Securing applied information systems with SAST integration into the Gulp pipeline*

Ievgen Zaitsev[1,*,2,†], Oleh Bondarenko[2,†], Oleksandr Golubenko[2,†], Artem Antonenko[3,†]and Andrii Savchenko[2,†]

[1] *Institute of Electrodynamics of NAS of Ukraine, Beresteyskyi 56, 03057 Kyiv, Ukraine*
[2] *Yuriy Bugay International Scientific and Technical University, Volodymyrska 7, 04025 Kyiv, Ukraine*
[3] *National University of Life and Environmental Sciences of Ukraine, Heroiv oborony 15, 03041 Kyiv, Ukraine*

Abstract

The growth of cyber threats in modern web development and the need to implement DevSecOps practices demand innovative approaches to code protection. Traditional security testing methods, applied in late stages, face significant limitations, including the inability to respond quickly to vulnerabilities and non-compliance with the "shift-left" principle. This article presents a practical approach to integrating static application security testing (SAST) tools directly into the automated build process for front-end projects using the Gulp task runner. By leveraging the ESLint linter, configured to detect dangerous constructs (such as no-eval and no-implied-eval), the proposed system provides continuous code monitoring and automatic vulnerability detection. The Gulp pipeline, implemented via the gulp-eslint-new plugin, is modified to immediately interrupt the build process (failAfterError) upon error detection. These capabilities contribute to strengthening cybersecurity capacities, serve as an effective first line of defense, and prevent obvious vulnerabilities corresponding to the OWASP Top Ten from entering the repository. Furthermore, the approach demonstrates simple implementation without the need for complex external tools. Through the integration of SAST into the Gulp pipeline, the proposed framework is well-equipped to address modern security challenges, enhancing the overall code reliability in the Node.js environment.

Keywords

Gulp, SAST, ESLint, build automation, cybersecurity, web development, static analysis, code security, JavaScript, DevSecOps

1. Introduction

Analysis of modern sources highlights the relevance of integrating security practices into automated development processes. On the one hand, build automation tools, such as Gulp [1], have become an integral part of modern web development, allowing for the optimization of routine tasks and acceleration of product delivery [1]. They provide a flexible mechanism for managing complex code and resource processing pipelines. On the other hand, the prevalence of JavaScript [2] as the primary front-end language carries significant security risks [3]. According to the OWASP Top Ten [4], many common web vulnerabilities, particularly Cross-Site Scripting (XSS), arise precisely from flaws in client-side code, requiring the implementation of robust secure coding practices [5]. These vulnerabilities in applied information systems can lead to data compromise, financial losses, or, in the context of hybrid threats, the spread of disinformation.

In response to these challenges, the industry is actively adopting DevSecOps [6] approaches, which involve integrating security at all stages of the development lifecycle, starting from code writing ("shift-left") [6]. A key tool for this is Static Application Security Testing (SAST) [7]. As

* *Applied Information Systems and Technologies in the Digital Society (AISTDS-2025), October 01, 2025, Kyiv, Ukraine*
[1] *Corresponding author.
† These authors contributed equally.

zaitsev@i.ua (I. Zaitsev); olegbon.ua@gmail.com (O. Bondarenko); o.golubenko@istu.edu.ua (O. Golubenko); artem.v.antonenko@gmail.com (A. Antonenko); an.savchenko@istu.edu.ua (A. Savchenko)
0000-0003-3303-471X (I. Zaitsev); 0009-0008-2516-143X (O. Bondarenko); 0000-0002-1776-5160 (O. Golubenko); 0000-0001-9397-1209 (A. Antonenko); 0000-0002-8314-6034 (A. Savchenko)

OWASP explains, SAST allows for the automatic detection of potential vulnerabilities directly in the source code before its execution, which is a proactive protection method [7]. Gulp [1] build automation, while accelerating development, does not solve these security problems on its own but merely creates a platform for their integration. This is especially important for runtime environments, such as Node.js, where security depends not only on the application code but also on the configuration of the environment itself [8]. This is precisely why the "shift-left" concept [6] requires the implementation of security control tools directly into the developer's environment and CI/CD processes.

