Zero Trust printing with rdp

- New Page

New Page

Decoupled Application Streaming & Print Pipeline

Architectural Blueprint — Guacamole-based HTML5 Delivery with Zero-Touch mTLS Printing

---

1. System Architecture

```
flowchart TB
    subgraph Client["Unmanaged Client PC (Branch/BYOD)"]
        Browser["Browser (Kiosk Mode)<br/>Guacamole HTML5 Client"]
        LPD["Local Print Daemon<br/>(Windows Service, port 9100 egress only)"]
        LocalPrinter["Local/Network Printer"]
        ClientCert["TLS Client Cert<br/>(machine-bound, in Windows Cert Store)"]
    end

    subgraph Edge["Edge / DMZ"]
        RP["Reverse Proxy<br/>Nginx or HAProxy<br/>mTLS Termination + WAF"]
    end

    subgraph Core["Core Infrastructure (Private Subnet)"]
        GW["Guacamole Gateway<br/>guacd + guacamole-client<br/>WebSocket Session Broker"]
        PrintAPI["Print Relay API<br/>(WebSocket/HTTPS)<br/>Job Queue + Auth Binding"]
        subgraph Host["Windows Server (RDS/App Host)"]
            App["Legacy Win32 EXE"]
            VDriver["Virtual Print Driver<br/>(Port Monitor)"]
```

```
            Spooler["Windows Print Spooler"]
        end
    end

    Browser <--wss 443 (RDP/VNC framebuffer)--> RP
    RP <--> GW
    GW <--RDP/VNC--> App

    LPD --outbound wss 443, long-poll/subscribe--> RP
    RP <--> PrintAPI
    App --emits job--> Spooler
    Spooler --captured by--> VDriver
    VDriver --HTTPS POST (PDF/RAW)--> PrintAPI
    PrintAPI --pushes job to bound session--> LPD
    LPD --RAW/ESC-P/PCL--> LocalPrinter

    ClientCert -. presented on TLS handshake .-> RP
```

Key design principle: two independent WebSocket planes ride the *same* port 443 through the *same* reverse proxy, but are logically separate:

Plane	Direction	Protocol	Purpose
Display plane	Server → Client (interactive)	Guacamole protocol over WSS	Framebuffer/keyboard/mouse
Print plane	Client → Server initiated (long-lived subscribe), Server → Client push	Custom JSON/binary over WSS	Print job delivery

Neither plane depends on RDP virtual channels (MS-RDPEFS is never invoked because Guacamole's RDP backend simply never advertises the printer redirection channel — you control this by omitting `enable-drive`/printer redirection GuacD parameters entirely).

---

2. Print Pipeline Architecture (Server Side)

2.1 Capture mechanism

Do not use RDP printer redirection at all — instead intercept at the Windows print subsystem on the **host** server itself:

1. **Install a custom Port Monitor / Virtual Printer Driver** on the Windows Server (the same technique PDFCreator, Bullzip, or PrintNode's server agent uses):
   - Register a printer object (e.g. `"Streamed-PDF-Printer"`) backed by a **Port Monitor DLL** (`AddMonitor`/language monitor API) or, more simply, a driver that writes to a **redirected port** pointing at a local named pipe or a "FILE:" port monitored by a watcher service.
   - The legacy EXE just prints to this printer as if it were any local printer — no application-side changes required.
2. **Rendering to a transport format:**
   - Configure the virtual driver's rendering pipeline as **PostScript → Ghostscript → PDF**, or if speed matters more than fidelity, capture the **RAW spool file (EMF/XPS)** directly and let the client-side daemon rasterize.
   - Recommendation: **PDF for LaserJet/inkjet-class documents**, **RAW passthrough for receipt/label printers** (ESC/POS, ZPL) where the app already emits printer-native command streams — don't touch these, just tunnel bytes 1:1.
3. **Spool Watcher Service** (Windows Service, .NET or Go):
   - Watches the spool directory (or named pipe) for completed jobs.
   - On job completion, reads job metadata via the Print Spooler API (`EnumJobs`, `GetJob`) to capture: originating session/user, target logical printer name (as selected inside the legacy app — this maps to the client's real printer via a naming convention, e.g. `Client:HP_LaserJet_M404`).
   - Wraps the payload: `{ session_id, client_printer_name, job_id, mime_type, payload_bytes (base64 or binary frame) }`.
   - POSTs (or streams over an already-open outbound WebSocket from server → Print Relay API) to the **Print Relay API**.

