Python for Security Tooling — RE, VR and analysis

Concept cards — generated 2026-08-05

Contents

Track P 1
Module P01-tool-shape 1
The shape of a security tool 1
The import landscape, and the three interpreters 6

Track P

Module P01-tool-shape

The shape of a security tool

Every tool in this course has the same shape, and it is worth fixing that shape now because you are going to build eleven more of them.

> **A security tool is a library function that returns data, with a CLI bolted on that prints it.**

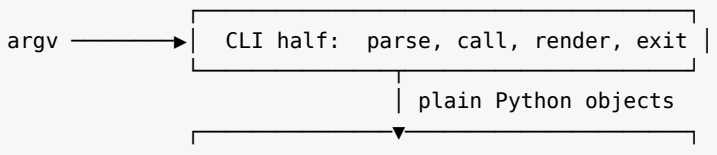
The two halves fail differently, are tested differently, and — on this platform — are graded differently. The library half has a right answer, so the trainer grades it. The CLI half is yours: you assemble it from the template at the bottom of this card, and it goes in your repository.

Why the split is not just tidiness

A tool written as one function that parses `sys.argv`, does the work, and prints as it goes cannot be:

- **called by another tool** — and the whole of P12 is composing these eleven tools into a pipeline;
- **tested** — you can test a return value; testing printed output means capturing stdout and matching strings, which breaks every time you improve a message;
- **reused at a different scale** — the function that analyses one file is the function you map over forty thousand.

So: the library half takes data and returns data. It does not print, it does not read `sys.argv`, and it does not call `sys.exit`. The CLI half does all three and nothing else.



```
                  │  library half:  data in, data out     │  ◀── graded
                  └──────────────────────────────────────┘
```

Two streams, and they are not interchangeable
Data goes to stdout. Everything else goes to stderr.

This is not style. It is what makes a tool composable:

```
binid --json ./firmware/* > findings.json      # data captured
                                                # progress still on your terminal
```

If your progress messages went to stdout, `findings.json` is now unparseable and you will not find out until something downstream chokes on it. The rule generalises: **if a human reads it, stderr; if a program reads it, stdout.**

A corollary that catches people: `print()` writes to stdout. In a tool, `print` is for results only. Progress, warnings and errors go through `logging`, which writes to stderr by default.

The exit-code contract
A tool that always exits 0 is telling every script that calls it that everything was fine. Frequently it was not.

The contract this course uses:

Exit	Meaning
`0`	The run completed and found nothing.
`1`	The run completed and found something worth a human's attention.
`2`	The run did not complete. Something failed to analyse.

Three things about it are worth stating explicitly, because they are where the mistakes live.

"Found nothing" and "could not look" are different answers. This is the one that matters. A scanner that fails to parse every single input and reports a clean run has not given you a clean run; it has given you no run at all, dressed as a clean one. If anything errored, the exit code says so — even if the parts that worked found nothing.

Findings from a target that errored do not count. If your tool collected two findings from a file and then hit a malformed header and gave up on it, you do not know whether those findings are real, and you certainly do not know what else was in there. Discard them and report the error. A partial analysis reported as a complete one is the worst output a tool can produce, because it looks exactly like a good one.

Errors and findings can both be true at once. Forty files, thirty-six clean, four unparseable, and one finding among the thirty-six. That run found something *and* failed. Report both; the exit code takes the finding branch, because a finding is the thing a human must act on, and the error count in the report is what tells them the picture is incomplete.

> This particular numbering is a convention, not a standard. `sysexits.h` and POSIX only agree that zero means success. What matters is that your tool has *a* contract, that it is documented in your README, and that it distinguishes "nothing found" from "could not look".

Determinism, and why a reviewer cares

Run your tool twice on the same input. If the output differs, nobody can review it, diff it, or trust it.

The four usual sources:

- **Filesystem order.** `os.walk` and `Path.rglob` return entries in whatever order the filesystem gives them, which differs between machines and between runs. Sort before you emit.
- **Set and dict iteration.** Dicts preserve insertion order in modern Python; sets do not preserve anything. Anything that reaches your output through a set needs an explicit sort with an explicit tie-break.
- **Unseeded randomness.** Covered properly in P11, where a fuzzer you cannot replay is a fuzzer you cannot debug. The same rule applies to any sampling.
- **Timestamps and paths in the output body.** Put them in a provenance block, not scattered through the findings, so two runs diff cleanly.