For the JavaScript [2] ecosystem, one of the most popular tools that can perform SAST functions is the ESLint [9] linter. This tool has evolved from a simple code style analyzer to a powerful SAST tool capable of detecting complex code patterns. Its flexible rule system allows configuring checks not only for code style but also for the use of dangerous constructs, such as direct eval calls (rule no-eval [10]) or passing strings to setTimeout/setInterval functions (rule no-implied-eval [11]). Practical integration of ESLint into Gulp is carried out using specialized plugins, for example, gulp-eslint-new, whose documentation is available on npm [12]. However, although the Gulp [1] and ESLint [9] documentation describes their individual capabilities, there is a lack of detailed examples of their joint integration specifically focused on automatically blocking the build upon vulnerability detection. Most existing solutions focus on bundlers, such as Webpack, or full-fledged CI platforms, leaving Gulp users with an insufficient number of practical guides. Thus, there is a need for practical demonstrations of how existing tools can be effectively used to enhance cybersecurity in automated Gulp pipelines [1].

The aim of this research is to develop and demonstrate a method for integrating Static Application Security Testing (SAST) tools into an automated build system for front-end projects based on Gulp. The object of the study is the web application development process, and the subject is the configuration and use of ESLint as a SAST tool in the Gulp pipeline for automatically detecting potential vulnerabilities in JavaScript code. The expected result is the creation of a practical example that shows how a Gulp build can be configured to automatically block code with known dangerous patterns, which will contribute to enhancing the cybersecurity capabilities of web applications in the early stages of development.

2. Methodology

To achieve the stated aim, an approach combining practical implementation and experimental verification was employed [13]. The foundation for the research was an existing Gulp pipeline, designed for automating the build process of front-end projects, which included compiling Pug and SCSS, as well as basic JavaScript [2] processing.

ESLint [9] was chosen as the Static Application Security Testing (SAST) [7] tool due to its widespread adoption in the JavaScript ecosystem, configuration flexibility, and the possibility of integration with Gulp.

The SAST integration was carried out in several stages:

1. ESLint Configuration. The .eslintrc.json configuration file was modified by explicitly enabling rules responsible for detecting common dangerous code patterns, such as no-eval [10] and no-implied-eval [11]. This was done by adding corresponding entries to the "rules" section.
2. Gulp Task Configuration. The gulp-eslint-new [12] Gulp plugin, compatible with ESLint v8 [9], was used. The existing scriptLint Gulp task was updated: cwd (for correctly locating the configuration file) and useEslintrc: true (for using the "legacy" .eslintrc.json format) options were added. A key change was the use of the .pipe(eslint.failAfterError()) method, which ensures the Gulp build stops (fails) if any ESLint errors are detected.
3. Integration Testing. To verify the functionality of the integrated SAST system, code snippets containing known dangerous constructs (eval, setTimeout with a string) were

intentionally introduced into the source JavaScript [2] file. Subsequently, the Gulp dev script was executed to observe its reaction – the expectation was that the build would be automatically interrupted with a corresponding error message.

This methodical approach allowed for the practical implementation and verification of SAST integration based on ESLint into the Gulp pipeline, forming the basis for the research results.

3. System Architecture and SAST Integration

The foundation for integrating static application security testing (SAST) tools was an automated build system based on the Gulp task runner. This pipeline was designed for typical front-end development tasks, allowing for the automation of routine operations and ensuring process consistency. It included the following key functions:

- Compilation of HTML templates from Pug, which simplifies the creation of modular markup.
- Compilation and processing of styles from SCSS, including adding vendor prefixes and minification for production.
- Transpilation and minification of JavaScript [2] code using Babel, ensuring compatibility with various browsers.
- Automatic browser page refresh (BrowserSync), which significantly accelerates the development cycle.

