gracefully with warning log
User accesses another user's scan	401 Unauthorized response returned
Non-admin accesses admin routes	403 Forbidden response returned
Scan cancelled mid-process	Process checks status flag and aborts; temp dir cleaned up

14. RESULTS AND KEY FINDINGS

During development and testing, Git Guardian successfully detected security issues in multiple real-world public repositories. The following table summarizes typical detection performance:

Table -11: Detection Performance Summary

Issue Type	Severity	Detection Accuracy
AWS Access Key ID (AKIA...)	Critical	High
JWT Tokens (eyJ...)	High	High
Private Key Blocks (PEM)	Critical	Very High
.env File Present in Repo	Critical	Very High
console.log Statements	Low	High
TODO Comments	Low	High
Generic API Key Assignment	High	Medium

15. LIMITATIONS AND FUTURE SCOPE

15.1 Current Limitations

- The scanner only supports public repositories. Private repositories require a GitHub personal access token with repo scope, which is not currently wired into the clone process.
- The detection engine uses static regex patterns and may produce false positives (e.g., flagging test fixtures that intentionally look like API keys).
- The system does not scan git commit history — only the latest state of the repository is analyzed.
- Binary files and files with encoding errors are silently skipped.
- Large repositories (>500MB) may exceed the 60-second clone timeout.

15.2 Future Enhancements

- Git History Scanning: Traverse all commits to detect secrets that were previously committed but later removed.
- SAST Integration: Integrate established tools like Semgrep or Bandit for deeper static analysis beyond regex.
- Webhook Support: Allow GitHub webhooks to trigger automatic scans on every push event.
- Remediation Guidance: Provide actionable fix recommendations alongside each detected issue.
- Email / Slack Notifications: Notify users when a scan completes or critical issues are found.
- PDF Report Export: Allow users to download a full PDF report of any scan.
- Private Repository Support: OAuth 2.0 GitHub integration for private repositories.
- Machine Learning Enhancement: Train an ML model to reduce false positives in API key detection.

16. CONCLUSIONS

Git Guardian successfully demonstrates the design and development of a practical, end-to-end web application for GitHub repository security scanning. The project integrates a wide range of modern web technologies — from JWT authentication and MongoDB data modeling on the backend to React component architecture and Tailwind CSS on the frontend.

The core security analysis engine provides meaningful detection of critical vulnerability categories including hardcoded credentials, AWS keys, and exposed private keys. The automated scheduling feature extends the tool's utility from a one-time audit tool into a passive security monitoring platform.

This project provided hands-on experience in full-stack web development, RESTful API design, database schema design, asynchronous job processing, cloud deployment, and security engineering principles. It represents a strong foundation for a production-grade security tool that could be extended with the enhancements outlined above.

ACKNOWLEDGEMENT

The authors gratefully acknowledge the guidance and support of the Department of Computer Science and Engineering at NITRA Technical Campus, Ghaziabad. We also thank the open-source community for providing the tools and frameworks that made this project possible.

REFERENCES

- [1] Express.js Official Documentation – <https://expressjs.com/>
- [2] Mongoose ODM Documentation – <https://mongoosejs.com/docs/>
- [3] React.js Official Documentation – <https://react.dev/>
- [4] Vite Build Tool – <https://vitejs.dev/>
- [5] Tailwind CSS – <https://tailwindcss.com/docs/>
- [6] BullMQ Documentation – <https://docs.bullmq.io/>
- [7] Octokit – GitHub REST API Client – <https://github.com/octokit/octokit.js/>
- [8] jsonwebtoken – <https://www.npmjs.com/package/jsonwebtoken>
- [9] OWASP Top 10 Security Risks – <https://owasp.org/www-project-top-ten/>
- [10] GitHub Docs: Secret Scanning – <https://docs.github.com/en/code-security/secret-scanning/>
- [11] MongoDB Atlas Documentation – <https://www.mongodb.com/docs/atlas/>
- [12] Vercel Deployment Documentation – <https://vercel.com/docs>