WINK Streaming

Command API Reference

Version 1.4.9.1

Document Updated: August 2026

WINK Streaming, Inc.

Table of Contents

Overview

This document provides a comprehensive reference for the WINK Streaming Command API, covering WINK Forge, WINK Archive, and WINK Media Router appliances. The API enables programmatic control of video streaming operations, user management, and system administration.

Supported Products

- **WINK Forge:** Video transcoding and stream management (Firmware 1.6+)
- **WINK Archive:** Video recording and storage (Firmware 1.1+)
- **WINK Media Router:** Stream distribution and OTP authentication (Firmware 1.2+)

API Architecture

The WINK API uses XML-based messaging over HTTPS, providing:

- Secure SSL/TLS encrypted communication
- XML request/response format for structured data
- Session-based authentication with optional shared keys
- RESTful design principles for predictable behavior

> **Note:** This document does not cover WINK Crossroad integration APIs or third-party integration commands, which are documented separately.

Getting Started

Authentication

All API calls require authentication using the following XML structure:

```
<wink_api user='username' pass='password' key='shared_key'>
    <req id='unique_id' command='command_name'>payload</req>
</wink_api>
```

Authentication Parameters

Parameter	Required	Description
user	Yes	API username (must have API access level)
pass	Yes	User password
key	Optional	Shared key for enhanced security

SSL Requirements

All API endpoints require SSL/HTTPS connections. Configure SSL certificates through:

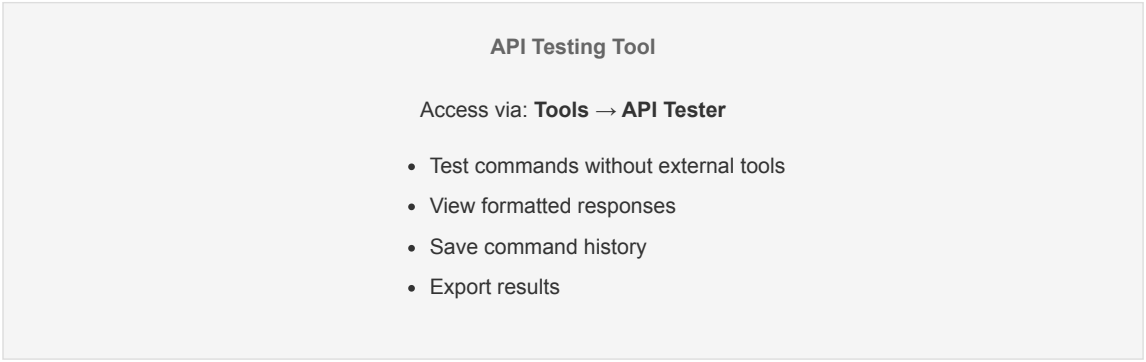
1. Navigate to **Settings → System Options → SSL Certificate**
2. Generate self-signed certificate or upload existing certificate
3. On a trusted network during bring-up, `--insecure` lets cURL skip validation of a self-signed certificate. Understand the trade before using it: credentials travel in the request body, so an unverified connection exposes them to anyone in the path. Prefer pointing cURL at the appliance CA with `--cacert /path/to/appliance-ca.pem`, and never use `--insecure` against a production system or across an untrusted link.

Example Connection

```
curl -X POST https://10.130.90.188/api/ \
    --cacert /path/to/appliance-ca.pem \
    --data-urlencode "<wink_api user='apiuser' pass='apipass'>
        <req id='1' command='version'></req>
    </wink_api>" \
    -H 'Content-Type: application/xml'
```

API Testing Tool

WINK Forge includes a built-in API testing interface:



Built-in API testing interface

WINK Forge API Commands

1. Version

Command	version
Description	Returns the API version number
Payload	None

Example	`<wink_api user='apiuser' pass='apipass'>`
	`  <req id='1' command='version'></req>`
	`</wink_api>`

Response	`<wink_api>`
	`  <resp id="1" command="version" status="ok">1.4</resp>`
	`</wink_api>`

Return Value	Numeric API version (e.g., "1.4")

2. Input Control - Start, Stop, Restart

Command	start | stop | restart
Description	Controls transcoding operations for input streams
Payload	GUID of the input stream or 'all' for all streams

Example	`<wink_api user='apiuser' pass='apipass'>`
	`  <req id='1' command='restart'>WF05-2657-1C04-7F1B-5EC0</req>`
	`</wink_api>`

Response	`<wink_api>`
	`  <resp id="1" command="restart" status="ok">WF05-2657-1C04-7F1B-5EC0</resp>`
	`</wink_api>`

Return Value	GUID of affected input or error message

> **Tip:** Use 'all' as the payload to control all streams simultaneously. This is useful for system-wide maintenance or emergency stops.

3. IP Based Input Control

Command	startip | stopip
Description	Controls transcoding based on IP:Port combination
Payload	IP address and port (format: IP:PORT)

Example	`<wink_api user='apiuser' pass='apipass'>`
	`  <req id='1' command='stopip'>10.130.90.68:554</req>`
	`</wink_api>`

Response	`<wink_api>`
	`  <resp id="1" command="stopip" status="ok">WF05-2657-1C04-7F1B-5EC0</resp>`
	`</wink_api>`

Return Value	GUID of the affected input

GUID Operations

4. Get GUID

Command	getguid
Description	Returns GUID for a specific IP:Port combination
Payload	IP and Port (e.g., 10.130.90.199:554)

Example	`<wink_api user='apiuser' pass='apipass'>`
	`  <req id='1' command='getguid'>10.130.90.199:8382</req>`
	`</wink_api>`

Response	`<wink_api>`
	`  <resp id="1" command="getguid" status="ok">WF05-2657-1C04-7F1B-5EC0</resp>`
	`</wink_api>`

5. GUID Status

Command	guidstatus
Description	Returns the current status of a stream
Payload	GUID string
Response Values	• 0 = Not publishing • 1 = Publishing • 2 = Blocked • 3 = Set as unavailable

6. Block GUID

Command	blockguid
Description	Blocks video output from viewers
Payload Options	• GUID - Block stream (default) • GUID@0 - Unblock stream • GUID@2 - Block stream • GUID@3 - Mark as unavailable

Example	`<wink_api user='apiuser' pass='apipass'>`
	`  <req id='1' command='blockguid'>WF05-2657-1C04-7F1B-5EC0</req>`
	`</wink_api>`

> **Note:** Blocked streams display a placeholder image instead of live video. This is useful for privacy compliance or content moderation.

