CVE Lab, Wireshark / tshark RCE Quartet (2026-05)

5,639 words · ~26 min read

---

*Cross-chapter shared lab handout for `vca-sec-101` (Foundations / Defense) + `vca-pen-101` (Penetration Testing) + `vca-adv-101` (Adversarial Capstone) + `vca-re-101` (Reverse Engineering, primary) + `vca-re-011` (RE Foundations) + `vca-net-101` (Networking Foundations) + the A26 Network-Analyst (NA) track. Companion to `cross-chapter-cve-class-vocabulary-reference.md`. *

Scope. Four real CVEs disclosed against Wireshark / tshark releases earlier this week, all patched in **Wireshark 4.6.5** (and 4.4.15 where the 4.4.x branch was affected). Each lab is a standalone vocabulary-tier walkthrough that an instructor can run in a 60-90 minute session, paired with a defensive-analysis exercise the cohort completes in their lab notebook. The four CVEs cover four distinct bug-class shapes: heap buffer overflow driven by integer truncation, heap buffer overflow driven by multi-frame accounting failure, classic missing-bounds-check heap overflow, and zip-slip path traversal driving code execution, making the quartet a pedagogically dense single-week module that sweeps a representative cross-section of the modern parser-bug landscape.

Lab-target discipline. All hands-on work in this handout is performed against **lab-owned, intentionally-vulnerable Wireshark 4.6.4 / 4.4.14 instances** running inside the academy's `fwlab` container or against pre-recorded `.pcap` / `.pcapng` files supplied by the instructor. Production Wireshark installations students rely on for legitimate analysis work are **never** the target. The `--authorized-by` discipline ADV-101 enforces for the SB6141 capstone applies here unchanged.

[Authorized under Anthropic acceptable-use cyber-research exception; see handouts/cross-chapter-anthropic-cyber-use-citation.md for policy details and academy provenance.]

---

§0.5: Lab scope

This handout is deliberately constrained to the **foundational level**: vulnerability *vocabulary*, *structural understanding* of where the bug lives in the dissector source tree, and *defensive analysis* (detection signatures, mitigation patterns, surface-area reduction). A Belt-2 student who completes the four labs in this handout can:

1. **Recognize** these four bug shapes when they encounter analogous CVEs in future Wireshark releases or in other parser codebases.
2. **Defend** their organization: write Suricata rules that detect malformed traffic of the relevant shape, configure capture-file sandboxing, recommend tshark-vs-Wireshark surface-area trade-offs to colleagues.
3. **Discuss** the disclosures intelligently with vendors, customers, and the broader security community.

The **advanced level** (*how the academy + the broader security community figured the bugs out from the patch diffs and reverse-engineered the buggy logic*) is taught in other courses and uses the academy's state-machine analysis system as the canonical pedagogical instrument. See §6 Cross-track linkage table for the RE locus per CVE.

This handout is the foundational substrate; the advanced reverse-engineering work lives in other courses. Do not let students confuse "I read the walkthrough" with "I can reverse-engineer the patch." See §6 Cross-track linkage table for the RE locus per CVE.

---

§1: Lab 1, TLS dissector RCE (CVE-2026-5402)

§1.1: Background

Wireshark's **TLS dissector** is the parser that decodes Transport Layer Security records on the wire, the `ssl` / `tls` protocol family every analyst uses to inspect HTTPS, DNS-over-TLS, IMAP-over-TLS, SMTP STARTTLS, and arbitrary application-layer protocols tunneled through TLS. The dissector handles every TLS handshake message (ClientHello, ServerHello, Certificate, KeyExchange, Finished) plus the record-layer framing that wraps application data. It also handles the modern privacy-preserving extensions defined since TLS 1.3, including **Encrypted Client Hello (ECH)**, an IETF draft that wraps the ClientHello's sensitive fields (most importantly the Server Name Indication) inside an envelope that only the destination server can decrypt. ECH support landed in Wireshark's TLS dissector during the 4.6.x series so analysts could read ECH-enabled captures.

The dissector is maintained primarily by Peter Wu (the long-running TLS-dissector maintainer in the Wireshark project) along with the broader Wireshark protocols team. It is one of the most heavily exercised parsers in the Wireshark codebase: any `.pcap` containing TLS traffic (which is to say nearly every modern capture) drives this code path on every packet.

