VTX Smart Cards 1: smart card readers in a session

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

§1 Status and scope

Smart Cards 1 is an extension of VTX 1, draft d01, in conformance profile 3. It lets programs in a session use the smart card readers of the user's device, as RDP's smart card redirection does: signing in to the hosted desktop with a card or a security key, signing documents and mail, and anything else that talks to a card through PC/SC.

It works at the level of PC/SC calls, not of USB or the reader's driver. The server turns each PC/SC call a program in the session makes into a request on the smartcard channel; the client makes the same call on the device's own PC/SC and sends back the answer. Three things follow from that choice:

- The device's PC/SC keeps arbitrating between the session and programs on the device, and a transaction keeps a session's commands to the card together, so neither side breaks the other's work.
- No USB device leaves the device and no reader driver is installed on the host: any reader the device's PC/SC knows works in the session.
- The client sees every command, so it can type PINs itself and keep payment applications out of reach (Sections 7 and 8).

What a session needs	How Smart Cards 1 gives it
Sign-in with a card	The host's sign-in reads the card through PC/SC, as on a local machine; the PIN is typed on the device
Signing and decrypting	Programs use their usual middleware (OpenSC, a PKCS#11 module, a Windows minidriver) over the shared readers
Locking when the card is pulled	READERS reports the removal at once; the host's own policy locks the session
Cards that need a PIN pad	Every reader can act as a PIN pad: the client asks for the PIN in its trusted zone

Out of scope. Signing in to VTX itself with a certificate stays with v1's sign-in methods; Smart Cards 1 serves programs inside the session. The FIDO side of a security key (WebAuthn over HID) is not a smart card and is not covered. Reader escape commands, direct connections to a reader without a card and the raw protocol are left out on purpose (Section 9). Redirecting other USB devices is a separate question for later.

§2 Capability, permission and channel

The client offers `redirect.smartcards` in HELLO, under the `redirect.` prefix of the redirections; the server grants it with `{}`. Once the user has agreed to share readers with this session, the client opens one dynamic channel of type `smartcard` (an odd ID, as for every channel the client opens) and lists the readers on it. The server never opens it, and a connection has at most one.

```
$permission-ext /= "smartcards"       ; the server asks the user to share card readers

cap-smartcards = {
  ? "max": 1..16,             ; client: most readers it lists (default 8)
  ? "pin": "device" / "any",  ; client: "device": PINs are typed on the device only
(default "any")
}
```

Field	Who	Meaning
`max`	client	The most readers READERS will list; 1 to 16, default 8
`pin`	client	`device`: every reader acts as a PIN pad and the client refuses commands that carry a PIN (Section 7). `any`: PINs may also come from the session, as in RDP. An organization sets `device` in its client policy

Asking. When a program in the session wants a card and no reader is shared, the server asks with the `smartcards` permission on the trusted zone channel, for example at the sign-in screen: “Sign in to your desktop with your security key.” The user can also share readers ahead of time in the client's settings, per session or always. Either way the user picks the readers, and the client lists only those.

Stopping. Closing the channel (v1's CLOSE) stops sharing every reader at once; before it closes, the client disconnects every handle the session holds with `reset`, so no card stays unlocked for the session (Section 10).

§3 Readers and cards

READERS lists the shared readers and the card in each. The client sends it when the channel opens and again after every change: a card put in or taken out, a reader plugged in or out, a card that a program on the device takes or releases.

```
sc-readers = {                                        ; 0x10 READERS, C->S
  "readers": [* sc-reader],
}
sc-reader = {
  "id": sc-reader-id,         ; chosen by the client for this connection; never a
serial number
  "name": tstr .size (1..128),  ; as the device's PC/SC names it
  ? "pinpad": true,           ; the client asks for PINs itself (Section 7)
  ? "contactless": true,
  ? "payment": true,          ; the user lets sessions select payment applications here
(Section 8)
  ? "card": sc-card,          ; absent: the reader is empty
}
sc-card = {
  "inserted": uint,           ; cards put in this reader since the channel opened, the
one already there counted
  ? "atr": sc-atr,            ; 2 to 33 bytes; absent only for a mute card
  ? "mute": true,             ; the card does not answer a reset
  ? "inuse": true,            ; a program on the device holds it exclusively
}
```

