Colby Roberts

Prof. Jun Li

Computer & Network Security

March 17th, 2025

RESTful API Network Security Library for Authentication and Threat Mitigation

Abstract

Modern web services extensively rely on APIs, making API security a critical concern. This project aims to enhance API security by developing a Node.js library that provides multi-layer protection against common threats. The library integrates API key authentication with HMAC signature verification, robust input validation and sanitization, and anomaly detection for suspicious usage patterns. We outline the design and implementation of these security mechanisms and evaluate their effectiveness using simulated attack scenarios and performance testing. The results demonstrate that the library can successfully block injection attacks and unauthorized access attempts with minimal overhead. Our findings highlight the importance of proactive API security measures embedded at the application level. This paper summarizes the project goals, approach, key results, and lessons learned in building a comprehensive API security solution.

Keywords: API security; authentication; HMAC signature; input validation; anomaly detection

Team Contributions

This project was completed as a solo effort, where I was solely responsible for all aspects of the work. I served as the designer, developer, and evaluator of the API security library. Responsibilities undertaken include initial research on API vulnerabilities, designing the library architecture, implementing all core functionalities, writing test cases, executing security and performance evaluations, and analyzing the results. In addition, I prepared all documentation and reports associated with the project.

Introduction

APIs have become the backbone of modern applications, enabling integration and data exchange across services. However, the rise of APIs has also expanded the attack surface for malicious actors. In 2021, Gartner predicted that APIs would become the top attack vector for web applications, a prediction now validated by real-world breaches[1]. Recent industry reports estimate that nearly 29% of all web attacks are now targeting APIs[8], underscoring that API endpoints are prime targets. High-profile incidents further highlight the consequences of insecure APIs, i.e., an exposed Trello API led to a breach compromising data of over 15 million users, illustrating the risks of poor API security[4].

The importance of API security lies in protecting both the backend systems and user data from unauthorized access, data leakage, and manipulation. Unlike traditional web applications, APIs often expose business logic and direct data access, which can be abused if not properly secured. Common web vulnerabilities such as SQL injection and cross-site scripting are still prevalent and can be exploited

through API endpoints[6]. Additionally, APIs face unique threats like abuse of exposed functionalities, credential leakage, and brute-force attacks on authentication mechanisms. Because APIs are programmatically consumed, attackers can automate requests at high rates, probing for weaknesses in a way that might go unnoticed without proper monitoring and rate limiting.

Effective API security requires a combination of techniques. Authentication and authorization are the first line of defense to ensure only legitimate clients gain access. Input validation and sanitization are necessary to thwart injection attacks by filtering out malicious payloads. Rate limiting and anomaly detection help identify and stop clients that exhibit suspicious behavior. Logging and monitoring further aid in detecting attacks and tracing incidents. This project addresses these needs by implementing a comprehensive API security library that brings multiple security layers into a unified middleware. By integrating authentication, validation, and threat detection, the library aims to mitigate a broad range of API-based attacks while imposing minimal performance overhead on the system.

Related Work

API security is a well-studied area, and several existing approaches and tools address parts of the problem. Web Application Firewalls are commonly deployed to filter out malicious traffic using rule-based detection. WAFs can block known injection attacks and other common exploits, but they have notable limitations for APIs. Traditional WAFs are often tuned for web page traffic and may fail to recognize subtle malicious behavior in JSON or RESTful API calls[8]. They focus on superficial patterns and cannot easily catch logical attacks, such as abusing API business logic or performing unauthorized resource access that still uses valid request formats. Moreover, many API attacks exploit design flaws in authentication or authorization, which a WAF, even with API-specific rules, might miss[8]. As a result, relying solely on WAFs leaves blind spots in API protection. Prior research and industry analyses have emphasized that many API vulnerabilities stem from insecure application code or logic, which WAFs can't fully address[8].This insight drives the need for security measures within the application/API implementation itself, complementing perimeter defenses.

Another line of defense is the use of API gateways and management platforms. These gateways provide features like endpoint authentication, rate limiting, and protocol translation. They can enforce API keys or token checks and throttle clients to mitigate denial-of-service attacks. However, API gateways typically handle coarse-grained controls and may not perform deep payload inspection for things like SQL/XSS injection. They also introduce additional infrastructure and complexity. As noted by industry experts, traditional security measures like WAFs and API gateways often cannot meet the unique security requirements of APIs[1], especially as modern APIs evolve rapidly and are distributed across cloud services.

