Rapid Insights for Malware Analysts

Marc Rivero | @seifreed

https://github.com/seifreed/r2inspect

WHO AM I

Marc Rivero López
Cybersecurity Researcher
@seifreed

- 🧠 I've spent more hours inside aaa than in actual sleep cycles.
- 🧩 I believe s + pdf@entry0 is a lifestyle, not a command.
- 💀 I've learned that "just one more afl" can easily become 3 AM.
- 🕵️ I trust Radare2's izz output more than some threat intel feeds.

The problem

The problem

- **Manual triage** — analysts still repeat the same initial inspection steps manually.
- **Ad-hoc scripts** — scattered one-liners without a unified structure.
- **Lack of standardization** — results vary between analysts and setups.
- **Limited reproducibility** — it’s hard to ensure the same outcome across runs.
- **Scalability gaps** — running consistent analyses on many samples is still challenging.

Goal: make Radare2-based inspection structured, reproducible, and shareable across teams

Why is this tool supported by r2?

WHY IS THIS TOOL SUPPORTED BY R2?

- **Built on r2pipe** – uses Radare2's APIs directly for analysis and data extraction.
- **Extends, not replaces** – complements existing workflows instead of reinventing them.
- **Same philosophy** – open-source, scriptable, modular, and community-driven.
- **Native compatibility** – works with r2 commands, JSON outputs, and plugins.
- **Encourages reproducibility** – makes r2 results easier to share, compare, and automate.

r2inspect exists because Radare2 made it possible.

WHAT IS R2INSPECT?

WHY IS THIS TOOL SUPPORTED BY R2?

- **A reproducible inspection framework** built on top of Radare2 and r2pipe.
- **Purpose:** automate binary triage and extract structured data (JSON/YAML) for further analysis.
- **Focus:** consistency, speed, and integration — from single samples to large batches.
- **Modular design:** each inspection unit (imports, sections, entropy, etc.) runs independently.
- **Output-first approach:** standardized, machine-readable results ready for scripting, CI, or visualization.

In short: r2inspect **turns Radare2 analyses into structured, shareable intelligence.**

WHY IS THIS TOOL SUPPORTED BY R2?

- 🔍 **Metadata extraction** — format, architecture, hashes, size, entropy.
- 📦 **Sections & segments analysis** — permissions, anomalies, entropy hints.
- 🧩 **Imports & exports mapping** — API usage, suspicious patterns, missing symbols.
- 🧠 **Heuristics detection** — packer signatures, anti-debug hints, uncommon sections.
- 🗒️ **String inspection** — keyword filtering and regex-based extraction.
- ⚙️ **Function summary** — count, complexity indicators, potential entry points.
- 🧮 **JSON/YAML export** — standardized structured output for further processing.
- 📂 **Batch mode** — process entire directories with aggregated results.
- 🧱 **Extensible modules** — easily add custom detectors or enrichers via Python.

Which features are available?

WHICH FEATURES ARE AVAILABLE?

- **Input** — a binary (PE, ELF, Mach-O, etc.) is passed to r2inspect.
- **Radare2 session** — launched internally through r2pipe, using native commands.
- **Inspection modules** — independent analyzers collect data:
    a. sections, imports, exports, strings, heuristics, hashes, entropy…
- **Normalization layer** — converts heterogeneous r2 outputs into a unified schema.
- **Exporters** — produce JSON, YAML, or CSV reports ready for scripting or pipelines.
- **Optional hooks** — custom post-processors for enrichment or policy checks.