A laptop with a YubiKey in it and a contactless reader on the desk:

```
READERS {"readers": [
  {"id": "rd-1", "name": "Yubico YubiKey OTP+FIDO+CCID", "pinpad": true,
   "card": {"inserted": 1, "atr": h'3BFD1300008131FE158073C021C057597562694B657940'}},
  {"id": "rd-2", "name": "Identiv uTrust 3700 F CL Reader", "pinpad": true,
"contactless": true}]}
```

The insertion count. PC/SC programs tell a new card from the old one by an event counter, and a card can be pulled and put back between two looks. `inserted` is that counter: it starts at 1 for a card that is already in the reader when the channel opens, never goes down, and grows every time a card is put in. A handle keeps to the card it connected to; once the reader shows no card, or a card with a higher count, that handle's card is gone (Section 5).

Rules. Reader ids and names are unique; PC/SC itself requires unique names. An ATR starts with 3B or 3F, the two conventions ISO/IEC 7816-3 allows. A card has an ATR unless it is mute, and a mute card has none. With `"pin": "device"` every reader is a PIN pad. At most `max` readers are listed.

Which readers. The client lists only the readers the user shared: USB readers, readers built into the laptop or keyboard, security keys that speak CCID, and contactless readers. Ids are made up for each connection, so a session cannot recognize a reader across connections. The ATR is what a card's middleware needs to recognize the card, and a session only sees it for cards in readers the user shared.

§4 At a glance

> **Diagram: the PIN path · a signature with the PIN typed on the device.** Drawn in the PDF version, `16-smart-cards-1.pdf`.

A program in the session asks its PC/SC to verify a PIN on a PIN pad reader. The server sends the client a PIN request that holds the command with an empty PIN block; the client asks Marcel in its trusted zone, puts the PIN in the command, sends it to the card through the laptop's own PC/SC and returns only the card's answer. Without a PIN, the same path carries every other command: CONNECT, BEGIN, TRANSMIT and END go down, RESULT or ERROR comes back.

§5 Connecting, sharing and transactions

Every request carries a number, `req`, and each PC/SC call in the session becomes one request. A connection to a card is a handle the client hands out in its answer to CONNECT.

```
sc-connect = {                                          ; 0x11 CONNECT
  "req": sc-req,
  "reader": sc-reader-id,
  "share": "exclusive" / "shared",
  "protocols": [1*2 ("t0" / "t1")],
}
sc-reconnect = {                                        ; 0x12 RECONNECT: after a reset,
or to change the sharing
  "req": sc-req,
  "handle": sc-handle,
  "share": "exclusive" / "shared",
  "protocols": [1*2 ("t0" / "t1")],
  "init": "leave" / "reset" / "unpower",
}
sc-disconnect = { "req": sc-req, "handle": sc-handle, "disposition": sc-
disposition }   ; 0x13 DISCONNECT
sc-begin = { "req": sc-req, "handle": sc-handle }                                ;
0x14 BEGIN: a transaction
sc-end = { "req": sc-req, "handle": sc-handle, "disposition": sc-disposition }       ;
0x15 END
sc-disposition = "leave" / "reset" / "unpower" / "eject"
```

Request	PC/SC call	RESULT carries
CONNECT	SCardConnect	`handle` and `protocol`, one of those asked for
RECONNECT	SCardReconnect	`protocol`
DISCONNECT	SCardDisconnect	nothing; the handle ends
BEGIN	SCardBeginTransaction	nothing; the transaction is open
END	SCardEndTransaction	nothing; the transaction is closed
TRANSMIT	SCardTransmit	`response` (Section 6)
PIN	SCardControl with a PIN pad feature	`response` (Section 7)

Handles. A handle exists from the RESULT of its CONNECT until its DISCONNECT, which ends it whether it succeeds or not. The server sends no request on a handle it was never given or that has ended. A handle keeps to the card that was in the reader when it connected: once READERS shows the reader empty, or holding a card with a higher insertion count, the client answers every request on that handle with `removed`, until the server sends DISCONNECT.

