

Testing and debugging in the Applet Lab

What the service can do, how to exercise every part of it, and how to set a breakpoint on a line of Java and stop the card on it.

1. What the service can and cannot do

Measured against the lab's own requirements, not a wish list.

CAPABILITY	STATE	WHAT THAT MEANS IN PRACTICE
Run real CAP bytecode	Full	A converter output runs instruction by instruction. Not a model of an applet.
GlobalPlatform install	Full	INSTALL [for load], LOAD blocks, INSTALL [for install and make selectable].
Proactive commands	Full	DISPLAY TEXT, GET INPUT, GET INKEY, SELECT ITEM, SEND SHORT MESSAGE, SEND USSD and the rest, through the generic <code>init / appendTLV / send</code> path. Gated on TERMINAL PROFILE, as a card is.
Toolkit events	Full	22 event-download codes, menu selection and timer expiration, routed by the export file's own numbers.
Class 2 SMS to an applet TAR	Full	The TAR is declared in the INSTALL parameters and the card routes by it.
Card file system	Full	<code>uicc.access.FileView</code> over a real UICC file tree, with real access conditions.
Source-line debugging	Full	Gutter breakpoints, per-instruction trace with source positions, stepping forwards and backwards.
S@T browser	Partial	A deck is parsed and its STK generic macros are read, which is the structure and the command. The other twelve bytecodes are refused by name rather than skipped, only the 8-bit text coding is decoded, and the commands are <i>listed</i> rather than issued through the terminal — so an S@T page cannot yet be refused, answered, or register a menu entry. The command catalogue marks each bytecode.
OTA security	Full	SPI policy, replay counters and Proof of Receipt work, and the PoR is decoded into the log — a Short Message answers <code>9000</code> when the card accepted the <i>envelope</i> , which says nothing about the script inside it. The card now holds a key set , so a cryptographic checksum is really computed and really verified, and a ciphered packet is really deciphered. It is derived from the session's own identifier: nothing is stored, and the same identifier yields the same octets on every replay, which is the one arrangement that survives a card being rebuilt from its history. The octets are never shown , and do not need to be — the lab builds the packet on the same side as the card that checks it, so what a reader needs is the SPI, the counter and the refusal. Nothing derived that way is secret in any real sense, which is exactly the distinction worth stating. The Security panel carries the Minimum Security Level, which is the half an attacker cannot write: a card that