Pipeline: Binary → Radare2 → r2inspect modules → Structured Output → Actionable Insights

WHICH FEATURES ARE AVAILABLE?

```python
import r2pipe
import hashlib

def get_imphash(path):
    r2 = r2pipe.open(path)
    r2.cmd('aa')  # analyze imports and symbols
    imports = r2.cmdj('iij')  # get imports in JSON

    if not imports:
        return None

    # Build canonical import list (module.function)
    names = []
    for imp in imports:
        mod = imp.get('libname', '').lower()
        func = imp.get('name', '').lower()
        if func:
            names.append(f"{mod}.{func}" if mod else func)

    # Sort alphabetically as per PE convention
    names = sorted(names)

    # Join with commas and hash using MD5 (standard ImpHash formula)
    data = ','.join(names).encode('utf-8')
    imphash = hashlib.md5(data).hexdigest()
    return imphash

if __name__ == "__main__":
    path = "sample.exe"
    print("ImpHash:", get_imphash(path))
```

Algorithm	Purpose	Use Case	Output Format
MD5	Basic file identification	Quick file fingerprinting	32-character hex
SHA1	File integrity verification	Legacy hash verification	40-character hex
SHA256	Cryptographic file hashing	Secure file identification	64-character hex
SHA512	Enhanced cryptographic hashing	High-security applications	128-character hex
ImpHash	Import table hashing	Malware family clustering	32-character hex
SSDeep	Fuzzy hashing	Similar file detection	Variable length string
TLSH	Locality sensitive hashing	Malware variant detection	70-character string
MACHOC	Function-level hashing	Code similarity analysis	64-character hex per function
RichPE	Rich header hashing	Compiler toolchain identification	32-character hex
Telfhash	ELF symbol hashing	Linux malware clustering	Variable length string
Impfuzzy	Import fuzzing hash	PE import similarity	Variable length string
CCBHash	Control flow graph hashing	Structural code analysis	Variable length string
SimHash	Similarity hashing	Document/code similarity	64-bit integer
Binlex	N-gram lexical analysis	Instruction pattern matching	SHA256 per n-gram size