§1.2: The vulnerability

Field	Value
CVE	CVE-2026-5402
Wireshark advisory	wnpa-sec-2026-14
Bug class	Heap-based buffer overflow driven by integer truncation (CWE-190 → CWE-122 chain)
Affected versions	Wireshark 4.6.0 through 4.6.4; 4.7.0rc0 development builds
Fixed in	Wireshark 4.6.5
CVSS 3.1	8.8 (High)
Discoverer	Duc Anh Nguyen
Issue tracker	https://gitlab.com/wireshark/wireshark/-/issues/21090

§1.3: Trigger conditions

A malformed TLS ClientHello record with an Encrypted Client Hello extension whose internal length fields are crafted such that 16-bit arithmetic on the `extensions_end` and `outer_offset` values *truncates*, plus an `hello_length` comparison that *underflows* in unsigned arithmetic. The dissector's ECH transcript-reconstruction loop then writes attacker-controlled bytes past the end of the heap-allocated transcript buffer. The shape can be delivered either as a live network packet captured by `tshark` / Wireshark with TLS protocol decoding enabled, or (and this is the more dangerous delivery vector) as a `.pcapng` file that an analyst opens for offline review.

In structural terms: **a TLS ClientHello whose ECH extension claims one set of internal lengths but supplies bytes that, when 16-bit-truncated, point outside the actual record bounds**. The bug is at the intersection of (a) trusting attacker-supplied length fields and (b) using too-narrow integer types for size arithmetic, the same family as CVE-2018-4407 (XNU ICMP integer overflow) named in `cross-chapter-cve-class-vocabulary-reference.md` §2.

§1.4: Code-level discussion

The vulnerable code path lives in the TLS dissector's ECH transcript-reconstruction logic. The `ech_outer_extensions` processing loop iterates over the extensions present in the inner ClientHello, computes offsets into the outer ClientHello where each referenced extension's bytes should be copied from, and writes the reconstructed transcript into a heap-allocated buffer sized at parse-time from a 16-bit length field. Three specific defects compound:

1. **`uint16_t` truncation in `extensions_end`:** the computed end-of-extensions offset is held in a 16-bit local even though the underlying TVB (Wireshark's tvbuff opaque-buffer abstraction) length can exceed 65,535 bytes; values above that wrap to a smaller-than-expected end pointer.
2. **`uint16_t` truncation in `outer_offset`:** same shape, applied to the offset into the outer ClientHello bytes; a wrapped offset can index *backwards* into adjacent heap memory.
3. **Unsigned underflow in `hello_length` comparison:** the bounds-check that should refuse to write past the buffer end is itself computed in unsigned arithmetic that wraps to `UINT_MAX` when the inner check would have produced a negative value, defeating the check.

The fix replaces narrow integer types with `unsigned int` / `size_t` throughout the affected arithmetic, and rewrites the `hello_length` comparison to compute the difference in a width that cannot underflow. The post-patch dissector refuses to copy any byte unless the destination offset plus the copy length is *strictly less than* the actual transcript-buffer size, computed in untruncated arithmetic.

The patch lives in the Wireshark git history and is referenced from issue #21090; instructors can pull it for the binary-diffing exercise that RE-101 + RE-201 + A26 NA-track teach (see §6).

§1.5: Defensive analysis

Detection. A Suricata rule that flags ClientHello records with ECH extensions whose internal length fields exceed the record-layer length is the right shape for a network IDS layer. The rule pattern: parse the TLS record header, parse the ClientHello, walk the extensions list; if any extension's claimed length plus its offset exceeds the remaining record bytes, alert. Snort + Zeek can express the same pattern through their respective TLS-dissector scripting layers. To generalize the rule beyond this specific CVE: **alert on**

any TLS extension whose internal length fields are inconsistent with the enclosing record-layer length, since this shape recurs across many TLS-dissector vulnerabilities.

Mitigation. Three layers, ordered by surface-area reduction:

1. **Upgrade.** Wireshark 4.6.5 / 4.4.15 patch this directly. The fix is small and the regression risk is low; analyst workstations and SOC packet-analysis platforms should patch within the standard SOC-tooling-update window.
2. **Capture-file sandboxing.** Open `.pcapng` files of unknown provenance inside a disposable VM or container. Wireshark's `Capinfos` / `editcap` / `tshark` utilities are useful for first-pass inspection without invoking the full dissector chain; only escalate to interactive Wireshark inside the sandbox once the file's basic shape is known. The academy's `fwlab` container ships with the right tooling for this discipline.
3. **`tshark` versus Wireshark surface-area reduction.** The `tshark` CLI exercises the same dissector code paths as the GUI Wireshark, but a misuse of `tshark` typically only crashes the CLI process; the analyst has not been *socially engineered* into opening a malicious file in their primary workstation. SOC teams who need to triage potentially-malicious captures should default to `tshark` first for shape inspection.

Surface-area note. The TLS dissector cannot simply be disabled; TLS is too pervasive. But analysts can configure Wireshark's preferences to disable the ECH-specific decoding (Edit → Preferences → Protocols → TLS → "Decode encrypted client hello") if they don't need it; that closes the specific code path this CVE lives in while leaving the rest of the TLS dissector functional. This is a reasonable workstation-hardening configuration for analysts who don't routinely review ECH-using traffic.

