

WAF for Small Ecosystems

- [Concepts](#)
- [Why This Design](#)
- [Architecture](#)
- [Standing Up the Edge](#)
- [Adding Inspection](#)
- [Operating It](#)
- [Lessons Learned](#)

Concepts

The single front door to a self-hosted environment: what it inspects, what it terminates, and what it refuses to pass.

This book is a working reference for putting a web application firewall (WAF) in front of every public-facing service in a self-hosted environment. It opens with the case for a single inspecting edge, then walks through standing up the reverse proxy, adding application-layer inspection with open-appsec, authenticating inward to the backends, and operating the whole thing day to day. The specific stack is nginx with open-appsec, but the pattern, one hardened front door that every request must cross, applies regardless of the tools.

Why a single inspecting edge

A reverse proxy solves routing. It takes a request for `app.example.org` and forwards it to the right backend. That alone is worth doing, because it means backends never face the internet directly. But routing is not inspection. A plain proxy forwards a malicious request just as faithfully as a legitimate one. It looks at where a request is going, not at what the request contains.

A WAF adds the missing half. It sits at the same place the proxy does, the one point every inbound request must cross, and it examines the content of each request against the shape of known attacks; injection attempts, cross-site scripting, path traversal, malformed methods, etc. The whole catalog of things that target the application layer rather than the network. Requests that match potential malicious activity patterns are refused before they ever reach a backend. The edge stops being a pass-through and becomes a filter.

Three distinct jobs happen at this one host, and it is worth keeping them separate in your head because the rest of the book configures them one at a time:

- **Reverse proxy.** Route each hostname to its backend. Nothing public-facing exists anywhere else.
- **TLS termination.** Public certificates live here and here only. This is where encryption to the browser begins and ends; what happens inward is separate.
- **Application-layer inspection.** Examine every request and refuse the malicious ones. This is the true WAF functionality.

Why this matters more at small scale

One person may struggle to harden every application individually. Each app has its own request handling, its own vulnerabilities, its own patch cadence, and keeping all of them individually defended against web attacks is not realistic for a solo operator. A single inspecting edge changes the equation. Harden one front door well, and every service behind it inherits that protection. Not that you shouldn't harden your applications and services, and the hosts running them. A WAF is another security layer that aligns with the Defense-in-Depth model.

The WAF enables consistent enforcement across all applications and services; one place to update rules, one place to review what is being blocked, one place that sees every request and can make decisions about it.

The honest limits

A WAF is only one layer in the Defense-in-Depth model, and may represent a false sense of security if it is your only means of protection.

- **It reduces exposure; it does not eliminate it.** A WAF blocks known attack patterns and anomalous requests. It is not a guarantee that nothing malicious gets through, and a novel or carefully crafted attack can evade it. It buys you a great deal, but it is not a reason to stop patching or to relax authentication.
- **It is not a substitute for the other layers.** Defense-in-Depth means the WAF sits alongside host, application, and data layer protections. One still needs other technical, operational, and administrative controls to further secure the environment.
- **A single edge is a single point of failure.** In my environment there is only one WAF host, and if it is down, every public service is unreachable. For a small ecosystem serving a small number of people, that is an acceptable availability tradeoff.. In any larger or genuinely critical environment it would not be. A production deployment would run the WAF as a high-availability pair behind a load balancer, so that one host failing does not take the whole environment offline, and so that the edge itself can be patched and updated without downtime. The single-host model in this book is right-sized for a small ecosystem and explicitly wrong for a large one.

None of this undermines the case for a WAF; it reinforces it. The WAF a valuable single control at the edge, because it is the one point everything crosses, and that same property is why it must not be the only control and why, at scale, it must not be a single host.