The system architecture involved a division into two main scenarios: dev for development (with a focus on speed and debugging) and prod for creating an optimized final build. An important feature of the base pipeline was its modular structure: the Gulp configuration was divided into separate files by task type, and all paths were centrally managed through a special path.js module, which provided high flexibility and ease of maintenance. The entire Gulp configuration was written using modern JavaScript Modules (ESM) syntax, adhering to current Node.js [8] environment standards. This existing, well-structured pipeline became the foundation for implementing automated code security checks during the build stage.

For the implementation of SAST, the ESLint [9] linter was chosen. This tool is the de facto standard for static analysis of JavaScript code, and although it is often used for style checking, its flexible rule system allows it to be effectively used as a SAST [7] tool as well. A significant advantage is the ability to integrate it into automated build processes, such as Gulp. The primary ESLint configuration is managed through the .eslintrc.json file, located in the project root, which is a "legacy" configuration format compatible with ESLint v8.

The standard ESLint configuration already included the eslint:recommended and google rule sets, which provide a baseline level of code quality checks. To integrate security checks, this configuration was supplemented by explicitly enabling specific rules in the "rules" section. Specifically, the following rules were added:

- no-eval: "error". This rule categorically prohibits the use of the eval() function [10]. As noted in the ESLint documentation and secure coding [5] recommendations, eval() is extremely dangerous as it allows arbitrary code execution from a string, which is a primary vector for XSS attacks [4].
- no-implied-eval: "error". This rule prohibits passing strings instead of functions to setTimeout(), setInterval(), and execScript() [11]. Such constructs are considered "implied eval" as they also execute code from a string and carry similar security risks [4, 5].

Additionally, the no-unused-vars rule (from eslint:recommended) was left active as an example of a general code quality check, which also indirectly contributes to security by reducing the amount of "dead" code.

```
{
  "extends": [
    "eslint:recommended",
    "google"
  ],
  "rules": {
    // ... other style rules ...
    "no-implied-eval": "error", // Disallow implied eval
    "no-eval": "error"         // Disallow direct eval
  }
}
```

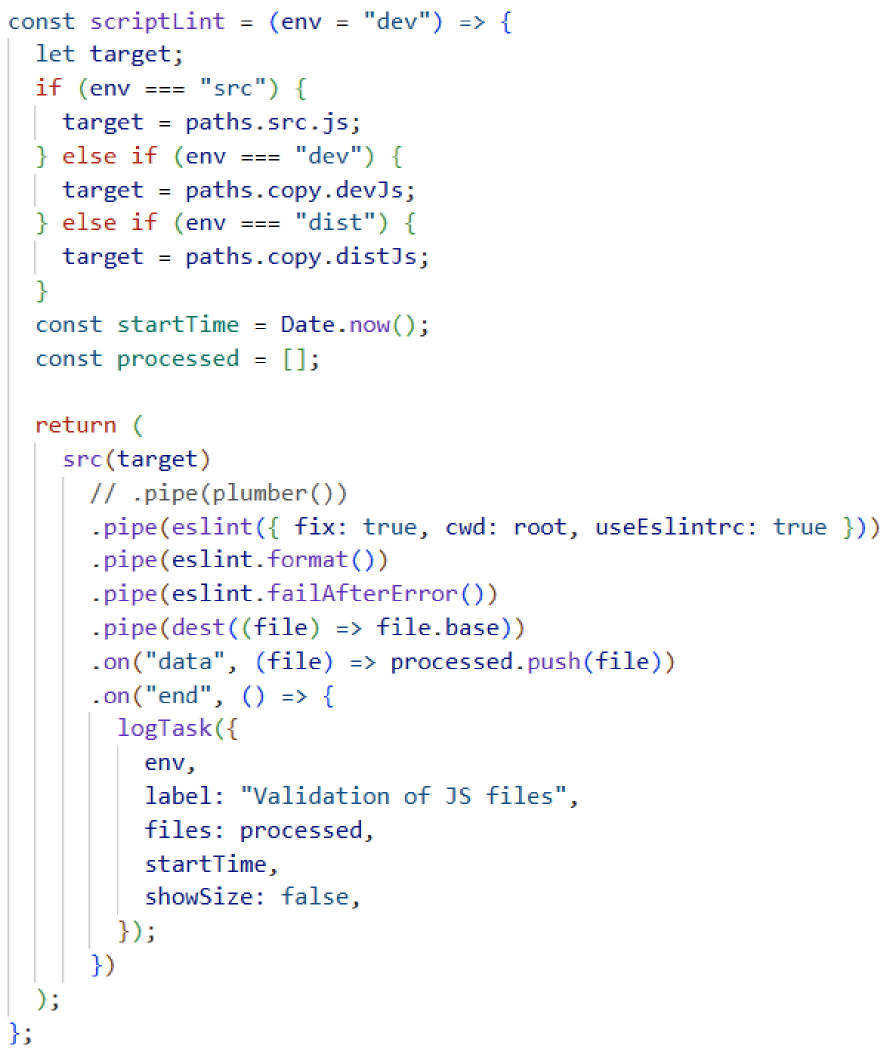
Code 1: Snippet of the .eslintrc.json file with added security rules

