VTX Exec 1: remote exec and port forwarding

Nov 15, 2026 · VTX 1 · extension · draft d01 · MIT licence · vtx.foundation/spec/v1/exec-1/

Exec 1 gives VTX what people still keep SSH for: running a command and reading its output and exit status, and forwarding TCP, UDP and Unix-socket connections in both directions, all started by the client and allowed by the server's policy.

§1 Status and scope

Exec 1 is draft d01 of an extension of VTX 1, in conformance profile 1, built on the core at draft d03. It adds three dynamic channel types and one scene node; no rule of the core changes.

- `exec` channels run a command on the server as the signed-in user, with stdin, stdout, stderr, signals and the exit status, and no terminal.
- `forward` channels carry one TCP connection, UDP flow or Unix-socket connection each.
- `listen` channels ask the server to listen for connections and send each one back to the client, as SSH's remote forwarding does.
- A `port` node lets an application, an agent for instance, offer a server-side port that the user opens with one click.

Everything starts at the client. The server only grants what its policy allows, and never runs anything on the client or reaches into the client's network on its own: the one way in is a listener the client asked for, leading to a target the client chose.

Not in Exec 1. Agent forwarding and X11 forwarding are left out on purpose. Interactive programs that need a terminal run as ordinary sessions with a VT grid. Scripted file copies use exec, since Transfer 1 always asks the user.

§2 Capabilities and policy

The client asks for what it wants to use; the server's answer, as in v1, lists what its policy grants the signed-in user.

Key	Value	Meaning
`exec`	`true`	Run commands (Section 3)
`forward`	`tcp`, `udp`, `unix`, `listen`, each `true` or absent	Forward these kinds of connection; `listen` allows listeners on the server (Section 4)

- Policy belongs to the server: which users may run commands, which hosts and ports they may reach, which addresses a listener may bind. A refusal is OPEN_REFUSED `refused` with a reason: `policy`, `not-found`, `unreachable`, `refused`, `timeout` or `in-use`.
- Both stay off until the deployment turns them on for a user. Listeners bind to loopback unless policy allows more, as SSH's `GatewayPorts` does.
- The server logs every command and forward with the user, the time and the target.

Headless sessions. A HELLO that offers `exec` or `forward` but no display capability (`cells`, `components` or any `surface.` key) asks for a headless session. The server starts no application, and the session ends with the connection instead of lingering. Scripts work this way; an interactive client opens the same channels inside its normal session.

§3 Exec channels

An exec channel runs one command as the signed-in user, without a terminal, and returns its output and its exit status. The client opens it with an odd ID.

OPEN param	Meaning
`argv`	The program and its arguments, run directly with no shell; or
`shell`	A command line for the user's login shell
`env`	Variables added to the environment, as policy allows (like SSH's `AcceptEnv`)
`cwd`	The working directory
`stdin`	`false`: the command reads end-of-file at once (default `true`)
`merge`	`true`: stderr goes into stdout

Type	Frame	Fields	Purpose
0x10	DATA	`fd`, `bytes`	fd 0 from the client; fd 1 and 2 from the server
0x11	EOF	`fd`	That stream has ended
0x12	SIGNAL	`signal`	From the client: `SIGINT`, `SIGTERM`, `SIGHUP`, `SIGKILL`, `SIGQUIT`, `SIGUSR1` or `SIGUSR2`

- When the command ends, the server sends EOF on stdout and, unless merged, on stderr, then CLOSE `no_error` with `exit` or `signal`, the fields v1's CLOSE already has.
- A CLOSE from the client hangs up: the server sends `SIGHUP` to the command, and `SIGKILL` if it still runs 5 seconds later.
- When the client stops granting window, output pauses and the command blocks on a full pipe, as it would locally.
- A command that needs a terminal, such as `top`, runs as a normal session with the command as its target, on a VT grid.

```
c → OPEN    channel 17, type exec, params {"argv": ["uptime"], "stdin": false}
s → OPEN_OK channel 17
s → DATA    fd 1, " 02:31:07 up 41 days,  3:12,  2 users,  load average: 0.41, 0.38,
0.35\n"
s → EOF     fd 1
s → EOF     fd 2
s → CLOSE   no_error, exit 0
```

§4 Forwarding

One forward channel carries one TCP connection, one UDP flow or one Unix-socket connection. The client opens it to reach a target from the server's side; for the opposite direction, it first opens a listen channel.

> **Diagram: forward, listen and exec · who opens each channel.** Drawn in the PDF version, `06-exec-1.pdf`.

A local forward and an exec go out from the client; the way back in exists only as a listener the client opened, leading to the local service it chose.

Local forwarding. For each local connection, or SOCKS request, the client opens `forward` with `proto`, `to` (`host` and `port`, or `path` for a Unix socket) and, for the server's logs, `from`. The server connects and answers OPEN_OK once connected, or OPEN_REFUSED with a reason.

Remote forwarding. The client opens `listen` with `proto` and `bind`; port 0 means any free port, and OPEN_OK's `bound` gives the real address. For each connection that arrives, the server opens `forward` with an even ID, the listener's channel in `listen`, and the remote peer in `from`. The client connects to the local target it chose for that listener and answers OPEN_OK or OPEN_REFUSED. Closing the listen channel stops listening; connections already forwarded carry on.