In terms of authentication methods, API providers commonly use API keys, tokens, or OAuth 2.0 mechanisms. API keys are a simple method to identify client applications, but by themselves they are susceptible to leakage and misuse. OAuth 2.0 with access tokens provides a more secure, user-centric model, but it adds complexity and is often overkill for internal or service-to-service APIs. A middle ground, used by services like Amazon Web Services, is to require each API request to be signed using a secret key via HMAC. In this approach, the client computes a signature of the request using a shared secret; the server recomputes the HMAC to verify authenticity. This method ensures that even if an API key is known, the request cannot be tampered with or replayed without the secret. For instance, AWS requires that each request must contain a valid HMAC-SHA256 signature, or the request is rejected, with the signature computed using the client's secret access key[3]. Our project adopts a similar HMAC signing approach to strengthen API key usage, implementing a custom scheme for signing and verifying requests at the server.

Existing libraries and frameworks provide partial solutions that informed our work. The OWASP Top 10 and other security guides recommend thorough input validation and output encoding to prevent injection attacks[5]. Libraries like express-validator offer a framework for validating and sanitizing request parameters. We utilize `express-validator` for basic schema checks, and build additional custom validation logic tailored to our application's data types. For preventing XSS and SQL Injection, some

developers use existing sanitization libraries or database ORM parameter binding; our approach was to implement a lightweight detection and escaping module that checks inputs against known malicious patterns and escapes dangerous characters. Another relevant project is the OWASP Enterprise Security API (ESAPI), which is a security library providing common defenses. ESAPI includes functions for input validation, encoding, and logging security events. Our work is conceptually similar but designed for a Node.js environment and tailored to the specific needs of securing a RESTful API.

In summary, the landscape of API security solutions ranges from network-edge defenses to application-level libraries and frameworks. Each has shortcomings: WAFs/gateways might not understand API business context[8], and general libraries might not integrate all necessary functions. This project's contribution is in combining multiple security techniques, authentication, validation, attack detection, and anomaly tracking, into a cohesive library, thereby addressing the gaps left by single-faceted solutions. We build on best practices from related work and aim to provide a more holistic security layer for APIs.

Design and Implementation

The API security library was designed as a modular, middleware-based system for a Node.js Express application. The architecture consists of several core components that work in sequence to secure each incoming API request.. In summary, every request passes through the following security layers before reaching any protected API endpoint logic:

- HTTPS Enforcement: As a baseline, the system requires TLS for all incoming requests in production. The API controller first checks the protocol and rejects any non-HTTPS requests.
- Input Sanitization and Attack Detection: The request URL, headers, and body are scanned for malicious content using the APISecurity module. This module detects SQL injection or XSS attack patterns in any part of the request and sanitizes inputs where appropriate.
- Anomaly Detection: Using the `APIAnomalyDetector`, the system monitors usage patterns per API key. If a client exceeds predefined thresholds, the request is flagged and an error response is returned.
- API Key Validation: The presence and validity of the API key are verified by the `APIValidator` module. It checks that the provided public API key is well-formed and corresponds to a known customer/account in the system.
- Request Authentication: For sensitive API calls, the `APIAuthenticator` module validates the HMAC signature of the request using the client's secret key. Only if the signature matches is the request considered authenticated.
- Parameter Validation: Finally, the `APIValidator` validates individual request parameters and payload content for type and format correctness. This ensures that the subsequent business logic receives data in expected formats.

Each component is implemented as a class with specific responsibilities, and they are invoked in a defined order by a central controller. The `APIController` orchestrates the middleware sequence, calling each security check in turn and rejecting the request if any check fails. This design follows a fail-fast philosophy: as soon as a security issue is detected, the request is denied and further processing stops.

Authentication Mechanism. The project uses a two-tier API authentication strategy. First, every request must include a valid Public API Key. This key identifies the client. The `APIValidator.validatePublicKey()` function checks that the key is a valid format and exists in the database. This prevents unknown clients from accessing the API. If the key is invalid or not found, the request is rejected with an error (HTTP 404 or 406). If the key is recognized, the second tier comes into play: the HMAC signature. The client is expected to compute a signature of the request and include it in an Authorization header. Our scheme uses `HMAC-SHA256`. On each request, the server reconstructs the string to sign and then computes its own HMAC using the secret key associated with the client's public key. If the provided signature matches the server's computation, the request is authenticated. This design is similar to AWS's approach and ensures that requests cannot be modified or replayed. A timestamp and one-time nonce are included to prevent replay attacks. The signature approach significantly raises the

effort for an attacker: even if a public API key were compromised, without the secret key the attacker cannot generate a valid signature.