Sharing. On one reader the session holds either a single exclusive handle or any number of shared ones, as PC/SC itself allows. The client answers a CONNECT or RECONNECT that would break this with `sharing`, and does the same when a program on the device holds the card exclusively (`inuse` in READERS). A CONNECT to an empty reader gets `nocard`, to a mute card `mute`, to a reader no longer listed `reader`.

Transactions. BEGIN opens a transaction on a handle: until END, no program on the device and no other handle can send the card a command, so the selected application and a verified PIN stay as the session left them. The server sends END only inside a transaction it opened on that handle, and BEGIN only outside one. RECONNECT and DISCONNECT close an open transaction. A BEGIN waits while a program on the device holds a transaction, so its answer may take a while.

Resets. When a program on the device resets the card, the next request on the handle gets `reset`, as SCARD_W_RESET_CARD in PC/SC. From then on the client answers only RECONNECT or DISCONNECT on that handle; everything else gets `reset` again until a RECONNECT succeeds. The card's application selection and PIN state are gone after a reset, so the program starts over.

§6 Commands and answers

TRANSMIT carries one command APDU to the card, as ISO/IEC 7816-4 defines it: four header bytes, then the data with its length and the expected answer length, short or extended. The client sends it unchanged and returns the card's answer unchanged, data first and the two status bytes SW1 SW2 last. Answers that come in parts (`61 xx`, fetched with GET RESPONSE) are the program's business, as on a local reader.

```
sc-transmit = { "req": sc-req, "handle": sc-handle, "apdu": sc-command }   ; 0x16
TRANSMIT
sc-command = bstr .size (4..65544)   ; header, Lc, data and Le; extended APDUs up to
65,544 bytes
sc-response = bstr .size (2..65538)  ; data, then SW1 SW2

sc-result = {                                        ; 0x20 RESULT
  "req": sc-req,
  ? "handle": sc-handle,      ; CONNECT
  ? "protocol": "t0" / "t1",  ; CONNECT, RECONNECT: one of those asked for
  ? "response": sc-response,  ; TRANSMIT, PIN: the card's answer
}
sc-error = { "req": sc-req, "code": ..., ? "message": tstr }   ; 0x21 ERROR
```

Requests and answers. Only the server sends requests, with numbers that grow, and at most 16 wait for an answer at once; a PC/SC context makes one call at a time, so 16 leaves room for several programs. The client answers every request exactly once, with RESULT or ERROR, in any order. A RESULT holds exactly the fields its request calls for (the table in Section 5) and nothing else; a card's error, such as `6A 82` for an application it does not have, is a RESULT, because the card did answer.

ERROR is for calls that did not reach the card or were refused. Each code maps to the PC/SC result the host gives the program:

Code	When	PC/SC result in the session
`reader`	The reader is no longer listed	SCARD_E_UNKNOWN_READER
`nocard`	The reader is empty	SCARD_E_NO_SMARTCARD
`removed`	The handle's card was taken out	SCARD_W_REMOVED_CARD
`reset`	A program on the device reset the card	SCARD_W_RESET_CARD
`sharing`	Held exclusively, or shared where exclusive was asked	SCARD_E_SHARING_VIOLATION
`protocol`	None of the protocols asked for	SCARD_E_PROTO_MISMATCH
`mute`	The card does not answer	SCARD_W_UNRESPONSIVE_CARD
`canceled`	The user closed the PIN prompt	SCARD_W_CANCELLED_BY_USER
`timeout`	The user did not type the PIN in time	SCARD_E_TIMEOUT

Code	When	PC/SC result in the session
`denied`	The client's rules refuse it (Sections 7 and 8)	SCARD_W_SECURITY_VIOLATION
`comm`	The reader failed	SCARD_E_COMM_DATA_LOST

`message` is for logs and is never shown as the user's own words.