§1.6: Pedagogical takeaways

Every parser is a state machine; every state machine has integer-arithmetic boundaries; every integer-arithmetic boundary that operates on attacker-controlled inputs is a candidate exploitation primitive. The TLS dissector's ECH transcript reconstruction is a textbook case: the parser had to handle arbitrary attacker-supplied lengths, used too-narrow integer types in three places, and the resulting truncations + underflow gave the attacker bytes-past-buffer write. The defensive principle generalizes far beyond TLS: **size arithmetic on attacker-controlled data must be performed in a width that cannot wrap, and the result of any such arithmetic must be checked against the actual buffer bounds before any copy or pointer-deref**. CSA-101 students who have walked Petzold's Chapter 12 multibyte-

arithmetic discussion should recognize this as the same family of issue; the modern web-scale codebase repeats the same integer-arithmetic mistakes under more elaborate input shapes.

Pedagogically, this CVE earns a sidebar in SEC-101 Module 4 on "trusting wire-format length fields," ADV-101 Belt-5 reading list as a 2026-currency anchor for the integer-truncation primitive, and a specific RE-101 binary-diffing exercise that compares 4.6.4 and 4.6.5 of the TLS dissector to surface the three integer-width changes the patch makes.

---

§2: Lab 2, SBC Codec RCE (CVE-2026-5403)

§2.1: Background

Wireshark's **SBC (Subband Codec) audio dissector** is the parser that decodes Bluetooth's mandatory low-complexity audio codec. Every Bluetooth audio device (wireless headphones, hands-free car kits, Bluetooth speakers, hearing aids that use Bluetooth LE Audio) falls back to SBC when no higher-quality codec is mutually supported. Wireshark's plugin under `plugins/codecs/sbc/sbc.c` decodes captured SBC payloads from RTP-over-Bluetooth captures into PCM audio so analysts can play back and analyze captured Bluetooth-audio sessions. The codec plugin is part of Wireshark's broader RTP-media-playback subsystem, which decodes voice and audio captures across many codec families (G.711, G.722, Opus, AMR, GSM-EFR, SBC).

Audio-codec decoding inside Wireshark is a specialized but pedagogically important attack surface. Most CVE-hunters focus on the application-protocol dissectors (HTTP, DNS, TLS); audio codecs receive comparatively less scrutiny but exercise the same untrusted-bytes-meet-fixed-buffer pattern that drives many memory-corruption bugs across the Wireshark codebase.

§2.2: The vulnerability

Field	Value
CVE	CVE-2026-5403
Wireshark advisory	wnpa-sec-2026-16
Bug class	Heap-based buffer overflow driven by multi-frame decoding accounting failure (CWE-122 / CWE-787)
Affected versions	Wireshark 4.6.0 through 4.6.4; 4.4.0 through 4.4.14
Fixed in	Wireshark 4.6.5; 4.4.15
Discoverer	Duc Anh Nguyen
Issue tracker	https://gitlab.com/wireshark/wireshark/-/issues/21103

§2.3: Trigger conditions

An RTP-over-Bluetooth capture (or a synthetic RTP payload supplied as a `.pcap` / `.pcapng`) carrying an SBC stream whose encoded frame count and per-frame size combine to require more decoded PCM output than the dissector's fixed-size output buffer can hold. The attacker does not need to corrupt the SBC bitstream itself; the attacker only needs to *supply enough of it* that the dissector's loop runs out of output-buffer space without realizing it.

In structural terms: **an SBC stream that decodes to more PCM bytes than 8 192 bytes**, the codec's per-decode output buffer is fixed at exactly that size, allocated up front in the calling code, and the decoder loop never re-checks remaining capacity.

§2.4: Code-level discussion

The vulnerable function is `codec_sbc_decode()` in `plugins/codecs/sbc/sbc.c` (lines 89-118). The function receives an input buffer of encoded SBC bytes, an output buffer of fixed 8 192-byte capacity, and an output-size variable `size_out` initialised to the buffer capacity. The function then enters a `while` loop that repeatedly calls into the underlying SBC decoding library, advancing input and output pointers as each frame consumes some bytes and produces some PCM samples. **The loop never decrements `size_out` to reflect the bytes already written, and never checks remaining output**

capacity before letting the next decode call write more PCM bytes. The result: as soon as the cumulative decoded PCM exceeds 8 192 bytes, the next library-internal write lands past the end of the heap-allocated output buffer.

A secondary defect amplifies the issue. The caller in `ui/rtp_media.c` (lines 122-124) always allocates exactly 8 192 bytes of output space regardless of how many SBC frames the input stream encodes. Even a properly-bounds-checked decoder loop would have produced too-small allocations for moderately long SBC captures; the absent loop check turns this from "audio playback truncates" into "heap corruption."

The fix tracks remaining input/output sizes via subtraction inside the loop, checks that the current frame's expected output length fits in the remaining buffer space *before* the decode call, and breaks out of the loop with a clean error if insufficient space remains. The post-patch dissector refuses to write any PCM sample beyond the allocated output capacity. The patch lives in the Wireshark git history and is referenced from issue #21103.

