/yara-rule-authoring

Source: `~/.claude/skills/tob-yara-authoring/skills/yara-rule-authoring/SKILL.md`

---

name: yara-rule-authoring description: > Guides authoring of high-quality YARA-X detection rules for malware identification. Use when writing, reviewing, or optimizing YARA rules. Covers naming conventions, string selection, performance optimization, migration from legacy YARA, and false positive reduction. Triggers on: YARA, YARA-X, malware detection, threat hunting, IOC, signature, crx module, dex module.

YARA-X Rule Authoring

Write detection rules that catch malware without drowning in false positives.

> **This skill targets YARA-X**, the Rust-based successor to legacy YARA. YARA-X powers VirusTotal's production systems and is the recommended implementation. See Migrating from Legacy YARA if you have existing rules.

Core Principles

1. **Strings must generate good atoms** — YARA extracts 4-byte subsequences for fast matching. Strings with repeated bytes, common sequences, or under 4 bytes force slow bytecode verification on too many files.
2. **Target specific families, not categories** — "Detects ransomware" catches everything and nothing. "Detects LockBit 3.0 configuration extraction routine" catches what you want.
3. **Test against goodware before deployment** — A rule that fires on Windows system files is useless. Validate against VirusTotal's goodware corpus or your own clean file set.
4. **Short-circuit with cheap checks first** — Put `filesize < 10MB and uint16(0) == 0x5A4D` before expensive string searches or module calls.
5. **Metadata is documentation** — Future you (and your team) need to know what this catches, why, and where the sample came from.

When to Use

- Writing new YARA-X rules for malware detection
- Reviewing existing rules for quality or performance issues
- Optimizing slow-running rulesets
- Converting IOCs or threat intel into detection signatures
- Debugging false positive issues
- Preparing rules for production deployment
- Migrating legacy YARA rules to YARA-X
- Analyzing Chrome extensions (crx module)
- Analyzing Android apps (dex module)

When NOT to Use

- Static analysis requiring disassembly → use Ghidra/IDA skills
- Dynamic malware analysis → use sandbox analysis skills
- Network-based detection → use Suricata/Snort skills
- Memory forensics with Volatility → use memory forensics skills
- Simple hash-based detection → just use hash lists

YARA-X Overview

YARA-X is the Rust-based successor to legacy YARA: 5-10x faster regex, better errors, built-in formatter, stricter validation, new modules (crx, dex), 99% rule compatibility.

Install: `brew install yara-x` (macOS) or `cargo install yara-x`

Essential commands: `yr scan`, `yr check`, `yr fmt`, `yr dump`

Platform Considerations

YARA works on any file type. Adapt patterns to your target:

Platform	Magic Bytes	Bad Strings	Good Strings
Windows PE	`uint16(0) == 0x5A4D`	API names, Windows paths	Mutex names, PDB paths
macOS Mach-O	`uint32(0) == 0xFEEDFACE` (32-bit), `0xFEEDFACF` (64-bit), `0xCAFEBABE` (universal)	Common Obj-C methods	Keylogger strings, persistence paths
JavaScript/Node	(none needed)	`require`, `fetch`, `axios`	Obfuscator signatures, eval+decode chains
npm/pip packages	(none needed)	`postinstall`, `dependencies`	Suspicious package names, exfil URLs
Office docs	`uint32(0) == 0x504B0304`	VBA keywords	Macro auto-exec, encoded payloads
VS Code extensions	(none needed)	`vscode.workspace`	Uncommon activationEvents, hidden file access
Chrome extensions	Use `crx` module	Common Chrome APIs	Permission abuse, manifest anomalies
Android apps	Use `dex` module	Standard DEX structure	Obfuscated classes, suspicious permissions

macOS Malware Detection

No dedicated Mach-O module exists yet. Use magic byte checks + string patterns:

Magic bytes:

```
// Mach-O 32-bit
uint32(0) == 0xFEEDFACE
// Mach-O 64-bit
uint32(0) == 0xFEEDFACF
// Universal binary (fat binary)
uint32(0) == 0xCAFEBABE or uint32(0) == 0xBEBAFECA
```

Good indicators for macOS malware:

- Keylogger artifacts: `CGEventTapCreate`, `kCGEventKeyDown`
- SSH tunnel strings: `ssh -D`, `tunnel`, `socks`
- Persistence paths: `~/Library/LaunchAgents`, `/Library/LaunchDaemons`
- Credential theft: `security find-generic-password`, `keychain`

Example pattern from Airbnb BinaryAlert:

```
rule SUSP_Mac_ProtonRAT
{
    strings:
        // Library indicators
        $lib1 = "SRWebSocket" ascii
        $lib2 = "SocketRocket" ascii

        // Behavioral indicators
        $behav1 = "SSH tunnel not launched" ascii
        $behav2 = "Keylogger" ascii

    condition:
        (uint32(0) == 0xFEEDFACF or uint32(0) == 0xCAFEBABE) and
        any of ($lib*) and any of ($behav*)
}
```

JavaScript Detection Decision Tree

```
Writing a JavaScript rule?
├─ npm package?
│  ├─ Check package.json patterns
```

```
│   ├─ Look for postinstall/preinstall hooks
│   └─ Target exfil patterns: fetch + env access + credential paths
├─ Browser extension?
│   ├─ Chrome: Use crx module
│   └─ Others: Target manifest patterns, background script behaviors
├─ Standalone JS file?
│   ├─ Look for obfuscation markers: eval+atob, fromCharCode chains
│   ├─ Target unique function/variable names (often survive minification)
│   └─ Check for packed/encoded payloads
└─ Minified/webpack bundle?
    ├─ Target unique strings that survive bundling (URLs, magic values)
    └─ Avoid function names (will be mangled)
```

JavaScript-specific good strings:

- Ethereum function selectors: `{ 70 a0 82 31 }` (transfer)
- Zero-width characters (steganography): `{ E2 80 8B E2 80 8C }`
- Obfuscator signatures: `_0x`, `var _0x`
- Specific C2 patterns: domain names, webhook URLs

JavaScript-specific bad strings:

- `require`, `fetch`, `axios` — too common
- `Buffer`, `crypto` — legitimate uses everywhere
- `process.env` alone — need specific env var names

Essential Toolkit

Tool	Purpose
yarGen	Extract candidate strings: `yarGen.py -m samples/ --excludegood` → validate with `yr check`
FLOSS	Extract obfuscated/stack strings: `floss sample.exe` (when yarGen fails)
yr CLI	Validate: `yr check`, scan: `yr scan -s`, inspect: `yr dump -m pe`
signature-base	Study quality examples
YARA-CI	Goodware corpus testing before deployment

Master these five. Don't get distracted by tool catalogs.

Rationalizations to Reject

When you catch yourself thinking these, stop and reconsider.

Rationalization	Expert Response
"This generic string is unique enough"	Test against goodware first. Your intuition is wrong.
"yarGen gave me these strings"	yarGen suggests, you validate. Check each one manually.
"It works on my 10 samples"	10 samples ≠ production. Use VirusTotal goodware corpus.
"One rule to catch all variants"	Causes FP floods. Target specific families.
"I'll make it more specific if we get FPs"	Write tight rules upfront. FPs burn trust.
"This hex pattern is unique"	Unique in one sample ≠ unique across malware ecosystem.
"Performance doesn't matter"	One slow rule slows entire ruleset. Optimize atoms.
"PEiD rules still work"	Obsolete. 32-bit packers aren't relevant.
"I'll add more conditions later"	Weak rules deployed = damage done.
"This is just for hunting"	Hunting rules become detection rules. Same quality bar.
"The API name makes it malicious"	Legitimate software uses same APIs. Need behavioral context.
"any of them is fine for these common strings"	Common strings + any = FP flood. Use `any of` only for individually unique strings.
"This regex is specific enough"	`/fetch.*token/` matches all auth code. Add exfil destination requirement.
"The JavaScript looks clean"	Attackers poison legitimate code with injects. Check for eval+decode chains.
"I'll use .* for flexibility"	Unbounded regex = performance disaster + memory explosion. Use `.{0,30}`.
"I'll use --relaxed-re-syntax everywhere"	Masks real bugs. Fix the regex instead of hiding problems.