This targeted configuration allows ESLint to act as an effective first-level SAST tool, automatically checking JavaScript code for specific, high-risk patterns [10, 11] associated with common web vulnerabilities [4], directly during the build process.

To make these ESLint checks an integral part of the automated build process, the corresponding Gulp task scriptLint was modified. The mere presence of the .eslintrc.json file is insufficient; it is necessary to actively integrate the linting process into the Gulp stream that processes JavaScript files. The gulp-eslint-new [12] Gulp plugin was used to interact with ESLint v8. The choice of this specific plugin was critical, as the original gulp-eslint has serious compatibility issues with modern ESLint versions, which blocks the ability to use "legacy" configurations. This often leads to "silent failures," where linting does not run, but the Gulp task reports no error, creating a false sense of security.

The plugin was configured with the cwd: root and useEslintrc: true options. These options are key for correct operation in this project configuration. The cwd: root option solves the problem of resolving the root directory, as Gulp tasks run from the nested gulpfile.js/ folder, whereas .eslintrc.json resides in the project root. Without this option, ESLint cannot find its configuration file. The useEslintrc: true option, in turn, explicitly tells the plugin to use the "legacy" .eslintrc.json file instead of the new eslint.config.js format, which was necessary for compatibility with ESLint v8 and the google config.

A key element of the integration was adding the .pipe(eslint.failAfterError()) method to the Gulp stream (see Figure 1). This method, provided by the gulp-eslint-new [12] plugin, analyzes the ESLint [9] results and, if at least one error is detected, generates an error in the Gulp stream, leading to an immediate halt of the script. It is important to note that this method must be called after formatting (eslint.format()) but before writing the file to disk (.pipe(dest(...))). If the order were different, then even if an error was detected, the compiled (but unsafe) file would end up in the dev build, negating the entire security effect. Thus, failAfterError() serves as a reliable technical "gatekeeper" that ensures only valid code proceeds. Consequently, code with critical vulnerabilities does not reach subsequent processing stages and cannot be accidentally deployed to a test or production environment. This modification of the Gulp task transforms ESLint from a simple checking tool into an active element of quality and security [7] control within the automated pipeline, aligning with DevSecOps principles.



```
const scriptLint = (env = "dev") => {
  let target;
  if (env === "src") {
    target = paths.src.js;
  } else if (env === "dev") {
    target = paths.copy.devJs;
  } else if (env === "dist") {
    target = paths.copy.distJs;
  }
  const startTime = Date.now();
  const processed = [];

  return (
    src(target)
      // .pipe(plumber())
      .pipe(eslint({ fix: true, cwd: root, useEslintrc: true }))
      .pipe(eslint.format())
      .pipe(eslint.failAfterError())
      .pipe(dest((file) => file.base))
      .on("data", (file) => processed.push(file))
      .on("end", () => {
        logTask({
          env,
          label: "Validation of JS files",
          files: processed,
          startTime,
          showSize: false,
        });
      })
  );
};
```

Figure 1: Snippet of the scriptLint Gulp task with integrated ESLint (Screen reader description: The screenshot shows the JavaScript code of the scriptLint Gulp task, where the gulp-eslint-new plugin is called with cwd and useEslintrc options, and the eslint.format() and eslint.failAfterError() methods are sequentially used to output errors and stop the build, respectively).