What the following pages cover

- **Why this design:** a single edge host running open-appsec on nginx, and the tradeoffs of that choice.
- **How it fits the architecture:** where the edge sits, what terminates here, and where internal trust takes over.
- **Standing up the edge:** the reverse proxy, public TLS, and the per-site pattern every new service follows.
- **Adding inspection:** installing open-appsec and the learn, then prevent lifecycle that keeps it from blocking legitimate traffic.
- **The internal handoff:** authenticating from the WAF to each backend, where this book intersects with the mTLS book.
- **Operating it:** logs, alerts, and telling a real attack from a false positive.
- **What I'd tell someone starting out:** the lessons, including the ones about running a single edge.

Why This Design

A single edge host that terminates public TLS, inspects every request, and forwards only what passes inspection. This page is the reasoning behind that shape and the tool choice

Why terminate everything at one host

The alternative to a single edge is exposing services directly, each with its own public certificate and its own internet-facing surface. That multiplies the attack surface by the number of services and scatters the security configuration across all of them. Every app becomes its own front door, and every front door is one you have to lock individually.

Funnelling all public traffic through one host inverts that. Backends get no public IP addresses at all; they are reachable only from the edge. The internet sees exactly one machine. That machine is the only thing that has to be hardened against direct attack, the only place public certificates live, and the only place request inspection has to run. Concentrating the edge concentrates the work of defending it, which for a solo operator is the difference between a defensible position and an unmanageable one.

Why open-appsec on nginx

The edge runs nginx as the reverse proxy, with [open-appsec](#) attached as the inspection layer. The reasons:

- **Open-source and self-hostable.** Consistent with the rest of the environment, it runs on infrastructure under my control with no per-request cost and no traffic routed through a third-party scrubbing service. For a privacy-focused setup, sending every request through someone else's cloud WAF would undercut the point.
- **Machine-learning-based, not only signatures.** Traditional WAFs match requests against signatures of known attacks. Open-appsec adds a model that scores requests by how anomalous they look, which catches variations that no signature covers and reduces the endless rule-tuning that signature-only WAFs demand.
- **Runs as an nginx module.** It attaches to the nginx already doing the proxying rather than being a separate appliance in the path. One host, one request flow, one place to

apply controls.

- **Right-sized.** It provides real application-layer protection, the OWASP attack categories, anti-bot, schema validation, without the cost and operational weight of a commercial WAF appliance built for enterprise traffic volumes.

The tradeoffs

A single edge is a single point of failure and a single chokepoint. Every request in the environment passes through this one host, which means its performance is everyone's performance, and its uptime is everyone's uptime. In a small ecosystem this is a reasonable trade. The simplicity of one well-understood edge outweighs the availability cost, and the traffic volume is nowhere near enough to strain a single host. It must be said, though, that this does not scale. A larger or more critical environment needs the WAF running as a redundant, load-balanced tier so that no single host failing takes the environment down, and so the edge can be updated without an outage window. The design in this book is deliberately right-sized for a small ecosystem, and deliberately unsuitable for anything larger.

Open-appsec's learning model needs time and tuning. The machine-learning approach is a strength, but it is not zero-configuration. The model has to observe real traffic before it can reliably tell normal from malicious, and requires some manual review and policy tuning before enabling blocking capabilities. This is a real operational cost, covered in detail on the inspection page: the learn phase is mandatory, not optional.

Pinning nginx creates a maintenance burden. Open-appsec attaches to specific nginx versions, which means nginx cannot be allowed to upgrade freely. It has to be held at a compatible version and updated deliberately in-step with open-appsec's compatibility. That is a small, periodic chore traded for the inspection capability.

Adapt this for...

Any environment consolidating public ingress behind one inspecting edge. The tool specifics are nginx and open-appsec, but the decisions generalize: put all public services behind one hardened proxy, terminate TLS there, inspect there, and give backends no other way in. The one decision that must change with scale is the number of edge hosts. One can be right for a small ecosystem, and a high-availability tier is required for anything larger.

Architecture

Where the edge sits and what it hands off to. The WAF is one boundary in a layered defense, and understanding what it does and does not own makes the configuration pages make sense.

Where the WAF sits

The WAF host lives at the edge of the private network. It is conceptually the boundary between the internet and everything internal, but it is not itself outside; it sits inside the private cloud, and a cloud-level firewall in front of it exposes a single port to the world.

