Reverse Proxy Security System: Design and Implementation Report

Isaac Dominguez
CECS 478

Problem Statement

Modern web servers face a wide range of security and privacy risks regardless of their scale or purpose. Whether hosting a simple static website, running a personal gaming server, or supporting a large-scale web application, any exposed server becomes vulnerable to attacks such as denial of service, buffer overflows, cross-site scripting, SQL injection, and various forms of unauthorized probing. Even with strong firewall rules, restrictive network policies, or hardened configurations, traditional defensive measures address only a narrow subset of possible threats and often lack adaptive or layered protections.

This project explores a reverse proxy–based security system designed to reduce exposure, regulate traffic behavior, filter malicious content, and protect backend infrastructure. The approach centers on intercepting client traffic before it reaches the backend server, enabling preemptive detection, enforcement, and mitigation. The system aims to demonstrate how a strategically placed proxy can provide a flexible, centralized, and extensible security layer capable of enhancing resilience against common attack patterns. The prototype implements rate limiting, content inspection, packet size controls, progressive warning mechanisms, and detailed logging as an integrated defensive framework.

System Architecture and Design

The system is made up of three main parts: a client that can send normal or malicious traffic, a proxy that filters and applies security rules, and a backend server that only handles traffic the proxy approves. The client can send regular messages as well as oversized packets, rapid bursts, and other payloads meant to imitate common attacks. The proxy sits between the client and the server and acts as the main protection layer. It checks every packet, allows safe ones to pass, blocks harmful ones, and keeps communication running smoothly when no issues are found. The backend server is kept simple on purpose and only echoes back messages it receives from the proxy, making sure no untrusted traffic ever reaches it directly.

The client module is a TCP program built with Python sockets. It has an interactive interface that lets you send messages in real time, and a separate thread listens for server responses at the same time. The client also includes a set of test packets with different sizes and formats to help check whether the proxy's filters work correctly.

The proxy module is the core of the system. It includes several features that work together to enforce security. The rate limiter tracks when each packet arrives and decides whether a client is sending messages too quickly. This prevents spam-like behavior without affecting normal traffic. The filtering system checks packet sizes, looks for patterns that resemble HTTP requests, and scans for simple signs of code injection. When the proxy detects a violation, it logs the event with a timestamp and packet details. It also keeps track of repeated violations and closes the connection if the client continues to break the rules.

The backend server is very simple. It only receives packets that the proxy forwards, sends back acknowledgment messages, and acts as a controlled environment to test how well the proxy blocks unwanted traffic. Since the server is hidden behind the proxy, outside clients never connect to it directly.

Communication between all components uses standard TCP/IP with a basic text-based protocol. Each message ends with a newline, which makes parsing easier. The proxy listens on port 8000 and forwards allowed traffic to the server on port 9000. This setup keeps the server hidden and ensures all traffic passes through the proxy first.

A reverse proxy design was chosen because it gives a single point of control for security and hides the backend server's real address. This reduces the attack surface and makes the system easier to manage. The sliding-window rate limiter was selected because it handles bursts of traffic more fairly and accurately than simpler methods. The warning system helps prevent accidental blocks by giving users feedback before cutting them off entirely.

Docker was used to separate the client, proxy, and server into different containers. This makes the system easier to deploy, avoids configuration issues, and allows each part to be tested or updated without affecting the others.

Threat Model and Security Assumptions

The system assumes that both the proxy and backend server operate within a controlled and uncompromised environment. Docker's virtual networking and isolation mechanisms are treated as trustworthy, and the logging infrastructure is considered secure from tampering. In contrast, all external clients are treated as untrusted entities capable of sending arbitrary, malformed, or malicious data. The network path between the client and proxy is assumed to be susceptible to monitoring or manipulation.

The attacker model includes capabilities common in real-world scenarios. An attacker may craft arbitrary TCP packets, manipulate timing patterns to simulate denial-of-service attempts, send oversized packets designed to overflow buffers, and generate payloads containing scripting commands, code fragments, or protocol confusion attempts. They may also attempt to imitate HTTP traffic or embed hidden malicious content. However, the attacker cannot directly communicate with the backend server, bypass the proxy’s rate limiting, modify proxy logic, or interfere with internal logs or configuration states.

The trust boundary exists at the proxy, which separates untrusted client traffic from the trusted backend server. Only traffic validated by the proxy crosses this boundary. The server therefore assumes any received packet has passed through a meaningful level of security inspection. Risks such as denial of service, buffer overflow attempts, protocol injection, and content based attacks are mitigated through the proxy’s filtering mechanisms. Remaining risks include distributed attacks from multiple originating IPs, the possibility of encrypted payloads concealing harmful content, and the potential for resource exhaustion under extreme loads. Although the proxy handles a wide range of attacks effectively, some advanced evasion techniques or zero-day vulnerabilities remain possible.