The result record

Give a finding a shape and use the same one everywhere. A `dataclass` costs three lines and buys you type checking, a free `__repr__`, and one place to change when the shape grows:

```
from dataclasses import dataclass, asdict, field


@dataclass(frozen=True, slots=True)
class Finding:
    id: str                     # stable, greppable: "unbounded-copy"
    severity: str               # high | medium | low
    target: str                 # what it was found in
    detail: str = ""            # one line, human-readable
    evidence: dict = field(default_factory=dict)

    def as_dict(self) -> dict:
        return asdict(self)
```

`frozen=True` because a finding should not be edited after it is made, and `slots=True` because you may end up with a great many of them.

The CLI wrapper — `toolcore`

This is the half you assemble. Paste it, keep it, and import it from every tool you build in this course. It is complete and runnable as it stands.

```python
"""toolcore — the CLI and report spine shared by every tool in this course."""

from __future__ import annotations

import argparse
import json
import logging
import sys

log = logging.getLogger("tool")

EXIT_CLEAN, EXIT_FINDINGS, EXIT_ERROR = 0, 1, 2


def build_parser(prog: str, description: str) -> argparse.ArgumentParser:
    p = argparse.ArgumentParser(prog=prog, description=description)
    p.add_argument("--json", action="store_true", help="emit JSON on stdout")
    p.add_argument("-v", "--verbose", action="count", default=0,
                   help="-v for info, -vv for debug")
    p.add_argument("--quiet", action="store_true", help="errors only")
    return p


def configure_logging(args: argparse.Namespace) -> None:
    """Exactly one handler, on stderr. Applications configure; libraries do not."""
    level = logging.WARNING
    if args.quiet:
        level = logging.ERROR
    elif args.verbose >= 2:
        level = logging.DEBUG
    elif args.verbose == 1:
        level = logging.INFO
    logging.basicConfig(
        level=level,
        stream=sys.stderr,
        format="%(levelname)-7s %(name)s: %(message)s",
    )


def emit(report: dict, *, as_json: bool) -> None:
    """Data to stdout. Sorted keys so two runs diff cleanly."""
    if as_json:
        json.dump(report, sys.stdout, indent=2, sort_keys=True, default=str)
        sys.stdout.write("\n")
        return
    for line in render_text(report):
        print(line)
```

```
def render_text(report: dict) -> list[str]:
    lines = [f"{report['targets']} target(s): "
             f"{report['analysed']} analysed, {report['errors']} errored"]
    for f in report.get("findings_list", []):
        lines.append(f"  [{f['severity']:<6}] {f['target']}: {f['id']}")
    lines.append(f"status: {report['status']}")
    return lines


def main(argv: list[str] | None = None) -> int:
    parser = build_parser("tool", "one-line description of your tool")
    parser.add_argument("paths", nargs="+", help="what to analyse")
    args = parser.parse_args(argv)
    configure_logging(args)

    results = []
    for path in sorted(args.paths):          # sorted: determinism
        try:
            results.append(analyse(path))    # <-- your library half
        except Exception as exc:             # noqa: BLE001 — recorded, not swallowed
            log.error("%s: %s", path, exc)
            results.append({"target": path, "findings": [], "error": str(exc)})

    report = summarize(results)              # <-- the graded function
    emit(report, as_json=args.json)
    return report["exit_code"]


if __name__ == "__main__":
    sys.exit(main())
```

Note the `except` block. It catches broadly *and records what it caught* as a result with an `error` set. That is the difference between handling an error and hiding one: the run continues, and the report still knows the picture is incomplete. `py-p1-hunt-silent-success` is the version of this loop that gets it wrong.

`pyproject.toml`

```
[project]
name = "toolcore"
version = "0.1.0"
requires-python = ">=3.11"
dependencies = []

[project.scripts]
tool = "toolcore:main"

[build-system]
requires = ["setuptools>=68"]
build-backend = "setuptools.build_meta"
```

`pip install -e .` and `tool` is on your PATH. The `[project.scripts]` entry point is what makes it a tool rather than a script you have to remember the path to.