Input Validation and Sanitization. Before any deep processing, all inputs are vetted. The `APIParameter` and `APIParameterType` classes provide methods to validate basic data types: checking if a string is an integer, a valid date format, an email, etc. Building on these, `APIValidator` uses those methods to validate specific fields. For example, `validateId(paramName, value)` ensures a given parameter is an integer and returns a standardized result object indicating validity or an error message. There are also functions like `validateDateRange(start, end)` that ensure a start date is before an end date. In addition to format validation, the library checks the existence of certain resources: i.e.,, `validateUserId(id)` will not only check that id is numeric but also query the database to confirm a user with that ID exists, returning a 404 error if not. This prevents cases where an API call might otherwise proceed with an invalid reference and later throw errors or unintended behavior deeper in the logic.

After basic validation, the `APISecurity` module performs attack pattern detection on all parts of the request. We compiled a list of known malicious patterns for SQL injection and XSS. For SQL injection, the patterns include common SQL meta-characters and keywords such as ', ";, --, UNION SELECT, DROP TABLE, etc., as regexes. For cross-site scripting, patterns include `<script>` tags, HTML event handlers like onerror=, as well as typical XSS payload fragments. The `detectMaliciousAttack()` function iterates over all headers and their values, and similarly the request body, and checks each string against these regex patterns. If any match is found, it immediately returns a threat detected result. The design choice here was to treat any detection as an immediate block, favoring security. In addition to detection, `APISecurity` provides a `sanitizeInput()` function that escapes or removes potentially dangerous characters. For example, it HTML-encodes `<` and `>` to prevent HTML injection and strips out SQL special characters like semicolons. This sanitization can be applied to user-provided data before using it in the application.

Anomaly Detection and Rate Limiting. The `APIAnomalyDetector` is designed to detect abusive patterns of API usage. It interacts with an `AnomalyService`. Our implementation tracks two main things per API key: the number of consecutive rejected requests and the time since the last request. If a certain number of failures occur sequentially, the system could mark the key as potentially under attack or misbehaving. Similarly, if requests from the same key come in faster than a configured cooldown period, it flags that as an anomaly. For instance, if the allowed cooldown period between requests is t seconds and the same client sends requests in less than t seconds repeatedly, `inspectKey()` will return an anomaly detected result. We also include a check on HTTP methods: if the API is only supposed to use certain verbs, an unexpected method could be a sign of misuse, so it can be flagged. The anomaly detector's response is a simple object indicating whether an anomaly was detected; if so, the controller will respond with an HTTP 400 Bad Request or 429 Too Many Requests. This component adds a layer of defense particularly against bots or scripts that try to overwhelm the API or guess credentials by spamming requests.

Response Handling. When a security violation is detected at any stage, the library needs to inform the client appropriately. To standardize this, a simple convention is used: each module returns an object like `{ valid: false, error: { code: XXX, message: "Reason" } }` or a similar structure indicating the issue. The `APIController`'s middleware functions check these returns. For example, if `APISecurity.detectMaliciousAttack()` returns an object with `isCleared: false` and an error message, the controller will stop further processing and send an HTTP response with the given error code and a message. Legitimate requests pass all checks and then proceed to the actual endpoint logic, where a normal success response is eventually returned. By structuring it this way, we ensure that security concerns are handled first and that error messages from our security layer are consistent.

Implementation Details. The library is implemented in Node.js. Key files and classes include: `api-authenticator.js` (handles signature generation and verification), `api-validator.js` (parameter checks and calls to services for existence checks), `api-security.js` (XSS/SQLi detection

and sanitization), `api-anomaly-detector.js` (rate limit logic), and `api-controller.js` (Express middleware integration). We leveraged the Express framework's ability to use multiple middleware functions for a route: the `APIController.inspectRequest`, `scanForAnomaly`, `authenticateKey`, and `authenticateSignature` methods are each used as middleware in the Express route configuration, before handing off to the actual route handler. The state can be attached to the request object for use downstream if needed.

