International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 09 , September - 2026

GhostJS: An Automated JavaScript Reconnaissance Framework for Detecting Exposed Secrets, Hidden Endpoints and Client-Side Security Risks

Sander Ruitenbeek
Co-Founder & Chief Technology Officer (CTO)
Trinetlayer.com

Ketan Jain
Chief Marketing Officer (CMO)
Trinetlayer.com

Yash Chavhan
Co-Founder & CEO
Trinetlayer.com

Lakshay Soni
Security Analyst
Trinetlayer.com

Co-Authors : Shalini Dhiman, Dhruv Sharma

Abstract - Modern web applications rely heavily on client-side JavaScript for user interfaces application logic, authentication workflows and communication with backend services. Because JavaScript resources are delivered to users they may contain artifacts such as API endpoints, administrative routes, internal service references, configuration values, source-map information and credential-like strings. Identifying artifacts can support authorized security assessment and improve visibility into the client-side attack surface.

This paper presents GhostJS, an automated JavaScript reconnaissance framework for identifying and analysing artifacts in web applications. The proposed framework integrates target and subdomain discovery, JavaScript enumeration and collection static analysis, detection endpoint discovery, source-map analysis, contextual analysis, validation finding classification and structured reporting into a unified workflow. The framework maintains traceability between discovered artifacts and their source context to support analysis.

A central principle of GhostJS is the separation of detection, contextual analysis, validation and final classification. This distinction helps prevent patterns from being treated as confirmed vulnerabilities without sufficient contextual or validation evidence. GhostJS is intended to provide an approach for analysing client-side JavaScript resources and supporting authorized web application security assessments.

Keywords: JavaScript Security, Web Security, JavaScript Reconnaissance, Static Analysis, Secret Detection, API Endpoint Discovery, Source-Map Analysis, Contextual Analysis, Security Validation, Client-Side Security

1. INTRODUCTION

Modern web applications have moved from server-rendered pages to very interactive client-side systems that rely heavily on JavaScript. Frameworks and libraries like React, Angular and Vue let applications run a lot of work inside the browser using JavaScript. JavaScript is at the heart of application interfaces, API communication, authentication flows, configuration management and dynamic content generation. At the time client-side resources are sent to users and can be inspected without needing access to the application's internal source code. Security testing guidance specifically recommends reviewing JavaScript resources for routes API keys, internal information, credentials and exposed source-map or debugging files [1].

Academic research has also shown how important JavaScript is for security. JavaScript runs inside a browser. Talks to browser APIs and sensitive data making script security a key area for web-security research. Previous work has looked at JavaScript security policies, information-flow control, browser enforcement mechanisms and static analysis techniques [2].

From an application-security perspective JavaScript resources can reveal details about an application's architecture and function. API paths, administrative routes, internal service references, configuration values and debugging artifacts can appear inside JavaScript bundles when the related features are not shown in the main user interface. This information helps authorized security analysts understand the application's attack surface.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 09 , September - 2026

API security is closely tied to this issue. Modern web applications often depend on APIs to connect client-side parts with backend services. The OWASP API Security Top 10 lists authorization weaknesses like Broken Object Level Authorization and Broken Function Level Authorization as major API security risks [3].

Administrative functionality is especially important because it can give access to operations. However, seeing an administrative route in JavaScript does not automatically mean a vulnerability. The security importance depends on the server-side controls that protect that functionality. Authentication, authorization, object-level access control and response handling must be checked on their own.

Another key concern is credential-like information in client-side resources. Security tools like Gitleaks and TruffleHog show why automated discovery matters, while endpoint-focused tools such as LinkFinder show the benefit of pulling endpoints from JavaScript files [4][6].

However, these capabilities are usually done separately. Security analysts may have to do JavaScript discovery, scanning, endpoint extraction, source-map review, contextual analysis and reporting with several tools. Matching the outputs by hand can add work. Make it hard to keep track of how findings relate.