CAPABILITY	STATE	WHAT THAT MEANS IN PRACTICE
		only verifies what the packet asked to be verified accepts a packet whose SPI says "no checksum", perfectly, with no error anywhere. Raising the floor re-judges the whole history, because the history is replayed against it.
On-card cryptography	Partial	<code>javacard.security</code> and <code>javacardx.crypto</code> are on the card, at the Java Card 3.0.4 API, and measured rather than claimed: upstream JCAIlgTest, unmodified, installed over GlobalPlatform and driven by its own client, reports 289 supported, 8,319 refused and no errors. What to add was chosen from JCAIlgTest's published results for 152 real cards: of the 236 algorithms that at least 20% of those cards support, this card has 235. <i>Keys</i>: DES and Triple DES, AES-128/192/256, Korean SEED, HMAC, RSA 512–4096 plain and CRT, and EC over prime fields from 112 to 521 bits, with default curves (P-192 to P-521, secp112/128/160r1, brainpoolP320r1 and P512r1) when the applet sets none. <i>KeyPair</i> generates EC pairs and RSA pairs up to 3072 bits. <i>Cipher</i>: DES, 3DES and AES in ECB and CBC with no padding, ISO 9797 M1 and M2, or PKCS#5; AES-CTR; SEED ECB and CBC; RSA with no padding, PKCS#1 v1.5 and OAEP over SHA-1 and SHA-2. <i>Signature</i>: DES and retail MAC, AES CBC-MAC, AES-CMAC, SEED MAC, HMAC over every digest, RSA PKCS#1 v1.5 and PSS, ISO 9796-2 with and without message recovery (<code>SignatureMessageRecovery</code>), and ECDSA with SHA-1 and SHA-2. <i>The rest</i>: SHA-1, SHA-2 and SHA-3 digests, MD5, RIPEMD-160, ECDH key agreement (<code>SVDP_DH</code> , <code>DHC</code> , plain, plain XY), ISO 3309 CRC16 and CRC32, and random data. Each is checked against an independent implementation or published test vectors. Randomness is reproducible on purpose. Key generation draws from a counter and ECDSA uses RFC 6979's deterministic k, because a card here is rebuilt by replaying its history. A key generated here is a teaching key, never a secret. Still refused: AES-GCM and AES-CCM, DSA, EC over binary fields, finite-field DH, <code>ALG_EC_PACE_GM</code> (specific to PACE, and left out rather than guessed), XDH, and SM2, SM3 and SM4. A refusal is a <code>CryptoException</code> with reason <code>NO_SUCH_ALGORITHM</code> , which an applet catches with a typed <code>catch</code> or with <code>catch (Exception)</code> — the way a portable applet probes a card for what it supports. It now survives past the first command, which it did not: a session stored after any command was rebuilt from four of its fields, so <code>javacard.security</code> and <code>javacardx.crypto</code> were silently deleted from the card the moment anything happened to it. The applet had already compiled against them, so the failure arrived later and looked like the applet's fault.
Legacy <code>sim.toolkit</code>	Partial	GSM 03.19 / TS 43.019 runs: <code>ToolkitInterface</code> , <code>ToolkitRegistry</code> , <code>ProactiveHandler</code> , <code>ProactiveResponseHandler</code> , <code>EnvelopeHandler</code> and <code>ToolkitConstants</code> . An applet written against the SIM API compiles, converts, installs, registers a menu entry and is triggered through <code>processToolkit(byte)</code>

CAPABILITY	STATE	WHAT THAT MEANS IN PRACTICE
		— a different method from the UICC API's <code>processToolkit(short)</code> , so an applet implements one interface or the other. Both kinds share one registry and one menu, because both are doors into the same CAT Runtime Environment. The event numbers differ and are translated by name against ETSI's own export file, so no UICC event number is written down twice. The numbers in the sim.toolkit export file are ours. 3GPP's export files are not in this workspace, so the API's <i>shape</i> is faithful — the class names, the method names, the byte-typed events — and its tokens are this project's inventions. A CAP converted against 3GPP's real export file will not run here, and one converted here will not run on a real card. That is the same bargain <code>javacard.framework</code> strikes, whose file is Oracle's and cannot be redistributed either, and it is stated for the same reason. There is no <code>sim.access.SIMView</code> , deliberately. The SIM API has no file access at all, which is why a 2G-era applet asks the terminal for its cell instead of reading <code>EF_LOCI</code> — and a <code>SIMView</code> quietly forwarding to <code>uicc.access</code> would teach that the two APIs are interchangeable when the point is that they are not. GSM 03.19's SMS-PP and SMS-CB envelope events are also absent: a formatted Short Message reaches an applet by TAR through the Remote Management path, which this card implements, and routing one by event as well would deliver it twice by two mechanisms. The Legacy SIM API sample is a worked example.
BIP	Partial	An applet can OPEN CHANNEL, SEND DATA, RECEIVE DATA, GET CHANNEL STATUS and CLOSE CHANNEL, and the terminal answers all five properly: a granted buffer that may be <i>smaller</i> than the one asked for, seven channels and a refusal on the eighth, and the right BIP problem when a channel identifier is not valid. It reaches no network and never will. A session here is rebuilt by replaying its history, and a live socket is the one thing that cannot survive that. The destination is parsed and shown in the trace, then ignored. (The same card runtime <i>does</i> reach a network elsewhere in RSP Lab: for remote management over HTTPS, the OTA console's lab cards open a BIP channel to the OTA platform and run a real TLS-PSK session over it — see <i>Remote management over HTTPS</i> below. That channel belongs to the card's security domain, not to an applet, and is not offered here.) What there is instead is a far end you control, and it is three things rather than one. <i>Echo</i> is the original loopback: what SEND DATA writes is what RECEIVE DATA reads, which demonstrates the API and tests nothing, because an applet's parser is only ever fed its own encoder's output. <i>Silent</i> is an open channel with nothing waiting — what a server that has not answered yet looks like, and the case that finds an applet treating "no data" as "the connection failed". <i>Scripted</i> answers each SEND DATA with the next reply from a list, so a parser can be handed a reply it did not write, including a malformed one. The list travels in the session's history, so the same replies arrive