The request path for user traffic is a sequence of narrowing gates:

1. **Cloud perimeter firewall** allows exactly one inbound path from the internet, HTTPS to the WAF host. Nothing else public is reachable.
2. **The WAF host** terminates public TLS, inspects the request, and, if it passes, forwards it to the backend.
3. **Service firewall** allows the backends to be reached only from the WAF, on the internal side. A backend will not accept a connection from anywhere else.
4. **The backend** serves the request, having never been exposed to the internet at any point.

Every public request runs that gauntlet. There is no path from the internet to a backend that does not pass through the WAF, which is the purpose of the whole design.

What terminates here, and what begins

Two different kinds of TLS meet at the WAF, and keeping them distinct is essential:

- **Public TLS terminates at the WAF.** The certificate a browser sees is a public certificate (from a public CA such as Let's Encrypt) for the site's public name, and it lives on the WAF. Encryption between the user and the environment ends here.
- **Internal mutual TLS (mTLS) begins at the WAF.** The connection from the WAF inward to a backend is a separate, mutually-authenticated TLS connection using private certificates from the internal certificate authority. The WAF proves its identity to the

backend with a client certificate, and the backend proves its identity to the WAF with a server certificate.

So the WAF is the seam between two trust domains, the public web on one side and the internally-authenticated network on the other. It is the one component that speaks both. The details of the internal mTLS side, the private CA, the certificates, the enforcement, are the subject of the [mTLS with a Private CA](#)

A note on the single edge

Because the architecture funnels all user traffic through one WAF host, that host's availability is the availability of every public service. As stated in the concepts and design pages, this is an accepted tradeoff for a small ecosystem and would be replaced by a high-availability tier in any larger environment. Architecturally, the thing to understand is that the WAF is not just a security boundary but a throughput and availability chokepoint, and every property of the environment that depends on "the edge is up" depends on this single host in this design.

Standing Up the Edge

The reverse proxy and public TLS, before any inspection is added. This is the baseline every public site sits on. Nginx routing a hostname to a backend over public TLS, with a consistent per-site configuration that new services slot into.

Install nginx from the official repository

Open-appsec attaches to specific nginx versions, so nginx has to come from nginx's own repository (not the distribution's package) and then be held at a compatible version. Review [Open-appsec nginx attachment documentation](#) to find the list of compatible nginx versions. Then add the official repo, install that version, and pin it:

```
curl -fsSL https://nginx.org/keys/nginx_signing.key \  
| sudo gpg --dearmor -o /usr/share/keyrings/nginx-archive-keyring.gpg  
  
echo "deb [signed-by=/usr/share/keyrings/nginx-archive-keyring.gpg]  
http://nginx.org/packages/ubuntu noble nginx" \  
| sudo tee /etc/apt/sources.list.d/nginx.org.list  
  
sudo tee /etc/apt/preferences.d/99nginx <<'EOF'  
Package: nginx*  
Pin: origin nginx.org  
Pin-Priority: 900  
EOF  
  
sudo apt update  
sudo apt install nginx<version> certbot python3-certbot-nginx
```

Then hold nginx so an unattended upgrade cannot move it to a version open-appsec does not yet support:

```
sudo apt-mark hold nginx
```

That hold is important because an automatic nginx upgrade can break the open-appsec attachment. Nginx must be updated deliberately, in step with open-appsec's compatibility, not automatically. This is a common way the setup breaks weeks later.