7. Unblock GUID

Command	unblockguid
Description	Restores video output to viewers
Payload	GUID string

IP-Based Operations

8. Block IP

Command	blockip
Description	Blocks video based on IP:Port without knowing GUID
Payload	IP:Port combination (e.g., 10.130.90.199:554)
Response	Returns the GUID of the blocked stream

9. Unblock IP

Command	unblockip
Description	Restores video based on IP:Port
Payload	IP:Port combination

Stream Management

10. Input Stream Properties

Command	input_source
Description	Creates or updates video input configuration
Required (New)	• title - Stream title • desc - Stream description • path - Input URI • type - Protocol (RTSP, MJPEG, HTTP, UDP, TCP, etc.)
Required (Update)	guid - Existing stream GUID
Optional Parameters	fps — Frame rate (decimal) width — Video width in pixels height — Video height in pixels croptop — Pixels to crop from top cropright — Pixels to crop from right cropbottom — Pixels to crop from bottom cropleft — Pixels to crop from left rotate — Rotation (90, 180, 270, 0) deinterlace — 1=on, 0=off ptz_avail — 1=PTZ available, 0=none enabled — 1=enabled, 0=disabled usrlogic_id — Logical ID number usrgroup — Group assignment usrlat — Latitude coordinate usrlong — Longitude coordinate

Example	<wink_api user='apiuser' pass='apipass'>
	  <req id='1' command='input_source'>
	  <input title='Test Camera'
	  desc='Testing stream'
	  type='RTSP'
	  path='rtsp://192.168.1.100:554/stream1'
	  fps='30'
	  width='1920'
	  height='1080'
	  enabled='1' />
	  </req>
	</wink_api>

11. Input Stream Delete

Command	input_delete
Description	Removes video input and associated outputs
Payload	GUID string

> **Warning:** Deleting an input stream permanently removes all associated output streams. This action cannot be undone.

User Management

12. Create User

Command	createuser
Description	Creates a local user account

Parameters	• name - Full name
	• username - Login username
	• password - User password
	• md5 - true/false (password hashing)
	• access - Access level:
	  ◦ 1 = Admin
	  ◦ 2 = Operator
	  ◦ 3 = Viewer
	  ◦ 4 = API Only

Example	`<wink_api user='apiuser' pass='apipass'>`
	`  <req id='1' command='createuser'>`
	`  <name>Billy Armstrong</name>`
	`  <username>billy</username>`
	`  <password>SecurePass123!</password>`
	`  <access>2</access>`
	`  </req>`
	`</wink_api>`

13. Delete User

Command	deleteuser
Description	Removes a user account
Payload	XML with username

14. Change Password

Command	changepassword
Description	Updates user password
Parameters	• username - Target user • password - New password • md5 - true if pre-hashed

System Operations

15. Event Logger

Command	log
Description	Adds entry to system event log
Payload	Log message string

Example	`<wink_api user='apiuser' pass='apipass'>`
	`  <req id='1' command='log'>API integration test successful</req>`
	`</wink_api>`

16. Flush Logs

Command	flushlogs
Description	Clears the event log
Payload	None

17. Reboot Server

Command	reboot
Description	Initiates system reboot
Payload	None
Response	Error message or immediate reboot

> **Warning:** This command immediately reboots the system. Ensure all critical operations are complete before executing.

WINK Archive API Commands

Firmware Requirements: Version 1.1 or greater

> **Archive File Naming:** Files are automatically named using the pattern:
> `TITLE_[YYYY-MM-DD_HH.MM.SS]_GUID.mp4`
> Example: `EastSideSony_[2015-04-03_13.06.18]_WMA5-6E0F-1C04-86F2-E5C4.mp4`

1. Version

Same as WINK Forge version command - returns API version number.

2. Archive Control

Commands	archiver_start | archiver_stop
Description	Controls video archiving operations
Payload	GUID or blank (blank = all archivers)
Behavior	If already recording, creates overlapping instance to ensure no content loss
Example	<wink_api user='apiuser' pass='apipass'> <req id='1' command='archiver_start'>WMA5-6E0F-1C04-86F2-E5C4</req> </wink_api>

> **Storage Management:** Archives automatically maintain 85% disk capacity by removing oldest files when threshold is reached.

3. Archiver Status

Command	archiver_status
Description	Returns archiver configuration and status
Payload	GUID or blank (blank = all archivers)

Response	<wink_api>
	  <resp id="11" command="archiver_status" status="ok">
	  <archiver guid="WMA5-6E0F-1C04-86F2-E5C4"
	  title="TV2"
	  desc="Conference Room Feed"
	  type="RTMP"
	  path="rtmp://10.130.90.10/public/stream_high"
	  segmenttime="5"
	  meta1="Building A"
	  meta2="Floor 2"
	  meta3="Room 201"
	  meta4="PTZ Camera"
	  enabled="1" />
	  </resp>
	</wink_api>

4. Archiver Source

Command	archiver_source
Description	Create or modify archiver configuration
Parameters	• guid - Required for updates • title - Archiver name • desc - Description • type - Input type (RTMP, RTSP, HTTP) • path - Source URI • segmenttime - Recording duration (5-60 minutes) • meta1-4 - Metadata tags • enabled - 1=enabled, 0=disabled

5. Archive Delete

Command	archiver_delete
Description	Stop and remove archiver
Payload	GUID

WINK Media Router API

One-Time Password (OTP) Authentication

OTP tokens provide short-lived, scoped access to a video stream without ever putting a permanent credential in front of a viewer. A token is minted by your server using your API credentials, handed to the browser as part of a finished stream URL, and it expires on its own. Used correctly, a leaked stream URL is worth little: it is narrow in scope and gone in minutes.

For live video, OTP is the only control that actually works

Live video is not a normal API resource. It is fetched by a *player* rather than by your application code, over a long-lived connection, across many segment requests, often through a CDN, and the URL is trivially copied out of any page. That combination defeats nearly every access control integrators reach for first:

- **Session cookies** — native and embedded players frequently do not send them, and they are unreliable cross-origin and through caching layers. The stream breaks for legitimate viewers long before it stops anyone else.
- **IP allow-listing** — viewers roam, share NAT, and move between networks mid-session. It locks out real users while a proxy walks straight through it.
- **Origin, Referer, User-Agent, CORS** — all chosen by the caller. See CORS Is Not Security.
- **An unguessable URL** — a link that never expires is a permanent credential the moment it is pasted into a chat, captured in a screenshot, or written to a proxy log. Obscurity has no expiry date.

A short-lived, per-session, scoped token carried in the stream URL is the one mechanism that survives all of it: every player supports it because it is just a URL, it expires on its own, it can be revoked for one viewer without touching the others, and it leaves an audit trail per session. That is why the OTP flow is the supported way to secure live video on the WINK Media Router — not one option among several.

The rest of this section is about using it *correctly*. The token API is deliberately simple, and the simplest thing an integrator can build with it — minting tokens straight from the browser — is also the one thing that must never ship. Read the callout below before you write any code.

⚠ Mint OTP tokens server-to-server. Never from the browser.