GhostJS fills this gap by combining these tasks into one unified JavaScript reconnaissance workflow. The goal is not to replace security tools but to give an integrated process for finding and organizing security-relevant artifacts that appear in client-side JavaScript.

The main contributions of this research are:

- A workflow for automated JavaScript reconnaissance.
- Automated identification of security-relevant JavaScript artifacts.
- Combined endpoint and secret discovery with analysis.
- Source-map analysis, as part of the reconnaissance process.
- A detection-to-validation methodology designed to reduce reporting.
- Experimental evaluation demonstrating endpoint discovery capability.

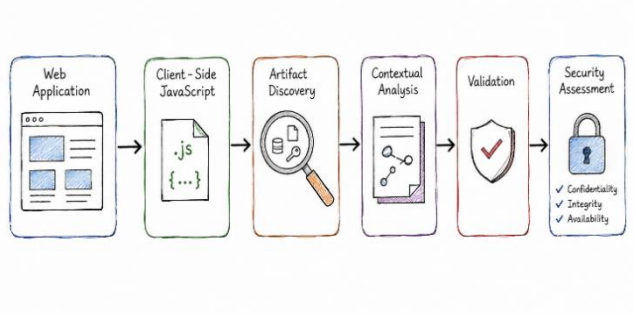

Figure 1. Conceptual Overview of the GhostJS JavaScript Security-Reconnaissance Framework

2. LITERATURE REVIEW

2.1 JAVASCRIPT SECURITY AND STATIC ANALYSIS

People have studied JavaScript security a lot because web applications run scripts inside a browser and those scripts can touch data and use browser features.

Research has looked at both dynamic ways to analyse JavaScript behavior, such as controlling information flow enforcing security policies and program analysis [2]. Static analysis is a way to examine JavaScript without running the app along every possible path.

Studies on JavaScript static analysis show that JavaScript's dynamic features and its complex use of browser APIs make it hard to be both precise and fast [7].

In reconnaissance you do not always need program verification. Security analysts often want to pull out items like URLs, routes, API references, configuration values and strings that look like credentials. Lightweight pattern-based and contextual analysis is handy for this step.

Checking client-side JavaScript is also part of web-security testing. The OWASP Web Security Testing Guide says you should collect JavaScript files and look for data, routes, credentials and exposed source maps [1].

2.2 SECRET DETECTION

Secret detection is another well-known area of automated security analysis. Credentials, API keys, tokens and other sensitive values can be accidentally committed to source code or included in app resources.

Gitleaks is built to find secrets like passwords API keys and tokens and it lets you set rules and reduce false alarms [4].

TruffleHog also looks for secrets by using verification techniques. Its workflow can tell the difference between found and verified secrets showing why validation matters of treating every match as a real credential [5]. These methods back an idea for GhostJS: a string that looks like a secret is just a discovery candidate, not proof that it is an active credential.

2.3 API AND ENDPOINT DISCOVERY

API endpoints are the points where client-side apps talk to backend services. Finding these endpoint references gives insight into how the app works. LinkFinder is a known tool that is made to find endpoints and their parameters inside JavaScript files [6].

Academic work has also used analysis of client-side JavaScript to spot server-side endpoints. These methods show that analysing client code can find server interaction points that simple dynamic crawling might miss [8].

GhostJS takes this idea further by adding endpoint discovery to a reconnaissance pipeline that also looks at JavaScript collection, secret detection, source-map analysis, contextual analysis and classifying findings.

2.4 API AUTHORIZATION AND SECURITY

Endpoint discovery matters a lot for API authorization

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 09 , September - 2026

because just because a route exists does not mean it is correctly locked down. The OWASP API Security Top 10 lists Object Level Authorization and Broken Function Level Authorization as big API risks [3]. Broken Function Level Authorization is especially important when you find admin routes because privileged functions should only be used by authorized users.

Likewise object-level authorization must happen on the server. Just because a user is logged in does not mean the user can access every object or record that the API shows. So, endpoint discovery should be seen as a reconnaissance step that flags areas that need authorization testing not as proof of a vulnerability.