Some versions of nginx handle activating websites differently. Personally, I follow the `sites-available` -> `sites-enabled` method, which requires a change in `nginx.conf`:

```
include /etc/nginx/sites-enabled/*;
```

The per-site pattern

Every public service follows the same server-block shape. Understanding it once means every new site is the same move. Here is the pattern, annotated, with the public name and the internal backend name as specific things that must change per site. Additional settings may need to be modified or added depending on the service hosted.

```
server {
    listen 443 ssl;
    server_name app.example.org;                # the public name

    # Public TLS – the certificate a browser sees (managed by Certbot)
    ssl_certificate      /etc/letsencrypt/live/app.example.org/fullchain.pem;
    ssl_certificate_key  /etc/letsencrypt/live/app.example.org/privkey.pem;
    include /etc/letsencrypt/options-ssl-nginx.conf;

    # Security headers
    add_header X-Frame-Options DENY;            # disallow framing (clickjacking)
    add_header X-Content-Type-Options nosniff;  # no MIME-type sniffing

    client_max_body_size 512m;                  # raise/lower per site; comment out if no
    uploads

    location / {
        limit_req zone=general burst=20 nodelay; # rate limiting (see note below)

        proxy_pass https://app.int.example;      # internal backend, over mTLS

        proxy_http_version 1.1;
        proxy_set_header Connection "";          # comment out if the app needs WebSockets
    }
}
```

```

proxy_set_header Host          $host;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto $scheme;

# --- internal mutual TLS to the backend ---
# Verify the backend's server certificate against the internal CA
proxy_ssl_server_name on;
proxy_ssl_name app.int.example;
proxy_ssl_trusted_certificate /etc/ssl/certs/internal-ca.crt;
proxy_ssl_verify on;
proxy_ssl_verify_depth 2;

# Present the WAF's client certificate to the backend
proxy_ssl_certificate      /etc/ssl/certs/waf-to-app.crt;
proxy_ssl_certificate_key /etc/ssl/private/waf-to-app.key;
}
}

```

The important pieces:

- **server_name** is the public name; **proxy_pass** is the internal name. The WAF translates a public request into an internal, mutually-authenticated one. The two names are different domains of trust meeting in one block.
- **The security headers** are set at the edge so every site gets them without each backend having to. **X-Frame-Options DENY** prevents the site being framed; **X-Content-Type-Options nosniff** stops browsers from second-guessing content types.
- **client_max_body_size** is tuned per site: large for a file-sync service, small or default for a forum. Set it to what the app legitimately needs and no larger, since it is also a limit on how much a malicious request can send.
- **limit_req** applies rate limiting per site, which is a basic protection against floods and brute-force attempts. Different services tolerate different rates; a sync-heavy or automation service needs a higher burst than a wiki.
- **The proxy_ssl_* block** is the internal mTLS handoff, covered in the mTLS book. It is what makes the connection inward authenticated rather than plaintext.

Public certificates

Public TLS certificates are obtained and renewed with Certbot on the WAF. Because this host is the only place public certificates live, it is the only place they are renewed:

```
sudo certbot --nginx -d app.example.org
```

To run renewal on a schedule, run `sudo crontab -e` and add the following line. This runs certbot on the 5th day of each month. Certbot will renew a certificate once it is within 30 days of expiration.

```
1 0 5 * * certbot renew
```

Certbot's nginx integration reloads nginx after a successful renewal, so unlike the internal certificates (which need an explicit reload, covered in the mTLS book), the public certificates take care of themselves here.

Adding Inspection

Attaching application-layer inspection to the reverse proxy. Standing up the edge gave you a proxy that routes and terminates TLS. This page adds the part that examines requests and refuses those identified as malicious, and sets up the principles that makes blocking safe later.

Installing open-appsec

Open-appsec attaches to nginx as a module plus a set of agent services. With nginx already installed from its official repository and pinned, install open-appsec:

```
cd /tmp
wget https://downloads.openappsec.io/open-appsec-install
chmod +x open-appsec-install
sudo ./open-appsec-install --auto --no-email
sudo ./open-appsec-install --download
```

The `--download` step prints the exact commands to install the nginx attachment module for your specific nginx version. Run the printed commands rather than assuming paths, because they are version-specific. They copy the attachment libraries into place and the nginx module itself:

```
# example – use the paths from your own --download output
sudo cp /tmp/open-appsec/nginx_module_<ver>/libosrc_shmem_ipc.so /usr/lib/
sudo cp /tmp/open-appsec/nginx_module_<ver>/libosrc_compression_utils.so /usr/lib/
sudo cp /tmp/open-appsec/nginx_module_<ver>/libosrc_nginx_attachment_util.so /usr/lib/
sudo cp /tmp/open-appsec/nginx_module_<ver>/ngx_cp_attachment_module.so
/usr/lib/nginx/modules/
```