CAPABILITY	STATE	WHAT THAT MEANS IN PRACTICE
		in the same order on every rebuild. One reply at a time can still be delivered by hand.
CAT_TP	Partial	The state machine, the PDUs, fragmentation and reassembly are all here and reachable from the CAT_TP panel: listen, dial, deliver a PDU, send a payload, keep alive, close. The lab now plays the far end. Attach it and a handshake completes in one click instead of three: the peer is the same automaton with its ports reversed, because CAT_TP is symmetric and a second implementation of one protocol disagrees with the first the moment either is fixed. It is exactly as deterministic as pasting PDUs by hand, which is what lets a stateful transport live in a session rebuilt by replay — it is a state machine, not a socket. The far end can also originate a payload, which an echo can never do. Pasting a PDU still works and is still the only way to hand a card something no correct peer would send. Detaching leaves the connection open with nobody answering, which is a real situation worth being able to produce. A card still cannot originate a real administration session against a live platform, because that needs ISD keys this deployment does not hold.
Remote management over HTTPS	Not here	SCP81 (GlobalPlatform Amendment B) is not part of the Applet Lab: its card has no PSK and no route out, and remote management here arrives by SMS (SCP80) through the Channel panel. It is live in the OTA console instead, on a provisioned lab card. Provisioning gives that card an SCP81 PSK (KVN 40 , KID 01). An Amendment B trigger arrives as an ordinary secured packet, the card opens BIP to the platform the trigger names, completes a TLS-PSK handshake, and runs the RFM or RAM script it is handed over HTTP inside that TLS session. The console's <i>Demos</i> show reading EF_ICCID over HTTPS and installing an applet over HTTPS next to the same install over SMS. A card provisioned before the PSK existed has to be provisioned again.
Contactless	Partial	A reader's field raises EVT_CONNECTIVITY on the front-end's connectivity gate, the terminal turns it into the CAT "HCI connectivity event", and an applet registered for that event is triggered — the chain TS 102 622 clause 11.2.2.1 describes, end to end. The HCP framing and the gate, pipe and event identifiers are faithful. A reader can now talk, not only wave. An APDU sent over the field arrives on the APDU gate '30' (clause 12.2) and reaches the card as an ordinary command APDU — an applet's process cannot tell which interface it came in on, which is the lesson. Which applet it reaches is simplified: on a real card that is GlobalPlatform's Contactless Registry Service and the implicit selection parameters an applet was installed with, and neither is modelled, so it goes to whatever is selected. SWP is not modelled and will not be: it is a physical layer,