2.5 SOURCE-MAP EXPOSURE

Source maps link minified JavaScript files to their source files. They help in development and debugging. If they are publicly available in production, they can give away extra information.

The OWASP Web Security Testing Guide says that source maps and front-end debugging files should be checked for information leakage [1]. When source maps are exposed, they can make JavaScript easier to read. Can show original filenames, source paths, folder structures and implementation details.

2.6 RESEARCH GAP

Current methods show strengths in separate areas, like secret detection endpoint discovery, static analysis and JavaScript security analysis. These tasks usually live in separate tools and workflows.

GhostJS research points out the practical gap: Gitleaks and TruffleHog mainly find secrets while LinkFinder mainly pulls endpoints; these tools alone do not give a workflow that covers JavaScript discovery, secret detection, endpoint identification, source-map analysis, contextual assessment and structured classification. GhostJS therefore concentrates on integration of replacing each tool. Its contribution is to merge reconnaissance steps into one traceable workflow.

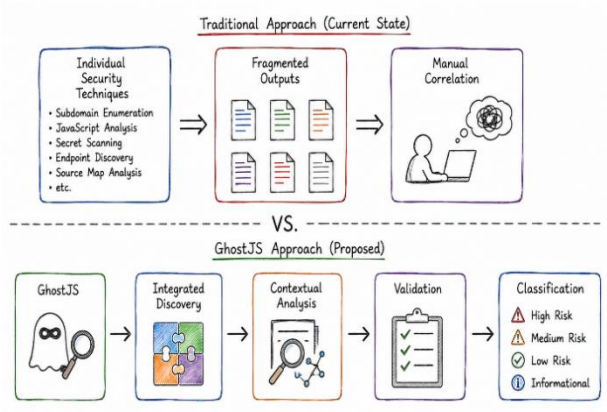

Figure 2. Research Gap and Integrated GhostJS Approach

3. PROBLEM STATEMENT AND RESEARCH OBJECTIVES

3.1 PROBLEM STATEMENT

Modern web applications send a lot of JavaScript code to the browser. This code might include things like API addresses, admin routes, internal services, configuration settings, source maps and strings that look like passwords. Even though this information can be helpful for security tests finding and connecting the dots between these pieces of information from different sources can be time-consuming and hard.

Most security tools focus on one task like finding secrets or finding endpoints. Because of that people who do security checks often have to use different tools and then put together the results to understand how each piece of information connects to the source code, the application and what it means for security.

The research problem that GhostJS is trying to solve is creating a way to collect JavaScript code find security-related information link the findings back to the original source and help with checking the information without saying that something is a real problem when it might not be.

3.2 RESEARCH OBJECTIVES

The objectives of GhostJS are:

1. To automatically find domains, hosts and JavaScript code.
2. To look at JavaScript code for security-related things.
3. To find secrets and passwords.
4. To find references to APIs, admin areas and internal endpoints.
5. To find source maps that are not protected.
6. To connect findings back to the source files and the context they came from.
7. To help with checking and classifying security issues.
8. To reduce alarms by keeping the detection separate from the final check.

To create reports that people can look at.

4. PROPOSED GHOSTJS FRAMEWORK

GhostJS is set up as a complete system, for collecting and analysing information. The process starts with finding the target and the JavaScript code then goes through looking at the code finding things understanding the context checking the results classifying them and making a report. The system is meant for security checks and safe research settings.

The overall process looks like this:

Target → Find domains and subdomains → Find JavaScript files → Collect bundles → Look at the code → Find secrets

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 09 , September - 2026

→ Find endpoints → Check source maps → Understand the context → Check the results → Classify the findings → Make a report

The system keeps track of where each thing was found. This makes it easier to follow where each discovery came from. It helps people not know what was found but also where it was found.