`README.md`

```
# toolname

One sentence: what it finds, and in what.

## Install
    pip install -e .

## Use
    toolname [--json] [-v] PATH [PATH ...]

## Exit codes
| 0 | completed, nothing found |
| 1 | completed, findings reported |
| 2 | did not complete — see stderr |

## Example
    $ toolname --json ./samples/
    { "targets": 3, "analysed": 3, "errors": 0, ... }

## What it does not do
Be honest here. A tool whose limits are documented is more useful than one
that appears to do everything.
```

That last section is not filler. Every tool in this course narrows the field rather than deciding — a scanner reports candidates, not bugs. Writing the limit down is what separates a tool a reviewer can use from one they have to second-guess.

What is graded

You assemble the wrapper above; nothing checks it, and it is yours.

What the trainer grades is the function the wrapper calls on its second-to-last line:

```
summarize(results) -> dict
```

Given one result record per target, it folds them into a report with an honest status and the exit code this card's contract requires. That is `py-p1-summarize-code`, and it is the load-bearing half — everything above it is plumbing you paste once and reuse eleven times.

The import landscape, and the three interpreters

Most of the skill in this course is knowing what to reach for. The libraries are well documented individually and nowhere collected, so this card is the map.

Two things it is trying to give you: a sense of which jobs the standard library already does well enough that a dependency is a mistake, and a sense of what each of the big third-party libraries is actually *for* — which is usually narrower than its README suggests.

The stdlib spine

These come with Python, work everywhere, and cover more of this course than you would expect. Every graded function you write in this course uses only these.

Module	What it is for here
`struct`	Fixed binary layouts. The single most-used module in the course.
`pathlib`	Paths, everywhere, instead of string concatenation.
`mmap`	Random access into an image too large to read into memory.
`hashlib`	Content hashing for identity, dedupe and provenance.
`binascii`	hex/base64 conversions at the byte level.
`ctypes`	Calling native code — and it is the *real* interop story, not a toy.
`socket`	Byte-level networking; the thing pwntools wraps.
`subprocess`	Driving another tool, which is most of P09.
`argparse`	The CLI half of every tool you build.
`itertools` / `collections`	The shapes in P05.
`re`	Text pattern matching — and note it works on `bytes` too.
`json`	The interchange format between your stages.

Reach for the stdlib first. A parser built on `struct` has no install story, no version skew, and runs inside a host application that will not let you pip install anything. That last point is the whole back half of this card.

The third-party field

Reading and disassembling

Capstone — *the disassembly library.*

> A note on the name, because this platform overloads it. On TheRange, "capstone" also means a kind of task (`kind: capstone`) and a scoring model. The library is a completely unrelated

thing that happens to share the word. Everywhere in this course, **Capstone** with a capital C is the disassembler.

Give it bytes and an architecture, get back instructions with mnemonics, operands and groups. It does not know about functions, sections or control flow — it decodes. That narrowness is why it is fast and why it is embedded in almost everything else. P08 builds a pass on it.

keystone — the mirror of Capstone: give it assembly text, get bytes. You need it far less often than you expect, and mostly when constructing a payload or a patch.

pyelftools — pure-Python ELF parsing, including DWARF. Read-only, and its DWARF support is the reason to pick it over the alternatives.

LIEF — parses PE, ELF and Mach-O *and writes them back out*. If you need to modify a binary and have it still load, this is the one. Heavier than pyelftools, and a C extension.

Driving, hooking and running

pwntools — the exploitation and CTF toolkit. What you will actually use most is the boring part: tubes (`remote`, `process`) that handle framing and buffering so you do not write another `recv` loop, and the packing helpers (`p64`, `u32`). P07 contrasts these with raw sockets so you know what they are doing for you.

Frida — dynamic instrumentation. Injects a JavaScript agent into a running process; your Python drives it and receives messages. The right tool when the question is "what does this program actually do at run time" and you have a running instance. P10.

unicorn — CPU emulation without an operating system. Run a function's instructions with no process around them. **Qiling** builds an OS emulation layer on top of it.

angr — symbolic execution and binary analysis. Powerful, slow, and prone to state explosion; correct for narrow, well-posed questions.