§7 PINs on the device

A reader marked `pinpad` behaves like a reader with its own keypad: the program in the session hands over the command with an empty PIN block, and the PIN is typed where the card is. PC/SC programs already know how to do this: they ask for the PIN pad features of PC/SC part 10 (FEATURE_VERIFY_PIN_DIRECT and FEATURE_MODIFY_PIN_DIRECT), which OpenSC and most middleware use. The host turns those calls into PIN requests.

```
sc-pin = {                                          ; 0x17 PIN, S->C
  "req": sc-req,
  "handle": sc-handle,
  "action": "verify" / "change",
  "apdu": sc-command,          ; the command, each PIN block holding its padding
  "format": "ascii" / "bcd" / "iso9564-2",
  "offset": uint,              ; where the PIN block starts in the command's data
  "block": 1..16,              ; bytes in one PIN block; a change has the new block
right after the old one
  "min": 1..16,                ; digits or characters
  "max": 1..16,
  ? "label": tstr .size (1..64),  ; what the user is asked for, e.g. "PIV PIN"
  ? "timeout": 5..300,         ; seconds the user has (default 30)
}
```

Format	How the client fills the block	Most digits
`ascii`	One byte per character from the start of the block; the rest keeps the template's padding (PIV pads with FF)	`block`
`bcd`	Two digits per byte from the start of the block; an odd last digit takes F in the low half	2 × `block`
`iso9564-2`	ISO 9564 format 2: 2 and the PIN's length in the first byte, then the digits, then F; the block is 8 bytes	12

The PIV application of a YubiKey, for example, verifies its PIN with `00 20 00 80 08 FF FF FF FF FF FF FF FF`, `ascii`, offset 0, block 8, 6 to 8 characters. A change carries two blocks, the old PIN and then the new one; the client asks for the new one twice.

Template rules. The command is a well-formed APDU; the PIN block, or both blocks for a change, lie inside its data; `min` is at most `max`; `max` fits the block in its format. The server sends PIN only for a handle on a reader marked `pinpad`.

The prompt. The client asks in its trusted zone, where the session cannot draw. The prompt names the session, the reader and the `label`, and after a wrong PIN it shows the tries the card has left (`63 Cx` in its answer). The client fills the template, sends the command through the device's PC/SC and returns the card's answer as the RESULT's `response`; it then wipes the PIN from memory. Closing the prompt gives `canceled`, letting `timeout` pass gives `timeout`. While the device is locked no prompt is shown and the request waits.

Device-only PINs. With `"pin": "device"` every reader is a PIN pad, and the client refuses with `denied` any TRANSMIT whose command carries a PIN in its data: VERIFY (INS 20 or 21), CHANGE REFERENCE DATA (24) and RESET RETRY COUNTER (2C). The same commands without data, which only ask how many tries are left, go through. A PIN can then reach the card only through the prompt, and a host that was broken into cannot learn it. Organizations that sign in with cards should set this policy once their middleware uses PIN pad readers; with `"pin": "any"`, programs may also send the PIN themselves, as with RDP.

§8 Payment cards

A contactless reader on the desk reads any card laid on it, and bank cards end up there. A session that can talk to a bank card's payment application could relay it to a payment terminal somewhere else and pay with it, while the card lies on the user's desk. No program in a hosted desktop needs that, so the client keeps payment applications out of the session's reach.

The rule. On a reader without `payment`, the client answers with `denied` every SELECT by name (INS A4, P1 04) whose name can reach a payment application: the payment system environments `1PAY.SYS.DDF01` and `2PAY.SYS.DDF01`, any name that starts with the registered application provider identifier (RID) of a card scheme, and any name that is the start of one, since a card selects the first application whose name begins with the bytes given. Every EMV payment starts with one of these SELECT commands, so refusing them stops a relay. The rest of the card stays reachable: a session can still find out that a card is there and try the applications it knows, which on a bank card fail with `6A 82`.

RID	Scheme
A0 00 00 00 03	Visa, V PAY
A0 00 00 00 04	Mastercard, Maestro
A0 00 00 00 25	American Express
A0 00 00 00 42	Cartes Bancaires
A0 00 00 00 65	JCB
A0 00 00 01 52	Discover, Diners Club
A0 00 00 02 77	Interac
A0 00 00 03 33	UnionPay
A0 00 00 03 59	girocard
A0 00 00 05 24	RuPay
A0 00 00 06 58	Mir

The list may grow with new drafts, and a client may refuse more schemes than it names. A client checks every command it passes, so the list lives on the device, not on the host.