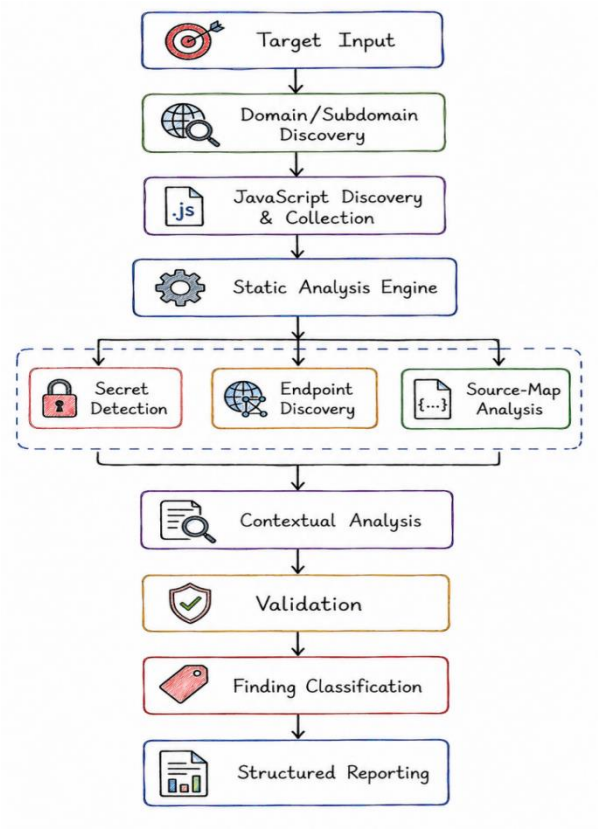

Figure 3. GhostJS System Architecture

4.1 RECONNAISSANCE AND JAVASCRIPT COLLECTION

The first stage identifies the target environment. Collects JavaScript resources that belong to the application. Domain and subdomain discovery can increase reconnaissance coverage because modern applications may spread APIs, authentication services, admin interfaces, static resources and other parts across hosts.

GhostJS then identifies JavaScript resources that belong to discovered application components. These may include JavaScript files, application bundles, dynamically loaded resources and other browser-accessible JavaScript assets.

Each collected resource is linked to its originating host. Kept for later analysis. This link is important because security findings should remain traceable to the source from which they were derived. The proposed workflow explicitly connects resource discovery and JavaScript collection before performing security analysis.

4.2 STATIC SECRET DETECTION

The static-analysis stage examines collected JavaScript resources without changing the target application. The framework searches for artifacts such as endpoint paths, admin routes, internal service references, cloud-storage references, configuration values, token-like strings, API keys and debugging information. A central principle is that Static Detection does not equal Confirmed Vulnerability.

Potential secrets are evaluated using pattern strength, contextual relevance and supporting evidence. A generic confidence model is represented as:

Cs = wpP + wcC + weE

where:

Cs = secret confidence score

P = pattern-match strength

C = contextual relevance

E = supporting evidence

wp, wc, we = corresponding weights

The model is meant for prioritization not proof of credential validity. A credential-like string must therefore move to a validation stage before being classified as a security credential. This model and its interpretation are part of the GhostJS methodology.

4.3 ENDPOINT AND SOURCE-MAP DISCOVERY

Endpoint discovery is a component of GhostJS. The framework searches JavaScript resources for route structures that may match application APIs, admin functionality, internal services or versioned API interfaces.

Typical route patterns may include:

/api/*, /admin/*, /internal/*, /v1/*, /v2/*

Each discovered endpoint is linked to the JavaScript resource in which it was found. This creates a relationship:

Finding → Endpoint → JavaScript File → Source Context

Such traceability helps security analysts determine the context of a discovered route. Source-map analysis adds another layer of reconnaissance.