Your API credentials (`apiuser` / `apipass`) authenticate token creation. Anyone who can reach them can mint valid tokens at will. Therefore:

- **Credentials live only on your server.** They are never delivered to, embedded in, or reachable from the browser — including through a public endpoint that mints a token for any caller. A minting endpoint the browser can call directly is the same as shipping the credential.
- **The browser calls your backend** (under a real, authenticated user session). Your backend creates the token with the server-side credentials and returns **only the finished, short-lived stream URL** to the browser.
- **A `Referer` or `Origin` header is not access control.** Both are trivially set by any server-side caller. Require a real session, not a header.
- **Scope tokens as narrowly as the deployment allows** — per viewer session, and per stream where possible — and keep them **short-lived**, renewed with `extend`.

This is not a style preference. An integration that skips it can be replayed by any third party to pull your feeds directly from the WINK origin, in bulk, until your token limit is exhausted and legitimate viewers are denied service. See Anti-patterns for real cases.

Default Settings: OTP tokens expire after 60 minutes by default, though for browser-facing use you should set a much shorter `duration` and renew with `extend` (see below). All tokens can be cleared via the Web Administration OTP section.

OTP Must Be Minted Server-to-Server

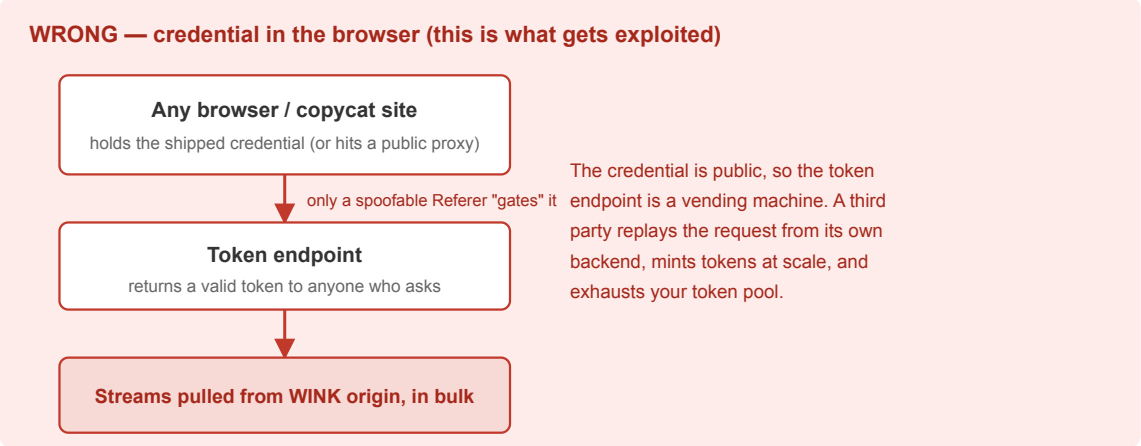
The rule above has one mechanism behind it. The credential that mints tokens is the asset being protected; everything else follows from keeping it on your server:

1. A viewer loads your page and authenticates to *your* application as they normally would.
2. The page asks *your own backend* for a stream — not the WINK Media Router directly.
3. Your backend, holding `apiuser` / `apipass` that never leave the server, calls the WINK Media Router and creates a token scoped to that viewer (and, where possible, that stream), with a short duration.
4. Your backend returns **only the finished stream URL** — playlist plus `otp` and `expires` — to the browser.
5. The browser plays it. Before it lapses, the page asks your backend again, which calls `extend` or mints a fresh token.

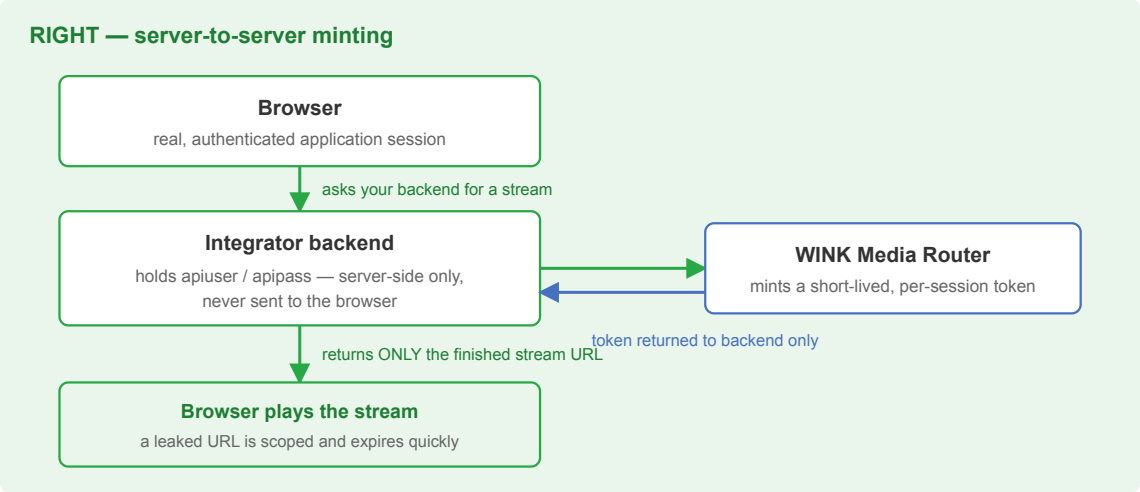
At no point does anything the browser holds let it — or a copycat replaying the same request — mint a new token. A stolen stream URL is scoped and expires; a stolen credential would not exist to steal.

The Two Flows, Side by Side

The difference between a secure and an exploitable integration is entirely in *where the credential lives and what the browser is allowed to ask for*. The diagrams below contrast the two.



Insecure: anything the browser holds can be replayed by anyone.



Secure: the credential never leaves the server; the browser only ever receives a finished, expiring URL.

Anti-patterns: What Not To Build

Each item below is a real failure mode, with the consequence. The insecure build "works" perfectly in a demo, which is exactly why it ships — the damage only shows up once someone replays it at scale.

✗ Do NOT

- Ship `apiuser` / `apipass` (or any equivalent bearer/credential) to the browser. Anyone can read it and mint tokens at will.
- Stand up a public endpoint that mints a token for any caller (a proxy the browser hits directly). That is the same as shipping the credential.
- Rely on `Referer`, `Origin`, or `User-Agent` as access control. All are attacker-controlled.
- Issue one shared/"public" token for all viewers. It defeats per-session revocation, audit, and quotas, and becomes a master key.
- Use long TTLs for browser-facing tokens. A leaked long-lived token streams for hours.
- Assume WINK's rate limits protect you. Put your own quotas and bot protection on your token endpoint.

✓ Do instead

- Keep credentials on your server; the browser receives only a finished stream URL.
- Require a real, authenticated session to obtain a stream — a header is not a session.
- Mint one token per viewer session, scoped per stream where possible.
- Keep tokens short-lived and renew with `extend`.
- Rate-limit and add bot protection (e.g. Turnstile/WAF) on your own token endpoint.
- Alert on abnormal token-issuance volume from a single session or source.

Real cases — every one of these actually shipped

None of the following are hypothetical, and none of them were clever attacks. They are the ordinary ways OTP integrations get built when the pattern is treated as a suggestion. Each one passed its own demo, which is precisely why it reached production.