Methods and Experimental Procedures

The system was implemented in Python using the socket and threading libraries. Docker containers were built with lightweight Python-based images to create clean and consistent environments for each component. The proxy’s main algorithms include the sliding-window rate limiter, pattern-based HTTP detection, and keyword-driven code-injection checks. The rate limiter works by tracking timestamps for each incoming request. When a new packet arrives, any timestamps older than one second are removed, and the remaining number is compared with the allowed limit. HTTP detection looks for common request lines, header names, and version markers, while code detection checks message contents for programming-related keywords and symbols that appear together.

Testing was performed by sending specific sequences of packets from the client to the proxy. Normal traffic consisted of simple alphanumeric messages that should pass through without issues. Oversized packets were generated by exceeding the 50-byte limit, and rate-limit tests involved sending more than five packets within a single second. Additional tests included HTTP style messages, code-like strings, packets with special characters, embedded null bytes, and multi-line inputs. Throughout testing, the proxy applied its rules by blocking unwanted packets, logging violation details, issuing warnings, and closing connections after repeated offenses.

The metrics gathered during testing included the number of blocked packets, the types of violations detected, and the timestamps for each logged event. General performance behavior, such as processing speed, memory usage for tracking clients, and thread activity, was observed through console output and log data. The logging system recorded packet sizes, parts of message contents, message direction, and violation categories, providing a clear record of how the proxy responded to different traffic types.

Results and Analysis

Analysis of the logged data demonstrated that the proxy successfully identified and blocked all oversized packets sent during experimentation. Twenty-four malicious or invalid packets were recorded, each exceeding the 50-byte threshold. The distribution of packet sizes revealed concentrations at 58 bytes and 201 bytes, corresponding to intentionally crafted test messages and stress-test payloads. All such packets were intercepted before reaching the backend server, confirming the reliability of the size-filtering mechanism.

The system showed strong performance in distinguishing legitimate traffic from harmful transmissions. All normal packets reached the server without interruption, and no false positives were observed during testing. Rate limiting prevented excessive packet submissions and ensured that bursts exceeding the allowed threshold triggered warnings or connection termination. Although content-based detection modules did not record violations in this dataset, their functionality remains integral to the broader defense strategy and is capable of responding to malicious payloads matching code-like or HTTP-like patterns.

Warning escalation operated as intended. Clients received clear notifications after each violation, and connections were terminated after surpassing the designated warning limit. The logging timeframe spanned approximately thirteen minutes, during which violations occurred at an average rate of nearly two per minute, with a peak cluster occurring within the first minute of testing. Representative log entries captured precise packet lengths, binary payload excerpts, and timestamps, providing valuable insight into the system’s behavior under varying traffic conditions.

Limitations

The development of this reverse proxy system was affected by several practical limitations. Time was a major factor, as I had other assignments and projects for different classes, which reduced the amount of attention I could give this work. Because of this, I was not able to explore more advanced features or test the system as thoroughly as I wanted.

My limited background in building proxies and working with network traffic also influenced the final design. I am still new to the concepts behind packet handling, filtering methods, and connection management, so a lot of the work involved learning the basics before I could begin implementing the system. This meant that some more complex features were out of reach for this version of the project.

In addition, my experience with low-level networking is still developing. While the system can inspect messages and enforce rules, it does not handle deeper packet structures or more advanced protocols. Working with raw network data often requires specialized knowledge that I am still gaining.

Finally, finding resources that explained traffic inspection and proxy behavior at a beginner level was difficult. Many articles and examples assume experience, which made it challenging to understand certain topics quickly. Because of these combined limitations, I focused on implementing core features rather than more advanced or experimental ideas.

Future Work

There are several ways this system could be improved in future versions. One major enhancement would be to expand the filtering engine so it can better understand different network protocols and detect more types of malicious traffic. With more knowledge and time, the proxy could include smarter detection methods instead of relying mainly on keywords and simple checks.

Adding support for encrypted traffic is another important next step. Most modern systems use HTTPS or other encrypted protocols, so allowing the proxy to handle secure connections would make it more realistic and useful. This would involve learning how to manage certificates and inspect data while still keeping the connection secure.

The system could also be improved by making it faster and more scalable. Instead of creating a new thread for every connection, switching to an asynchronous or event-based approach would help the proxy handle many clients at once. Adding configuration files or adjustable security rules would also make the system easier to update without changing the code.

Finally, stronger testing tools would help measure how well the proxy performs under real-world conditions. Automated traffic generators, larger test sets, and more diverse attack patterns would give a clearer picture of how the system behaves. As my experience grows, these improvements would help turn this basic prototype into a more reliable and feature-rich security tool.