2.2 Print Relay API

A lightweight stateful broker (Node.js/Go), colocated with or adjacent to Guacamole:

- Maintains a **session-to-daemon binding table**: `guac_session_id ↔ client_daemon_connection_id ↔ mTLS cert fingerprint`.
- This binding is established at login time (see §5), so a print job tagged with `session_id` is routed to the *exact* browser tab/client machine that originated it — critical in a multi-user central-server model.
- Delivers jobs over the **already-open outbound WSS connection** from the client daemon (server-push, no inbound firewall holes needed on the client side — this is what makes BYOD/branch feasible).
- Implements **at-least-once delivery** with job ACK/NACK from the client daemon, and a dead-letter/retry queue (Redis-backed) for transient disconnects.

```
Windows Host                Print Relay API                  Client Daemon
     |  spool job captured          |                               |
     |------ HTTPS POST job ------->|                               |
     |                              |-- lookup session binding ----->|
     |                              |------ WSS push: job frame ---->|
     |                              |<----------- ACK ---------------|
     |<------- 200 OK --------------|                               |
```

---

3. Client-Side Silent Printing Service

A small, code-signed Windows service (PrintNode-client-equivalent), installed once via GPO/script or self-installing MSI on first login (no admin rights required if scoped to `HKCU` + user-mode Windows service alternative, or a scheduled task run at logon).

3.1 Responsibilities

1. **Enrollment**: On first run, generates a keypair, submits a CSR to your internal CA (or uses SCEP/ACME-internal), receives a machine-bound client certificate, stores it in `Cert:\LocalMachine\My` or `CurrentUser\My` protected by DPAPI.
2. **Discovery**: Enumerates locally installed/shared printers via `Get-Printer` / Win32 Print API, reports the list to the server on connect (`{printer_name, driver, status}`), so the server-side dropdown/naming convention in §2.1 stays in sync automatically — no manual mapping.
3. **Transport**: Opens a **single persistent outbound WSS connection** to `wss://gateway.company.com/print-channel`, authenticated via the mTLS cert from step 1 (no username/password, no VPN).
4. **Job handling**:
   - Receives a job frame → writes payload to a temp spool file.
   - If PDF: shells out to a bundled minimal PDF renderer (e.g., **SumatraPDF** `-print-to` silent mode, or **Ghostscript** `gswin64c -dPrinter=`) targeting the resolved local printer name.
   - If RAW/ESC-POS/ZPL: writes bytes directly to the target port — either the printer's TCP/IP **port 9100** (raw socket send) if it's a network printer, or via `WinSpool.WritePrinter` if it's a locally-attached USB/shared printer.
   - Sends ACK with job status back over the same channel.
5. **Resilience**: exponential backoff reconnect, job queue persisted to local disk (SQLite) so jobs aren't lost across a temporary network blip or client reboot.