4. Experimental Results and Discussion

To practically verify the effectiveness of the integrated SAST system, an experiment was conducted. Its goal was to prove the Gulp pipeline's ability to automatically detect and block code containing known dangerous patterns. Three types of errors, intended to be detected by ESLint configured according to the previous section, were intentionally introduced into the source script.js file:

1. A call to eval() – a direct security threat allowing arbitrary code execution and a vector for XSS [4]. Rule: no-eval [6].
2. A call to setTimeout() with a string argument – "implied eval", carrying similar risks [4] and violating secure coding practices [5]. Rule: no-implied-eval [7].
3. An unused variable – a code quality error that can complicate maintenance and hide logical flaws. Rule: no-unused-vars.

console.log("JS file is connected");

// For demonstrating XSS vulnerability

eval("console.log('This is a security risk!')"); // Error no-eval

// For demonstrating XSS vulnerability (implied eval)

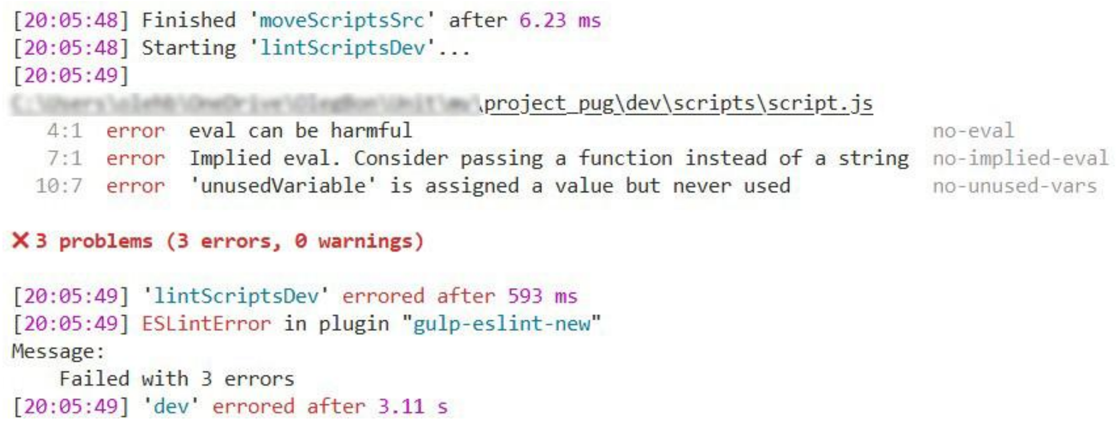
setTimeout("console.log('Implied eval risk!')", 100); // Error no-implied-eval

// For demonstrating ESLint error (unused variable)

const unusedVariable = "This should cause an error"; // Error no-unused-vars

Code 2: Test file script.js with intentionally introduced errors

After saving this file, the standard Gulp script for development (npm run dev) was executed. As expected, the build process was automatically interrupted during the execution of the scriptLint task. A detailed ESLint report was output to the console, clearly identifying all three introduced errors, specifying the rule type (no-eval, no-implied-eval, no-unused-vars) and the error location in the code (see Figure 2). This demonstrates the correct operation of the ESLint configuration and its ability to detect the specified patterns.