Load the module by adding this at the very top of `nginx.conf`:

```
load_module /usr/lib/nginx/modules/ngx_cp_attachment_module.so;
```

Then install the agent services that do the actual inspection work behind the nginx module:

```
sudo /tmp/open-appsec/openappsec/install-cp-nano-agent.sh --install --hybrid_mode --server
'NGINX Server'
sudo /tmp/open-appsec/openappsec/install-cp-nano-service-http-transaction-handler.sh --install
```

```
sudo /tmp/open-appsec/openappsec/install-cp-nano-attachment-registration-manager.sh --install
sudo systemctl restart nginx
```

`--hybrid_mode` runs open-appsec in standalone mode, managed by the local policy file rather than a cloud console. That keeps management local and self-hosted, consistent with the rest of the environment.

The learn-then-prevent lifecycle

This is the single most important operational fact about running open-appsec: **the machine-learning model must observe real traffic before it can be trusted to block**. Open-appsec scores requests by how anomalous they are relative to what it has learned normal traffic looks like. Before it has learned, it does not have a reliable baseline, and turning on blocking prematurely means legitimate but unusual requests potentially get refused.

So the lifecycle is:

1. **Install and run in a learning posture.** Let it watch real traffic for a period, weeks, not hours, so it sees the normal range of what your applications actually do: large uploads, automation calls, the odd-looking-but-legitimate request patterns each app has.
2. **Review what it would have blocked.** During learning, the logs show what the model flags. This is where you find the false positives before they become outages.
3. **Switch to prevention, one protection at a time.** Once the baseline is solid, enable blocking. From then on, flagged requests are actually refused.

Skipping the learn phase, or switching to prevention too soon, is the path breaking legitimate traffic and ripping out open-appsec in frustration. The learning period is the cost of the model-based approach, and it is what makes the approach work. Budget time to do it right.

Open-appsec expresses this as **modes** on each protection, and the naming is worth understanding because it appears throughout the policy file:

- **detect-learn**: observe and learn, log what would be blocked, block nothing. This is where everything starts.
- **prevent-learn**: block, and keep learning to refine the model. You move a protection here only once you trust its baseline, which is covered on the *Operating It* page.

You move a protection from `detect-learn` to `prevent-learn` when you trust its baseline.

The policy file

The policy lives in `/etc/cp/conf/local_policy.yaml`. Below is the working policy as an **initial deployment**, with every protection in `detect-learn`. It has three parts: the default policy applied to traffic, the practice that defines what protections run and in what mode, and the log trigger.

```
policies:
  default:
    mode: detect-learn          # overall posture: observing and learning, blocking
    nothing
    practices: [appsec-best-practice]  # which practice (protection set) applies
    triggers: [appsec-log-trigger]     # how events are logged
    custom-response: 403-forbidden    # what a blocked client receives, once blocking is
    enabled
    source-identifiers: ""
    trusted-sources: ""
    exceptions:
      - name: allow-backup-sync
        condition:
          source-ip: "203.0.113.10"    # placeholder for a specific trusted source IP
          action: allow                # exempt this source from inspection
    specific-rules: []

practices:
  - name: appsec-best-practice
    openapi-schema-validation:
      configmap: []
      override-mode: detect-learn      # schema validation: learning only
    snort-signatures:
      configmap: []
      override-mode: detect-learn      # signature matching: learning only
    web-attacks:
      max-body-size-kb: 1000000
      max-header-size-bytes: 102400
      max-object-depth: 40
      max-url-size-bytes: 32768
      minimum-confidence: critical     # when promoted, only block at highest confidence
    (see note)
      override-mode: detect-learn      # web attacks: learning only for now
    protections:
      csrf-protection: detect-learn
      error-disclosure: detect-learn
```

```
non-valid-http-methods: true
open-redirect: detect-learn
anti-bot:
  injected-URIs: []
  validated-URIs: []
  override-mode: detect-learn          # anti-bot: learning only
```