3.2 Minimal service skeleton (C#, .NET Worker Service)

```
public class PrintDaemonWorker : BackgroundService
{
    private ClientWebSocket _ws;
    private readonly X509Certificate2 _clientCert;

    protected override async Task ExecuteAsync(CancellationToken ct)
    {
        while (!ct.IsCancellationRequested)
        {
            try
            {
                _ws = new ClientWebSocket();
                _ws.Options.ClientCertificates.Add(_clientCert); // mTLS
                await _ws.ConnectAsync(new Uri("wss://gateway.company.com/print-channel"),
ct);
                await AnnouncePrinters(ct);
                await ReceiveLoop(ct);
            }
            catch (Exception ex)
            {
                Log.Warn($"Print channel dropped: {ex.Message}, retrying...");
                await Task.Delay(BackoffDelay(), ct);
            }
        }
    }

    private async Task ReceiveLoop(CancellationToken ct)
    {
        var buffer = new byte[64 * 1024];
        while (_ws.State == WebSocketState.Open)
        {
            var msg = await ReceiveFullMessage(buffer, ct);
            var job = JsonSerializer.Deserialize<PrintJob>(msg);
            await DispatchToLocalPrinter(job);
            await SendAck(job.JobId, ct);
        }
```

```
    }

    private async Task DispatchToLocalPrinter(PrintJob job)
    {
        var path = Path.Combine(Path.GetTempPath(), $"{job.JobId}.{job.Ext}");
        await File.WriteAllBytesAsync(path, job.Payload);

        if (job.Mime == "application/pdf")
        {
            // Silent print via bundled SumatraPDF
            Process.Start(new ProcessStartInfo("SumatraPDF.exe",
                $"-print-to \"{job.ResolvedLocalPrinterName}\" -silent \"{path}\"")
                { CreateNoWindow = true, UseShellExecute = false });
        }
        else if (job.Mime == "application/raw")
        {
            using var tcp = new TcpClient(job.PrinterHost, 9100);
            using var stream = tcp.GetStream();
            await stream.WriteAsync(job.Payload);
        }
    }
}
```

4. Zero-Touch Mutual TLS (mTLS) — Nginx Reverse Proxy

Both the Guacamole WebSocket (display) and the print-channel WebSocket terminate mTLS at the same edge tier before reaching internal services.

```
# /etc/nginx/conf.d/gateway.conf

# Internal CA used to issue client machine certificates
ssl_client_certificate   /etc/nginx/certs/internal-ca-bundle.pem;
ssl_verify_client        on;
ssl_verify_depth         2;

# Strong TLS baseline
```

```
ssl_protocols             TLSv1.2 TLSv1.3;
ssl_ciphers                HIGH:!aNULL:!MD5;
ssl_prefer_server_ciphers on;

upstream guac_backend {
    server 10.10.20.11:8080;   # guacamole-client / tomcat
}

upstream print_relay {
    server 10.10.20.12:9443;   # Print Relay API
}

server {
    listen 443 ssl;
    server_name gateway.company.com;

    ssl_certificate     /etc/nginx/certs/gateway-server.crt;
    ssl_certificate_key /etc/nginx/certs/gateway-server.key;

    # Reject connections without a valid client cert up front
    if ($ssl_client_verify != SUCCESS) {
        return 403;
    }

    # --- Display plane: Guacamole WebSocket tunnel ---
    location /guacamole/ {
        proxy_pass http://guac_backend;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_set_header Host $host;

        # Pass verified cert identity down to the app for session binding
        proxy_set_header X-Client-Cert-CN   $ssl_client_s_dn_cn;
        proxy_set_header X-Client-Cert-Hash $ssl_client_fingerprint;

        proxy_read_timeout 3600s;
        proxy_send_timeout 3600s;
    }
```

```
    # --- Print plane: persistent WebSocket from client daemon ---
    location /print-channel {
        proxy_pass http://print_relay;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";

        proxy_set_header X-Client-Cert-CN   $ssl_client_s_dn_cn;
        proxy_set_header X-Client-Cert-Hash $ssl_client_fingerprint;

        proxy_read_timeout 86400s;   # long-lived, near-permanent connection
        proxy_send_timeout 86400s;
    }
}
```

HAProxy equivalent (if you prefer L4/L7 split) — key stanza:

```
frontend fe_gateway
    bind *:443 ssl crt /etc/haproxy/certs/gateway.pem ca-file /etc/haproxy/certs/internal-ca-
bundle.pem verify required
    http-request set-header X-Client-Cert-CN %{+Q}[ssl_c_s_dn(cn)]
    acl is_print path_beg /print-channel
    use_backend be_print if is_print
    default_backend be_guac

backend be_guac
    server guac1 10.10.20.11:8080 check

backend be_print
    server relay1 10.10.20.12:9443 check
    timeout tunnel 24h
```

Why this satisfies "zero-touch": the client OS's TLS stack presents the machine certificate automatically on handshake — the end user never sees a login prompt for network access. Application-level auth (which user, which permitted session) can still be layered on top via SSO/SAML *inside* the browser session, but network admission itself is fully silent.

