TLS Fingerprinting — The First Gate

March 28, 2026

> *This post is purely informational security research. All observations reference publicly observable browser behavior via standard DevTools. The techniques discussed are documented to help defenders understand the structural limitations of client-side bot defense.*

Before your Javascript loads, before your cookies are set, before a single pixel renders — the TLS handshake has already happened. And in that handshake, your client told the server exactly what it is.

TLS fingerprinting is the first line of defense in modern bot mitigation stacks. It operates below the application layer, analyzing the cryptographic parameters a client advertises during connection setup. The logic is relatively straightforward: real browsers produce distinctive TLS handshakes; automation tools don't. If the handshake doesn't look like a browser, reject the connection before wasting resources on JS challenges or behavioral analysis.

It's a clean, efficient gate. It's also structurally limited in ways that matter.

This is Part 1 of *Trusting the Client*, a series examining why client-side bot defense operates on fundamentally unstable ground. Each part dissects one defense layer through the same lens: how it works, how it breaks, and what defenders should do instead. This series is based on my own case study of a GoDaddy WHOIS endpoint protected by Akamai & Kasada, documented in the companion post Free the WHOIS: Reclaiming Public WHOIS Data (/series/).

How TLS Fingerprinting Works

Every TLS connection begins with a ClientHello message. It is as it sounds: the client offers up their hand in marriage to the server they wish to wed. This message contains a set of fields that describe the client's cryptographic capabilities:

- **TLS version**: The highest protocol version the client supports (e.g. TLS 1.2, 1.3)[1]
- **Cipher suites**: A list of encryption algorithms the client is willing to use
- **Extensions**: Optional features the client advertises: Server Name Indication (SNI), signature algorithms, ALPN (Application-Layer Protocol Negotiation), supported groups (the elliptic curves used in the key exchange), point formats (compression methods for those curves), and dozens more. JA3 extracts supported groups and point formats as separate fingerprint fields, but on the wire they're extensions like any other

These fields aren't secrets — they're broadcast in plaintext (or, in TLS 1.3, partially encrypted but still observable at the CDN/WAF termination point)[2]. And critically, different TLS implementations populate them differently.

Python's `requests` library (backed by OpenSSL) advertises differently than Chrome (backed by BoringSSL). Go's `crypto/tls` package will produce a different ClientHello than `curl`. Even different versions of the same browser produce measurably different handshakes.

This variation is the signal that TLS fingerprinting exploits.

JA3: The First Generation

JA3, developed by Salesforce researchers John **A**lthouse, Jeff **A**tkinson, and Josh **A**tkins in 2017, was the first widely adopted fingerprinting method. It constructs a fingerprint by concatenating five fields from the ClientHello:

```
JA3 = MD5(TLSVersion, Ciphers, Extensions, EllipticCurves,
EllipticCurvePointFormats)
```

Each field is a comma-separated list of numeric values joined by hyphens. The resulting string is MD5-hashed into a 32-character hex fingerprint.

The fingerprint is deterministic for a given TLS stack.

JA4: Addressing JA3's Limitations

JA3 has known weaknesses that attackers learned to exploit. Looking back at how JA3 is represented, you'll notice that the hash is completely opaque. Two fingerprints that differ by a single cipher suite produce completely unrelated hashes, making analysis difficult. In fact, the number of hashes that could exist is literally the ceiling of what MD5 as an algorithm produces: 2^128. Two fingerprints that are nearly identical on the wire produce completely unrelated hashes, so there is no way to cluster, compare, or partially match fingerprints without a lookup table mapping each hash back to its raw input.

JA3 also only captures the TLS layer and has no visibility into application-level signals.

JA4, developed by FoxIO in 2023, addresses these issues with a multi-part, human-readable fingerprint:

```
JA4 = Protocol_CipherInfo_ExtensionInfo_SignatureAlgorithms
```

Key improvements:

- **Sorted cipher suites and extensions**: Eliminates ordering randomization as an evasion vector

- **Human-readable structure**: The fingerprint is inspectable without a lookup table
- **Suite of fingerprints**: JA4 covers only TLS; JA4+ extends the methodology to server responses, certificates, HTTP headers, and TCP parameters
- **Truncated hashes per section**: Each component is independently comparable, enabling partial matching and clustering

Cloudflare's Bot Management uses JA4 fingerprints as part of its bot scoring pipeline. Fastly integrates TLS fingerprinting into its CDN-based bot management. Akamai Bot Manager performs TLS analysis at the edge before any content is served.

What TLS Fingerprinting Catches

The defense is effective against a specific and common threat class: **automation tools that don't use real browser TLS stacks.**

For the majority of scraping and automation activity — which as of 2024 has surpassed 50% of total web traffic [Imperva, 2025] — TLS fingerprinting is an efficient first filter. It operates at the connection level, costs minimal compute, and eliminates lazy clients before they consume any application resources.

