HTTP2 DDoS SEED Lab

Roei Ben Zion, Ohad Libai

Tel Aviv University

October 10, 2025

https://github.com/OhadLibai/DDoS-SEED-Lab

Overview

The repository's lab of the project contains a set of hands-on labs for studying and experimenting with DDoS attacks that target modern web stacks by exploiting vulnerabilities and backdoors arise from inherent flaw design or implementation of HTTP/2 and legacy HTTP/1.1 protocols. The lab focuses on the the comparison of two deployments: local infrastructure versus cloud infrastructure. It compares and studies the success of the network attacks in each setup in terms of performance delta and collect some metric in regard. The technical, step-by-step deployment and configuration instructions for each lab are in that lab's own `README.md` — please follow those files for environment setup, scripts and exact CLI flags.

Learning objectives

- Understand and fill partially implemented protocol-level and application-level DDoS techniques (HTTP/2 multiplexing, flow control abuse, Slowloris).
- Launch controlled experiments to measure attack impact on response time, connections, CPU, memory and other resource metrics.
- Compare attack behaviour and mitigation effects between local deployments and cloud (GCP) deployments.
- Produce a short technical report that analyzes monitoring results, draws conclusions and suggests mitigations.

Lab overview

Each lab is packaged as its own folder. Read the `README.md` inside each lab before running anything; those per-lab READMEs contain the full technical setup, script usage and parameters.

HTTP/2 Application Flood (http2-app-flood)

Focus: Abuses of HTTP/2 multiplexing and application-level floods that cause worker exhaustion (CPU, request handling). Scenarios include single-worker vs multi-worker server setups and several synthetic CPU-heavy workloads that emulate real application behavior (e.g., CAPTCHA/crypto/gaming endpoints).
Key concepts: stream multiplexing, request concurrency, application CPU exhaustion.

Abstract

HTTP/2's stream multiplexing lets many logical requests (streams) share a single TCP connection. An application-layer flood exploits this by opening a set of TCP connections and creating very large numbers of concurrent streams per connection; each stream issues a legitimate-looking request that triggers CPU- or I/O-intensive application work (for example: CAPTCHA verification, cryptographic hashing, image processing or synthetic "game logic" tasks). Because stream creation is inexpensive compared with establishing new TCP/TLS sessions, an attacker can concentrate load on server workers and request handlers while keeping network bandwidth and observable connection counts deceptively low.

From the server internals viewpoint this attack primarily manifests as high CPU utilization across worker processes or threads, long request queuing times and an increase in application-level latencies. Instrumentation to collect includes per-request latency percentiles, per-worker CPU/memory (e.g., `docker stats` or process-level metrics), application logs showing request start/finish timestamps, and stream-level server logs (RST_STREAM or stream error counts) if available. Experiment controls that are useful for a rigorous comparison include fixed connection counts (e.g., 8, 32, 128), varying streams-per-connection (8–1024).

The advanced (so-called "cloud") version of this attack enhances the attack mainly by the following additions:

1. Spawning multiple independent processes, each running its own set of TCP connections and streams, dramatically multiplying the total attack capacity beyond what a single process can achieve.
2. Adding an *AdaptiveStreamManager* class implements intelligent, self-adjusting attack behavior that learns from server responses in real-time. It dynamically switches between "burst mode" (aggressive, high-volume attacks when the server appears healthy), "stealth mode" (throttled requests to evade detection when error rates increase), and normal operation, making the attack harder to detect.

HTTP/2 Flow-Control Attacks (flow-control)

Focus: Attacking HTTP/2 flow control (e.g., zero-window, slow/incremental window updates, adaptive slow attacks) to keep server threads or workers blocked and exhaust available connection state.
Key concepts: window update manipulation, connection state exhaustion, protocol-level resource management.