A few settings are worth understanding now, even though nothing is enforcing yet, because they govern what happens once you promote protections to blocking:

- **minimum-confidence: critical** on web-attacks is a deliberately conservative choice for when that protection is eventually promoted. Open-appsec will only block a web-attack when its confidence is at the highest level, minimising false positives at the cost of letting lower-confidence suspicious requests through to be logged rather than blocked. For a small environment where a false block locks out a real person and there is no one else on call to fix it, erring toward fewer false positives is the right call. A higher-security posture with a team to tune continuously would lower this threshold and accept more false positives.
- **non-valid-http-methods: true** is a plain boolean rather than a mode: malformed or non-standard HTTP methods are rejected outright, since there is no legitimate reason for them.
- **custom-response: 403-forbidden** defines what a blocked client eventually receives. It has no effect while everything is in **detect-learn**, because nothing is being blocked yet, but it is in place for when protections are promoted.
- **The exceptions block** exempts a specific trusted source from inspection entirely. This is sometimes a real necessity, a backup or sync endpoint whose legitimate traffic trips the model, but every exception is a hole in the inspection. Keep each one as narrow as possible (a single source IP, not a range), document why it exists, and revisit it periodically. An exception whose reason no one remembers is a liability.

Applying policy changes

After editing the policy, apply it:

```
open-appsec-ctl --apply-policy
```

Operating It

A WAF is not a set-and-forget install. It is a control you have to be able to see working, tune when it is wrong, and trust when it fires. This page is about running it after it is standing.

Logging

A control you cannot observe is a control you cannot trust. Open-appsec's logging is driven by the log trigger configured in the policy. Events, requests inspected, requests blocked, what rule or model score triggered a block, are logged locally.

“ Logs that live only on the WAF are lost if the WAF is the thing that fails, and they cannot be correlated with what the backends saw. When centralised with the rest of your host, application, and service logs, they become part of one picture. How I manage log forwarding, centralizing, parsing, and alerting, will be covered in a future deep dive.

The two questions the logs exist to answer:

- **What is being blocked?** A blocked legitimate request is a person locked out. A blocked malicious request is the WAF doing its job and you want to confirm it is happening.
- **What is being flagged but not blocked?** The protections in learning mode are logging what they *would* block. That stream is how you decide whether a protection is ready to promote to blocking.

Telling a real attack from a false positive

This is a core operational skill, and it is a judgment call the logs inform rather than make for you. When a request is blocked, the question is whether it was a genuine attack or a legitimate request that looked like one.

- A **false positive** typically comes from a known source, correlates with a real user action, and involves an application's own legitimate but unusual behaviour, a large structured

upload, an API call with an odd-looking payload, a automation client's request shape. The fix is a narrow, documented exception or promoting the relevant protection's tuning, not turning inspection off.

- A **real attack** typically comes from an unfamiliar source, does not correlate with any legitimate user action, and matches attack shapes, injection strings, traversal sequences, probes for known-vulnerable paths. The right response is to confirm the block worked and, if the source is persistent, consider blocking it further upstream.

The reason the conservative `minimum-confidence: critical` setting from the inspection page matters here is that it biases toward fewer false positives, which keeps this triage manageable for a solo operator. The tradeoff is some lower-confidence suspicious traffic is logged rather than blocked. This is deliberate. A flood of false positives trains you to ignore the alerts, which is worse than a slightly more permissive block threshold.

Alerting

Raw logs are necessary but not sufficient. You also need to be told when something warrants attention rather than having to go looking. In my environment, an automation workflow reviews the forwarded logs and raises an alert on patterns worth a human's attention, for example, a spike in blocked requests or repeated hits from one source. The same log-and-alert approach covers the identity provider's authentication events, so the WAF's alerting is one instance of a general pattern: forward the logs, let an automation watch them, and surface only what matters.