Decision Trees

Is This String Good Enough?

```
Is this string good enough?
├─ Less than 4 bytes?
│  └─ NO — find longer string
```

```
├─ Contains repeated bytes (0000, 9090)?
│  └─ NO — add surrounding context
├─ Is an API name (VirtualAlloc, CreateRemoteThread)?
│  └─ NO — use hex pattern of call site instead
├─ Appears in Windows system files?
│  └─ NO — too generic, find something unique
├─ Is it a common path (C:\Windows\, cmd.exe)?
│  └─ NO — find malware-specific paths
├─ Unique to this malware family?
│  └─ YES — use it
└─ Appears in other malware too?
   └─ MAYBE — combine with family-specific marker
```

When to Use "all of" vs "any of"

```
Should I require all strings or allow any?
├─ Strings are individually unique to malware?
│  └─ any of them (each alone is suspicious)
├─ Strings are common but combination is suspicious?
│  └─ all of them (require the full pattern)
├─ Strings have different confidence levels?
│  └─ Group: all of ($core_*) and any of ($variant_*)
└─ Seeing many false positives?
   └─ Tighten: switch any → all, add more required strings
```

Lesson from production: Rules using `any of ($network_*)` where strings included "fetch", "axios", "http" matched virtually all web applications. Switching to require credential path AND network call AND exfil destination eliminated FPs.

When to Abandon a Rule Approach

Stop and pivot when:

- **yarGen returns only API names and paths** → See When Strings Fail, Pivot to Structure
- **Can't find 3 unique strings** → Probably packed. Target the unpacked version or detect the packer.
- **Rule matches goodware files** → Strings aren't unique enough. 1-2 matches = investigate and tighten; 3-5 matches = find different indicators; 6+ matches = start over.
- **Performance is terrible even after optimization** → Architecture problem. Split into multiple focused rules or add strict pre-filters.

- **Description is hard to write** → The rule is too vague. If you can't explain what it catches, it catches too much.

Debugging False Positives

```
FP Investigation Flow:
│
├─ 1. Which string matched?
│     Run: yr scan -s rule.yar false_positive.exe
│
├─ 2. Is it in a legitimate library?
│     └─ Add: not $fp_vendor_string exclusion
│
├─ 3. Is it a common development pattern?
│     └─ Find more specific indicator, replace the string
│
├─ 4. Are multiple generic strings matching together?
│     └─ Tighten to require all + add unique marker
│
└─ 5. Is the malware using common techniques?
      └─ Target malware-specific implementation details, not the technique
```

Hex vs Text vs Regex

```
What string type should I use?
│
├─ Exact ASCII/Unicode text?
│  └─ TEXT: $s = "MutexName" ascii wide
│
├─ Specific byte sequence?
│  └─ HEX: $h = { 4D 5A 90 00 }
│
├─ Byte sequence with variation?
│  └─ HEX with wildcards: { 4D 5A ?? ?? 50 45 }
│
├─ Pattern with structure (URLs, paths)?
│  └─ BOUNDED REGEX: /https:\/\/[a-z]{5,20}\.onion/
│
└─ Unknown encoding (XOR, base64)?
```

```
   └─ TEXT with modifier: $s = "config" xor(0x00-0xFF)
```

Is the Sample Packed? (Check First)

Before writing any string-based rule:

```
Is the sample packed?
├─ Entropy > 7.0?
│  └─ Likely packed — find unpacked layer first
├─ Few/no readable strings?
│  └─ Likely packed — use entropy, PE structure, or packer signatures
├─ UPX/MPRESS/custom packer detected?
│  └─ Target the unpacked payload OR detect the packer itself
└─ Readable strings available?
   └─ Proceed with string-based detection
```

Expert guidance: Don't write rules against packed layers. The packing changes; the payload doesn't.

When Strings Fail, Pivot to Structure

If yarGen returns only API names and generic paths:

```
String extraction failed — what now?
├─ High entropy sections?
│  └─ Use math.entropy() on specific sections
├─ Unusual imports pattern?
│  └─ Use pe.imphash() for import hash clustering
├─ Consistent PE structure anomalies?
│  └─ Target section names, sizes, characteristics
├─ Metadata present?
│  └─ Target version info, timestamps, resources
└─ Nothing unique?
   └─ This sample may not be detectable with YARA alone
```

Expert guidance: "One can try to use other file properties, such as metadata, entropy, import hashes or other data which stays constant." — Kaspersky Applied YARA Training

Expert Heuristics

String selection: Mutex names are gold; C2 paths silver; error messages bronze. Stack strings are almost always unique. If you need >6 strings, you're over-fitting.