CAPABILITY	STATE	WHAT THAT MEANS IN PRACTICE
		one wire with S1/S2 signalling, and an emulator cannot do physical. Only the ARRIVING edge of a field is an event, so an applet cannot tell how long a tap lasted.
SMPP	Encoder, decoder and a local SMSC	The Channel panel builds the <code>submit_sm</code> an OTA platform would send, to SMPP v3.4 clause 4.4.1, and now reads one back: paste a PDU from your own tooling and it is decoded field by field, with the two fields that decide delivery named. A decoder matters more than an encoder for learning, because an encoder only ever shows you a PDU you already described. The loop closes without a socket. What an SMSC does between an ESME and a handset is move fields from one structure into another — a transformation, not a network — so the lab can be the SMSC. <i>Deliver through the SMSC</i> takes the <code>short_message</code> out of the PDU, verbatim, puts it in an SMS-DELIVER and hands it to the card. Change the destination address, deliver again, and watch the same applet answer: the address chose a <i>handset</i>, and the TAR four layers down chose the applet. The other direction works too — an applet's SEND SHORT MESSAGE becomes the <code>deliver_sm</code> an SMSC would push to an ESME, with the protocol identifier, the data coding and the destination read out of the applet's own SMS-SUBMIT rather than invented. Still no socket and no credentials. A transient bind/submit/unbind would mean storing an SMSC's credentials to gain a delivery path this lab does not need.
ADF USIM files	Partial	The card carries an ADF USIM: EF_IMSI, EF_LOCI, EF_EPSLOCI, EF_Keys, EF_KeysPS, EF_UST, EF_AD, EF_SPN, EF_ACC, EF_HPPLMN, EF_FPLMN and EF_ARR, with the identifiers and sizes 3GPP TS 31.102 clause 4.2 gives them. Their numbers are not in <code>uicc.access.UICCConstants</code>, which is ETSI's list, so an applet selects them by number. READ is behind PIN1 on almost all of it — that refusal is the lesson, and now so is the other half of it: the Files panel shows the tree, its access conditions and the PIN, and can present PIN1 so the same call succeeds. It sends no digits, because nothing in the exercise depends on which four the card was personalised with and a page that shipped them would put a PIN value in a repository. Typing digits does an ordinary VERIFY, which is how a wrong one and the attempt counter behind it are reached. EF_KEYS and EF_KEYSPS are never printed, empty or not — a habit that is harmless on this card is the habit that gets applied to one where it is not. Contents are sparse on purpose: a file pre-filled with invented network data would teach you to trust numbers this lab made up. The IMSI is the reserved test network, MCC 001 / MNC 01.
Timing fidelity	Counts, never	No clock cycles are modelled and none ever will be. An instruction here is a step in a Java interpreter on a Lambda, and the time it takes has no relationship

CAPABILITY	STATE	WHAT THAT MEANS IN PRACTICE
	time	to the time the same instruction takes on a card — where an EEPROM write dwarfs everything else and none of that is here. Draw no performance conclusion from a clock. What every command now reports is the number of instructions it executed, and that number <i>is</i> faithful, because the bytecode is real: an applet that executes 4,000 instructions where another executes 400 is doing ten times the work on a card too. The comparison is sound; turning it into milliseconds is not, and nothing here does.

It is not a conformance tool. Both ends of every exchange here are ours, so agreement proves consistency, not correctness. Evidence of interoperability comes only from real cards and real terminals.

2. First run, end to end

Ten minutes, and it exercises every layer.

1. **Sign in** at `appletlab.rsplab.click` with an RSP Lab account. The editor opens on a starter applet called `wallet`.
2. **Build & install.** The service compiles with a real `javac`, converts to a CAP, creates a card, loads the package, installs the applet and selects it. The Card log shows each APDU in both directions. Expect `9000` throughout and `installed` beside the button.
3. **Send it an APDU.** Type `00 10 00 00 01 07` into the box and press Send. The applet adds 7 to its balance and answers `0007 9000`. Send it again: `000E`. The field survived, because the card is rebuilt from its history on every request and the arithmetic is done again each time.
4. **Open the menu.** Press `≡` on the phone. Two entries appear — they came from `initMenuEntry` in the applet's constructor, which ran during INSTALL. Press one: the terminal sends an ENVELOPE (MENU SELECTION) and the applet is triggered while nothing is selected, which is what a toolkit applet is for.
5. **Send the TERMINAL PROFILE.** *CAT* panel → *Send TERMINAL PROFILE*. Nothing proactive works until this arrives, and that is deliberate — see Traps.
6. **Move the phone.** Change MCC/MNC in the Network panel and press *Move & send EVENT DOWNLOAD*. The starter applet registered for `EVENT_EVENT_DOWNLOAD_LOCATION_STATUS`, so it is triggered and rewrites a menu entry with the new operator. Delete that `setEvent` line, rebuild, and the move goes unnoticed — which is the card behaving correctly, not the lab failing.
7. **Install over the air.** Tick *over the air* and rebuild. The same INSTALL now arrives inside a class 2 Short Message addressed to the Issuer Security Domain, whose TAR is `00 00 00`. The card manager cannot tell the difference: the remoteness is entirely in the transport.