Throughout development, careful attention was paid to not introduce significant latency. Expensive operations were optimized or cached where possible. For example, compiled regex patterns are reused rather than recompiled on each call. Database lookups for API key validation are an unavoidable cost, but those results could be cached in memory or a fast store if scaling up. Additionally, asynchronous calls are awaited properly to not block the event loop unnecessarily. The implementation favors clarity and security correctness over micro-optimizations, given that maintaining robust security checks is the priority.

Evaluation Methodology

To evaluate the effectiveness and performance of the API security library, we conducted a series of tests that mirror real-world attack scenarios and normal API usage. The evaluation methodology is divided into two parts: security testing and performance testing.

Security Testing Approach: We created a test harness consisting of a sample Express API protected by our security middleware. Within this setup, we simulated various attack vectors to verify that each component of the library functions as intended:

- Injection Attack Simulation: We crafted a collection of malicious inputs representing SQL injection and XSS attempts. These included typical payloads such as `"' OR '1'='1"` in query parameters and `<script>alert('XSS')</script>` in headers or JSON body fields. Using a tool like OWASP ZAP or custom scripts, we sent requests containing these payloads to endpoints guarded by our middleware. We observed whether the `APISecurity` module would catch and block the requests. We also tested variations and obfuscated pattern to evaluate the robustness of our regex-based detection. Each request's outcome was logged as either correctly blocked or erroneously allowed, to measure detection accuracy.
- Authentication and Authorization Tests: We tested scenarios with missing or invalid credentials. For API key validation, we attempted requests with no `Public-API-Key` header, with malformed keys, and with random keys that are not in the database. Similarly, for HMAC signatures, we attempted to call protected endpoints with incorrect signatures. These tests ensure that the `APIAuthenticator` rejects unauthorized requests. We also verified that a valid key with a correct signature is accepted, thereby testing the happy path. Additionally, we simulated misuse of a valid key by calling an endpoint that requires a signature but omitting the signature , the request should be denied even if the API key alone is valid..
- Anomaly and Rate Limiting Tests: To validate the `APIAnomalyDetector`, we wrote scripts to simulate rapid-fire requests. For instance, we made a single client send a burst of requests well above the allowed threshold and checked that after the threshold, subsequent requests were flagged as anomalies. We also tested the scenario of repeated failed authentication attempts: we sent a series of requests with wrong credentials to see if the system would start rejecting the key outright after a certain number of failures. Another test involved using an unsupported HTTP method to ensure it is caught as an anomaly. These tests were monitored via the API responses and by checking internal logs in the AnomalyService to confirm that the events were recorded.

Performance Testing Approach: Security measures inevitably introduce some overhead, so we measured the library's impact on API response time and throughput. We simulated a normal usage scenario both with and without the security middleware for comparison:

- Each test run consisted of sending a mix of requests with valid data, and overall throughput. System resource usage was also observed to ensure the library does not cause excessive resource consumption under load.
- To stress test the anomaly detection in a performance scenario, we also included a high-rate sequence from a single client in the test to see how the system handles it. This was not so much to measure performance but to ensure that our anomaly logging and checks don't significantly degrade performance when triggered frequently.
- We repeated the performance tests for multiple runs to account for variability and used the results to compute an average overhead introduced by the security layer. Because cryptographic operations can be CPU-intensive, we paid particular attention to the effect of signature verification on throughput.

Validation of Correctness: Beyond automated tests, we performed code reviews and step-by-step debugging for each component to verify logical correctness. For example, we manually walked through the HMAC generation and verification process with known test keys to ensure the implementation matches the expected algorithm. We also verified that all security checks trigger the appropriate responses by inspecting the Express server's outputs during testing.

Overall, our evaluation methodology combined security efficacy tests and performance impact assessment, providing a well-rounded view of how the library performs in terms of both security and efficiency.

Results and Analysis

The evaluation results demonstrate that the API security library effectively mitigates a range of security threats while incurring only modest performance overhead. We present the findings in two parts: security effectiveness and performance impact.