How it Breaks: Real Browsers and Fingerprint Spoofing

TLS fingerprinting has two distinct failure modes, and they differ dramatically in their sophistication.

Using a Real Browser

The simplest case requires no sophistication at all. If the attacker uses an actual Chrome binary, the JA3/JA4 fingerprint is legitimate Chrome. It's not *mimicking* Chrome's TLS stack; it *is* Chrome's TLS stack.

In my case study, a Chrome extension with native messaging produced a TLS fingerprint indistinguishable from a normal Chrome browsing session. Akamai's TLS fingerprinting, the first of two checks in GoDaddy's defense, passed the connection without challenge. The fingerprint was correct because the software was correct.

This is not a bypass. It's a non-event. The defense is checking whether the client is a browser; the client *is* a browser. The check passes.

Spoofing Without a Browser

For attackers who need higher throughput than a real browser allows, several libraries enable TLS fingerprint spoofing:

curl-impersonate is a modified build of curl that reproduces Chrome and Firefox TLS handshakes by using the same TLS libraries with the same configuration parameters[3]. It also matches HTTP/2 SETTINGS frames and header ordering, addressing the composite fingerprint problem that trips up naive spoofing attempts.

utls is a Go library that provides programmatic control over the ClientHello message. Developers can construct a ClientHello that exactly matches any target browser's fingerprint, field-by-field. It's used extensively in censorship circumvention tools (hence "refraction-networking") where TLS fingerprinting is used by state firewalls to identify and block VPN traffic.

curl_cffi is a Python binding to curl-impersonate, bringing TLS fingerprinting to Python's scraping ecosystem with a `requests`-compatible API.

These tools defeat JA3/JA4 fingerprinting in isolation, but spoofing the TLS layer is only the first problem. The deeper challenge is **cross-layer consistency**: making every protocol layer tell the same story. The spoofing ecosystem ranges from transport-only libraries that leave headers to the developer, to higher-level wrappers that automate header generation, to anti-detect browsers that are effectively real browsers packaged for automation. But at each level, the attacker must keep every layer consistent or risk detection.

Why the Limitation is Structural

TLS fingerprinting identifies **software**, not **intent**.

The ClientHello is a product of the TLS library, its version, and its configuration. It tells you what software initiated the connection. It cannot tell you *why* the connection was initiated, *who* is controlling the software, or *what* they intend to do with the response.

This creates a logical boundary:

- If the attacker uses non-browser software -> TLS fingerprinting detects it (assuming the fingerprint isn't spoofed)
- If the attacker uses browser-grade TLS (real browser or library-level spoofing) -> TLS fingerprinting has no signal

The defense assumes that bot traffic originates from non-browser software. This assumption holds for the majority of automated traffic, but fails completely against an attacker who is willing to write a couple lines of automation scripting or use library-level TLS mimicry. And the trend is clear: as TLS

fingerprinting becomes more widespread, the attacker tooling for successfully spoofing it becomes more accessible and more polished.

This is Kerckhoff's principle applied to bot defense: TLS and browser behavior are public specifications, so the attacker knows exactly what "correct" looks like and can reproduce it, while the defense must accept every client that does. **The defense is at a disadvantage from the beginning. Spoofing a fingerprint is a one-time engineering cost, but maintaining detection is an ongoing burden that scales with every browser release and every new evasion tool.**

What Defenders Should Do Instead

TLS fingerprinting isn't useless — it's *insufficient as a standalone gate*. The remediation isn't to abandon it, but to position it correctly within a layered defense architecture.

Use TLS Fingerprints as One Signal, Not the Only Signal

Treat the fingerprint as an input to a bot score, not a binary pass/fail gate. A connection with Python's `requests` fingerprint can be blocked with high confidence. A connection with a Chrome fingerprint should be passed to the next layer, not trusted implicitly.

Invest in Cross-Layer Correlation

The most valuable signal isn't the TLS fingerprint alone, it's the *consistency* between TLS, transport framing (HTTP/2, HTTP/3), and application-layer signals. Flag sessions where:

- TLS says Chrome but HTTP/2 SETTINGS or QUIC transport params don't match Chrome's known configuration
- TLS says Chrome 120 but the `sec-ch-ua` header advertises Chrome 118
- TLS says Firefox but the cipher suite order is sorted (Firefox doesn't sort; JA4's sorting removes this signal, but the raw ClientHello preserves it)
- HTTP/2 pseudo-header ordering doesn't match any known browser
- TCP SYN params suggest an OS inconsistent with the claimed browser

These are all examples of cross-layer validations that significantly raise the bar for spoofing libraries, which must now match the target browser across multiple layers simultaneously.

Maintain Fingerprint Databases and Update Cadence

Browser TLS fingerprints change with every major release. Chrome ships a new stable version roughly once a month, and each release can modify the cipher suite list, extension set, or supported groups. A fingerprint database that's three months stale will produce false positives on legitimate traffic and false negatives on spoofed traffic.