> **1. The shipped default credentials were never changed**
>
> A production deployment went live still using the factory username and password. The documentation that told the integrator how to call the API was the same documentation that told everyone else what the defaults were.
>
> **Outcome: the system was open to anyone who had read the manual. No replay, no interception, no spoofing — just the published default, still in place.**

> **2. The API credentials were in the page source**
>
> An integrator wrote the token logic in client-side JavaScript and pasted `apiuser` and `apipass` straight into it. Anyone who pressed *View Source* had the account.
>
> **Outcome: the credential was public from the moment the page went live. Not leaked — published.**

> **3. Token minting was moved into the browser**
>
> Another built the whole mint-and-play flow in the browser. Even where the credential was not literally printed in the source, the browser still had to hold something that could mint, and whatever the browser holds, the viewer holds.
>
> **Outcome: every visitor became a token factory. Per-session revocation, audit and quotas all became meaningless, because every token looked the same.**

> **4. The `Referer` header was the only lock on the door**
>
> A public endpoint minted a token for any caller that arrived with the right `Referer`. The header is set by the client, so it is not a check — it is a formality. Two lines of server-side code reproduce it exactly.
>
> **Outcome: the endpoint was open to anyone who read the page once and noticed which header it wanted.**

> **5. One maximum-duration token, handed to everybody**
>
> A customer minted a single token at the longest TTL available and shared it with the entire user base, reasoning that one token was simpler than many.
>
> **Outcome: a master key with a long life and no way to revoke one user without cutting off all of them. The first time it escaped the intended audience, there was no way to tell who had it or to take it back.**

> **6. A two-second token, refreshed every two seconds**
>
> Over-correcting in the other direction, one integrator minted two-second tokens and re-minted on a two-second timer for continuous live video. Short TTLs are good; a TTL shorter than the thing it protects is not. The renewal raced the expiry on every cycle.
>
> **Outcome: visible stream stalls whenever a refresh lost the race, and a self-inflicted flood against the token endpoint that looked indistinguishable from abuse. Use a sensible TTL and `extend`; do not rebuild the session on a stopwatch.**

> **7. All of it at once**
>
> An integrator embedded a shared minting credential in its public web page and exposed a token endpoint gated only by a `Referer` header. A third-party site replayed that request from its own backend, minted tokens in bulk, and streamed the feeds directly from the WINK origin. A single token in that deployment also unlocked *every* camera, so one replayed request was enough to pull the whole set.

> **Outcome: the integrator's token limit was exhausted and legitimate viewers were intermittently denied service. Every "Do NOT" above was in play simultaneously.**

Nothing here required breaking anything. The credential was public, the header was spoofable, the token was broad and long-lived, and there was no independent rate limiting. In each case the secure pattern would have made the abuse impossible rather than merely harder — which is the entire point of implementing it exactly as written, rather than approximately.

Implement it exactly — there is no partial credit

Every case above is a deployment where most of the flow was right and one piece was taken as optional. The OTP design only holds if all of it holds; each rule below closes a door that the others leave open, so a near-miss implementation is an insecure one.

1. **Change every default credential before the system is reachable.** Defaults are published; treat a shipped username and password as already compromised.
2. **The credential never leaves your server.** Not in JavaScript, not in a config the browser can fetch, not in a mobile app bundle, not in a repository. If the browser can reach it, it is public.
3. **Your server mints the token, and only for an authenticated session.** A header, a cookie the client sets, or a URL that is merely hard to guess are not sessions.
4. **One token per viewer session**, scoped to the narrowest set of streams that session actually needs. Never one token for the whole audience, and never one token for the whole camera set when a subset will do.
5. **Short TTL, but longer than a frame.** Pick a duration measured in minutes, renew with `extend` before expiry, and do not re-mint on a tight timer.
6. **Rate-limit and bot-protect your own token endpoint.** WINK's limits protect WINK; they will not stop your endpoint being drained, and hitting them is what takes your viewers offline.
7. **Destroy tokens on logout** and alert on abnormal issuance volume from a single session or source.

CORS Is Not Security

This comes up on nearly every integration, so it is worth stating plainly: **CORS is not an access control, and it never was.** It is a browser convention. Treating it as a security boundary is one of the most common reasons an integration that "felt safe" turns out not to be.

Cross-Origin Resource Sharing exists to protect *the user* from a page they did not trust, by stopping that page from silently reading responses from another origin where the user is already logged in. It runs inside the browser, at the browser's discretion, and it decides what *a page* may read. It says nothing whatsoever about who may call your endpoint.

Why it cannot protect an endpoint

> **The check happens on the client, not on your server**
>
> Your server receives the request and does the work regardless. CORS only governs whether the browser hands the response back to the calling script afterwards. By then the request has already been served.

> **Anything that is not a browser ignores it completely**
>
> `curl`, a server-side script, a mobile app, a scraper, or any HTTP library will not read your `Access-Control-Allow-Origin` header, let alone honour it. There is no enforcement outside the browser, and an attacker is not obliged to use one.

> **The browser itself can be told to skip it**
>
> Chrome and the other major browsers can be launched with the same-origin policy disabled by a command-line flag, and extensions exist whose whole purpose is stripping CORS headers. A user who wants past the check is a browser restart away from it.

> **A proxy erases it in one line**
>
> Any request can be relayed through a trivial server-side proxy that forwards it, receives the response, and returns it with whatever headers the attacker likes. This is the standard workaround and takes minutes to build.

> **We have seen client-side tricks used to get around it in practice**
>
> Integrators have worked around their own CORS restrictions with in-page techniques rather than fixing the design. If a developer can defeat the check to make their own product work, so can someone with less friendly intentions.

Put together: a permissive CORS policy does not create a vulnerability, and a restrictive one does not remove one. The header is simply not part of the security model.

What to do instead

Configure CORS correctly anyway — it is good hygiene and it prevents other people's pages from misusing a logged-in session. Just do not count it as protection.

- **Authenticate every call** that mints or manages a token, server-side, against a real session. That is the control.
- **Keep the minting credential off the browser entirely**, at which point the origin of a browser request stops mattering.
- **Rate-limit by session and by source**, not by origin.
- **Scope and time-limit tokens** so that a request you did not anticipate is still contained.

> **Rule of thumb:** if the only thing standing between a caller and a token is something the caller sends — an `Origin`, a `Referer`, a `User-Agent`, or a header your own JavaScript adds — there is no control in place. Every one of those values is chosen by whoever makes the request.

OTP Endpoints & Token Usage

The OTP actions live on two different paths. Token creation and token management are handled separately:

Action	Endpoint
create	`http://router.example.com/api/otp/`
extend | query | destroy	`http://router.example.com/api/v1/otp/`

Both endpoints accept form-encoded HTTP POST data and require `apiuser` and `apipass` on every request.

> **Legacy path:** earlier integrations created tokens at `/otp/api/` (the segments reversed). That path still responds and existing integrations keep working, but it is **not the correct endpoint and may be deprecated at any time**. Use `/api/otp/` for token creation in all new work, and migrate existing callers when convenient. If you see `/otp/api/` in older code or documentation, treat it as the old form of `/api/otp/` .