The principle, consistent across this environment: monitor the control working, not just its output. A WAF that has silently stopped inspecting, because an nginx upgrade broke the attachment, because a service died, looks exactly like a WAF that is inspecting and finding nothing. The way you tell them apart is by watching for the *absence* of the logs you expect, not only the presence of alarming ones. A sudden silence from a component that normally logs steadily is itself a signal.

Testing that inspection actually works

Before trusting the WAF, and periodically after, confirm it is actually seeing and flagging malicious-looking requests. You do this by sending requests that mimic common attacks and checking that they show up in the logs (in `detect-learn`) or are refused (once promoted to `prevent-learn`). These are safe to run against your own environment.

Run them from an external host, not from an exempted source, or the exception will skip inspection and you will learn nothing.

SQL injection in a query parameter:

```
curl -k "https://app.example.org/?id=1' OR '1'='1"
curl -k "https://app.example.org/?id=1;DROP TABLE users--"
```

Cross-site scripting in a parameter:

```
curl -k "https://app.example.org/?q=<script>alert(1)</script>"
```

Path traversal, attempting to escape the web root:

```
curl -k "https://app.example.org/../../../../etc/passwd"
curl -k "https://app.example.org/?file=../../../../etc/passwd"
```

Command injection in a parameter:

```
curl -k "https://app.example.org/?host=127.0.0.1;cat%20/etc/passwd"
```

A non-standard HTTP method (this one is blocked outright by `non-valid-http-methods`, even early, since it is a boolean rather than a learned protection):

```
curl -k -X BADMETHOD "https://app.example.org/"
```

A known-scanner user agent, the kind of probe that hits every public host:

```
curl -k -A "sqlmap/1.0" "https://app.example.org/"
```

What to expect at each stage:

- In `detect-learn`: these requests are not blocked. The application responds normally (or with its own 404), but each should appear in the open-appsec logs as something the WAF *would* have blocked. If they do not appear in the logs at all, inspection is not working, the module did not load, the agent is not running, or the request never reached the WAF.
- In `prevent-learn`: the malicious requests should receive the `403-forbidden` response instead of reaching the application. Seeing the 403 is the confirmation that blocking is live.

Running this small battery after install (to confirm detection works) and again after promoting a protection (to confirm blocking works) turns "I think the WAF is protecting us" into "I watched it catch these." It is also a good periodic check, since a WAF that has silently stopped inspecting will let all of these through with no log entry.

Adding exceptions when legitimate traffic trips inspection

Reviewing the `detect-learn` logs will surface legitimate requests the model flags such as a backup client's sync traffic, an app's large structured upload, an automation tool's unusual request shape. Before promoting the relevant protection to blocking, these need an exception, or promotion will start legitimate traffic.

An exception goes in the `default` policy's `exceptions` block. Keep the condition as narrow as the situation allows, a specific source IP is far safer than a broad range:

```
exceptions:
  - name: allow-backup-sync
    condition:
      source-ip: "203.0.113.10"      # the specific trusted source
      url: "/sync"                  # scope appropriately, don't blanket exempt an IP
    action: allow
# exempt a specific endpoint rather than a whole source
- name: allow-large-upload-path
  condition:
    url: "/api/upload"
  action: allow
# exempt a parameter that legitimately carries markup
- name: allow-html-body-field
  condition:
    paramName: "post_body"
  action: skip
```

Apply it with `open-appsec-ctl --apply-policy`, then re-run the legitimate traffic and confirm it is no longer flagged. Every exception is a standing hole in the inspection, so each one should be as specific as possible, named clearly, documented as to why it exists, and revisited periodically to confirm it is still needed. An exception whose reason no one remembers is a liability sitting inside your security control.

Promoting protections to prevent-learn

Once a protection has a few weeks, or possibly several months depending on your ecosystems overall traffic, of detection behind it and you have added exceptions for the legitimate traffic it flags, promote it to blocking. Do this **one protection at a time**, so that if a promotion starts refusing legitimate requests, you know exactly which change caused it and can revert just that one. Flipping everything to `prevent-learn` at once and hoping is precisely the mistake the learning phase exists to prevent.