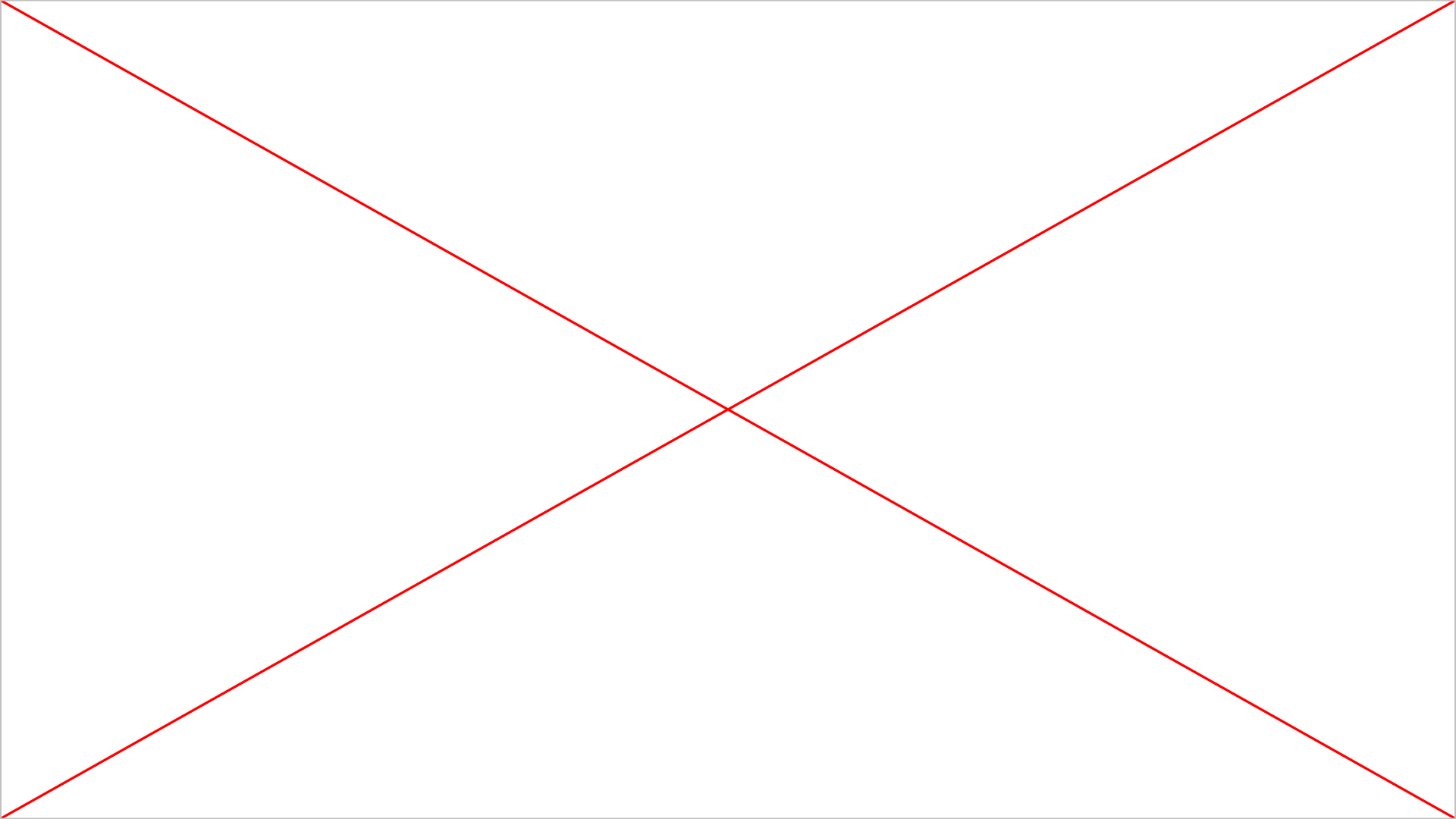
OUTPUT FORMATS

OUTPUT FORMAT

- 🗒️ **JSON** — primary structured output, ideal for automation, enrichment, and scripting.
    a. Machine-readable, deterministic, and easily parsed with jq, Python, or SIEMs.
- 📘 **YAML** — human-friendly format for quick inspection or config-driven pipelines.
    a. Same data model as JSON, but easier to review manually.
- 📊 **CSV / JSONL (batch mode)** — aggregate results from multiple binaries.
    a. Enables triage at scale, diffing, and statistical analysis.
- 📑 Markdown / HTML (optional exporters) — for readable reports or CI attachments.
    a. Useful for sharing findings without additional tooling.

Goal: make Radare2 insights portable — from CLI to dashboards, reproducible everywhere.

DEMO



How to extend the framework?

How to extend the framework?

- 🧩 **Modular architecture** — each feature is implemented as an independent inspection module.
- ⚙️ **Simple interface** — create a new Python class that inherits from Inspector.
- 📚 **Direct access to r2pipe** — every module can call native Radare2 commands or JSON APIs.
- 🧠 **Custom logic** — extract, normalize, and emit findings with your own heuristics or metrics.
- 🔄 **Automatic integration** — new modules are discovered and executed dynamically at runtime.
- 🧾 **Standardized output** — all modules produce structured JSON entries for aggregation.

HOW TO EXTEND THE FRAMEWORK?

```python
#!/usr/bin/env python3
"""
RWX Section Detector Module for r2inspect
Detects executable sections that are also writable (RWX),
which often indicate unpacking stubs or shellcode regions.
"""

from typing import Any, Dict, List
from ..utils.logger import get_logger
from ..utils.r2_helpers import safe_cmdj

logger = get_logger(__name__)

class RWXSectionDetector:
    """Detects sections with RWX permissions"""

    def __init__(self, r2, config):
        self.r2 = r2
        self.config = config

    def analyze_rwx_sections(self) -> List[Dict[str, Any]]:
        """Return sections that have RWX permissions"""
        findings = []
        try:
            sections = safe_cmdj(self.r2, "iSj", [])
            if not sections:
                return findings

            for section in sections:
                flags = section.get("perm", "")
                if all(x in flags for x in ["r", "w", "x"]):
                    findings.append({
                        "name": section.get("name", "unknown"),
                        "vaddr": section.get("vaddr", 0),
                        "size": section.get("size", 0),
                        "perm": flags,
                        "entropy": section.get("entropy", 0.0),
                        "severity": "high",
                        "description": "Section has RWX permissions (potential unpacked code)"
                    })

        except Exception as e:
            logger.error(f"RWXSectionDetector error: {e}")

        return findings
```