Passing the Token in a Stream URL

The token is appended to the playlist request as an `otp` query parameter, alongside an `expires` timestamp:

```
http://router.example.com/200/public/hls/<stream>.m3u8?otp=a1b2c3d4e5f6&expires=1691234567
```

Only the `.m3u8` playlist request carries the token; the segment (`.ts`) requests that follow are authorized off the same playlist request, so you do not rewrite individual segment URLs.

> **Understand the token's scope before you rely on it.** By default a token authorizes the stream(s) it is presented against for the life of the token — it is *not* automatically limited to a single camera. That convenience becomes a liability if the token can be minted by the wrong party: one broad token then unlocks everything it can reach. Scope tokens as narrowly as your deployment allows (per viewer session, and per stream where the deployment supports it), and keep them short-lived so the window in which a leaked token is useful is small. If you need per-camera scoping and are unsure whether your deployment enforces it, contact support before going live.

Token Scope Across Multiple Routers

Tokens can be cluster-wide across a deployment: a token created on one Media Router is valid on every Media Router where the API credentials are defined identically, so a five-router deployment can operate on one token per viewer session rather than one per router.

> **Cluster-wide scope cuts both ways.** The same property that lets one token span the cluster means one *stolen* token unlocks the whole fleet. Treat cluster-wide tokens as a deliberate choice, not a default to reach for: where your threat model warrants it, scope tokens more narrowly, and always keep TTLs short so a leaked token's blast radius is bounded in time even when it is broad in space.

> **One Token Per Session:** A single token must not be shared by all visitors to a site. Issue a token per viewer session so that expiry, extension, and revocation apply to one viewer at a time and the audit trail stays meaningful. A shared "public" token is a master key: it cannot be attributed to a viewer, rate-limited per viewer, or revoked without cutting off everyone.

1. Create OTP Token

Command	create
Method	HTTP POST
Endpoint	`/api/otp/`
Parameters	

apiuser / apipass	API credentials	Required
action	create	Required
duration	Minutes, 1 to 900 (default: 60). For browser-facing use, set this short — on the order of a visit — and renew with `extend`. The 900-minute maximum is rarely appropriate for public or browser-facing tokens.	Optional
hash_type	alpha | numeric | alphanumeric	Default: alphanumeric
hash_length	Max 128 (default: 32)	Optional
expiry_url	Redirect URL on expiry	Optional
expiry_message	Custom expiry message	Optional

Example

```
curl -d
"apiuser=apiuser&apipass=apipass&action=create&duration=15&hash_type=numeric&hash_length=20"
\
     -X POST http://router.example.com/api/otp/
```

Response	24814928371014572819
Notes	The create call returns the raw token string on its own, with no wrapper or formatting. Errors are returned as `0:` followed by a description (for example `0:Invalid Login`).

2. Destroy/Extend OTP

Commands	destroy | extend
Description	Manage existing OTP tokens
Endpoint	`/api/v1/otp/`
Parameters	• apiuser / apipass - API credentials • action - destroy or extend • token - Existing token string • duration - Minutes, 1 to 900 (for extend only)
Example	`curl -d "apiuser=apiuser&apipass=apipass&action=extend&token=2481492837101457 2819&duration=900" \ -X POST http://router.example.com/api/v1/otp/`
Response	Success | Failed

3. Query Token

Command	query
Description	Returns token expiration time
Endpoint	`/api/v1/otp/`
Parameters	• apiuser / apipass - API credentials • action - query • token - Token to query
Example	`curl -d "apiuser=apiuser&apipass=apipass&action=query&token=2481492837101457 2819" \ -X POST http://router.example.com/api/v1/otp/`
Response Format	YYYY-MM-DD HH:MM:SS

OTP Integration FAQ

The questions below come up regularly during web portal and partner integrations.

How is the token passed in the video URL?

As an `otp` query parameter on the playlist request, alongside an `expires` timestamp:

```
http://router.example.com/200/public/hls/<stream>.m3u8?otp=a1b2c3d4e5f6&expires=1691234567
```

Only the `.m3u8` request needs it. Segment requests inherit the authenticated session, so individual `.ts` URLs do not need to be rewritten.

Can one token be shared by all visitors?

No. Issue a token per viewer session. Sharing a single token across every visitor removes the ability to expire, extend, or revoke access for one viewer, and it makes the access log useless for audit purposes.

Do we need one token per Media Router, or one for the whole deployment?

One per viewer session is enough: tokens can be cluster-wide, so a session's token spans every Media Router where the API credentials are defined identically. Note the trade-off — a cluster-wide token that leaks unlocks the whole fleet, so keep durations short and scope more narrowly where your threat model calls for it.

Which endpoint handles which action?

Both paths are in use, and the actions live on different ones:

- **Create:** `http://router.example.com/api/otp/`
- **Extend / query / destroy:** `http://router.example.com/api/v1/otp/`

Create parameters are `action=create`, `duration=<minutes>`, `hash_type=<alpha|numeric|alphanumeric>`, and `hash_length=<max 128, default 32>`, plus `apiuser` and `apipass`. The create call returns the raw token string on its own. `action=query` returns the expiration as `YYYY-MM-DD HH:MM:SS`.

What duration should we set, and how should expiry be handled?

`duration` is in minutes and accepts 1 through 900. With per-session tokens, start short - on the order of the length of a typical visit - and call `extend` for sessions that are still watching, rather than issuing long-lived tokens up front.

On `expiry_url` versus `expiry_message`: for a public website, neither should be the primary mechanism. Have the page request a fresh token when one lapses; that is a cleaner experience than a redirect mid-stream. Setting `expiry_message` is worth doing as a fallback, so anything that does slip through shows a sensible message in the player instead of a bare failure.

Are there rate limits on the OTP API?

There are per-account and per-session quotas on token creation, and they exist to protect the platform — not to protect *your* integration. Do not treat WINK's limits as your abuse defense. Unbounded minting is both an abuse vector and an availability risk: a caller hammering an open token endpoint can exhaust your account's token cap and deny service to your legitimate viewers.

Put your own rate limiting and bot protection in front of your token endpoint (per-session and per-IP quotas, plus a challenge such as Cloudflare Turnstile or a WAF rule), and alert on abnormal issuance volume from a single session or source as an early abuse signal. Contact support if a legitimate deployment is expected to need a higher quota.

Do existing stream URLs survive a firmware upgrade?

Yes. Stream URLs are unchanged by the upgrade; the paths documented for a deployment stay the same.

Integrator Security Checklist

Confirm every item below before an OTP integration goes live in front of the public. If any answer is "no," the integration is exploitable in the way described in Anti-patterns.