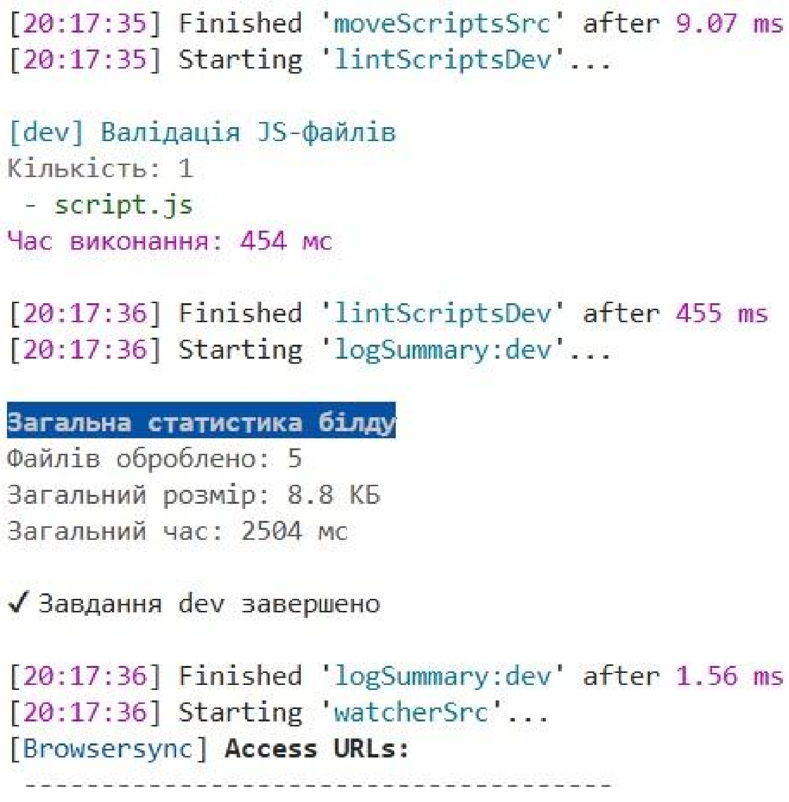


```
[20:05:48] Finished 'moveScriptsSrc' after 6.23 ms
[20:05:48] Starting 'lintScriptsDev'...
[20:05:49]
[redacted]\project_pug\dev\scripts\script.js
   4:1  error  eval can be harmful                                        no-eval
   7:1  error  Implied eval. Consider passing a function instead of a string  no-implied-eval
  10:7  error  'unusedVariable' is assigned a value but never used        no-unused-vars

✖ 3 problems (3 errors, 0 warnings)

[20:05:49] 'lintScriptsDev' errored after 593 ms
[20:05:49] ESLintError in plugin "gulp-eslint-new"
Message:
    Failed with 3 errors
[20:05:49] 'dev' errored after 3.11 s
```

Figure 2: Result of executing the Gulp dev script with errors in the script.js file (Screen reader description: The screenshot shows the console output after running Gulp. ESLint messages indicating three errors in the script.js file are visible: no-eval, no-implied-eval, and no-unused-vars. Below these, a Gulp error message indicates that the build process was halted.)

Due to the presence of .pipe(eslint.failAfterError()) [12] in the Gulp task, the Gulp process itself terminated with an error, preventing further processing of the file and browser page refresh. This is the key outcome: potentially vulnerable code [4, 5] was stopped automatically even before it could enter the repository, testing, or, especially, production.

To demonstrate the full operational cycle and the system's correct behavior in the absence of threats, the Gulp dev script was run again after all three errors in the script.js file were fixed. In this case, the lintScriptsDev validation task completed successfully, detecting no violations. As a result, the entire build process finished correctly, indicated by the "✓ Завдання dev завершено" (✓ dev task completed) message. Immediately after, Browsersync was launched for the local server, which automatically opened the project page in the browser and activated "live reload" mode. This confirms that "clean" code passes through the pipeline unhindered. The result of the successful build is shown in Figure 3.



```
[20:17:35] Finished 'moveScriptsSrc' after 9.07 ms
[20:17:35] Starting 'lintScriptsDev'...

[dev] Валідація JS-файлів
Кількість: 1
 - script.js
Час виконання: 454 мс

[20:17:36] Finished 'lintScriptsDev' after 455 ms
[20:17:36] Starting 'logSummary:dev'...

Загальна статистика білду
Файлів оброблено: 5
Загальний розмір: 8.8 КБ
Загальний час: 2504 мс

✓ Завдання dev завершено

[20:17:36] Finished 'logSummary:dev' after 1.56 ms
[20:17:36] Starting 'watcherSrc'...
[Browsersync] Access URLs:
 --------------------------------------
```

Figure 3: Result of executing the Gulp dev script after fixing the errors in the script.js file (Screen reader description: The screenshot shows the Gulp console output. The 'lintScriptsDev' task starts, validates the 'script.js' file, and completes successfully without any errors. Below, the "Загальна статистика білду" (Overall build statistics) and the "✓ Завдання dev завершено" (✓ dev task completed) message are displayed. The final lines show the launch of 'Browsersync', indicating the successful completion of the entire build process).