Condition design: Start with `filesize <`, then magic bytes, then strings, then modules. If >5 lines, split into multiple rules.

Quality signals: yarGen output needs 80% filtering. Rules matching <50% of variants are too narrow; matching goodware are too broad.

Modifier discipline:

- **Never use `nocase` or `wide` speculatively** — only when you have confirmed evidence the case/encoding varies in samples
- `nocase` doubles atom generation; `wide` doubles string matching — both have real costs
- "If you don't have a clear reason for using those modifiers, don't do it" — Kaspersky Applied YARA

Regex anchoring:

- Regex without a 4+ byte literal substring **evaluates at every file offset** — catastrophic performance
- Always anchor regex to a distinctive literal: `/mshta\.exe http:\/\/.../` not `/http:\/\/.../`
- If you can't anchor, consider hex pattern with wildcards instead

Loop discipline:

- Always bound loops with filesize: `filesize < 100KB and for all i in (1..#a) : ...`
- Unbounded `#a` can be thousands in large files — exponential slowdown

YARA-X tips: `$_unused` to suppress warnings; `private $s` to hide from output; `yr check` + `yr fmt` before every commit.

When to Use Modules vs. Byte Checks

```
Should I use a module or raw bytes?
├─ Need imphash/rich header/authenticode?
│  └─ Use PE module — too complex to replicate
├─ Just checking magic bytes or simple offsets?
│  └─ Use uint16/uint32 — faster, no module overhead
├─ Checking section names/sizes?
│  └─ PE module is cleaner, but add magic bytes filter FIRST
├─ Checking Chrome extension permissions?
│  └─ Use crx module — string parsing is fragile
└─ Checking LNK target paths?
```

```
    └─ Use lnk module — LNK format is complex
```

Expert guidance: "Avoid the magic module — use explicit hex checks instead" — Neo23x0. Apply this principle: if you can do it with uint32(), don't load a module.

YARA-X New Features

Key additions from recent releases:

- **Private patterns** (v1.3.0+): `private $helper = "pattern"` — matches but hidden from output
- **Warning suppression** (v1.4.0+): `// suppress: slow_pattern` inline comments
- **Numeric underscores** (v1.5.0+): `filesize < 10_000_000` for readability
- **Built-in formatter**: `yr fmt rules/` to standardize formatting
- **NDJSON output**: `yr scan --output-format ndjson` for tooling

YARA-X Tooling Workflow

YARA-X provides diagnostic tools legacy YARA lacks:

Rule development cycle:

```
# 1. Write initial rule
# 2. Check syntax with detailed errors
yr check rule.yar

# 3. Format consistently
yr fmt -w rule.yar

# 4. Dump module output to inspect file structure (no dummy rule needed)
yr dump -m pe sample.exe --output-format yaml

# 5. Scan with timing info
time yr scan -s rule.yar corpus/
```

When to use `yr dump`:

- Investigating what PE/ELF/Mach-O fields are available
- Debugging why module conditions aren't matching
- Exploring new modules (crx, lnk, dotnet) before writing rules

YARA-X diagnostic advantage: Error messages include precise source locations. If `yr check` points to line 15, the issue is actually on line 15 (unlike legacy YARA).

Chrome Extension Analysis (crx module)

The `crx` module enables detection of malicious Chrome extensions. Requires YARA-X v1.5.0+ (basic), v1.11.0+ for `permhash()`.

Key APIs: `crx.is_crx`, `crx.permissions`, `crx.permhash()`

Red flags: `nativeMessaging` + `downloads`, `debugger` permission, content scripts on `<all_urls>`

```
import "crx"

rule SUSP_CRX_HighRiskPerms {
    condition:
        crx.is_crx and
        for any perm in crx.permissions : (perm == "debugger")
}
```

See crx-module.md for complete API reference, permission risk assessment, and example rules.

Android DEX Analysis (dex module)

The `dex` module enables detection of Android malware. Requires YARA-X v1.11.0+. **Not compatible with legacy YARA's dex module** — API is completely different.