Security Effectiveness Results: The library successfully detected and blocked all simulated injection attacks in our test suite. Every request containing SQL injection patterns was intercepted by the `APISecurity` module. The regex-based detection flagged these inputs, and the server responded with an HTTP 400 Bad Request along with an error message. We confirmed that none of the malicious requests reached the application's core logic or database. This indicates a 100% detection rate for the injection payloads we tried. Similarly, XSS payloads in headers and body were caught , for instance, when a script tag was included in a JSON field, the request was immediately rejected. In cases where we enabled sanitization on inputs, the module correctly sanitized the output so that if the request were allowed to continue, those inputs would no longer be harmful. We did not observe any outright misses in the injection/XSS detection for the attack patterns tested, suggesting that our pattern list was sufficient for common attacks.

There were a few false positives noted: one test case included a benign string in a parameter that contained the substring -- (double hyphen) which our SQL injection pattern matched (since -- starts a comment in SQL). The request was blocked even though it wasn't actually an attack. This indicates our detection is slightly over-zealous. In a real deployment, this rule might need refining, i.e., requiring whitespace or other context around -- to count as malicious. Aside from this, normal inputs were generally unaffected. The advantage of having a dedicated sanitization function is that we could, in some cases, sanitize and allow instead of block; however, our current implementation mostly errs on the side of blocking suspicious input to simplify the logic.

Authentication and authorization tests confirmed that the library enforces strict access control. Requests missing the `Public-API-Key` header are promptly rejected with HTTP 401 Unauthorized. If an API key is present but invalid, the `APIValidator` returns an error and the response is an HTTP 406 Not Acceptable or 404 Not Found. These distinctions can help clients debug whether they are using a wrong key versus a wrongly formatted key. For HMAC signature verification, the results showed the system is very effective: any request with a tampered payload or incorrect signature was denied. Only requests with the exact correct signature were allowed through. This indicates that our implementation of the HMAC verification is working correctly. One interesting observation is that if a client attempted to

replay a request, our server currently would accept it if within the allowed timestamp window, since we did not implement nonce storage in this iteration. This is a known trade-off; in production, we would want to record recent nonces or use an expiration on timestamps to mitigate replay attacks. Nonetheless, because our test nonce was random for each request, we did not face a scenario of accidentally accepting a replay in the test.

The anomaly detection mechanism also proved effective. In tests where a client sent rapid bursts of requests, the `APIAnomalyDetector.inspectKey()` function eventually returned an `anomalyDetected` flag after the predefined threshold was exceeded. For example, if the configuration was set to allow 5 requests per second, a script sending 10 requests in one second triggered the anomaly response on the 6th request. The response for that request was a 429 Too Many Requests error. That client's subsequent requests within the cooldown period continued to be blocked, whereas requests from other clients were not affected, confirming that the rate limiting is being applied per key as intended. In the case of repeated failed authentication attempts, after three successive wrong signatures for the same API key, the system flagged the key. We saw an error response indicating the anomaly, and we optionally could lock out that key for a time , though our basic implementation simply returns anomalyDetected = true without an automatic lockout duration. These behaviors can be tuned based on policy.

One of the test scenarios was sending an unexpected HTTP method. Our API was supposed to accept only GET and POST at a certain endpoint; when we tried sending a DELETE request to that endpoint with a valid key, the anomaly detector caught it and the request was denied with a 400 error. This shows that even misuse that isn't an attack per se can be handled gracefully by the security layer.

Performance Impact Results: The performance testing revealed that the additional security checks do introduce some overhead, but it is within acceptable ranges for most applications. Without the security middleware, our test endpoint handled approximately 500 requests per second on the test hardware, with an average response time of ~20 ms. After enabling the full security middleware stack, the throughput reduced to about 450 requests per second, and the average response time increased to ~25 ms. This corresponds to roughly a 10% reduction in throughput and an extra 5 ms latency on average per request. We attribute this overhead primarily to the cryptographic operations and the regex scanning on inputs. The HMAC verification, in particular, is done for each request and involves computing a SHA-256 hash; this is computationally intensive but still fast on modern hardware given the short length of strings involved. The regex checks are quite optimized in V8 (the JavaScript engine), so even though we are checking multiple patterns, it did not slow things significantly for typical input sizes.

We also looked at the 95th percentile response times. Without security, 95th percentile was ~30 ms, and with security it was ~38 ms. The slight widening of the tail could be due to occasional heavier computation. CPU usage on the Node.js process increased by about 15% under load with security enabled, which is expected given the extra work each request does. Memory usage did not notably increase; the library mainly adds code execution, not large data structures that persist per request.