Abstract

HTTP/2 uses flow-control windows (per-stream and per-connection) managed by $WINDOW\_UPDATE$ frames to regulate the sender's data transmission. Flow-control attacks exploit this mechanism by holding streams open while preventing progress or by performing very slow, incremental window updates that keep server-side stream state allocated with minimal byte transfer. Primary lab variants are:
Zero-Window: The attack manipulates HTTP/2's INITIAL_WINDOW_SIZE setting by setting it to 0, which tells the server "I have no buffer space to receive data." This is a legitimate protocol feature designed to prevent overwhelming slow clients, but when weaponized, it forces the server to hold responses indefinitely while waiting for window updates that the attacker never sends.

Slow-Incremental: This variant employs a "slow drip" approach instead of complete blocking—it sends tiny WINDOW_UPDATE frames (1-10 bytes at a time) that technically allow data transmission but at an artificially throttled rate far below normal network speeds. The server must maintain active worker threads and memory buffers while transmitting large responses at a painfully slow pace, with deliberate delays (0.1-0.5 seconds) between each minimal window increment. This creates a more subtle form of resource exhaustion that mimics legitimate slow clients (like mobile devices on poor connections), making it harder to detect and block while still forcing servers to hold resources for extended periods.

Adaptive-Slow: This advanced variant adds machine learning-like adaptive behavior that continuously monitors server responses and automatically adjusts attack parameters in real-time to optimize resource exhaustion while evading detection. The AdaptiveRateController tracks timeout ratios and response times, using a PID-controller-inspired algorithm to dynamically tune window increment sizes (via multipliers) and delay ranges.

Mitigations to evaluate include enforcing minimum-progress or bandwidth-per-stream policies (close streams that do not make measurable progress within a timeout), limiting total state per-client (streams per IP/connection), and protocol-aware heuristics that deprioritize or close pathological $WINDOW\_UPDATE$ patterns. Practical deployments often combine conservative defaults (shorter idle timeouts, per-client stream caps) with monitoring that alerts on many long-lived, low-throughput streams.

Slowloris and Legacy Slow-Header Attacks (bonus-slowloris)

Focus: Legacy partial-header/slow header attacks and modern variants that target servers with HTTP/1.1 fallback or compatibility layers. Includes basic and advanced variants and cloud-targeted experiments.
Key concepts: slow header attacks, resource starvation via half-open requests, protocol fallback vulnerabilities.

Abstract

Slowloris-style attacks target HTTP/1.1 request parsing behavior by opening many connections and sending headers only incrementally so that the server keeps the socket and request parsing state reserved without ever completing the request. In modern stacks this class of attack remains relevant because many servers and compatibility layers still accept HTTP/1.1 traffic (or maintain fallback code paths). This advanced Slowloris variant adds intelligent self-adaptation and resilience features that make it significantly more effective and harder to mitigate. It uses a multi-threaded architecture where each socket runs in its own dedicated thread managed by a central controller, allowing automatic detection and replacement of dropped connections to maintain constant pressure on the target. The key innovation is dynamic pacing logic that monitors connection stability in real-time: when more than 5% of connections drop (indicating aggressive server defenses like timeout enforcement), it automatically decreases the keep-alive interval to send headers more frequently and prevent timeouts; conversely, when connections remain stable, it gradually increases intervals to operate more stealthily and reduce detectability. The attack also sends larger fake headers (configurable size, default 100 bytes) to bypass minimum byte-rate defenses that some servers use to detect slow clients, and randomizes sleep intervals within a range to avoid predictable timing patterns that signature-based detection systems could identify. The cloud's version novelty focuses on evasion through legitimacy mimicry—rather than just maintaining connections adaptively, it actively disguises attack traffic as normal browser behavior through User-Agent rotation, realistic request paths, standard HTTP headers, and steganographic keep-alive headers that resemble modern web application tracing systems, making it significantly harder for security systems to distinguish malicious connections from legitimate slow clients or mobile users.

Key indicators during an attack are very large numbers of established TCP connections with minimal bytes transferred, server accept-queue growth or inability to accept new connections, and many sockets stuck in early request-parsing states visible via `ss` or `netstat`. For measurement gather snapshots of socket states over time, accept-queue metrics, server log excerpts showing many half-complete requests, and response failures for new legitimate clients. Mitigation experiments should include aggressive header-read timeouts, per-IP connection limits, SYN-proxy or TCP handoff at an edge proxy (terminating suspicious half-open flows at the edge), and deployment of an intermediary reverse proxy/WAF that enforces complete-request validation before forwarding to the origin. These defenses illustrate how edge termination and conservative socket-timeout policies shift burden away from the origin server and blunt Slowloris-style impacts.