The exception. The user can allow payment applications on one reader in the client's settings, for the rare program that has to read a payment card; READERS then marks that reader `payment`. The client never sets it on its own.

§9 The session host and the application interface

Programs in the session see the shared readers as ordinary PC/SC readers and need no change. On a Linux host the server runs a PC/SC service for each session that the session's pcsc-lite library talks to, as xrdp and FreeRDP do; the readers appear under their own names with `(via VTX)` added. On a Windows host the server answers the session's WinSCard calls, as RDP's smart card redirection does. OpenSC, p11-kit, the sign-in through PAM and SSSD, browsers, mail clients and LibreOffice then work as they would with a local reader.

What the host answers itself. Some PC/SC calls need nothing from the device:

PC/SC call	Answered from
SCardListReaders	The last READERS
SCardGetStatusChange	READERS as it arrives; the insertion count becomes the event counter in the high 16 bits of the reader state
SCardStatus, SCardGetAttrib	The handle, the reader's name and the card's ATR and protocol
SCardControl with GET_FEATURE_REQUEST	FEATURE_VERIFY_PIN_DIRECT and FEATURE_MODIFY_PIN_DIRECT for a `pinpad` reader, nothing for others
SCardCancel, contexts	The host's own bookkeeping

What never reaches the device. SCardControl with any other code (reader escape commands, firmware updates, LED and buzzer control), direct connections to a reader without a card, and the raw protocol. The host answers these with the PC/SC error a reader gives when it does not support them. A reader's vendor commands stay between the device and its reader.

Application events. The server tells the session manager about readers and cards with the `smartcard` event, so it can, for example, lock the session when the card used to sign in is pulled, as SSSD and Windows' smart card removal policy do:

`action`	When	Fields
`added`	A reader appeared in the session	`id`, `name`, `reader` (its PC/SC name on the host)
`removed`	A reader went away	`id`
`inserted`	A card was put in	`id`, `atr` in lowercase hex
`withdrawn`	The card was taken out	`id`

```
{"kind":"smartcard","action":"added","id":"rd-1","name":"Yubico YubiKey
OTP+FIDO+CCID","reader":"Yubico YubiKey OTP+FIDO+CCID (via VTX) 00 00"}
{"kind":"smartcard","action":"inserted","id":"rd-1","atr":"3bfd1300008131fe158073c021c057597562694b657
{"kind":"smartcard","action":"withdrawn","id":"rd-1"}
```

Smart Cards 1 adds no scene op and no application line: programs reach the cards through PC/SC, not through the application interface.

§10 Security and privacy

A smart card holds keys that sign in, sign documents and decrypt mail in the user's name, so sharing a reader is sharing those keys with the session. Smart Cards 1 keeps the user aware of that and limits what a session can do with it:

- **The user decides.** Only readers the user shared are listed. The trusted zone shows every shared reader, lights up while the session is using a card, and stops sharing all of them with one click.
- **PINs on the device.** PIN prompts appear only in the trusted zone and name the session asking. With `"pin": "device"` no PIN ever reaches the host (Section 7).
- **Nothing unattended.** While the device is locked, the client holds every request until it is unlocked and shows no PIN prompt, so no program can use the card while the user is away from the device.
- **No verified card left behind.** When the channel closes or the connection drops, the client disconnects every handle the session holds with `reset`. The card forgets its verified PIN, and a session that has gone, or a host that is taken over afterwards, finds it locked again.
- **No payment.** Payment applications stay out of reach unless the user allows them on a reader (Section 8).
- **Only what PC/SC shows.** Reader ids are made up for each connection; names and ATRs are what any local program sees, for readers the user shared. Reader escape commands never leave the device (Section 9).