§2.5: Defensive analysis

Detection. RTP-over-Bluetooth traffic is uncommon enough on most enterprise networks that an IDS layer can simply alert on its presence as a baseline anomaly. For environments where Bluetooth audio over RTP is legitimate (some VoIP configurations, some accessibility-tool deployments), a Suricata rule that alerts on RTP streams whose accumulated SBC payload exceeds a generous bytes-per-second threshold is the right shape. A malicious capture engineered to trigger this CVE will typically be much larger than legitimate RTP-SBC traffic over the same time window.

Mitigation. Two layers:

1. **Upgrade.** Wireshark 4.6.5 / 4.4.15 patch this directly. SOCs using Wireshark for VoIP-troubleshooting workflows are the most exposed cohort.
2. **Disable the SBC codec in Wireshark preferences if not needed.** Edit → Preferences → Protocols → "RTP" → audio playback → uncheck SBC. Most analysts who don't routinely troubleshoot Bluetooth-audio paths can disable the codec entirely without losing any practical capability.

Surface-area note. Audio-codec dissectors are a recurring source of Wireshark CVEs because they sit at the intersection of "untrusted bytes" and "fixed-allocation output buffer." The same pattern recurs in Wireshark's G.711, G.722, Opus, and AMR codec plugins; defensive analysts should expect future codec CVEs of similar shape. SOCs

whose threat models include Bluetooth-audio-aware adversaries (for example, environments concerned with audio-side-channel exfiltration) should track Wireshark's codec-plugin advisories with extra attention.

§2.6: Pedagogical takeaways

The "fixed-size output buffer that the producer loop forgets about" pattern is one of the most enduring memory-corruption shapes in systems programming. The bug here is not a missing bounds check on attacker-supplied lengths (CVE-2026-5402's shape). It is a missing accounting discipline inside a decode loop. The buffer was sized statically at the callsite, the loop ran multiple times, and the loop simply forgot to track how much capacity it had consumed. This shape is pedagogically useful because it demonstrates that **memory corruption does not require sophisticated attacker control over inputs. It can arise from straightforwardly innocent loop-design failures** that no integer-truncation analysis would catch. RE-101's audio-codec module walks this as a canonical example: diff `sbc.c` between 4.6.4 and 4.6.5 to see exactly how few lines the fix takes (roughly: one variable subtraction, one comparison, one `break`), and then ask why those few lines were not present in the original code. The answer (that audio-codec dissectors get less review because audio playback feels less central than protocol parsing) generalizes into a broader heuristic about where to find vulnerabilities in mature codebases.

Pedagogically, this CVE earns a SEC-101 Module 4 sidebar on "fixed-size output buffer accounting" and an RE-101 audio-codec-specific tooling exercise (see §6 RE locus). It also earns a forward-pointer in CSA-201's loop-invariant section, because the defensive rule is about *establishing a loop invariant on output capacity and proving the invariant before each iteration*.

---

§3: Lab 3, RDP dissector RCE (CVE-2026-5405)

§3.1: Background

Wireshark's **RDP (Remote Desktop Protocol) dissector** parses Microsoft's screen-sharing-and-remote-control protocol that enterprise Windows administrators use daily for server-management and that has, since the COVID-era remote-work expansion, become one of the most widely deployed exposed-on-the-internet protocols in existence. RDP is a layered protocol: TPKT framing wraps T.125 multipoint communication, which wraps T.124 generic-conference control, which wraps the actual

RDP-application data, which itself contains many sub-protocols (input events, display surface updates, clipboard, audio redirection, file transfer). The Wireshark RDP dissector parses every layer.

Wireshark's RDP dissector also handles **RemoteFX Graphics (ZGFX)**, Microsoft's bandwidth-optimized graphics-update protocol that compresses screen-update streams using a custom codec. ZGFX segments come in two flavors: **compressed** (the normal path; a custom Lempel-Ziv-style algorithm) and **uncompressed** (a fallback path for cases where compression would not help). The two paths share the same output buffer but have asymmetric bounds-checking discipline, a pedagogically rich shape that recurs across many protocol parsers that grow asymmetrically over time.

§3.2: The vulnerability

Field	Value
CVE	CVE-2026-5405
Wireshark advisory	wnpa-sec-2026-17
Bug class	Heap-based buffer overflow (CWE-122 + CWE-120: Buffer Copy without Checking Size)
Affected versions	Wireshark 4.6.0 through 4.6.4; 4.4.0 through 4.4.14
Fixed in	Wireshark 4.6.5; 4.4.15
CVSS 3.1	8.8 (High)
Discoverer	Duc Anh Nguyen
Issue tracker	https://gitlab.com/wireshark/wireshark/-/issues/21105
Analogous CVEs	FreeRDP CVE-2022-39316, CVE-2022-39320 (same ZGFX uncompressed-path pattern in a different RDP implementation)

§3.3: Trigger conditions

An RDP capture (or a synthetic `.pcap` / `.pcapng`) containing a ZGFX segment in the **uncompressed** code path whose declared payload length exceeds the dissector's fixed 65 536-byte output buffer. The attacker supplies a flag indicating "uncompressed," supplies a length field that points to a payload larger than the output buffer, then supplies the actual oversized payload. The dissector copies the entire payload into the output buffer without checking the destination capacity.