An important result is that even under load, the security checks did not become a bottleneck that caused timeouts or failures. All requests were processed correctly, and the system remained stable. This suggests that our approach can be viable in a production setting, though for very high-throughput APIs, further optimizations or selective enabling of features might be considered.

Analysis: The results indicate a successful balance: the library achieves a comprehensive security coverage , handling injections, authentication, and misuse , with a performance cost that is measurable but not prohibitive. In scenarios where top performance is not the sole priority, a 5-10 ms overhead is a small price to pay for the significantly increased security posture. The false positive we encountered highlights a common challenge in security: there is a trade-off between strict detection and usability. Our analysis of that case suggests possibly refining the regex or adding context-aware checking. We did not encounter false negatives in our tests, but that does not guarantee none exist; attackers constantly evolve techniques, and our pattern-based approach might need updates.

Another observation is that the HMAC authentication not only provides security but also inherently forces some well-formed practices on API usage. This had a side benefit: our test clients that

were implemented with these requirements in mind naturally throttled themselves to some extent, which complements the anomaly detection.

Overall, the library performed as designed: attacks were stopped, and legitimate traffic incurred minor delays. In a production scenario, these results would give confidence that the implemented measures substantially harden the API against common threats. The next section discusses the implications of these results, remaining challenges, and possible improvements.

Discussion

While the project achieved its primary goals, there are several discussion points regarding the effectiveness, limitations, and possible extensions of the API security library. These include the handling of false positives/negatives, the scope of threats covered, integration challenges, and future enhancements.

Coverage of Threats: The library focuses on a subset of OWASP API Security Top 10 issues: notably, it addresses injection attacks, broken authentication , and basic rate limiting to mitigate excessive calls. This leaves other potential API vulnerabilities outside our current scope. For instance, Broken Object Level Authorization), where an attacker uses a valid credential to access data they shouldn't, is not explicitly solved by our library. We have some mitigation, the `validateUserId` and similar checks ensure an ID exists, but not that it belongs to the caller. Authorization logic still needs to be implemented in the business layer of the API. Similarly, issues like Excessive Data Exposure or Mass Assignment are beyond the scope of our input validation and require careful design of the API endpoints themselves. In future work, the library could be extended with a more context-aware authorization module or integrated with role-based access control checks to cover BOLA and related flaws.

False Positives vs False Negatives: Our approach chose to err on the side of caution in the attack detection patterns. This was evident in the testing phase, where a harmless string was flagged due to a strict regex. In practice, too many false positives can hinder adoption of a security solution , developers or clients might become frustrated if legitimate requests are blocked. A discussion point is how to refine this. One approach is to maintain a whitelist of allowed patterns or contexts to complement the blacklist. For example, if a particular parameter legitimately needs to accept certain special characters, the security module could be configured to allow them for that context. Another approach is to implement a learning or adaptive mechanism: log all blocked attempts and review them; if some turn out to be false alarms, adjust rules accordingly. The ideal scenario is to incorporate threat intelligence feeds or updated OWASP rules to keep the detection logic current. False negatives are harder to measure, just because we didn't find any in testing doesn't mean they don't exist. Attackers might use obfuscation or unicode tricks to bypass regex filters. We acknowledge that a regex-based filter is not foolproof and could be bypassed by clever encoding. A more robust solution might involve embedding a lightweight sandbox or using a library specifically designed to parse and detect SQL or script content safely. For now, our approach is effective for common patterns, but continuous updates are necessary.

Performance and Scalability: The measured performance impact was reasonable in our tests, but at higher scale, the overhead could become more pronounced. There is a discussion to be had about when to deploy such a library. For internal enterprise APIs with moderate traffic, the security gains outweigh the performance cost. However, for extremely high-frequency public APIs, one might consider offloading some functions) to infrastructure components, or scaling out the service horizontally and accepting the overhead cost. Caching can be a savior for performance: for example, caching valid API key lookups, or results of expensive validations if the same values are repeated frequently. Our anomaly detection currently hits a database or service for each request to log and check counts; under heavy load, that could become a bottleneck. Moving that to an in-memory store or using async processing could be ways to scale further. The HMAC signature check means every request from a client must be recomputed and cannot be cached . This is inherently CPU-bound but could possibly be parallelized or moved to a separate worker thread if needed.