8. **Open a sample.** Press *Samples* and pick one. Each carries its own instructions in the Card log.

Where everything lives

The card pane is three regions: the phone, which is the output; the tabbed **deck**, which is the input; and the Card console, pinned below, which never scrolls away. The tabs are named for what ETSI calls them rather than for what is convenient — which matters most for the first two, because TS 102 223 uses TERMINAL for the handset:

TAB	WHAT IT IS
Network	the PLMN and cell — the Location Information object. Radio, MCC, MNC, LAC/TAC, Cell ID/ECI/NCI, service state, signal.
Terminal	what the terminal reports about <i>itself</i> , which is what PROVIDE LOCAL INFORMATION returns: IMEI, clock and time zone, measurements.
CAT	Card Application Toolkit messages the terminal sends the card: ENVELOPE, and the TERMINAL PROFILE that unlocks everything proactive.
S@T	a page of bytecode pushed to the card's browser.
Channel	how an OTA message is addressed and carried, every layer shown — and which layer carries what is the point. It also reads a PDU back, and delivers one to the card through the lab's own SMSC.
Files	the card's file tree, its access conditions and its PINs. Present PIN1 here and a refused READ starts working.
Security	what the card holds and what it insists on: the key set, the Minimum Security Level, and where each TAR's replay counter has reached.
NFC	a reader's field, the HCI events it raises, and an APDU sent over it.
BIP	the channels an applet opened, and what the far end answers — echo, silent or scripted.
CAT_TP	the transport above BIP, driven a step at a time or against a far end the lab plays.
Campaign	a named list of card commands: run, save, import, export.

Controls that need a card are **disabled until one exists**, with a title saying so. They used to be enabled and answer a click by writing a note into a console that was, at the time, below four panels you touch once a session — which is indistinguishable from a broken button.

3. Debugging: breakpoints and the trace

Setting a breakpoint

1. Click in the **gutter** beside a line of your applet — the margin left of the line number. A red dot appears.
2. Tracing switches itself on. A breakpoint with the trace off is armed and can never fire, which is indistinguishable from a broken one.
3. Do the thing that runs that code: send an APDU, press a menu entry, fire an event.
4. The Trace pane fills, ending on your line. The label says *stopped at a breakpoint*.

What actually happens. The browser sends the file and line number. It does *not* know where that is in bytecode — only the card does, from the CAP's Debug component — so the service resolves the line to a method offset and a program counter and matches on that.

A consequence worth relying on: a breakpoint on a line that has moved simply stops matching, the way it does in every debugger. And a breakpoint on a blank line, a comment or a closing brace is quietly dropped rather than refusing the whole request.

Reading the trace

Each row is one instruction, with the source position it came from:

```
Wallet.java: 42  +  18  getfield_s      stack 2
Wallet.java: 42  +  20  sadd           stack 2
Wallet.java: 42  +  21  putfield_s     stack 0
```

- **Drag the slider** to walk the recording. The editor highlights the line and follows along — forwards *and backwards*, at the speed of an array index.
- **Click a row** to jump the editor to that line.
- **Press More** when the label says the window filled. That re-records a later window of the same command; nothing is executed that has not already been executed.

A breakpoint ends the recording, not the run. There is no live machine to leave suspended — the next command rebuilds the card by replaying its history — so the applet finishes and answers normally. Nothing is lost by that, and a great deal is avoided.

When an applet does nothing

Three causes, in the order worth checking:

- 1. **It threw.** A throw inside `processToolkit` is caught by the card — the terminal still gets its answer — so the status word says nothing. The lab surfaces it in the Card log as a note naming the exception. Look there first.
- 2. **It never registered.** An applet is only triggered by events it asked for. Check the `setEvent` calls in the constructor.
- 3. **No TERMINAL PROFILE.** Every `send()` fails until one arrives, and the failure message says so.