Key APIs: `dex.is_dex`, `dex.contains_class()`, `dex.contains_method()`, `dex.contains_string()`

Red flags: Single-letter class names (obfuscation), `DexClassLoader` reflection, encrypted assets

```
import "dex"

rule SUSP_DEX_DynamicLoading {
    condition:
        dex.is_dex and
        dex.contains_class("Ldalvik/system/DexClassLoader;")
}
```

See dex-module.md for complete API reference, obfuscation detection, and example rules.

Migrating from Legacy YARA

YARA-X has 99% rule compatibility, but enforces stricter validation.

Quick migration:

```
yr check --relaxed-re-syntax rules/  # Identify issues
# Fix each issue, then:
yr check rules/  # Verify without relaxed mode
```

Common fixes:

Issue	Legacy	YARA-X Fix
Literal `{` in regex	`/{/`	`/\{/`
Invalid escapes	`\R` silently literal	`\\R` or `R`
Base64 strings	Any length	3+ chars required
Negative indexing	`@a[-1]`	`@a[#a - 1]`
Duplicate modifiers	Allowed	Remove duplicates

> **Note:** Use `--relaxed-re-syntax` only as a diagnostic tool. Fix issues rather than relying on relaxed mode.

Quick Reference

Naming Convention

```
{CATEGORY}_{PLATFORM}_{FAMILY}_{VARIANT}_{DATE}
```

Common prefixes: `MAL_` (malware), `HKTL_` (hacking tool), `WEBSHELL_`, `EXPL_`, `SUSP_` (suspicious), `GEN_` (generic)

Platforms: `Win_`, `Lnx_`, `Mac_`, `Android_`, `CRX_`

Example: `MAL_Win_Emotet_Loader_Jan25`

See style-guide.md for full conventions, metadata requirements, and naming examples.

Required Metadata

Every rule needs: `description` (starts with "Detects"), `author`, `reference`, `date`.

```
meta:
    description = "Detects Example malware via unique mutex and C2 path"
    author = "Your Name <email@example.com>"
    reference = "https://example.com/analysis"
    date = "2025-01-29"
```

String Selection

Good: Mutex names, PDB paths, C2 paths, stack strings, configuration markers **Bad:** API names, common executables, format specifiers, generic paths

See strings.md for the full decision tree and examples.

Condition Patterns

Order conditions for short-circuit:

1. `filesize < 10MB` (instant)
2. `uint16(0) == 0x5A4D` (nearly instant)
3. String matches (cheap)
4. Module checks (expensive)

See performance.md for detailed optimization patterns.

Workflow

1. **Gather samples** — Multiple samples; single-sample rules are brittle
2. **Extract candidates** — `yarGen -m samples/ --excludegood`
3. **Validate quality** — Use decision tree; yarGen needs 80% filtering
4. **Write initial rule** — Follow template with proper metadata
5. **Lint and test** — `yr check`, `yr fmt`, linter script
6. **Goodware validation** — VirusTotal corpus or local clean files
7. **Deploy** — Add to repo with full metadata, monitor for FPs

See testing.md for detailed validation workflow and FP investigation.

For a comprehensive step-by-step guide covering all phases from sample collection to deployment, see rule-development.md.

Common Mistakes

Mistake	Bad	Good
API names as indicators	`"VirtualAlloc"`	Hex pattern of call site + unique mutex
Unbounded regex	`/https?:\/\/.*/`	`/https?:\/\/[a-z0-9]{8,12}\.onion/`
Missing file type filter	`pe.imports(...)` first	`uint16(0) == 0x5A4D and filesize < 10MB` first
Short strings	`"abc"` (3 bytes)	`"abcdef"` (4+ bytes)
Unescaped braces (YARA-X)	`/config{key}/`	`/config\{key\}/`

Performance Optimization

Quick wins: Put `filesize` first, avoid `nocase`, bounded regex `{1,100}`, prefer hex over regex.

Red flags: Strings <4 bytes, unbounded regex (`.*`), modules without file-type filter.

See performance.md for atom theory and optimization details.