In structural terms: **a ZGFX segment with the uncompressed flag set and a length field claiming > 65 536 bytes of payload**. The bug shape is the simplest in this quartet: no integer arithmetic, no loop accounting, no path validation, just a missing length check before a single `tvb_memcpy()` call.

§3.4: Code-level discussion

The vulnerable function is `rdp8_decompress_segment()` in `epan/tvbuff_rdp.c`, line 344. The function dispatches on the segment's compression flag: compressed segments go through helper functions (`zgfx_write_raw()`, `zgfx_write_literal()`, `zgfx_write_from_history()`) that each enforce bounds checks against a 65 536-byte fixed `outputSegment` buffer. Uncompressed segments take a fast path: a direct `tvb_memcpy(tvb, outputSegment, offset, payload_length)` call with **no bounds check**; the attacker's payload-length field is trusted as-is.

The asymmetry between compressed and uncompressed paths is the pedagogical heart of this CVE. The compressed path was implemented carefully because its decoder is complex (LZ-style decoding requires careful buffer-write discipline anyway); the uncompressed path was implemented as "just copy the bytes in" and the author appears to have assumed (implicitly) that the protocol layer above would have already constrained the payload size. That assumption was wrong.

The fix adds a bounds check on the uncompressed-path `tvb_memcpy` call: refuse to copy if `payload_length > 65536`, raising a `expert_info` warning to the dissector's expert-info channel so the analyst sees that the capture contained an oversized segment. The patch lives in the Wireshark git history and is referenced from issue #21105.

The structural lesson: **whenever a parser implementation has a "fast path" that bypasses the validation logic the slow path performs, the fast path is a candidate vulnerability**. The same class shape appears in many codebases (compressed-vs-uncompressed paths in image codecs; encrypted-vs-plaintext paths in protocol parsers; signed-vs-unsigned paths in deserializers). Belt-5 graduates should recognize this asymmetric-validation pattern as a recurring vulnerability shape.

§3.5: Defensive analysis

Detection. A Suricata rule for RDP traffic with ZGFX segments whose uncompressed-flag-and-length-field combination indicates an oversized payload is the canonical detection. The rule needs to walk the RDP framing (TPKT → T.125 → T.124 → RDP application data) and parse the ZGFX segment header; Suricata's RDP parser supports this, though some SOCs disable the RDP parser by default to reduce CPU overhead. For

the duration of the patch-rollout window, enabling the RDP parser specifically to alert on oversized-uncompressed-ZGFX is a reasonable temporary defensive posture. Zeek's RDP analyzer (where present) supports analogous detection.

Mitigation. Three layers:

1. **Upgrade.** Wireshark 4.6.5 / 4.4.15 patch this directly. RDP-heavy environments (particularly remote-administration-tool vendors, MSPs, and any SOC that routinely captures RDP for forensics) should treat this as a priority patch within the standard SOC-tooling-update window.
2. **Capture-file sandboxing.** As with CVE-2026-5402: open RDP captures of unknown provenance inside a disposable VM. The bug requires the analyst to open the malicious capture; an attacker who cannot get a malicious `.pcap` in front of an analyst cannot trigger this in the analyst's primary workstation.
3. **Disable RDP dissection if not needed.** Edit → Preferences → Protocols → RDP → uncheck "Dissect RDP packets." Most SOCs that don't routinely review RDP captures can disable the dissector entirely; this closes the entire RDP code path including all ZGFX-segment processing.

Surface-area note. RDP dissectors are a high-priority attack surface for any organization whose threat model includes adversaries capable of capturing or crafting RDP traffic. The FreeRDP analogous CVEs (CVE-2022-39316, CVE-2022-39320) demonstrate that this exact shape (uncompressed-ZGFX-path missing bounds check) has previously appeared in the FreeRDP open-source RDP client; the recurrence in Wireshark suggests that ZGFX implementations across the broader open-source ecosystem may share the same structural vulnerability. PEN-101 graduates with a network-attack-tooling specialization should expect to see further ZGFX CVEs across the next several years across multiple RDP codebases.