4. The USIM and its files

The card carries a real UICC file tree. Every `FileView` call an applet makes is built into an APDU and run against it, so the access conditions, the record pointer and the status words are the card's own.

FILE	ID	SHAPE	READ	UPDATE
EF_ICCID	2F00 ... 2FE2	10 octets	ALWAYS	NEVER
EF_DIR	2F00	32 octets	ALWAYS	ADM
EF_ARR	2F06	32 octets	ALWAYS	NEVER
DF_TELECOM	7F10	directory	—	—
EF_SMS	6F3C	10 × 176	PIN1	PIN1
EF_SMSP	6F42	2 × 28	PIN1	PIN1
EF_SMSS	6F43	2 octets	PIN1	PIN1
EF_ADN	6F3A	10 × 30	PIN1	PIN1
EF_FDN	6F3B	5 × 30	PIN1	PIN2
EF_MSISDN	6F40	2 × 30	PIN1	PIN1

Every one of those identifiers is declared by ETSI's own `uicc.access.UICCConstants` , which is why an applet can name them with a constant rather than a literal.

EF_ICCID is nibble-swapped. Each *pair* of digits is stored with the second digit in the high nibble, so a card whose ICCID begins 8944 holds 98 44 . Read it back without swapping and you get a different, entirely plausible serial number — and the mistake round-trips through itself perfectly, so nothing catches it. `CardReader` shows both readings side by side.

Seeing the result. The **Files** panel is the tree, its access conditions and its PINs, read from the card as it stands (`GET /sessions/{id}/files` if you are scripting). An applet that wrote a record into EF_SMS has left something behind, and a status word does not show it — the file does.

The refusal, and the other half of it

Most of the ADF USIM answers `6982` to a READ, and a refusal is the first half of what a PIN teaches. The second half is watching the same call succeed, which needs the PIN presented — so the panel has two ways to do it, and they answer different questions.

CONTROL	WHAT IT MODELS	WHAT YOU LEARN
Present PIN1	A subscriber who knows their own PIN. No digits are sent.	That the access condition is the only thing standing between the applet and the file. Refresh, and a <code>6982</code> becomes <code>9000</code> with nothing else changed.
Verify these digits	An ordinary VERIFY APDU with what you typed.	What a <i>wrong</i> PIN does: <code>63CX</code> , with the attempts remaining in the low nibble, and the counter on screen going down. Three of those and PIN1 blocks, and only the PUK recovers it.

No digits are shipped, on purpose. Nothing in this exercise depends on which four the card was personalised with, and a page that carried them so it could type them would put a PIN value in a repository for no gain. The wrong-PIN path is the one that needs digits, and there any digits will do.

EF_KEYS and EF_KEYSPS are never printed, empty or not. On a real card they hold the ciphering and integrity keys; here they hold nothing, which is exactly how a habit that is harmless today gets applied to a card where it is not.

5. Toolkit events

Pick one in the CAT panel and press *Send ENVELOPE*. The list is filled from the service, not from a table in the browser, so it can only offer events your export file actually declares.

What happens next is the card's decision, and there are exactly two outcomes:

- `9000` — an applet was registered for it and consumed it.
- `6A88` — nothing was. That is a card behaving correctly: a terminal told a message was delivered has no reason to send it again.

There is no SMS-PP event. `uicc.toolkit` declares five envelope tags and no more: `D0`, `D3`, `D4`, `D6`, `D7`. A formatted Short Message does not raise an event — it reaches an application by **TAR**, through the Remote Management path of TS 102 225/226. The `EVENT_*_SMS_PP_*` constants belong to the legacy `sim.toolkit` API, which this lab does not implement.

6. The eleven sample applets

Samples in the editor toolbar. Each replaces the workspace and prints its own instructions into the Card log.