Extending r2inspect is straightforward:

- Drop a new Python file in r2inspect/modules/.
- Use safe_cmdj() to query Radare2 via r2pipe.
- Implement your own logic, return structured findings.
- Register it in core.py — and it's part of the pipeline.

CI/CD INTEGRATION

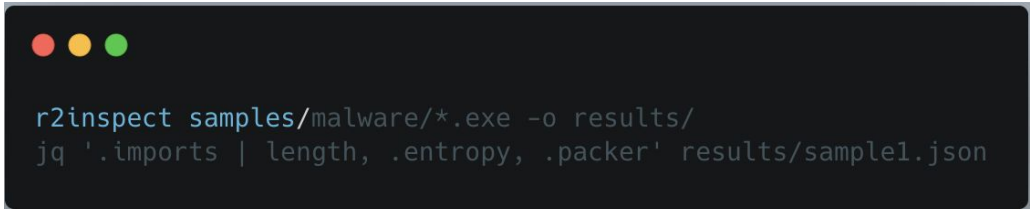
CI/CD INTEGRATION

- 🧪 **Shift-left approach** — bring binary inspection into the build pipeline.
- 🧰 **CLI-first design** — r2inspect runs headless inside CI environments (GitHub Actions, GitLab, Jenkins…).
- 📦 **Batch mode** — scan all build artifacts automatically before release.
- ⚙️ **Policy checks** — fail the build if forbidden imports, RWX sections, or packers are detected.
- 🗒️ **Machine-readable outputs** — JSON/YAML reports can be parsed and diffed between builds.
- 📈 **Continuous visibility** — track compiler changes, size, entropy, or function count over time.
- 🔄 **Reusable workflows** — easy to integrate with Docker, Python virtualenv, or r2pm environments.

USE CASES

Use case 1 - Malware Triage & Clustering

- Detects packers, high-entropy sections, and RWX anomalies.
- Extracts hashes (Imphash / SSDeep / TLSH) for clustering or similarity scoring.
- Outputs unified JSON/CSV/JSONL — easy to ingest into Python, ELK, or MISP.



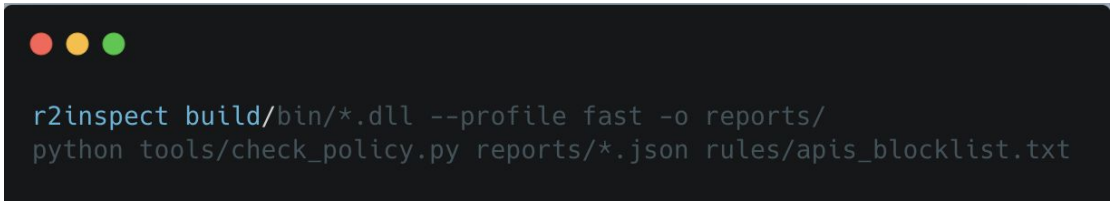
I can triage 100 binaries in under 30 seconds — reproducibly and scriptably.