Source maps can connect generated JavaScript with source files and may reveal original filenames, directory structures, source paths, debugging information and implementation details. GhostJS treats source-map analysis, as a related component of the reconnaissance pipeline.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 09 , September - 2026

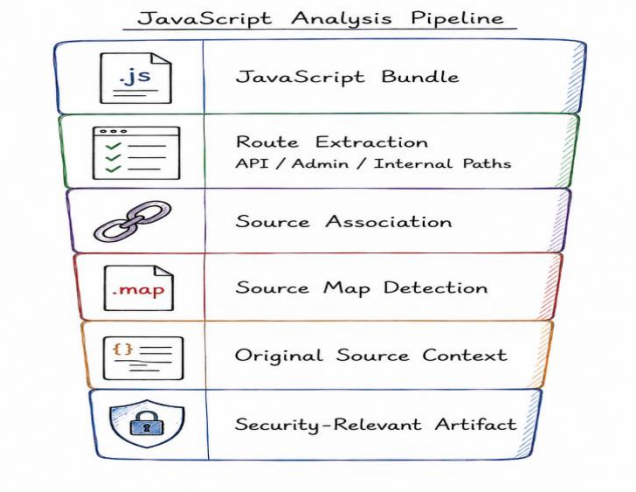

Figure 4. Endpoint and Source-Map Discovery Pipeline

4.4 CONTEXTUAL ANALYSIS AND VALIDATION

Contextual analysis and validation are parts of the process. Just looking at patterns isn't enough to make sure security classifications are correct. A string that looks like an API key could be a placeholder, an example in documentation, a test value or a credential that is not active. Similarly, a route with /admin/ might be properly protected, so it doesn't mean there's a vulnerability.

GhostJS connects the things it finds with details like domain JavaScript file, where the code is, what's around it the type of thing found how sure we are, how serious it is and whether it's been checked. This extra information helps the system tell the difference between things that might matter and things that are harmless.

Checking is done in a step. For something found during endpoint discovery checking might involve seeing if the endpoint is real if it needs a login if the rules for access are working and if people, without permission can get to functions or data.

Just looking at the code on its own can't show what happens on the server. So, an endpoint that is found is a possible security issue until its checked properly with the right access.

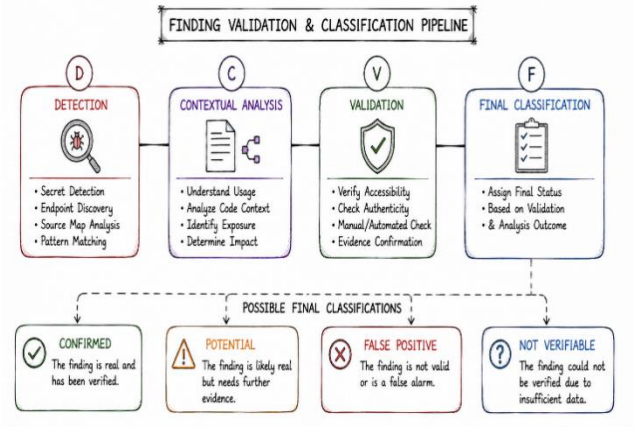

Figure 5. GhostJS Detection-to-Validation Model

4.5 FINDING CLASSIFICATION AND REPORTING

The last step turns the analysis results into security findings. GhostJS looks at things like how serious the issue's how sure we are about it what proof we have if it's been checked, which part of the system is affected and what type of finding it is.

A simple way to show risk is:

$R = I \times L$

where:

R = relative risk

I = how bad the impact could be

L = how likely it is to happen

This way of showing risk is for putting issues in order. It doesn't take the place of other ways to rate vulnerabilities.

A structured finding can have the title of the finding what category it is in, which part of the system is affected, the JavaScript file involved what was found, how sure we are, how serious it is, if it is been checked what the impact is and what to do to fix it. This way the basic information from a scan can be turned into a repeatable security check.

5. EXPERIMENTAL METHODOLOGY

5.1 EXPERIMENTAL DESIGN

The real test was done using the GhostJS scanning tool in a place where it was allowed. The test was about seeing how well the system can work with JavaScript files and find information that matters for security.