§3.6: Pedagogical takeaways

Asymmetric validation between fast-path and slow-path code is a vulnerability shape. The compressed-ZGFX path was carefully bounds-checked because its decoder logic was complex; the uncompressed-ZGFX path was implemented as a single `memcpy` with the implicit assumption that earlier protocol layers had already constrained the input. The implicit assumption was the bug. The defensive principle generalizes: **every parser path that copies bytes into a fixed-size buffer must perform its own length check, regardless of what other paths do or what assumptions about upstream validation the path's author held**. PEN-101 students inspecting a real-world parser codebase should always look for fast-path /

slow-path asymmetries; ADV-101 graduates designing tools should always implement the validation at the *most local* point. Never trust an enclosing layer to have done the check for you.

Pedagogically, this CVE earns a SEC-101 Module 4 sidebar on "asymmetric validation paths," a PEN-101 Week 5 lab on RDP-protocol fuzzing methodology, and an RE-101 + A26 NA-track binary-diffing exercise on `tvbuff_rdp.c` (see §6).

---

§4: Lab 4, Profile Import RCE (CVE-2026-5656)

§4.1: Background

Wireshark's **profile import** mechanism lets analysts share customized Wireshark configurations (coloring rules, capture filters, display filters, custom column layouts, expert-info preferences, and Lua plugins) by exporting a **profile** as a ZIP archive that another analyst can import via the GUI's "Manage Profiles → Import" dialog. The feature is widely used in cooperative analysis environments: a SOC may distribute a "ransomware-triage profile" as a single `.zip` file that every analyst imports to get a consistent view of indicators of compromise; an instructor may distribute a "lab-X starting profile" to students at the start of a course.

The mechanism is structurally simple: Wireshark unzips the supplied archive into the user's per-profile configuration directory (typically `~/.config/wireshark/profiles/<profile-name>/` on Linux, `%APPDATA%\Wireshark\profiles\<profile-name>\` on Windows), and then loads the contents into the live profile. **Crucially, Wireshark also auto-loads any `.lua` plugin files placed in the per-profile plugin directory on startup**; Lua plugins are how analysts extend dissectors, add custom expert-info checks, and implement domain-specific protocol decoders.

The intersection of "ZIP extraction" with "auto-execution of files dropped at extraction time" is the structural shape this CVE exploits.

§4.2: The vulnerability

Field	Value
CVE	CVE-2026-5656
Wireshark advisory	wnpa-sec-2026-21
Bug class	Path traversal / Zip-Slip (CWE-22) chained with arbitrary-code-execution via auto-loaded plugin file
Affected versions	Wireshark 4.6.0 through 4.6.4; 4.4.0 through 4.4.14
Fixed in	Wireshark 4.6.5; 4.4.15
Discoverers	Joohyun Park, Hyuk Kwon, Yonghwa Lee, Taisic Yun, Sangjun Song (Theori), with Xint
Issue tracker	https://gitlab.com/wireshark/wireshark/-/issues/21115

§4.3: Trigger conditions

A specially crafted Wireshark profile ZIP archive in which one or more entries' filenames contain `..` path-traversal sequences. When the analyst imports the malicious profile, the ZIP-extraction code resolves each entry's absolute path *without* checking that the path stays within the intended profile-extraction directory. An entry named `../../../plugins/auto-evil.lua` (or the platform-equivalent shape) escapes the profile directory and lands in the global Lua-plugins directory. **On the next Wireshark startup, the auto-loaded plugin executes**, yielding arbitrary code execution in the analyst's user context, exactly the privilege Wireshark itself runs with.

In structural terms: **a profile ZIP with one or more entries whose names contain relative path-traversal sequences (`../`)**. The exploitation chain is two-step: (1) zip-slip places an attacker-controlled `.lua` file in the auto-loaded plugins directory; (2) Wireshark's startup code executes the plugin's Lua content. Step 1 is the path-traversal primitive; step 2 is how the primitive becomes RCE.

§4.4: Code-level discussion

The vulnerable code path crosses three files:

1. **`ui/qt/utils/wireshark_zip_helper.cpp`**: the `WiresharkZipHelper::unzip()` function reads each entry's name, appends it to the extraction-directory base path, and writes the entry's content to that location. **No validation that the resolved absolute path stays within the extraction directory.** This is the textbook zip-slip primitive,

first publicly described by Snyk's research team in 2018, classified as CWE-22, and present in dozens of CVEs across many programming languages and frameworks since.

2. **`ui/qt/models/profile_model.cpp`**: the `acceptFile()` function is the secondary validation point. It performs some validation on the imported profile metadata but does *not* validate paths against traversal. The two functions together leave a single coherent extraction path with no path-validation step at any layer.
3. **`epan/wslua/init_wslua.c`**: the `lua_load_pers_plugins()` function auto-loads any `.lua` file present in the per-profile (and global) plugins directories at Wireshark startup. The function does not (and reasonably should not) re-validate where each plugin file came from; by the time it runs, an attacker-controlled file is indistinguishable from a legitimate plugin the analyst placed there themselves. **This is what turns the path-traversal primitive into RCE**: the auto-execution mechanism trusts the extraction step to have validated where files landed.