~/tools/dataset/malware/**Windows**

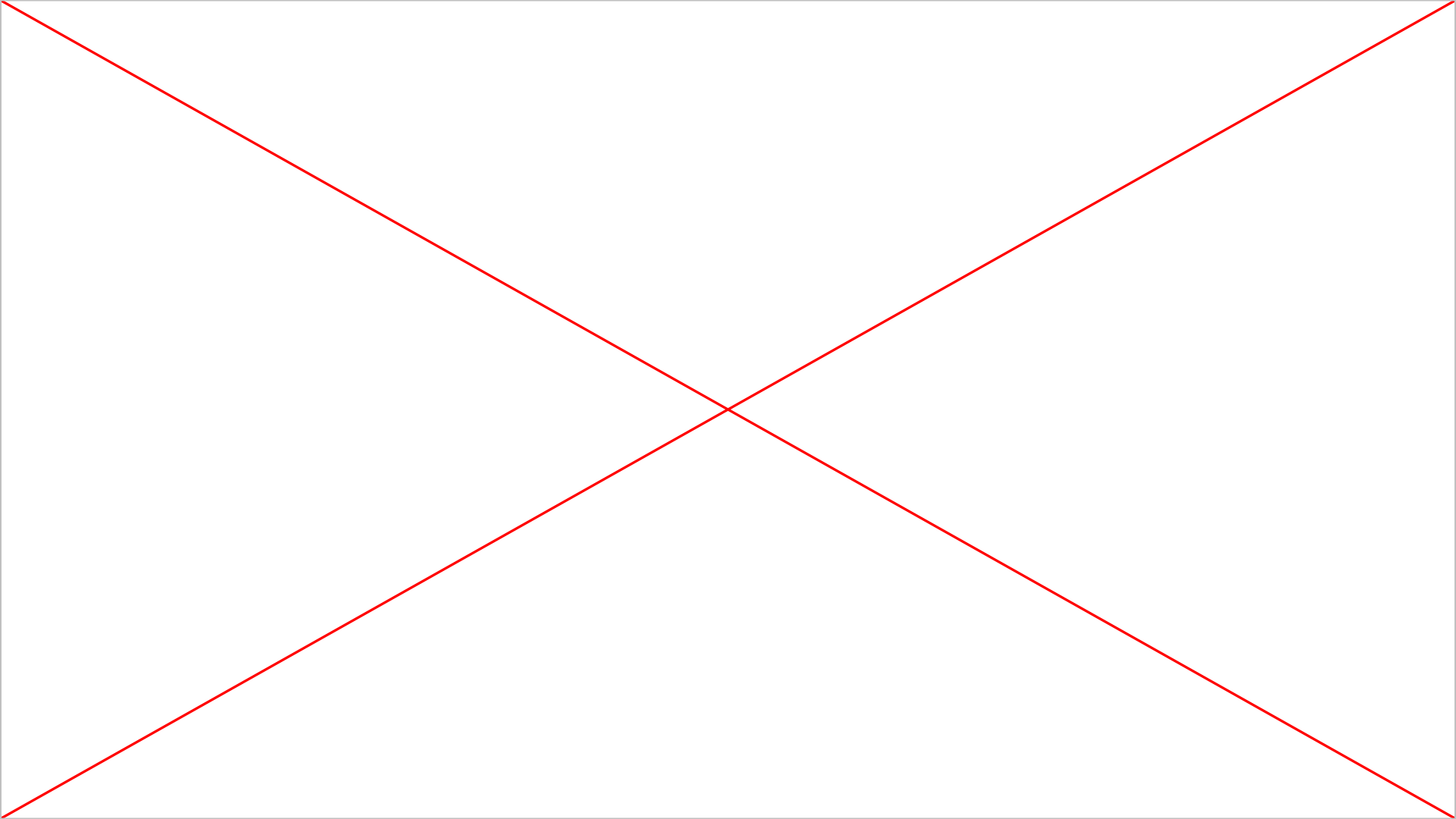
✔ at 22:31:24

Use case 2 - Build Integrity / Supply-Chain Validation

- Detects forbidden API imports (VirtualAlloc, LoadLibraryA, etc.).
- Identifies entropy or section changes between builds.
- Fails the pipeline automatically if binary policies are violated.