- [ ] API credentials (`apiuser` / `apipass`) live on the server only, and are never delivered to or reachable from the browser.
- [ ] The token endpoint requires a real, authenticated application session — not a `Referer` / `Origin` header.
- [ ] The browser receives only a finished stream URL, never anything it can use to mint a new token.
- [ ] Tokens are per viewer session, and scoped per stream where the deployment allows it.
- [ ] Tokens are short-lived, renewed with `extend` rather than issued long up front.
- [ ] Your own rate limiting and bot protection sit in front of your token endpoint.
- [ ] You alert on abnormal token-issuance volume from a single session or source.
- [ ] Grid / overview pages use periodic snapshot images, not a live stream per tile; only the focused camera goes live.

OTP Code Examples

The cURL calls above show the raw requests. The examples below show the same calls in context, issuing and maintaining one token per viewer session. In every example the credentials stay on the server and only a finished, short-lived URL reaches the browser — copy that shape, not just the API calls.

> **⚠ Keep credentials server-side**
>
> `apiuser` and `apipass` appear only in the server-side code below (the PHP and C# examples). They must never be exposed to a browser. The JavaScript example deliberately holds no credential — it asks its own backend for a token and receives only the finished URL. If you find yourself putting `apiuser` / `apipass` into client-side JavaScript, stop: that is the exploited anti-pattern.

PHP: Per-Session Token

Creates a token on the visitor's first request, reuses it for the rest of the session, and extends it when it is close to lapsing:

```php
<?php
session_start();

// Mint only for a caller you have already authenticated. Without this check the page
// is a public minting endpoint - the anti-pattern in "Real cases" above.
if (empty($_SESSION['user_id'])) {
    http_response_code(401);
    exit('Not authenticated');
}

// Create and manage live on different paths
$otp_create = 'https://router.example.com/api/otp/';
$otp_manage = 'https://router.example.com/api/v1/otp/';
$api_user   = 'apiuser';
$api_pass   = 'apipass';
$playlist   = 'https://router.example.com/200/public/hls/WF05-333F-4D5C-E636-2937.m3u8';
$duration   = 15;   // minutes, 1-900

function otp_post($url, $fields) {
    $options = array(
        'http' => array(
            'header'  => "Content-type: application/x-www-form-urlencoded\r\n",
            'method'  => 'POST',
            'content' => http_build_query($fields),
            'timeout' => 10,
            ),
        );

    $result = @file_get_contents($url, false, stream_context_create($options));
    return ($result === false) ? '' : trim($result);
}

// Still watching and the token is nearly up - extend rather than issue a second one
if (!empty($_SESSION['otp_token']) && ($_SESSION['otp_expires'] - time()) < 60) {

    $resp = otp_post($otp_manage, array(
        'apiuser'  => $api_user,
        'apipass'  => $api_pass,
        'action'   => 'extend',
        'token'    => $_SESSION['otp_token'],
        'duration' => $duration
        ));

    if ($resp === 'Success') {
        $_SESSION['otp_expires'] = time() + ($duration * 60);
    } else {
        unset($_SESSION['otp_token']);   // extend failed - fall through and create
    }
}

// No token yet (or the extend failed) - create one for this session
if (empty($_SESSION['otp_token'])) {

    $token = otp_post($otp_create, array(
        'apiuser'     => $api_user,
        'apipass'     => $api_pass,
        'action'      => 'create',
        'duration'    => $duration,
        'hash_type'   => 'alphanumeric',
        'hash_length' => 32
        ));
```

```
    // Errors are returned as "0:" followed by a description
    if ($token === '' || substr($token, 0, 2) === '0:') {
        die('OTP API error: ' . htmlspecialchars($token));
    }

    $_SESSION['otp_token']   = $token;
    $_SESSION['otp_expires'] = time() + ($duration * 60);
}

// Only the .m3u8 carries the token - segment requests inherit the session
$video_url = $playlist
           . '?otp=' . urlencode($_SESSION['otp_token'])
           . '&expires=' . $_SESSION['otp_expires'];

// Serving the same values as JSON is all the JavaScript example below needs:
// header('Content-Type: application/json');
// echo json_encode(array('token' => $_SESSION['otp_token'], 'expires' => $_SESSION['otp_expires']));
?>
```

JavaScript: Player Integration

The page asks its own backend for the session's token, builds the playlist URL, and requests a fresh token shortly before the current one lapses:

```
// hls.js instance, created once by your player setup. Declared here so the example
// is complete; swap in whatever player you use.
let hls = null;
let session = null;   // { token: "a1b2c3d4e5f6", expires: 1691234567 }
let refreshTimer = null;

async function getSessionToken() {
    const res = await fetch('/otp-token.php', { credentials: 'same-origin' });
    if (!res.ok) {
        throw new Error('Token request failed: ' + res.status);
    }
    return res.json();
}

function playlistUrl(playlist) {
    return playlist
        + '?otp=' + encodeURIComponent(session.token)
        + '&expires=' + session.expires;
}

function loadIntoPlayer(playlist) {
    const video = document.getElementById('player');
    const url = playlistUrl(playlist);

    if (video.canPlayType('application/vnd.apple.mpegurl')) {
        video.src = url;           // Safari and other native HLS players
    } else {
        hls.loadSource(url);       // hls.js or the player of your choice
    }
}

async function startStream(playlist) {
    session = await getSessionToken();
    loadIntoPlayer(playlist);
    scheduleRefresh(playlist);
}

// Refresh before expiry rather than letting playback fail. This is cleaner than
// redirecting mid-stream with expiry_url; keep expiry_message set as a fallback.
// Renew with headroom - do not re-mint on a tight timer (see "Real cases").
function scheduleRefresh(playlist) {
    const msRemaining = (session.expires * 1000) - Date.now() - 60000;   // 60s headroom

    clearTimeout(refreshTimer);
    refreshTimer = setTimeout(async function () {
        try {
            session = await getSessionToken();   // backend extends, or creates a new token
            loadIntoPlayer(playlist);
            scheduleRefresh(playlist);
        } catch (err) {
            // Do not let the chain die silently - back off and try again.
            console.error('Token refresh failed, retrying:', err);
            refreshTimer = setTimeout(function () { scheduleRefresh(playlist); }, 10000);
        }
    }, Math.max(msRemaining, 5000));
}

startStream('https://router.example.com/200/public/hls/WF05-333F-4D5C-E636-2937.m3u8')
    .catch(function (err) { console.error('Could not start stream:', err); });
```

C#: OTP Client

A small client wrapping the three actions, suitable for an ASP.NET application holding one token per user session:

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Net.Http;
using System.Threading.Tasks;

public class WinkOtpClient
{
    private static readonly HttpClient Http = new HttpClient();

    private readonly string _createUrl;   // https://router.example.com/api/otp/
    private readonly string _manageUrl;   // https://router.example.com/api/v1/otp/
    private readonly string _user;
    private readonly string _pass;

    public WinkOtpClient(string createUrl, string manageUrl, string user, string pass)
    {
        _createUrl = createUrl;
        _manageUrl = manageUrl;
        _user = user;
        _pass = pass;
    }

    private async Task<string> PostAsync(string url, Dictionary<string, string> fields)
    {
        fields["apiuser"] = _user;
        fields["apipass"] = _pass;

        using (var content = new FormUrlEncodedContent(fields))
        {
            var response = await Http.PostAsync(url, content);
            response.EnsureSuccessStatusCode();
            return (await response.Content.ReadAsStringAsync()).Trim();
        }
    }