Type	Frame	Fields	Purpose
0x10	DATA	`bytes`	Stream bytes, or exactly one UDP datagram
0x11	EOF	—	This side sends no more (TCP and Unix only)

- On QUIC and WebTransport, UDP flows travel as datagrams, one UDP datagram per DATA, like live media; elsewhere they ride the channel's stream. A DATA frame that does not fit `max_datagram` (default 1,200 bytes) goes on the channel's stream instead, so a large UDP datagram is never cut or dropped; every DATA that fits goes as a datagram. TCP and Unix data never use datagrams.
- A forward ends with CLOSE `no_error` once both sides sent EOF, or with reason `reset` when a connection was reset.
- Dynamic forwarding (SOCKS) is the client's business: it runs the SOCKS proxy and opens a forward per connection with the requested target. Nothing new crosses the wire.
- The client lists its active listeners and their local targets in its own controls, and the user can stop any of them.

§5 Ports in the scene

A `port` node offers a server-side port that the user opens with one click: an agent that starts a dev server puts one next to its message, and the user sees the app.

```
{"op":"open","id":"app","type":"port","parent":"dev","label":"Open the dev
server","port":3000,"scheme":"http","path":"/"}
```

Prop	Meaning
`label`	Required
`port`	1–65535
`host`	As the server reaches it; default `127.0.0.1`
`scheme`	`http` or `https`: open the forwarded port in the browser
`path`	Path to open; needs `scheme`
`url`	An https URL on the server's own origin that reaches the port, for clients that cannot listen

- A click opens a local listener on `127.0.0.1` at a free port, forwarding to `host`:`port` on the server's side. With `scheme`, the client then opens the browser at that local address and `path`. It needs `forward.tcp` from the server's policy.
- The click is the user's gesture: the server cannot start a forward by itself.
- A browser client cannot listen. It opens `url` when the server provides one, a proxy for this session only; otherwise the node shows its label as unavailable.

§6 Coming from SSH

Every everyday SSH feature has a VTX counterpart; the two left out are left out on purpose.

SSH	VTX with Exec 1
`ssh host cmd`	An exec channel with `argv` or `shell`; the exit status arrives in CLOSE
`ssh -t host top`	A normal session with `top` as its target, on a VT grid
`ssh -L 5432:db:5432 host`	One forward channel per connection, `to` db:5432
`ssh -R 9000:localhost:3000 host`	A listen channel bound to port 9000; one server-opened forward per connection; the client connects to localhost:3000
`ssh -D 1080 host`	The client's own SOCKS proxy, one forward channel per connection
`ssh -L /tmp/d.sock:/var/run/docker.sock host`	A forward channel with `proto: "unix"`
UDP, which SSH lacks	A forward channel with `proto: "udp"`, as datagrams on QUIC
`scp`, `sftp`	Transfer 1 for people; exec with `cat` or `tar` for scripts
`ssh -A` (agent forwarding), `ssh -X` (X11)	Left out: passkeys and smart cards sign in directly, and graphical applications run as VTX sessions

§7 Fallbacks and security

Nothing runs on the client, and the server reaches into the client's network only through a listener the client opened, to the target the client chose.

- The server's policy decides who may run what and reach where; every refusal carries a reason, and every command and forward is logged.

- Exec runs with the user's own rights, so it can do no more than the user could in a session. Servers SHOULD refuse environment variables that change how programs load, such as `LD_PRELOAD`.
- Listeners bind to loopback by default. The client lists them and the user can stop them at any time.
- A server-opened forward must name an open listener of the client's; anything else is a protocol error.
- Headless sessions end with the connection, so a script leaves nothing running behind.

Fallbacks. A server without Exec 1 never grants `exec` or `forward`, and a client without it never asks. A client without `forward` shows a port node's label as text; with `url` it may still open the server's proxy.

§8 Definitions and checks

The definitions live in `ext/exec-1/` of the bundle, and `tools/validate.py` runs 93 checks on them; every check in the bundle passes.

File	What it holds
`vtx-exec-1.cddl`	The capability value, OPEN params for exec, forward and listen, OPEN_OK for listeners, the frames, and the refusal reasons
`vtx-ops-exec-1.schema.json`, `vtx-app-exec-1.schema.json`	Every earlier op and application line, plus the port node
`examples/valid/`	A scene with port nodes; two wire traces: commands and a UDP flow over QUIC, and local and remote forwarding over TLS
`examples/invalid/`	8 broken port nodes and 31 broken traces, each labelled with the error it must raise
`tools/gen_ext_exec.py`	Generates the schemas

The 93 checks:

- 23: the schemas are valid, the CDDL parses, and each scene line passes its schema, the application schema and the scene rules;
- 3: the schemas without Exec 1 reject the port nodes;
- 8: each broken port node fails its own schema;
- 26: the CDDL accepts 15 samples and rejects 11;
- 33: a channel simulator replays both traces and catches every broken one for its labelled reason: who opens what, the server's grants, listeners named by server-opened forwards, endpoints, which side writes which fd, closed streams, the exit status after EOF, signals, datagrams for UDP only and within max_datagram, refusal reasons, bound ports, and v1 windows.

One more pycddl workaround. pycddl also mixes keys inside a field whose type is a choice of maps, such as `"to": endpoint`, and accepts any map for the empty map `{ }`. The validator checks such fields against their rule on its own, and requires empty maps to be empty; v1's `ctl-sessions` gains from the second fix too.

§9 Wire summary

Exec 1 adds entries to registries that take new entries by specification; it changes no v1 rule.

Registry or extension point	Exec 1 adds
Dynamic channel types	`exec`, `forward`, `listen`
Frame types	exec: 0x10 DATA, 0x11 EOF, 0x12 SIGNAL. forward: 0x10 DATA, 0x11 EOF. listen: none
Capability keys	`exec`, `forward`
OPEN_REFUSED reasons	`policy`, `not-found`, `unreachable`, `refused`, `timeout`, `in-use`
CLOSE fields	None new: `exit` and `signal` end an exec channel
Node types	`port`
Application interface kinds	None: exec and forwarding are started by the client