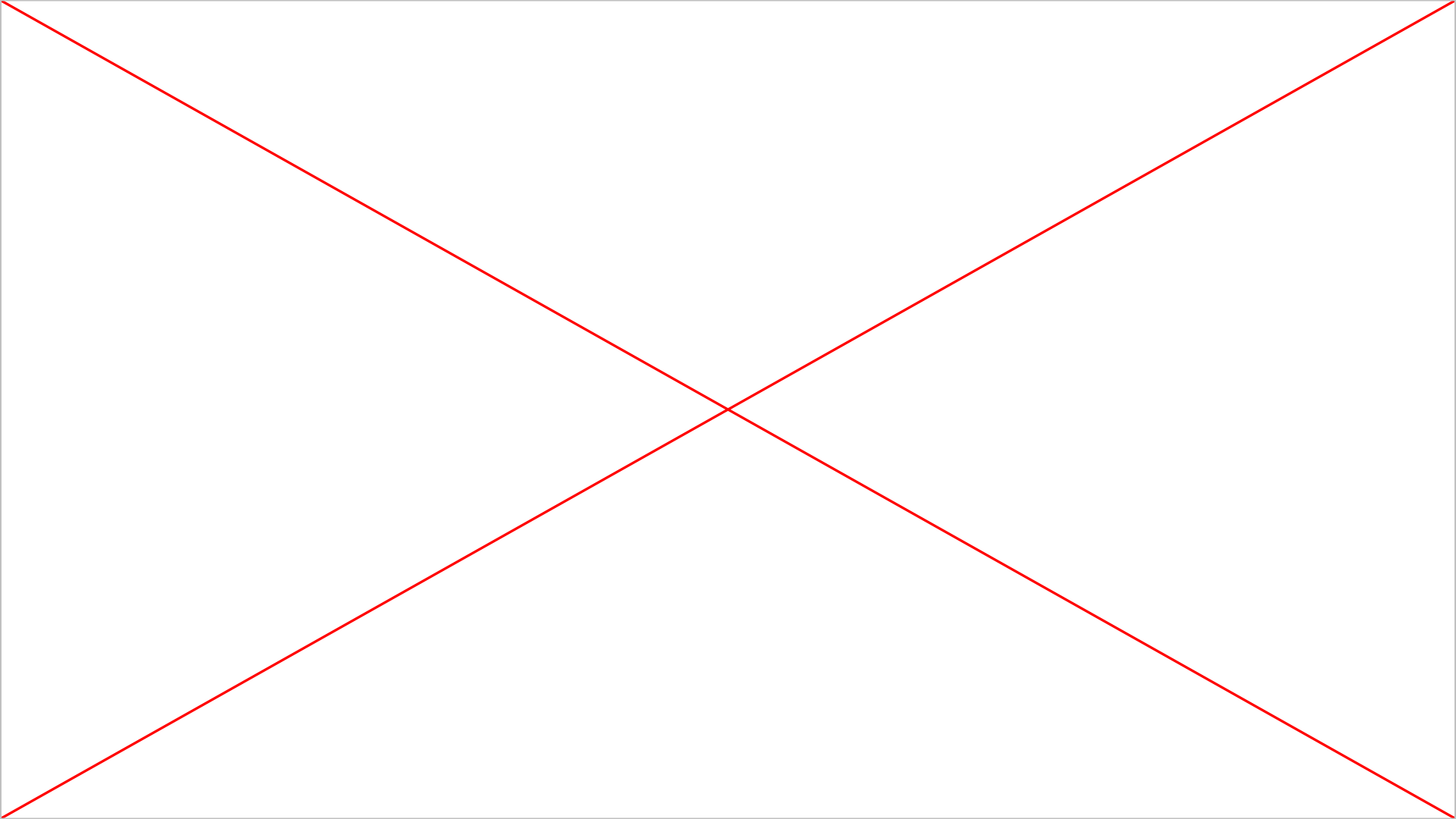
r2inspect enforces binary hygiene before deployment



Use case 3 - Threat-Intelligence Enrichment

- Extracts structured IOCs (hashes, imports, sections).
- Feeds directly into SIEM, MISP, or custom threat-intel dashboards.
- Enables MITRE ATT&CK correlation from static indicators.