    // Returns the raw token string. Errors arrive as "0:Invalid Login"
    public async Task<string> CreateAsync(int durationMinutes = 15)
    {
        var token = await PostAsync(_createUrl, new Dictionary<string, string>
        {
            { "action", "create" },
            { "duration", durationMinutes.ToString() },   // 1-900
            { "hash_type", "alphanumeric" },
            { "hash_length", "32" }
        });

        if (token.StartsWith("0:"))
        {
            throw new InvalidOperationException("OTP API error: " + token);
        }

        return token;
    }

    public async Task<bool> ExtendAsync(string token, int durationMinutes = 15)
    {
        var result = await PostAsync(_manageUrl, new Dictionary<string, string>
        {
            { "action", "extend" },
            { "token", token },
```

```
            { "duration", durationMinutes.ToString() }
        });

        return result == "Success";
    }

    public async Task<DateTime> QueryAsync(string token)
    {
        var expires = await PostAsync(_manageUrl, new Dictionary<string, string>
        {
            { "action", "query" },
            { "token", token }
        });

        // Errors arrive in the same body as a valid answer - check before parsing.
        if (expires.StartsWith("0:"))
        {
            throw new InvalidOperationException("OTP API error: " + expires);
        }

        // The router reports UTC; keep the kind explicit so callers cannot misread it.
        var parsed = DateTime.ParseExact(expires, "yyyy-MM-dd HH:mm:ss",
                                         CultureInfo.InvariantCulture);

        return DateTime.SpecifyKind(parsed, DateTimeKind.Utc);
    }

    public async Task<bool> DestroyAsync(string token)
    {
        var result = await PostAsync(_manageUrl, new Dictionary<string, string>
        {
            { "action", "destroy" },
            { "token", token }
        });

        return result == "Success";
    }

    // Only the .m3u8 carries the token - segments inherit the session
    public static string BuildPlaylistUrl(string playlist, string token, DateTimeOffset expires)
    {
        return string.Format("{0}?otp={1}&expires={2}",
                             playlist,
                             Uri.EscapeDataString(token),
                             expires.ToUnixTimeSeconds());
    }
}
```

Usage, holding one token per session:

```
// Credentials come from configuration or a secret store - never from source control.
var otp = new WinkOtpClient("https://router.example.com/api/otp/",
                            "https://router.example.com/api/v1/otp/",
                            Configuration["Wink:ApiUser"],
                            Configuration["Wink:ApiPass"]);

var token   = await otp.CreateAsync(15);
var expires = new DateTimeOffset(await otp.QueryAsync(token), TimeSpan.Zero);

var videoUrl = WinkOtpClient.BuildPlaylistUrl(
    "https://router.example.com/200/public/hls/WF05-333F-4D5C-E636-2937.m3u8",
    token, expires);

// Later, while the session is still watching:
await otp.ExtendAsync(token, 15);

// When the session ends:
await otp.DestroyAsync(token);
```

Code Examples

JavaScript Integration

> **⚠ Command API credentials never belong in browser JavaScript**
>
> The Command API authenticates with `apiuser` and `apipass` in the request body. A browser that can send that request is a browser that holds the credential, and anyone can read it. The pattern below is therefore the **same anti-pattern** documented for OTP in Anti-patterns: What Not To Build — and it is worse here, because the Command API can start, stop, block, reconfigure and reboot, not merely view. Put the credential on your server and let the browser talk to *your* backend.

The correct shape: the page calls your own endpoint under an authenticated session, and your server holds the credential and speaks to the appliance.

```
// Browser - holds no credential, talks only to your own backend
async function guidStatus(guid) {
    const res = await fetch('/api/forge-proxy.php', {
        method: 'POST',
        credentials: 'same-origin',              // your session cookie, not an API credential
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ command: 'guidstatus', payload: guid })
    });

    if (!res.ok) {
        throw new Error('Proxy request failed: ' + res.status);
    }

    return res.json();
}

guidStatus('WF05-1EDA-1C04-6E33-ABF9')
    .then(function (r) { console.log('Status:', r.data); })
    .catch(function (e) { console.error(e); });
```

The matching backend. It authenticates the caller first, allows only the commands this application is meant to expose, and adds the credentials itself:

```php
<?php
// /api/forge-proxy.php - credentials live here, never in the browser
session_start();

// 1. Require a real, authenticated session. A header is not a session.
if (empty($_SESSION['user_id'])) {
    http_response_code(401);
    exit(json_encode(array('error' => 'Not authenticated')));
}

// 2. Allow-list the commands this application may issue. Never proxy arbitrary commands.
$allowed = array('version', 'guidstatus', 'inputstatus');

$body    = json_decode(file_get_contents('php://input'), true);
$command = isset($body['command']) ? $body['command'] : '';
$payload = isset($body['payload']) ? $body['payload'] : '';

if (!in_array($command, $allowed, true)) {
    http_response_code(403);
    exit(json_encode(array('error' => 'Command not permitted')));
}

// 3. Build the request server-side, escaping anything that came from the client.
$xml = sprintf(
    "<wink_api user=%s pass=%s><req id='1' command=%s>%s</req></wink_api>",
    quoteattr(FORGE_API_USER),          // from config/environment, not source control
    quoteattr(FORGE_API_PASS),
    quoteattr($command),
    htmlspecialchars($payload, ENT_XML1 | ENT_QUOTES, 'UTF-8')
);

$ch = curl_init('https://forge.example.com/api/');
curl_setopt_array($ch, array(
    CURLOPT_POST           => true,
    CURLOPT_POSTFIELDS     => $xml,
    CURLOPT_HTTPHEADER     => array('Content-Type: application/xml'),
    CURLOPT_RETURNTRANSFER => true,
    CURLOPT_TIMEOUT        => 10,
    CURLOPT_SSL_VERIFYPEER => true,     // leave on in production
    ));

$response = curl_exec($ch);
curl_close($ch);

$node = @simplexml_load_string($response);

header('Content-Type: application/json');
echo json_encode(array(
    'status' => $node ? (string) $node->resp['status'] : 'error',
    'data'   => $node ? trim((string) $node->resp)      : 'Malformed response',
));

function quoteattr($s) {
    return "'" . htmlspecialchars($s, ENT_XML1 | ENT_QUOTES, 'UTF-8') . "'";
}
?>
```

> **The one exception:** the appliance's own API Testing Tool runs on the appliance, behind its admin login, on a trusted network. Typing credentials into that form is fine because they never leave the appliance. That is not the same as shipping them in a page you publish.

Python Integration

A server-side client. Values are escaped before they reach the XML, every call has a timeout, and certificate verification stays on:

```
import requests
import xml.etree.ElementTree as ET
from xml.sax.saxutils import escape, quoteattr


class WinkAPIError(Exception):
    pass


class WinkAPI:
    def __init__(self, host, username, password, verify_ssl=True, timeout=10):
        self.host = host
        self.username = username
        self.password = password
        self.verify_ssl = verify_ssl
        self.timeout = timeout
        self.api_url = f"https://{host}/api/"

    def execute_command(self, command, payload="", req_id="1"):
        # quoteattr/escape matter: a password containing ' or & would otherwise
        # break the document, and an unescaped payload could inject elements.
        xml_request = (
            f"<wink_api user={quoteattr(self.username)} pass={quoteattr(self.password)}>"
            f"<req id={quoteattr(str(req_id))} command={quoteattr(command)}>"
            f"{escape(str(payload))}"
            f"</req></wink_api>"
        )

        try:
            response = requests.post(
                self.api_url,
                data=xml_request.encode("utf-8"),
                headers={"Content-Type": "application/xml"},
                verify=self.verify_ssl,
                timeout=self.timeout,
            )
            response.raise_for_status()
        except requests.RequestException as exc:
            raise WinkAPIError(f"Request to {self.api_url} failed: {exc}") from exc

        try:
            root = ET.fromstring(response.text)
        except ET.ParseError as exc:
            raise WinkAPIError(f"Malformed response: {response.text[:200]}") from exc

        resp = root.find("resp")
        if resp is None:
            raise WinkAPIError(f"No <resp> element in response: {response.text[:200]}")

        return {
            "status": resp.get("status"),
            "command": resp.get("command"),
            "data": (resp.text or "").strip(),
        }


# Example usage - credentials come from the environment, not from source control
import os

api = WinkAPI(
    "forge.example.com",
```

```
    os.environ["WINK_API_USER"],
    os.environ["WINK_API_PASS"],
)

try:
    version = api.execute_command("version")
    print(f"API Version: {version['data']}")

    status = api.execute_command("guidstatus", "WF05-1EDA-1C04-6E33-ABF9")
    print(f"Stream Status: {status['data']}")
except WinkAPIError as exc:
    print(f"API call failed: {exc}")
```

Self-signed certificates

An appliance running its factory self-signed certificate will fail verification. Reach for `verify_ssl=False` only on a trusted network during bring-up, and understand the trade: the credentials travel in the request body, so an unverified connection means anyone in the path can read them. The fix is to install a real certificate, or to point `verify` at the appliance's CA bundle (`verify="/path/to/appliance-ca.pem"`) rather than disabling the check.

cURL Examples

Basic Commands

```
# Get API version
curl -X POST https://forge.example.com/api/ \
    --cacert /path/to/appliance-ca.pem \
    --data-urlencode "<wink_api user='apiuser' pass='apipass'><req id='1' command='version'></req>
</wink_api>" \
    -H 'Content-Type: application/xml'

# Start all streams
curl -X POST https://forge.example.com/api/ \
    --cacert /path/to/appliance-ca.pem \
    --data-urlencode "<wink_api user='apiuser' pass='apipass'><req id='1' command='start'>all</req>
</wink_api>" \
    -H 'Content-Type: application/xml'

# Create OTP token (server-side only - apiuser/apipass never reach the browser)
# Keep the duration short for browser-facing use and renew with extend.
curl -d "apiuser=apiuser&apipass=apipass&action=create&duration=15" \
     -X POST https://router.example.com/api/otp/
```

Credentials on a command line are not private

Anything passed with `-d` is visible to other users via `ps` while the command runs, and is written to your shell history. That is acceptable at a console you control; it is not a way to drive the API from a script or a shared host. Use `--data` , or an environment variable your shell does not log, and keep the real integration in server-side code as shown above.

Advanced Examples

```
# Create new input with full configuration
curl -X POST https://forge.example.com/api/ \
    --cacert /path/to/appliance-ca.pem \
    --data-urlencode "<wink_api user='apiuser' pass='apipass'>
        <req id='1' command='input_source'>
            <input title='Conference Room'
                   desc='Main conference room HD camera'
                   type='RTSP'
                   path='rtsp://admin:password@192.168.1.100:554/stream1'
                   fps='30'
                   width='1920'
                   height='1080'
                   ptz_avail='1'
                   usrlat='41.8781'
                   usrlong='-87.6298'
                   enabled='1' />
        </req>
    </wink_api>" \
    -H 'Content-Type: application/xml'
```

Notes & Best Practices

API Versioning

- API methods maintain backward compatibility across minor versions
- Major version changes may introduce breaking changes
- Always verify API compatibility before firmware upgrades
- Test API calls on non-production systems first

Performance Considerations

- Batch operations when possible to reduce API calls
- Implement exponential backoff for retry logic
- Monitor API response times for system health
- Use connection pooling for high-frequency operations

Security Best Practices

- **Mint OTP tokens server-to-server.** API credentials stay on your server; the browser receives only a finished, short-lived stream URL. Never ship `apiuser` / `apipass` to the client or expose a public endpoint that mints tokens for any caller. See OTP Must Be Minted Server-to-Server and the Integrator Security Checklist.
- Require a real authenticated session on your token endpoint — a `Referer` / `Origin` header is not access control.
- Keep browser-facing tokens per-session, scoped as narrowly as possible, and short-lived.
- Put your own rate limiting and bot protection in front of your token endpoint; do not rely on WINK's limits as your abuse defense.
- Always use HTTPS for API communications
- Implement API-specific user accounts with minimal permissions

- Rotate API credentials regularly
- Log all API access for audit purposes, and alert on abnormal token-issuance volume
- Use shared keys for additional security layers

Integration Tips

- OTP expiry should match application session duration
- Start short and call `extend` for sessions still watching, rather than issuing long-lived tokens up front
- Create one OTP per viewer session, not one shared token per site
- Define API credentials identically on every Media Router so cluster-wide tokens validate everywhere
- Implement proper error handling for all API responses
- Cache GUID lookups to reduce API calls
- Use the built-in API testing tool for development

> **Development Tip:** The API Testing tool in WINK Forge (Tools → API Tester) can export command templates in various programming languages.

Customer Support

Contact Information

Email Support	support@wink.co
Phone Support	+1-312-281-5433
Corporate Website	https://www.wink.co

Support Requirements

When contacting support, please provide:

- Customer ID
- System serial number or license key
- Firmware version (from version API command)
- Specific API commands causing issues
- Error messages or unexpected responses
- Network topology if relevant

Additional Resources

- **Developer Portal:** Access code samples and SDKs
- **Community Forum:** Share experiences with other developers
- **Training Videos:** API integration tutorials
- **Release Notes:** Latest API changes and additions

Programming Language Examples: Contact WINK Support for examples in your preferred programming language. While availability isn't guaranteed, we maintain samples for Python, PHP, Java, C#, and Node.js.

Document Version: 1.4.9.1 - Updated September 2026

Original Issue: August 2022