> Unicorn, Qiling and angr are taught properly in the **firmware course, module E4**, which is built. This course does not re-teach them.

r2pipe — drives radare2 as a subprocess and speaks JSON to it. The pattern generalises, and P09 is about that pattern.

Formats, patterns and protocols

construct — declarative binary structures. A nice middle ground between hand-rolled `struct` and a full parser library, and it handles length-dependent and conditional fields well.

yara-python — pattern matching over files, productised. Rules with wildcards, string sets and conditions. P08 has you build a small version of it first so the real one is not magic.

scapy — packet crafting and dissection, and pcap reading. Layer 2 and 3.

pycryptodome / **cryptography** — the two general crypto libraries. P06 uses them for recognising and undoing, not for breaking; the mathematics is the crypto course's.

Choosing: three questions

1. **Does the stdlib already do it?** Parsing a fixed header does not need a library. Reading DWARF does.
2. **Do I need to write, or only read?** Reading ELF is pyelftools. Writing ELF is LIEF. Getting this backwards costs you a rewrite.
3. **Where will this run?** The most important question, and the reason for the rest of this card.

The three interpreters

You will write Python for three different runtimes in this course, and they do not have the same rules. Assuming there is only one is a reliable way to write a script that works on your machine and fails inside the tool.

1. CPython 3.x — your own environment

The normal case. Anything on PyPI, a virtual environment per project, pinned versions.

```
python -m venv .venv && . .venv/bin/activate
pip install capstone lief
pip freeze > requirements.txt      # pin, or your tool rots
```

Use `pipx` for tools you want on your PATH globally rather than per-project.

2. Ghidra's bundled Jython — Python 2.7

Ghidra ships an embedded Jython interpreter so scripts can talk to its Java API directly. It is genuinely convenient and it is genuinely Python 2.7.

What that costs you:

- **Python 2.7 language semantics.** No f-strings. `print` behaves differently. Integer division differs. Text and bytes are not cleanly separated.
- **The Python 2.7 standard library only** — so any stdlib module that arrived in Python 3 simply is not there, no matter how ordinary it feels to you now.
- **No C extension modules at all.** Jython runs on the JVM; a library that compiles against CPython's C API cannot load. This rules out most of the interesting half of the list above.
- **No practical pip story.** Pure-Python packages can sometimes be made to work; assume nothing does.

The upshot: **a script that must run inside Ghidra has a dependency budget of approximately zero.** You write it against the Java API and the 2.7 standard library, and you keep it small. Anything heavier belongs on the host side.

> Before you write a script for Ghidra, check every import against two questions: did this module exist in Python 2.7, and is it pure Python? If either answer is no, that import has to go — or the script has to move out of Ghidra.

3. Ghidrathon — CPython 3 inside Ghidra

Ghidrathon is an extension that hosts a real CPython 3 interpreter inside Ghidra, giving scripts the Java API *and* modern Python *and* your installed packages. It is the answer to almost every complaint in the previous section.

The cost is that it is an extension: it has to be installed and version-matched to your Ghidra, so a script that depends on it is a script other people cannot simply run. That is a real trade-off, not a formality.

The fourth option, and often the right one: do not run inside the tool at all. Drive it from outside with `subprocess` — Ghidra's `analyzeHeadless`, radare2 via r2pipe, IDA in batch mode — and keep your logic in an environment you control. P09 is that pattern in full.

Pinning, and why a tool nobody can install is not a tool

Every third-party library above has broken an API between minor versions at some point. Your portfolio repository should say exactly what it was built against:

```
capstone==5.0.1
lief==0.15.1
```

Not `capstone>=5`. A reviewer who clones your repository in a year and gets a stack trace will conclude your tool does not work, and they will be right in every way that matters.

Where each of these is picked up

- **Capstone, keystone, yara-python** → P08
- **pyelftools, LIEF, construct** → P04
- **pwntools, scapy** → P07
- **Ghidrathon, `ghidra_bridge`, IDAPython, r2pipe** → P09
- **`ctypes`, cffi, Frida** → P10
- **Atheris, boofuzz** → P11
- **AFL++, QEMU mode, crash triage** → *not here.* Firmware **E1**, and it is already built.
- **unicorn, Qiling, angr** → *not here.* Firmware **E4**, also built.