It’s not just inspection — it’s intelligence you can plug anywhere.



PERFORMANCE & LIMITS

PERFORMANCE & LIMITS

- ⚡ **Memory-aware design** — each analysis runs under a monitored ***MemoryAwareAnalyzer*** with real-time control.
- 🧠 **Automatic garbage collection** — triggered dynamically when usage exceeds 75 % of allocated limits.
- 📉 **Safety thresholds:**
    a. 2 GB max process memory
    b. 512 MB per file
    c. 100 MB per section
    d. 50 K strings / 10 K functions analyzed
- 🔒 **Fail-safe behavior** — large or corrupted files are skipped gracefully instead of crashing.
- 🧰 **Adaptive analysis depth** — configurable “fast” and “deep” modes for triage or full inspection.
- 📊 **Resource reporting** — peak memory and CPU time logged for profiling and reproducibility.
- 🔄 **Scales well in batch mode** — internal checks prevent runaway memory in multi-file scans.

Goal: keep analysis stable, predictable, and reproducible — even on large or complex binaries.

ROADMAP

PERFORMANCE & LIMITS

- 🧠 **Cross-module correlation** — unify findings across imports, sections, entropy and packer detections.
- 🔬 **Advanced heuristics engine** — improved detection for virtualization, obfuscation, and in-memory loaders.
- 🧩 **Dynamic plugin loader** — allow external analyzers to register automatically without core modification.
- 📈 **Scoring system** — generate per-binary risk or suspicion scores from multiple detectors.
- 🗒️ **Reporting templates** — export HTML/Markdown dashboards directly from structured results.
- ⚙️ **Parallel processing** — multiprocessing for large-scale batch analysis with memory safety.
- 🌐 **REST API / Service mode** — optional headless server for remote inspection and integration.
- 🧪 **Test & benchmark suite** — unify validation of new modules, hashes, and performance metrics.

Next phase: evolve from static inspection tool → to a modular, extensible inspection framework for large-scale binary intelligence.

Why you should be an r2inspect user?

WHY YOU SHOULD BE AN R2INSPECT USER?

- ⚡ **Fast triage** – get structured insights from binaries in seconds.
- 🧩 **Built on Radare2** – leverages the power, flexibility, and openness of the r2 ecosystem.
- 🧠 **Reproducible results** – same analysis, same output, every time.
- 🗒️ **Standardized output** – JSON/YAML data ready for automation, SIEMs, or further enrichment.
- 🧰 **Modular & extensible** – easy to add your own detectors or analysis logic.
- 🧪 **CI/CD ready** – integrate binary inspection into your secure build pipeline.
- 🧱 **Lightweight & cross-platform** – works anywhere Radare2 does (Linux, macOS, Windows).
- 🌍 **Community-driven** – open source, transparent, and constantly evolving.

r2inspect turns Radare2's raw power into structured, reproducible intelligence – ready for your workflows.

CONCLUSIONS

CONCLUSIONS

- 🧩 **r2inspect bridges automation and analysis** — it brings structure and reproducibility to binary inspection without losing Radare2's flexibility.
- ⚡ **Practical impact:** faster triage, scalable workflows, and reliable results across analysts and pipelines.
- 🧠 **Designed for collaboration:** consistent outputs that teams can share, enrich, and build upon.
- 🌍 **Community-driven evolution:** open, extensible, and aligned with the Radare2 philosophy.
- 🔭 **Next step:** continue expanding modules, heuristics, and integrations to make large-scale binary intelligence accessible to everyone.

r2inspect isn't just a tool — it's a framework for reproducible insight, built on the spirit of Radare2.

THANKS!

https://github.com/seifreed/r2inspect