SAMPLE	SHOWS	TRY
EventProbe	Registers for all 22 routable events and tallies which fired. Displays each arrival.	Fire several events, then 00 40 00 00 00 .
SmsCourier	Class 2 SMS both ways. Receives on its own TAR; sends a long message split across two parts with a real concatenation header.	Send it an SMS to TAR B00010 , then press its menu entry.
UssdDialer	SEND USSD to the network, and reading the reply out of the TERMINAL RESPONSE.	Press <i>Check balance</i> , then 00 60 00 00 00 .
CardReader	Reading EF_ICCID, and being refused on EF_SMS for want of a PIN. The refusal is the lesson.	00 70 00 00 00 succeeds; 00 71 00 00 00 gives 6982 .
MenuKiosk	DISPLAY TEXT, GET INPUT and SELECT ITEM, and the difference between the card's own menu and a one-off list.	Press each entry, then 00 80 00 00 00 .
BipEcho	Opens a channel, writes to it and reads the answer back. With the far end on <i>echo</i> , what returns is what you sent; set it to <i>scripted</i> and the applet is handed a reply it did not write.	Press "Open a channel", then 00 90 00 00 00 .
BipParser	Exists to be given bad input. With the far end on <i>echo</i> a parser only ever meets its own encoder's output and passes for the same reason a mirror agrees with you; on <i>scripted</i> the replies are yours. It checks the length a reply CLAIMS against the length that arrived — the one field a parser must never trust — and reports which check refused.	Far end <i>scripted</i> , replies 0103414243 and 01FF414243 . Press "Fetch a record" once per reply, then 00 B0 00 00 00 .
LegacyAlive	The 2G-era applet, on <code>sim.toolkit</code> rather than <code>uicc.toolkit</code> : a byte event, handlers reached through their own classes, and no file access at all — so it asks the terminal where it is instead of reading EF_LOCI. It also does the thing most real applets of this shape do not: it checks what <code>send()</code> actually answered. PROVIDE LOCAL INFORMATION returns general result 20 and no object at all with no service, and an applet that copies the location anyway reports the last cell it saw as if it were current.	Press "Am I alive" on normal service, set the Network panel to <i>no service</i> , and press it again. Over the air: send it an SMS to TAR B00011 , then 00 A0 00 00 00 .
UsimReader	Reads EF_AD from the ADF USIM and is refused on EF_IMSI and EF_UST. Same directory, same FileView, opposite answers: the access condition is the only difference.	00 70 00 00 00 reads; 00 71 and 00 72 are 6982.

SAMPLE	SHOWS	TRY
LocationScout	Three files. Location both ways — PUSHED as an event when the phone moves, PULLED with PROVIDE LOCAL INFORMATION when the applet asks. UTRAN and E-UTRAN location are both nine octets and mean different things, so it asks which radio FIRST.	Move the phone, then “Where am I?”. “Everything” sweeps all eight qualifiers; 00 A0 00 00 00 reads the result.
FieldWatcher	A contactless reader, three layers away. A card never sees radio: the front-end raises an HCI event, the terminal turns it into a CAT event, and only that last hop reaches the applet. Its envelope tag is D6 — an ordinary EVENT DOWNLOAD.	NFC panel: Field on, off, on. The count rises by one each time, because only the arriving edge is an event. 00 B0 00 00 00 reads the tally.

How a proactive command is actually built

There is no `initSendShortMessage` and no `initSendUSSD`. TS 102 241 gives `ProactiveHandler` five convenience initialisers — `DisplayText`, `GetInkey`, `GetInput`, `CloseChannel`, `MoreTime` — and everything else is built by hand. On a real card too:

```
ProactiveHandler h = ProactiveHandlerSystem.getTheHandler();

h.init(ToolkitConstants.PRO_CMD_SEND_USSD, (byte) 0x00,
      ToolkitConstants.DEV_ID_NETWORK);          // to the NETWORK, not the terminal

h.appendTLV(ToolkitConstants.TAG_ALPHA_IDENTIFIER,    // optional: no CR bit
            TITLE, (short) 0, (short) TITLE.length);

h.appendTLV((byte) (ToolkitConstants.TAG_USSD_STRING | 0x80), // mandatory: CR bit set
            ussd, (short) 0, (short) ussd.length);

byte result = h.send();                          // the general result
```

7. Traps worth knowing before you hit them

Nothing proactive happens before TERMINAL PROFILE

A card knows nothing about its terminal until the profile arrives, and refuses to ask it for anything until then. The lab does *not* send it for you at start-up, because doing that quietly would hide the single most common reason a real

toolkit deployment does nothing at all.