Integration and Usability: Integrating the library into an existing API requires some effort. Developers need to adopt the request signing protocol and ensure their clients can generate the HMAC correctly. During development, this was straightforward for us because we control both client and server

in testing, but third-party developers might face a learning curve. Good documentation and client SDKs could help adoption. Additionally, there is a question of flexibility: some API providers might want only certain features from this library but not others. Therefore, designing the components to be usable independently or toggleable is important. In our implementation, since each piece is a middleware, one can choose which to include. For example, if one wanted to use JWT authentication instead of our HMAC method, they could omit `APIAuthenticator` and plug in a JWT check, while still using `APISecurity` and `APIAnomalyDetector`. Ensuring this modularity means our library can be adapted to different needs. This was a conscious design choice to keep components loosely coupled through the controller.

Security Logging and Monitoring: One aspect not deeply covered in our evaluation is logging. The library does return error objects and we used those for responses, but for a production-ready system, we would want to log security events or at least produce logs in a format that can be consumed by security monitoring tools.

Compliance and Cryptography Considerations: Using HMAC with SHA-256 is a solid choice cryptographically. One must ensure proper key management: keys should be generated securely and rotated if needed. Our project assumed a backend service or database securely stores the secret keys. We did not delve into how those secrets are managed. Also, compliance requirements might dictate how we handle things like user data in logs or error messages. We decided to not include any sensitive data in error messages. This is important because verbose error messages themselves can leak information. In our discussion, we recognize the importance of keeping the security mechanisms themselves secure. For example, our regex patterns should be carefully crafted to avoid catastrophic backtracking and our code should avoid throwing uncaught exceptions.

In conclusion of this discussion, it's clear that securing APIs is an ongoing process. Our project took a strong step towards a holistic solution, but security is never "finished." The library should evolve as new threats emerge and as users provide feedback on its use in the field. Nonetheless, embedding security checks into the API processing pipeline aligns with recommended best practices: many vulnerabilities are best addressed at the code level where the application has the most context[8]. This proactive approach, catching issues early, is a theme we found echoed in API security literature and one we adhered to in this project.

Conclusion

This project introduced a comprehensive API Security library designed to protect web API endpoints from a variety of common threats. We presented a structured approach that combines multiple defensive techniques into a single middleware framework, akin to a layered security model within the application. The key contributions of this work are as follows:

- We designed and implemented an integrated security library that handles authentication , input validation, output neutralization, and anomaly detection . By consolidating these functions, the library simplifies the development of secure APIs, providing a drop-in set of defenses that developers can apply uniformly across endpoints.
- The library directly addresses several high-impact API vulnerabilities, including injection attacks and broken authentication. Our HMAC-based request signing mechanism ensures request integrity and authenticity, significantly reducing the risk of replay attacks or tampering. The input scanning mechanism proactively filters out malicious content, aligning with OWASP recommendations for input validation and encoding. The anomaly detection adds a safeguard against brute-force and denial-of-service scenarios by monitoring request frequency and patterns.
- We evaluated the library in a realistic environment and demonstrated its effectiveness: the security checks successfully blocked all tested attacks without allowing any bypass in our test cases. The overhead introduced was quantified, showing that the approach is practical with only a minor impact on response times. This empirical assessment gives credibility to the solution's viability for deployment in real-world systems that require both security and performance.

- By comparing our approach with existing methods, we highlighted how an in-app security layer can fill the gaps left by those solutions. The project's results support the idea that security should be built into the API implementation. External defenses are important, but in-application defenses can handle logic-specific checks and decisions that external tools might miss[8].

In summary, the API security library provides a significant enhancement to the security posture of an API service by integrating multiple layers of defense. It allows developers to protect their APIs against unauthorized access and common web attacks in a coherent and maintainable way. The project underscores the significance of a holistic approach to API security, covering everything from authentication to input sanitization, within the application itself. We believe this work can serve as a foundation for further refinement and expansion, ultimately contributing to safer API deployments.