---

5. End-to-End Execution Flow

1. **Boot / login**: User's branch PC starts. The Print Daemon service auto-starts, loads its machine cert from the local store, and opens the persistent `/print-channel` WSS connection through the mTLS-enforcing reverse proxy. Nginx validates the cert chain against the internal CA before the connection is ever proxied to the Print Relay API. The daemon announces its locally discovered printers.
2. **Kiosk launch**: The browser opens in kiosk mode to `https://gateway.company.com/guacamole/`. The same client certificate is presented (browser reads it from the Windows cert store — configure via GPO/registry `AutoSelectCertificateForUrls` so there's no cert-picker prompt). Nginx validates it, forwards the verified CN/fingerprint as headers to Guacamole.
3. **Session binding**: Guacamole authenticates the user (LDAP/SAML/local — your choice, layered above the mTLS network trust) and starts an RDP/VNC session to the Windows Server hosting the legacy EXE. The Print Relay API records a binding: `{guac_session_id ↔ client_cert_fingerprint ↔ daemon_connection_id}`, using the same client certificate identity seen on both planes to tie them together.
4. **Interactive use**: All screen/keyboard/mouse traffic flows over the display WebSocket, indistinguishable from a normal HTTPS site to any BYOD firewall/proxy — no VPN client, no RDP port exposure.
5. **Print trigger**: User clicks "Print" inside the legacy EXE. The app sends the job to the pre-configured virtual printer. The Port Monitor/driver on the Windows Server captures it, the Spool Watcher Service reads job metadata (including the *target client printer name*, resolved from the printer list the daemon announced in step 1) and posts the rendered payload to the Print Relay API.
6. **Routing**: The Print Relay API looks up which `daemon_connection_id` is bound to the originating `guac_session_id`, and pushes the job frame down that client's already-open WSS connection.
7. **Local rendering**: The client daemon receives the frame, writes it to a temp file, and silently prints via SumatraPDF/Ghostscript (for PDF) or a raw TCP write to port 9100 (for RAW/label/receipt formats) — landing on the user's physically local printer with no dialog boxes.
8. **Acknowledgement & cleanup**: Daemon ACKs the job back over the WebSocket; Print Relay API marks it delivered; server-side temp spool artifacts are purged on a TTL; client-side temp files are purged after successful print confirmation.
9. **Disconnect handling**: If the WSS connection drops mid-job, the Print Relay API holds undelivered jobs in a Redis-backed dead-letter queue and redelivers on reconnect (matched by cert fingerprint), so a flaky branch-office link doesn't silently drop print jobs.

Operational Notes & Hardening Checklist

- **Certificate lifecycle**: short-lived client certs (30–90 days) reissued automatically by the daemon before expiry (like ACME) avoids a manual re-enrollment fleet exercise.

- **Least privilege**: the print daemon should run as a low-privilege service account, not LocalSystem, with write access scoped only to its temp spool folder.
- **Egress-only client footprint**: client machines need zero inbound firewall rules — both planes are client-initiated outbound WSS on 443, which is what makes this workable on unmanaged/branch networks without VPN.
- **Auditability**: log every job at the Print Relay API with session, user, printer, byte count, and delivery status — this is your compliance trail in lieu of native Windows print auditing (which you've bypassed).
- **Rate limiting / abuse control** at the Nginx layer on `/print-channel` to prevent a compromised daemon from flooding the relay.
- **Guacamole config**: explicitly disable `enable-drive`, `printing-enabled`, and `enable-audio` if unused, in your `guacd` connection parameters — you want RDP printer redirection *structurally absent*, not just unused, to shrink the attack surface (RDPEFS abuse, drive redirection exfil, etc.).