Reference Documents

Topic	Document
Naming and metadata conventions	style-guide.md
Performance and atom optimization	performance.md
String types and judgment	strings.md
Testing and validation	testing.md
Chrome extension module (crx)	crx-module.md
Android DEX module (dex)	dex-module.md

Workflows

Topic	Document
Complete rule development process	rule-development.md

Example Rules

The `examples/` directory contains real, attributed rules demonstrating best practices:

Example	Demonstrates	Source
MAL_Win_Remcos_Jan25.yar	PE malware: graduated string counts, multiple rules per family	Elastic Security
MAL_Mac_ProtonRAT_Jan25.yar	macOS: Mach-O magic bytes, multi-category grouping	Airbnb BinaryAlert
MAL_NPM_SupplyChain_Jan25.yar	npm supply chain: real attack patterns, ERC-20 selectors	Stairwell Research
SUSP_JS_Obfuscation_Jan25.yar	JavaScript: obfuscator detection, density-based matching	imp0rtp3, Nils Kuhnert
SUSP_CRX_SuspiciousPermissions.yar	Chrome extensions: crx module, permissions	Educational

Scripts

```
uv run {baseDir}/scripts/yara_lint.py rule.yar      # Validate style/metadata
uv run {baseDir}/scripts/atom_analyzer.py rule.yar  # Check string quality
```

See README.md for detailed script documentation.

Quality Checklist

Before deploying any rule:

- ☐ Name follows `{CATEGORY}_{PLATFORM}_{FAMILY}_{VARIANT}_{DATE}` format
- ☐ Description starts with "Detects" and explains what/how
- ☐ All required metadata present (author, reference, date)

- [ ] Strings are unique (not API names, common paths, or format strings)
- [ ] All strings have 4+ bytes with good atom potential
- [ ] Base64 modifier only on strings with 3+ characters
- [ ] Regex patterns have escaped `{` and valid escape sequences
- [ ] Condition starts with cheap checks (filesize, magic bytes)
- [ ] Rule matches all target samples
- [ ] Rule produces zero matches on goodware corpus
- [ ] `yr check` passes with no errors
- [ ] `yr fmt --check` passes (consistent formatting)
- [ ] Linter passes with no errors
- [ ] Peer review completed

Resources

Quality YARA Rule Repositories

Learn from production rules. These repositories contain well-tested, properly attributed rules:

Repository	Focus	Maintainer
Neo23x0/signature-base	17,000+ production rules, multi-platform	Florian Roth
Elastic/protections-artifacts	1,000+ endpoint-tested rules	Elastic Security
reversinglabs/reversinglabs-yara-rules	Threat research rules	ReversingLabs
imp0rtp3/js-yara-rules	JavaScript/browser malware	imp0rtp3
InQuest/awesome-yara	Curated index of resources	InQuest

Style & Performance Guides

Guide	Purpose
YARA Style Guide	Naming conventions, metadata, string prefixes
YARA Performance Guidelines	Atom optimization, regex bounds
Kaspersky Applied YARA Training	Expert techniques from production use

Tools

Tool	Purpose
yarGen	Extract candidate strings from samples
FLOSS	Extract obfuscated and stack strings
YARA-CI	Automated goodware testing
YaraDbg	Web-based rule debugger

macOS-Specific Resources

Resource	Purpose
Apple XProtect	Production macOS rules at `/System/Library/CoreServices/XProtect.bundle/`
objective-see	macOS malware research and samples
macOS Security Tools	Reference list

Multi-Indicator Clustering Pattern

Production rules often group indicators by type:

```
strings:
    // Category A: Library indicators
    $a1 = "SRWebSocket" ascii
    $a2 = "SocketRocket" ascii

    // Category B: Behavioral indicators
    $b1 = "SSH tunnel" ascii
    $b2 = "keylogger" ascii nocase

    // Category C: C2 patterns
    $c1 = /https:\/\/[a-z0-9]{8,16}\.onion/

condition:
    filesize < 10MB and
    any of ($a*) and any of ($b*)  // Require evidence from BOTH categories
```

Why this works: Different indicator types have different confidence levels. A single C2 domain might be definitive, while you need multiple library imports to be confident. Grouping by `$a*`, `$b*`, `$c*` lets you express graduated requirements.

Revision #5
Created 2026-02-18 08:40:13 UTC by John
Updated 2026-07-05 20:01:52 UTC by John