The test process consists of these steps:

1. Setting up the target.
2. Finding.
3. Subdomains.
4. Finding JavaScript files.
5. Collecting JavaScript files.
6. Looking at JavaScript files without running them.
7. Finding endpoints.
8. Creating security findings.
9. Looking at the context.
10. Checking if the findings are correct.
11. Classifying the findings.

The test was about showing how the system can find things on its own and put them in categories. It didn't try to use the endpoints it found.

5.2 EVALUATION PROCESS

After setting up the target GhostJS found JavaScript files that were part of the test environment. These files were looked at using the analysis process.

The analysis checked the JavaScript files for structures that

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 09 , September - 2026

show endpoints and other things that matter for security. Any endpoints found were connected to the JavaScript files they came from and sent to the step, which is looking at the context.

The final findings were put into categories based on how sure we're how bad they are and if they have been checked. This process made sure that just because a route was, in client code didn't mean it was a problem that could be used.

5.3 ETHICAL AND SECURITY CONSIDERATIONS

The test was done in a place where it was allowed. The work didn't involve breaking into things without permission using passwords wrong trying to damage systems or taking information more than needed.

The point of finding endpoints was to show how well the system can look around and find places that might need checking. Using the endpoints was not part of the test.

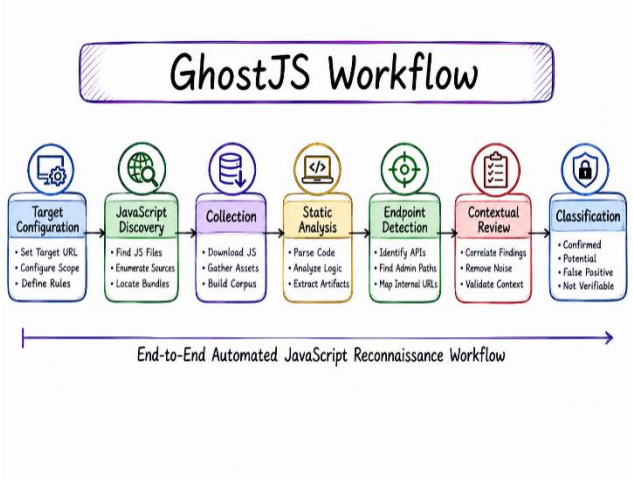

Figure 6. Experimental Evaluation Workflow

6. RESULTS

6.1 JAVASCRIPT PROCESSING

The completed GhostJS evaluation processed 60 JavaScript files connected to the target environment. The processing showed that the framework could collect and analyse a set of client-side resources within a single reconnaissance workflow.

The result shows the usefulness of automated JavaScript collection as an initial attack-surface mapping mechanism. Of relying solely on visible application navigation, the framework looks at client-side resources that may have references to more application functionality.

6.2 ENDPOINT DISCOVERY

The evaluation found 16 administrative/API endpoint paths within the JavaScript source. The discovered functionality was connected with application operations including users, revenue, payments, auditing, scanning, integrations, quotas, broadcasting, engagement and related administrative functionality.

The finding shows that client-side JavaScript can provide information about application functionality. Endpoint names and route structures can help authorized analysts find areas that need security review.

However, the discovery of an endpoint does not mean that the endpoint is publicly available or improperly protected.

6.3 FINDING CLASSIFICATION

The evaluated endpoint observation was given 85% confidence and Low severity while its validation state remained Not Verifiable.

This classification is important because the evidence showed that endpoint references were in JavaScript source. It did not show unauthorized access, authentication bypass, IDOR or information disclosure.

The 85% confidence value should be seen as confidence in the discovery of the endpoint-related artifact than confidence that the endpoint was exploitable.

6.4 INTERPRETATION OF RESULTS

The results show the value of automated JavaScript reconnaissance for attack-surface mapping. The framework was able to process JavaScript resources and find administrative/API references that might not be visible through normal application navigation.

The experiment therefore supports using JavaScript inspection as a reconnaissance layer in web application security assessments.

At the time the results show the importance of telling the difference between discovery and vulnerability confirmation.