A suggested order, with reasoning:

1. `non-valid-http-methods` is effectively already enforcing (it is a boolean, not a learned protection) and has essentially no false-positive risk, since legitimate clients do not send malformed methods. It is the safe first thing to have blocking, and confirms your `custom-response` and the block path work end to end.
2. `web-attacks` next. It is the protection you most want enforcing, injection, XSS, traversal, the core of what a WAF is for, and with `minimum-confidence: critical` it is tuned conservatively to keep false-positive rate low. Promote the `web-attacks` `override-mode` to `prevent-learn`, and its sub-protections (`csrf-protection`, `error-disclosure`, `open-redirect`) as each proves reliable in the logs.
3. `snort-signatures` once you have reviewed what it flags. Signature matching can be noisier against real application traffic, so it benefits from a longer look before blocking, but it is worth enforcing once tuned.
4. `openapi-schema-validation` only where you actually have a defined API schema for the traffic. Without one it has little to enforce, and promoting it can reject legitimate requests that simply do not match an assumed shape. Promote deliberately, per the applications that warrant it.
5. `anti-bot` last, and most cautiously. Bot detection has the highest false-positive potential of the set, since legitimate automation, monitoring, and sync clients can look bot-like. Spend the most detection time here and promote it only once you are confident it will not lock out your own tooling.

After each promotion, re-run the relevant curl tests from the section above to confirm blocking is live, and watch the logs for a few days for legitimate traffic newly caught. The whole sequence will likely span several months from install to a fully-enforcing policy.

Lessons Learned

The lessons I've learned from running a single-edge WAF in a self-hosted environment.

- **Run learning mode longer than feels necessary.** The instinct is to turn on blocking as soon as it is installed, because an inspecting WAF that isn't blocking feels pointless. Resist it. The model needs weeks, if not months, of real traffic to learn what your applications legitimately do, and blocking before it has that baseline means block legitimate requests. The learning period is not a delay before the real work, it *is* the work that makes blocking safe.
- **A single edge means the edge's own hardening and uptime are now critical.** Concentrating all traffic through one host is what makes it defensible by one person, and it is also what makes that host a single point of failure. In a small ecosystem that is an acceptable trade, but it raises the stakes on that one host. For an ecosystem larger than a dozen applications and services with a few dozen users, a high-availability WAF tier stops being optional. Run two WAFs behind a load balancer.
- **Tune false positives before they train you to ignore alerts.** A WAF that cries wolf useless, because a stream of false positives teaches you to dismiss its alerts, with the risk that you miss a real one. Keeping the block threshold conservative, reviewing what gets blocked, and fixing false positives with narrow exceptions is what keeps the alerts meaningful enough to act on.
- **Get the forwarded-protocol header right, or chase phantom auth bugs.** The single most disproportionate source of pain at the edge I encountered is the `X-Forwarded-Proto` header. If an application backend doesn't know the original request was HTTPS, it builds `http://` redirects, browsers refuse them, and you get login loops that look like an application bug but are actually a one-line proxy header. Set it explicitly, and remember it exists the next time a login mysteriously loops.
- **Keep every inspection exception narrow and documented.** Sometimes a legitimate source genuinely trips the WAF and needs an exception. Every exception is a hole in the inspection, so make each one as specific as possible. A single source rather than a range, to a URL, and include any relevant conditions. Write down why they exist, comments in the yaml is appropriate, and revisit it periodically. An undocumented exception whose purpose no one remembers is a liability sitting in your security control.
- **Remember nginx is pinned.** Because open-appsec attaches to specific nginx versions, an automatic nginx upgrade can silently break inspection. Hold nginx, check open-appsec's compatibility before updating, and move the two together. A WAF that has stopped inspecting because nginx moved out from under it looks exactly like a WAF finding nothing, which is the most dangerous failure mode there is.