Don’t Treat TLS Fingerprinting as a Security Boundary

TLS fingerprinting is a **cost filter**. It cheaply eliminates lazy automation attempts. It is not a **security boundary**. It does not prevent determined access. Design your API auth, rate limiting, and data protection as if TLS fingerprinting doesn’t exist. If your security model depends on the client not being a browser, you’ve done something wrong.

What’s Next

TLS fingerprinting operates before JavaScript executes; it’s the transport-layer gate. But once that connection is established and the page loads, a different defense layer takes over: **environment interrogation**. Defense SDKs like Kasada’s `p.js` probe the browser’s JS runtime across hundreds of data points: WebGL renderers, canvas hashes, audio fingerprints, navigator properties. You get the picture. This probing builds a composite fingerprint that *should* be unforgeable.

In Part 2, we’ll examine why it isn’t.

References

Althouse, J., Atkinson, J., & Atkins, J. (2017). JA3 — A Method for Profiling SSL/TLS Clients. Salesforce Engineering. https://github.com/salesforce/ja3 (https://github.com/salesforce/ja3)

FoxIO. (2023). JA4+ Network Fingerprinting. https://github.com/FoxIO-LLC/ja4 (https://github.com/FoxIO-LLC/ja4)

Cloudflare. (2024). JA3/JA4 Fingerprint. Cloudflare Bot Solutions Documentation. https://developers.cloudflare.com/bots/concepts/ja3-ja4-fingerprint/ (https://developers.cloudflare.com/bots/concepts/ja3-ja4-fingerprint/)

lwthiker. (2022). curl-impersonate: A special build of curl that can impersonate Chrome & Firefox. https://github.com/lwthiker/curl-impersonate (https://github.com/lwthiker/curl-impersonate)

refraction-networking. (2019). uTLS — low-level access to the ClientHello for mimicry purposes. https://github.com/refraction-networking/utls (https://github.com/refraction-networking/utls)

Jarad, G. & Bicakci, K. (2026). When Handshakes Tell the Truth: Detecting Web Bad Bots via TLS Fingerprints. ResearchGate preprint.

Kasada. (2023). The Bot Mitigation Game Has Changed Again. https://www.kasada.io/ (https://www.kasada.io/)

Kasada. (2022). Fortifies Anti-Bot Platform to Disrupt Solver Service Supply Chain. BusinessWire. https://www.businesswire.com/ (https://www.businesswire.com/)

Peakhour. (2024). JA3 vs. JA4 Fingerprinting: What's New and Why It Matters. https://www.peakhour.io/ (https://www.peakhour.io/)

Imperva. (2025). 2025 Bad Bot Report. https://www.imperva.com/ (https://www.imperva.com/)

Stamus Networks. (2024). JA3 Fingerprints Fade as Browsers Embrace TLS Extension Randomization. https://www.stamus-networks.com/ (https://www.stamus-networks.com/)

Akamai. (2019). Bots Tampering with TLS to Avoid Detection. https://www.akamai.com/ (https://www.akamai.com/)

lwthiker. (2022). HTTP/2 Fingerprinting. https://lwthiker.com/networks/2022/06/17/http2-fingerprinting.html (https://lwthiker.com/networks/2022/06/17/http2-fingerprinting.html)

Stenberg, D. (2022). curl's TLS fingerprint. https://daniel.haxx.se/blog/2022/09/02/curls-tls-fingerprint/ (https://daniel.haxx.se/blog/2022/09/02/curls-tls-fingerprint/)

Fastly. (2023). The State of TLS Fingerprinting: What's Working, What Isn't, and What's Next. https://www.fastly.com/ (https://www.fastly.com/)

Brotherston, L. (2015). TLS Fingerprinting. https://blog.squarelemon.com/tls-fingerprinting/ (https://blog.squarelemon.com/tls-fingerprinting/)

Footnotes

1. In TLS 1.3, the ClientHello's `legacy_version` field is frozen at `0x0303` (TLS 1.2) for backwards compat. The actual supported versions are advertised via the `supported_versions` extension. JA3 captures the legacy field value; JA4 accounts for this by examining the extension directly. ↩
2. Encrypted Client Hello (ECH), currently in draft at the time of writing, encrypts the inner ClientHello, including SNI and other identifying fields. Once widely deployed, ECH will limit what passive observers can extract from the handshake. However, the CDN/WAF terminating the TLS connection still sees the full plaintext content, so fingerprinting at the edge remains unaffected for now. ↩
3. curl-impersonate targets specific browser version snapshots (e.g., Chrome 110, Firefox 117), not the latest release. As browsers ship updates roughly every four weeks, the spoofed fingerprints age, creating a potential detection vector for defenders who track fingerprint freshness against known release timelines. ↩