The experiment results unequivocally confirm that the integrated SAST system based on ESLint and Gulp successfully performs its function: it automatically detects potentially dangerous and low-quality constructs in JavaScript code and blocks the build process. This implements a crucial element of security control at an early development stage, aligning with DevSecOps principles and Node.js security best practices, minimizing the risk of vulnerabilities entering the final product.

5. Conclusion

The conducted research demonstrates the practical effectiveness of integrating static application security testing (SAST) tools into an automated Gulp pipeline for front-end development. By configuring ESLint and integrating it into Gulp tasks using the gulp-eslint-new plugin, a system was successfully created that automatically detects and blocks potentially dangerous constructs in JavaScript code during the build stage.

Experimental verification confirmed that the modified Gulp pipeline successfully identifies and halts the build process when code violating established security rules (no-eval, no-implied-eval) is present. This proves that even simple, widely available tools, such as ESLint, can serve as an effective first-level SAST means for preventing common vulnerabilities listed in the OWASP Top Ten. Such automated control helps enforce secure coding practices consistently.

Integrating SAST into the Gulp pipeline is an accessible and effective technical measure for enhancing the cybersecurity capabilities of web applications. This approach implements the "shift-left" principle (moving security to earlier stages), aligning with modern DevSecOps practices, and allows for increased overall code reliability without significantly complicating the development process. It also aligns with security best practices for the Node.js environment in which Gulp itself runs.

It is worth noting the limitations of this approach: it is focused on the static analysis of known patterns and cannot replace comprehensive dynamic application security testing (DAST) or penetration testing. Nevertheless, it serves as a critically important first line ofD defense. Prospects for further research lie in expanding the ESLint rule set with specialized security plugins and integrating this Gulp pipeline into more comprehensive CI/CD processes for continuous security monitoring.

The proposed solution can be easily adapted and implemented in existing Gulp projects for the immediate enhancement of their security profile.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] Gulp, "Gulp Documentation," 2025. [Online]. Available: https://gulpjs.com/docs/en/getting-started/quick-start.

[2] Mozilla, "JavaScript," MDN Web Docs, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[3] Mozilla, "Security," MDN Web Docs, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security.

[4] OWASP Foundation, "OWASP Top Ten," 2024. [Online]. Available: https://owasp.org/www-project-top-ten/.

[5] Mozilla, "Website security," MDN Web Docs, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Website_security.

[6] Red Hat, "What is DevSecOps?," Red Hat, 2024. [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-devsecops.

[7] OWASP Foundation, "Static Code Analysis," OWASP Community Pages, 2025. [Online]. Available: https://owasp.org/www-community/controls/Static_Code_Analysis.

[8] Node.js Foundation, "Security Best Practices," 2025. [Online]. Available: https://nodejs.org/en/docs/guides/security.

[9] ESLint, "About ESLint," 2025. [Online]. Available: https://eslint.org/docs/latest/about/.

[10] ESLint, "Rule documentation: no-eval," 2025. [Online]. Available: https://eslint.org/docs/latest/rules/no-eval.

[11] ESLint, "Rule documentation: no-implied-eval," 2025. [Online]. Available: https://eslint.org/docs/latest/rules/no-implied-eval.

[12] npm, Inc., "gulp-eslint-new," npm, 2025. [Online]. Available: https://www.npmjs.com/package/gulp-eslint-new.

[13] Zaitsev I., Golubenko O., Tkachenko O., Pidmohylnyi O., Antonenko A. (2023). Exploring advanced hypothesis generation in astronomy through the implementation of a mathematical model of linguistic neural networks. CEUR Workshop Proceedings, 2023, 3687, 121–128 (Scopus)