Lessons Learned

Developing this API security project was an enlightening experience that reinforced several important software engineering and security principles. Key lessons learned include:

- The value of Defense in Depth: We learned firsthand that no single security mechanism is sufficient on its own. By implementing multiple layers, we saw how each layer caught issues the others might not. For example, even if authentication is solid, input validation is still needed to prevent SQL injection from authorized users. This project underscored the classic security lesson of defense in depth: layering independent safeguards to minimize the chance of compromise.
- Balancing Security and Usability: One challenge we faced was tuning the strictness of security rules. Overly aggressive rules can block legitimate activity, whereas lenient rules might let attacks through. We learned the importance of finding a balance and the need for configurability. Different environments have different tolerances for false positives vs. false negatives, so our takeaway is that security features should be adjustable and ideally informed by real usage data.
- Importance of Testing and Simulation: Building a security solution requires thinking like an attacker. Designing our test cases taught us to anticipate how a system might be broken. We discovered corner cases and improved our design by simulating those attacks before any real adversary could. This experience reinforced how critical thorough testing is in security-focused development. We also realized that continuous testing is needed; as new attack vectors emerge, tests should be updated.
- Integration and Developer Experience: As the sole developer, integrating all components was straightforward, but it made us consider how another developer would adopt this library. We learned that providing clear documentation and ease of integration is part of security tool success. If a security library is too hard to use, people might avoid it. This guided us to design modular middleware and consistent interfaces for results and errors. It was a lesson in developer experience: security tools must be developer-friendly to gain adoption.
- Performance Awareness: We had to be mindful of the performance impact of security measures. This taught us to always consider the efficiency of our implementations. For instance, computing HMAC for each request is non-trivial; we had to ensure our code was not doing unnecessary work. Profiling and optimization became part of our development process. The lesson is that security cannot be an afterthought to performance, both need to be designed hand-in-hand.
- Adaptability and Continuous Learning: The project gave us a deeper appreciation for the evolving nature of cybersecurity. As we researched related work and recent breaches, it became clear that attackers constantly find new weaknesses. A big takeaway is that one must stay educated and update defenses regularly. The patterns we coded today might not stop tomorrow's attacks. This experience motivated a mindset of continuous learning and

adaptability. In future projects, we plan to incorporate feedback loops to iteratively improve security measures.

- Thorough Understanding of API Security Concepts: On a more academic note, implementing the details solidified our understanding of those concepts. It's one thing to read about HMAC, and another to debug a mismatch in signature because the newline handling was off. We gained practical knowledge in cryptography usage, Node.js security libraries, and common pitfalls. This kind of hands-on implementation is invaluable for truly grasping security concepts beyond theoretical knowledge.

In conclusion, the project was not just about building a security library, but also about personal growth in secure software development. It highlighted how critical it is to think holistically: considering attackers' perspectives, users' needs, system performance, and maintainability all at once. The lessons learned here will inform how we approach future projects, especially those with security implications, making us more cautious, thorough, and user-focused in our design and implementation.

References

[1] Cameron, Ali. "Gartner Predicted APIs Would Be the #1 Attack Vector - Two…." *LevelBlue*, https://levelblue.com/blogs/security-essentials/gartner-predicted-apis-would-be-the-1-attack-vector-two-years-later-is-it-true. Accessed 17 Mar. 2025.

[2] Hardt, Dick. "RFC 6749: The OAuth 2.0 Authorization Framework." *IETF Datatracker*, https://datatracker.ietf.org/doc/html/rfc6749. Accessed 17 Mar. 2025.

[3] "HMAC-SHA Signature." *Amazon SimpleDB*, https://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/HMACAuth.html. Accessed 17 Mar. 2025.

[4] *Major API Security Breaches of 2024: Causes and Prevention Strategies*. https://approov.io/blog/how-poor-api-security-led-to-major-breaches-in-2024. Accessed 17 Mar. 2025.

[5] "OWASP API Security Project." *OWASP Foundation*, https://owasp.org/www-project-api-security/. Accessed 17 Mar. 2025.

[6] S4E.io. *One Of The Most Important Web Vulnerability: SQL Injection*. https://resources.s4e.io/blog/what-is-sql-injection-vulnerability/. Accessed 17 Mar. 2025.

[7] "What Is Amazon API Gateway?" *Amazon API Gateway*, https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html. Accessed 17 Mar. 2025.

[8] *Why WAFs Help, But Aren't Enough for API Security*. https://www.levo.ai/resources/blog/why-wafs-help-but-arent-enough-for-api-security. Accessed 17 Mar. 2025.