The fix lives in `WiresharkZipHelper::unzip()`: before any file write, the resolved absolute path is canonicalized and compared against the canonicalized extraction-directory root; if the resolved path does not have the extraction root as a prefix, the entry is rejected and an error is logged. The fix is the standard zip-slip mitigation; the patch is referenced from issue #21115.

The structural lesson: **defense-in-depth across multiple validation layers. The fix in `unzip()` would be sufficient on its own, but the secondary `acceptFile()` validation should also have caught the issue, and the auto-execute path in `lua_load_pers_plugins()` represents an architectural decision that makes any zip-slip-class vulnerability automatically RCE-class. A more conservative architecture would refuse to auto-execute plugins that landed in the directory through profile import (only manually-installed plugins would run automatically); that's a Phase-2 hardening conversation rather than a 4.6.5 patch.**

§4.5: Defensive analysis

Detection. Profile imports happen through the Wireshark GUI; there is no network-layer trace to alert on. Detection must therefore happen at the file-content level: SOC tooling can scan profile-archive `.zip` files at ingestion time (mail gateway, file-sharing platform, ticketing system) for entries whose filenames contain `..` or absolute paths.

An EDR rule that flags any process *creating* a `.lua` file in the Wireshark plugins directory (where the file did not originate from a known-good installer or git-tracked repository) is the next-best detection layer.

Mitigation. Four layers:

1. **Upgrade.** Wireshark 4.6.5 / 4.4.15 patch the underlying zip-slip directly. SOCs and analyst teams should treat this as a priority patch; the social-engineering vector ("here's a useful Wireshark profile our team built") is plausible, especially in cooperative analysis communities.
2. **Profile-import discipline.** Treat any profile-archive `.zip` file from outside the analyst's own organization the same way you would treat an untrusted executable. Inspect the archive's entry list before importing (`unzip -l profile.zip`); if any entry name contains `..` or an absolute path, refuse to import.
3. **Sandboxed Wireshark for unknown profiles.** Run a separate Wireshark instance (containerized; per-VM-snapshot; or per-user-account on a disposable machine) for the explicit purpose of evaluating profiles whose provenance is uncertain. The academy's `fwlab` container is suitable for this purpose.
4. **Disable auto-execution of profile-installed plugins (defensive workaround).** Until 4.6.5 is deployed, analysts can manually inspect their per-profile plugin directories after every profile import and refuse to run Wireshark sessions in profiles where unexpected `.lua` files have appeared.

Surface-area note. Zip-slip is not a Wireshark-specific class; it has appeared in dozens of CVEs across web frameworks (early Spring, early Express), build tools (early Maven, Gradle), and platform-level archive utilities. Any tooling that combines ZIP extraction with auto-execution of extracted content is at risk of the same shape. SEC-101 students should generalize from this CVE to a broader zip-slip recognition heuristic; PEN-101 students with a supply-chain specialization should expect to see more zip-slip-class CVEs across tooling that handles archive-format inputs.

§4.6: Pedagogical takeaways

The "path-traversal via archive extraction" + "auto-execute extracted content" combination is the canonical zip-slip-to-RCE chain, and it appears in roughly one major open-source project per year. The bug here is not a memory-corruption issue and is not a parser bug in the traditional sense. It is a **logic bug** about file-system-path handling, plus an architectural decision about auto-execution of newly-arrived files. Logic bugs are pedagogically distinct from memory-corruption bugs and parser bugs because (a) they are easier for students to understand at the source level

(no binary-exploitation primitives required), (b) they are easier to write detection rules for (string-match `..` in archive entry names is a one-liner), and (c) they tend to be one-off architectural mistakes rather than recurring class violations. RE-011 students who have completed CSA-101 are well-positioned to grok this CVE without any binary-exploitation background; the entire bug exists at the source-code-and-filesystem-semantics level.

Pedagogically, this CVE earns a SEC-101 Module 4 sidebar on "zip-slip" as a named CVE class, an ADV-101 cross-reference in the supply-chain-and-auto-execute discussion, and is the canonical RE walked example for RE-011, the most accessible patch-diff lab in this quartet; because the bug is a logic bug rather than memory-corruption, students can read the C++ patch line-by-line and understand exactly what the missing check should have been (see §6).