7. DISCUSSION

7.1 ATTACK-SURFACE VISIBILITY

The major benefit shown by GhostJS is better attack-surface visibility.

Modern web applications may have functionality that is not directly linked through the navigation interface. JavaScript bundles can still have references to routes used by application components.

The finding of 16 administrative/API endpoint paths shows that client-side resources can provide information about application functionality. Such information can help authorized security analysts decide what to test.

The result is in line with established web-security testing advice, which says to look at JavaScript resources for sensitive routes and other information that can show application functionality [1].

7.2 ADMINISTRATIVE FUNCTIONALITY AND AUTHORIZATION

Administrative endpoints are very important because they may relate to functionality.

However, endpoint visibility should not be mixed up with authorization failure. An administrative route can be in JavaScript. Still need strong server-side authentication and authorization.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 09 , September - 2026

This difference is in line with API security principles. The OWASP API Security Top 10 lists authorization weaknesses as API risks, including Broken Object Level Authorization and Broken Function Level Authorization [3].

Therefore, GhostJS endpoint discovery should be seen as a way to find functionality that needs controlled authorization assessment.

7.3 CONFIDENCE AND FALSE-POSITIVE CONTROL

A part of the GhostJS method is separating detection, contextual analysis, validation and final classification. Observed finding was classified as Low Severity, 85% Confidence and Not Verifiable of being reported as a confirmed vulnerability.

This approach makes automated security reporting more reliable because it stops the framework from treating every route- string as an exploitable issue.

The same idea applies to detection. A pattern that looks like an API key or token might be a test value, a placeholder, an inactive credential or a public identifier. So, discovery should come before validation not replace it.

7.4 PRACTICAL SECURITY VALUE

The experiment results show that GhostJS can help security analysts by cutting down the work needed to find security-relevant details in client-side resources.

The framework does not try to prove that every found item is a problem. Instead, it gives a path from discovery to validation.

This makes GhostJS very useful as a reconnaissance and prioritization step in an authorized security-testing plan.

8. SECURITY IMPLICATIONS

8.1 CLIENT-SIDE JAVASCRIPT SHOULD BE CONSIDERED PUBLIC

JavaScript sent to a browser should not be thought of as a place for private information. Users can look at client-side resources so secrets inside front-end code might be seen by others.

The OWASP Web Security Testing Guide also says to look at JavaScript for routes API keys, credentials and other leaks [1].

8.2 SECRETS SHOULD NOT BE IN FRONTEND CODE

API credentials, private keys, privileged tokens and other sensitive secrets should not be put directly into sent JavaScript. If a like value is found it should lead to controlled checking and fixing not immediate assumptions about it being a problem.

8.3 ADMINISTRATIVE APIS NEED SERVER-SIDE AUTHORIZATION

Administrative functions must be protected by server-side authorization. Frontend controls like hidden buttons or not accessible navigation should not be seen as security limits. The backend must check if the user has permission to do the action.

8.4 ENDPOINT INVENTORY

Organizations should keep a list of exposed APIs and application features. Automated JavaScript reconnaissance can help with endpoint inventory by finding route references in client-side resources.

8.5 SOURCE-MAP REVIEW

Source maps can be helpful during development. Might show the original code structure if they are public. Organizations should check production source maps. Decide if they need to be exposed in the application environment. The OWASP testing advice says to look at source maps and frontend debug files for leaks [1].

9. LIMITATIONS

The current evaluation has some limits.

First the test was done on one authorized target environment so the results can't be applied to all web apps.

Second manual testing of the found endpoints wasn't done. So, the experiment shows endpoint discovery, not confirmed vulnerabilities.

Third the evidence doesn't show authentication bypass, IDOR or confirmed leaks. These need authorized checking.

Fourth static endpoint extraction might find route- strings that don't match real backend endpoints. So, checking at runtime is important.

Fifth, a string that looks like a secret doesn't mean it's active. More checking is needed to see its validity, power and impact.