What you (students) must do — Tasks & Deliverables

First, you need to complete the implementation of each attack. To find the lines you are expected to fill, search for *# FILL CODE HERE* comment on every attcack .py partial implementation.

Once you are done filling the missing lines, run the monitoring tests (Live Monitoring and Testing) for each assigned lab:

1. Deploy the server locally (per lab README) and run the corresponding attack locally against it.
2. Deploy the server to GCP (per lab README) and run the attack from your local machine (or as the lab README specifies).
3. For each run collect the monitoring data described below.
4. Write a concise lab report (per lab or combined, depending on instructor).

The report should include

- Objective & test configuration (short — you can reference the lab README for full setup).
- What attack variant(s) you ran (names and key parameters).
- Monitoring data captured (see "Monitoring & metrics" below).
- Direct comparison between local and GCP runs (latency, throughput, CPU/memory, connection counts, observed failure modes).
- Short conclusions: which deployment was more/less impacted and why; any mitigation ideas or observations.

Deliverables

- Raw monitoring logs (curl outputs, `docker stats`, `ss/netstat` snapshots, any perf/Prometheus logs you collected).
- A PDF or Markdown report (2–5 pages) containing the analysis and comparison.
- Any command snippets or scripts you used to collect data (optional, useful for reproducibility).

Monitoring & metrics — suggested commands and checks

You should adapt and expand these according to the lab README. The commands below are the two basic (mandatory) checks we will use for the cloud experiments; there's an option to add short set of extra monitors that you may enable for deeper analysis.

Core checks (use these for every cloud run)

```
curl --http2-prior-knowledge -w "Time: %{time_total}s\n" http://<IP>:8080
```

Listing 1: HTTP/2 response time (prior-knowledge, total time)

```
gcloud compute ssh <VM_NAME> --zone=us-central1-a --command="sudo docker stats --no-
    stream"
```

Listing 2: Container/host resource usage (gcloud -> docker stats)

```
    docker exec <VICTIM_CONTAINER> sh -c "netstat -an | grep :8080 | grep ESTABLISHED
    | wc -l"
```

Listing 3: Check number of connections established to victim

You can always add extra monitoring from the lab repo or any other appropriate monitoring method (in this case, provide a short explanation for it).

How to compare GCP vs Local (practical checklist for the report)

1. **Baseline:** measure response time and resource usage before running the attack in each environment.
2. **Peak/Under Attack:** measure same metrics during sustained attack (same attack parameters where possible).
3. **Network differences:** note differences due to network latency and bandwidth (local loopback vs internet path).
4. **Resource differences:** VM size, CPU core counts, container worker counts, and any autoscaling or cloud NIC limits.
5. **Failure modes:** dropped connections, increased queueing, worker crashes, timeouts, or graceful degradation.
6. **Cost / practical considerations:** cloud egress or VM time (if applicable) and safety/ethics — do not exceed lab-approved usage.

Safety & legal notes

- ONLY run these experiments in the provided lab environment and with the servers you control.
- Running DDoS attacks against third-party systems or without explicit authorization is illegal and unethical.
- Use smallest resource sizes possible for cloud tests to limit costs and risk. Destroy any cloud VMs when you finish (lab READMEs include teardown scripts).

Cleanup (high level)

- Stop local attacks and servers using the lab scripts (see per-lab `deploy-*.sh` or Docker Compose commands).
- For cloud labs: run the repo's GCP teardown script or follow the lab README instructions to destruct VMs and release addresses.

Where the technical details are

Important: all technical setup steps, environment variables, exact script flags and troubleshooting tips are contained in each lab folder's `README.md`. This top-level file intentionally omits the detailed setup section — open the folder for the lab you are running and follow its `README`.