`send()` does not block here

On a card it suspends the applet until a TERMINAL RESPONSE arrives. Nothing in this lab can suspend — the interpreter runs inside one request and sessions are rebuilt by replay — so the terminal answers immediately from a script. What is lost is *timing*, and nothing else. An applet cannot observe a slow user.

A Text string begins with its coding scheme

The octet is not optional and is not a character. `copyTextString` copies it too, and `getTextStringLength` does not count it — so an applet that forgets shows a stray glyph in front of everything the user typed.

An Item object does not

SELECT ITEM's Item objects are an identifier followed by text, with no coding scheme. Getting that backwards puts a stray character at the front of every entry on screen.

TP-UDL counts the header

For 8-bit data the User Data Length includes the User Data Header, and TP-UDHI — bit 6 of the first TPDU octet — must be set. Without it the handset renders the six header octets as six leading characters instead of joining the parts.

A remote APDU script needs five header octets per command

There is nothing between one command and the next, so the script is split on header length. A four-octet case-1 APDU is refused as mis-parsed rather than truncated.

A menu item identifier is assigned, not chosen

`initMenuEntry` returns the identifier the card gave you. It is what arrives in the envelope and the only way to tell your own entries apart.

8. The API, for scripting

Every route needs a bearer token minted for this estate. Same origin as the page, so there is no preflight and no CORS to configure.

ROUTE	DOES
POST /compile	Java in, CAP out. Touches no session.
POST /sessions	A new card. Takes the export files it runs on.
POST /sessions/{id}/commands	Run one command, with optional recording and breakpoints.
POST /sessions/{id}/trace	Re-record a command that already ran. Executes nothing new.
GET /sessions/{id}/files	The card's file tree, its access conditions and its PINs. Key files' contents are withheld.
GET /sessions/{id}/security	What the card holds, what it insists on, and where each TAR's replay counter has reached.
POST /sessions/{id}/security	Personalise it: the key set and the Minimum Security Level. Re-judges the whole history, because the history is replayed against the new one.
GET /sessions/{id}	The history, which <i>is</i> the session.
POST /api	The methods an applet may call, read from the export files themselves.
POST /events	The events this card routes, and the constant each raises.
POST /envelope	Build an ENVELOPE without encoding TLV.
POST /sms	Wrap an APDU script as an OTA Short Message. Send <code>sessionId</code> with it for an SPI that needs the card's key set.
POST /channel	The whole delivery chain for one message: packet, envelope and <code>submit_sm</code> .
POST /smpp	Read a PDU, or turn an applet's SMS-SUBMIT into the <code>deliver_sm</code> an SMSC would push. A <code>submit_sm</code> that would reach a UICC comes back with the SMS-DELIVER to hand the card.
POST /sat	Build an S@T deck.

A breakpoint, over HTTP

```
POST /sessions/{id}/commands
{
  "command": { "kind": "apdu", "command": "0010000001 07" },
  "recording": {
    "enabled": true,
    "window": 4000,
    "breakpoints": [ { "sourceFile": "Wallet.java", "sourceLine": 42 } ]
  }
}
```

The response carries the trace, each entry with `sourceFile`, `sourceLine`, `methodName`, the opcode and its mnemonic — plus `stop`, which says whether the recording ended by completing, by filling its window, or on your breakpoint.

A session is its history: every command in order, replayed to rebuild the card. That is why a bug report here is a session, and reproducing one is running it.