Finally, the experiment doesn't have enough numbers for precision, recall, F1-score or other large-scale tests. More testing is needed before making claims about detection accuracy.

10. FUTURE WORK

Several improvements can make GhostJS better in research.

10.1 AUTHENTICATED SCANNING

Future versions can support controlled authenticated sessions. This would let endpoint behavior be checked under user roles while keeping authorization limits.

10.2 AUTOMATED ENDPOINT VALIDATION

Automated checking could see if a found endpoint exists, what HTTP methods it uses, if it needs authentication and if authorization changes with roles.

10.3 AST-BASED JAVASCRIPT ANALYSIS

Using Abstract Syntax Tree analysis could help find more by looking at the JavaScript structure of just matching text. This could help find API calls, made routes, token use and configuration objects.

10.4 SOURCE-MAP CORRELATION

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 09 , September - 2026

Future versions could link found issues to source-map positions. This would give analysts readable code context and make checking faster.

10.5 LIFECYCLE VALIDATION

A future secret-checking part could sort credential-like finds by format, service, active status, power, scope and when they expire. Such checking should stay inside authorized areas.

10.6 DIFFERENTIAL SCANNING

Differential scanning could compare GhostJS scans over time. Find new endpoints changed JavaScript files, new secrets or new source maps.

10.7 LARGE-SCALE BENCHMARKING

Future research should test GhostJS with data and security standards. Precision, recall and F1-score can measure detection performance:

Precision = TP / (TP + FP)

Recall = TP / (TP + FN)

F1 = 2 × (Precision × Recall) / (Precision + Recall)

Where TP's true positives FP is false positives and FN is false negatives.

11. CONCLUSION

This research showed GhostJS, an automated JavaScript reconnaissance framework made to find and organize security- items in modern web apps.

The framework includes domain discovery, JavaScript scanning, resource collecting, static analysis, secret finding endpoint discovery, source-map analysis, analysis, validation finding classification and reporting into one process.

The experiment showed the framework could process 60 JavaScript files and find 16 administrative/API endpoints from client-side code. The found features included routes for users, revenue, payments, auditing, scanning, integrations, quotas and more.

The result was 85% confidence, severity and Not Verifiable. That was right because the evidence showed endpoints were found but didn't show access, bypass, IDOR or leaks.

The findings show an idea for good security automation: finding an item doesn't mean it's a problem.

A good reconnaissance tool should give analysts details to know what was found where why it's important and what more checks are needed.

GhostJS does this by connecting JavaScript resources with found items, info, confidence, validation status and reporting. The results show that automated JavaScript reconnaissance can give attack-surface visibility and help, with later API and web app security checks.

REFERENCES

[1] OWASP Foundation, *Web Security Testing Guide (WSTG): Review Web Page Content for Information Leakage*, OWASP Web Security Testing Guide.
[2] S. Maffeis, A. S. Tan, and other contributors, *Survey on JavaScript Security Policies and Their Enforcement Mechanisms in a Web Browser*, Journal of Logic and Algebraic Programming, Vol. 82, Issue 8, 2013.
[3] OWASP Foundation, *OWASP API Security Top 10 – 2023*, OWASP API Security Project, 2023.
[4] Gitleaks Project, *Gitleaks: Detecting Secrets in Source Code and Repositories*.
[5] Truffle Security, *TruffleHog: Secret Detection and Verification*.
[6] Gerben Javado, *LinkFinder: A Python Script that Finds Endpoints in JavaScript Files*.
[7] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, *JSAI: Designing a Sound, Configurable, and Efficient Static Analyzer for JavaScript*, 2014.
[8] D. A. Sigalov, A. Khashaev, and D. Yu. Gamayunov, *Detecting Server-Side Endpoints in Web Applications Based on Static Analysis of Client-Side JavaScript Code*.
[9] T. Brito, M. Ferreira, M. Monteiro, P. Lopes, M. Barros, J. Fragoso Santos, and N. Santos, *Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node.js Packages*, 2023.