What remains. As on a local machine, a card stays verified after the PIN until it is reset or pulled, and a program in the session can use it in that time without asking again. Cards can require the PIN for each use of a key: the PIV signing key does, which is why the trace asks for the PIN again before signing the invoice. Organizations that care about this should use keys with that policy, set `"pin": "device"`, and lock sessions when the card is pulled.

§11 Definitions and checks

The definitions live in the bundle under `ext/smartcards-1/`:

File	What it holds
`vtx-smartcards-1.cddl`	The `redirect.smartcards` capability, the `smartcards` permission and the smartcard channel's frames
`vtx-app-events-smartcards-1.schema.json`	Server lines to an application, with the `smartcard` event; generated
`examples/valid/trace-signin-quic.jsonl`	The trace below, 61 frames
`examples/valid/app-smartcards.jsonl`	Six `smartcard` events

File	What it holds
`examples/invalid/readers-rules.jsonl`	Five reader lists that pass the CDDL but break a rule
`examples/invalid/traces.jsonl`	The trace broken 22 ways, one at a time
`examples/invalid/app-schema.jsonl`	Two events the schema rejects

`tools/gen_ext_smartcards.py` writes the schema, built on the Media Devices 1 generator, and `tools/make_smartcards.py` writes the examples. Certificates, challenges and signatures in them are made-up bytes from fixed seeds, so every run writes the same files.

The trace. Marcel signs in to his hosted desktop with the PIV application of his YubiKey: the host reads the authentication certificate in three parts, he types the PIN on his laptop, and the key signs the sign-in challenge. Later he signs an invoice in LibreOffice; a program on the laptop has reset the key meanwhile, so the first BEGIN gets `reset` and the host reconnects. His first PIN is wrong (`63 C2`, two tries left), the second is right, and the signing key signs. A bank card laid on the contactless reader is probed: the PIV application is not there, and the client refuses the payment environment. He pulls the YubiKey while the lock screen is checking it; the waiting command gets `removed` and the session locks.

The validator's Section 24 runs 61 checks, and every one passes:

Part	Checks
The schema, the CDDL alone and with every earlier extension, no rule defined twice, earlier schemas and the core rejecting the additions	6
CDDL samples, 9 valid and 10 malformed, and the 5 broken reader lists	24
The trace, and each broken trace failing for its labelled reason	23
Application events, 6 valid and 2 rejected	8

Label	What a broken trace does
`caps`, `direction`, `channel`	A channel the server did not grant; the server opening it; the client sending CONNECT; a second channel; a CONNECT after CLOSE
`readers`	Two readers with one id; a reader that is no PIN pad where PINs are typed on the device only; an insertion count going down
`card`, `sharing`, `handle`	A handle for an empty reader; an exclusive handle beside a shared one; a handle never given; a handle used after DISCONNECT
`result`, `transaction`, `limit`	A protocol not asked for; one request answered twice; TRANSMIT answered with a handle; END without BEGIN; 17 requests waiting
`pin`, `payment`	A PIN sent in TRANSMIT and answered; a PIN block running past the data; a payment application selected
`reset`, `removed`	An answer after a reset, before RECONNECT; an answer after the card was pulled

Section 17, the extensions together and the registry, gained three checks for the new channel type. The whole bundle runs 2,543 checks, all passing.

§12 Wire summary

Everything Smart Cards 1 adds to the wire is on its own channel type, `smartcard`, which only the client opens (an odd ID), at most once per connection:

Type	Frame	Direction	Purpose
0x10	READERS	C→S	The shared readers and their cards; on open and after every change
0x11	CONNECT	S→C	Connect to the card in a reader
0x12	RECONNECT	S→C	After a reset, or to change the sharing
0x13	DISCONNECT	S→C	End a handle
0x14	BEGIN	S→C	Open a transaction
0x15	END	S→C	Close it
0x16	TRANSMIT	S→C	One command APDU
0x17	PIN	S→C	A command whose PIN the user types on the device
0x20	RESULT	C→S	The answer to a request
0x21	ERROR	C→S	A call that did not reach the card, or was refused

Besides the channel, it adds the `redirect.smartcards` capability and the `smartcards` permission. It adds no frame on v1's well-known channels, no close code and no datagram.