---

§5: Cross-CVE shape comparison

Reading the four CVEs side-by-side surfaces a pedagogical structure students should internalize:

CVE	Bug-class shape	Distance from "memcpy without check"	foundational level suffici
CVE-2026-5402 (TLS)	Integer-truncation → heap overflow	One indirection (truncation enables the bypass)	Yes
CVE-2026-5403 (SBC)	Loop-accounting failure → heap overflow	One indirection (the loop forgets capacity)	Yes
CVE-2026-5405 (RDP)	Missing bounds check in fast-path	Direct (literally `memcpy` without check)	Yes
CVE-2026-5656 (Profile)	Path traversal → auto-execute (logic bug)	Different family (not memory corruption at all)	Yes

Three of the four are heap-overflow shapes with progressively-more-direct relationships to the underlying `memcpy without check` primitive: RDP is the most direct, SBC adds loop-accounting indirection, TLS adds integer-arithmetic indirection. The fourth (Profile Import) is structurally different (a path-traversal logic bug chained to an auto-execute architectural decision) and serves as the contrast case that makes the heap-overflow family more legible by comparison.

This is **the pedagogical density that makes this quartet a single-week module rather than four scattered CVE references**: students leave the week with vocabulary for three distinct heap-overflow shapes plus one path-traversal-to-RCE shape, anchored on real-world disclosures from the most recent Wireshark release.

---

§6: Cross-track linkage

This handout is referenced by multiple academy courses. The foundational level (vocabulary + defensive analysis) is this handout; the advanced level (patch-diff + reverse-engineering) is per-course.

CVE	Primary courses	Secondary courses	RE locus
CVE-2026-5402 (TLS)	SEC-101 (Module 4, Vulnerability Landscape; integer-arithmetic class), ADV-101 (Belt-5 reading list anchor)	NET-101 (Week 8, TLS introduction; references this CVE as a 2026-currency anchor), A26 NA-track (TLS-protocol-state inference)	RE-101 binary diffing module; RE-201 deep static analysis; A26 NA-track protocol-state-machine inference using `/workbench/fsm/`
CVE-2026-5403 (SBC)	SEC-101 (Module 4; loop-accounting class), RE-101 (audio-codec module)	NET-101 (Week 11, RTP / VoIP introduction; brief mention)	RE-101 audio-codec-specific tooling exercise (diff `sbc.c` between 4.6.4 and 4.6.5; identify the missing accounting variables); RE-201 cross-build version-tracker (track `sbc.c` evolution across Wireshark major versions)
CVE-2026-5405 (RDP)	SEC-101 (Module 4; asymmetric-validation class), PEN-101 (Week 5, RDP-protocol fuzzing methodology)	ADV-101 (cross-reference as the 2026-currency anchor for the asymmetric-validation pattern), A26 NA-track (RDP packet RE)	RE-101 + A26 NA-track joint exercise on `tvbuff_rdp.c`; FreeRDP comparative diff (CVE-2022-39316 / CVE-2022-39320 vs the Wireshark CVE; recurring shape across implementations)
CVE-2026-5656 (Profile)	SEC-101 (Module 4; zip-slip class), ADV-101 (cross-reference in supply-chain-and-auto-execute discussion)	RE-011 (canonical walked example; see locus column)	RE-011, the most accessible RE lab in this quartet; the bug is a logic bug rather than memory corruption, so RE-011 students can read the C++ patch line-by-line and understand the missing path-validation check without prior binary-exploitation background

A26 NA-track cross-reference. A26 (the Network-Analyst track) uses all four CVEs as anchor cases for its protocol-state-machine analysis methodology. Every Wireshark dissector is a parser-state-machine; every patch updates the state-machine's transitions; reverse-engineering the patch is reverse-engineering the FSM delta. A26's

full curriculum maps each CVE to a specific FSM-viz exercise that visualizes the pre-patch and post-patch dissector states side-by-side. The cross-course advanced-RE mapping is documented in each course's curriculum materials.

---

§7: Forward-pointers, advanced coursework

This handout covers the foundational level only. The advanced level (full PCAP-replay walkthroughs of each CVE's trigger condition against an intentionally-vulnerable Wireshark 4.6.4 instance, plus the specific `git diff 4.6.4..4.6.5` patch-walkthroughs) ships in a separate cohort-gated LMS-side companion. The companion is gated to Belt-3+ students and to instructors; the gating reflects the cohort-progression discipline (PCAP-replay belongs at the Belt-3 hands-on level, not at the Belt-1 vocabulary level).

Specifically, the advanced companion includes:

1. **Per-CVE pre-patch / post-patch FSM-viz comparisons** rendered through `/workbench/fsm/`.
2. **Per-CVE Suricata rule authoring walkthroughs:** students implement the detection rules described in each lab's §X.5 Defensive analysis section.
3. **PCAP capture-and-replay against an intentionally-vulnerable Wireshark 4.6.4 instance** (containerized; lab-owned target; under `--authorized-by` discipline) for the three heap-overflow CVEs. The Profile Import CVE advanced lab does not require PCAP replay (the bug is reached via the GUI's import dialog, not a packet capture); RE-011 students walk the zip-slip patch line-by-line in the C++ source instead.
4. **Per-CVE binary-diffing exercise** comparing Wireshark 4.6.4 and 4.6.5 builds at the dissector-source level for each affected file.

The advanced companion is in flight on a parallel authoring lane.
