Cybernuke

pre-β release version 20260911 by Ben Smyth

Cybernuke is offensive-defensive tooling for specification-backed systems, especially implementations of IETF standards wherein normative language (MUST, SHOULD, …) has proven fruitful in guiding agents to vulnerability sites.

Six zero days in six days for a fiver - unfettered access to classified material, remote kill switches, kernel vulnerabilities - day forty-nine: nineteen zero days disclosed, twenty-one under review, five-hundred-plus reports unopened.

Quickstart: Point your preferred agent at this file and at Cybernuke's published corpus.

Eradicate internet vulnerabilities by 2027

Governed by the Internet Engineering Task Force, driven by working groups drafting open standards, overwhelmingly implemented as open-source software maintained by small teams — this community is responsible for reliable host-to-host packet delivery, establishing security on top, and delivering content through those guarantees.

```
content delivery   HTTP · DNSSEC · SMTP · MQTT · OAuth · Kerberos · LDAP
 ↳ Apache · nginx · BIND · Unbound · Postfix · Exim · Mosquitto · Keycloak ·
   krb5 · OpenLDAP ...

transport security TLS · QUIC · SSH · DTLS · IKEv2 · OpenVPN
 ↳ BoringSSL · Botan · dropbear · GnuTLS · JSSE · libreswan · LibreSSL · libssh ·
   mbedTLS · msquic · neqo · ngtcp2 · NSS · OpenIKED · OpenSSH · OpenSSL ·
   OpenVPN · paramiko · picoquic · quiche · quic-go · rustls · s2n · Schannel ·
   strongSwan · wolfSSL ...

host-to-host comms Ethernet · IP · TCP · UDP · ARP · NDP · ICMP · VLAN
 ↳ Linux · FreeBSD · OpenBSD · NetBSD · macOS · Windows · Zephyr · DPDK · QNX ·
   lwIP · seL4 ...
```

Three thousand mandatory clauses across twenty-one stacks, hundreds of codebases, tens of languages, several-hundred-million lines of code. Nobody ever checked they're right. They're not. Billions spent, decades of effort — specifying, auditing, deploying these protocols — five-hundred-plus causes for concern (we've barely touched the surface).

Non-compliance as vulnerability proxy

Most implementations don't comply with their specification. Not every gap is a vulnerability, but every one is worth exploring.

Pick your favourite RFC. Catalogue normative clauses. Rank by the vulnerability non-compliance creates, then prove the worst on the wire.

1. **Enumerate normative clauses.** RFC 2119 & 8174 fix the vocabulary (MUST, SHALL, SHOULD, REQUIRED, RECOMMENDED, MAY, OPTIONAL, and their negations) and require normative statements to use it, so any line carrying that vocabulary in the RFC you're auditing is a normative clause; grep the keywords to extract them. (Modal verb is orthogonal to security: a security-sensitive SHOULD outranks an interoperability-only MUST.)

2. **Rank.** Score clauses by consequence: `P(violation breaks security)`, the likelihood non-compliance creates vulnerability. Three questions estimate:

   a. Who is the attacker: the peer they are talking to, on-path (can read and alter traffic in flight), off-path (can only inject spoofed packets blind), or an unprivileged local user?
   b. Given the attacker's role, what can they do here that they *couldn't already do*? Properties in play: authentication, availability, confidentiality, downgrade, forward-secrecy, integrity, replay, …
   c. Is the impact bounded to a relationship they already control?

   If (b) is "nothing new" or (c) is "yes", consequence is nil (it ranks last). Three principles for answering (b):

   - *full-chain walk* — trace everything the attack needs; if it only fires when the attacker already holds the capability it grants (a credential valid for what they attempt: the session keys, a certificate for that name and use), it confers nothing new, so it's non-compliance, not a zero day.
   - *adversary controls the wire* — don't under-credit them: "the other side would detect and abort" is no defence when an attacker on the connection with the session keys swallows the abort and forges its own messages, so detection only saves you if the reaction reaches an honest party.
   - *cryptographic backstop* — protocols bind their transcript (TLS's Finished MAC, SSH's session-hash signature, IKE's AUTH payload). Can the attacker get past it? *No* is a common way findings evaporate; *yes* is shown rather than argued, by an exchange that completes past the backstop (Step 4).

Rank is a rough guess, not a verdict; it only orders an exhaustive sweep the wire ultimately arbitrates (Step 4), so wrong guesses harm speed, not correctness. (Clauses too vague to judge are parked, not forced into a rank.) Work down the ranking a batch at a time, each batch carried through to conclusion before the next begins.

3. **Build probes.** A probe pairs a wire event with the verdict the clause demands; one probe per clause, derived from the RFC text alone:
   - **Invert constraints; establish violation space.** Read the clause as a condition over its subject matter. Where the clause states what is required, the condition is

that statement; where it states what is forbidden, the condition is its negation. Either way the violations are what fails it — for a condition `C` over subject matter `S`, the violation space is `{ x ∈ S : ¬C(x) }`, where `C` may test:

- *a value* — membership (`x ∈ M`) gives two classes, in and out, fails on out; a range (`min <= x <= max`) gives three, below, within and above, fails above and below. A length is a range; a position (e.g., MUST come last) is membership over an index.
- *a cardinality* — how many times something may appear. Fails too few or too many: a field that MUST appear once is violated by omission or replication; one that MUST NOT appear, by its presence.
- *a collection* — every element satisfies Q. Fails when some element does not.
- *a relation between fields* — an equality (`a == b`) or a comparison (`a < b`) fails when the relation is false. A derived value (a MAC over earlier fields) fails when it does not match the value recomputed from those fields.
- *a gated condition* — `Q → R` fails on `Q ∧ ¬R`.
- *a set carried down a chain* — state the clause accumulates as it walks, each step narrowing or widening as the clause directs. Fails where a step drops what an earlier one imposed, so the violation needs the tier above the one the probe would minimally reach.

- **Build wire event.** Embody the violation space in wire form, each carrying exactly the event under test and nothing else: the fewer variables on the wire, the cleaner the verdict and the easier every step that follows. Which vehicle you pick is this step's key call. *Patch a peer* when reaching the deviation needs real protocol state (handshake, transcript, live keys): take a correct one, change only the behaviour under test, leave the rest stock. *Craft packets* by hand when it's a lone packet or short sequence, no peer worth patching.

A clause worded at the sender is tested at the receiver: emit the violation as the peer, and read what the target does with it.

In practice, the patch-or-craft choice splits by layer. For TLS 1.3, picoTLS works well: a small, legible stack you patch surgically, so a few lines introduce one wire deviation and carry the exchange to the target's decision point. (OpenSSL covers corners picoTLS lacks.) One patch per clause. For example, a patch that forces CertificateVerify onto legacy RSASSA-PKCS#1-v1.5 in place of the RSA-PSS that RFC 8446 §4.4.3 mandates: a downgrade to a weaker signature the receiver must reject. Everything else is stock, so the few-line diff *is* the deviation. For host-to-host comms there is rarely a peer worth patching: raw IP/ICMPv6/NDP has no handshake to drive and no convenient library to hack, so those adversaries are usually built from scratch.

Never re-implement protocol you can inherit by patching a stock implementation; kills hand-rolled adversary anti-pattern wherein attempts to reinvent the protocol result in buggy code. *Craft by hand* only where no stock peer is worth patching — a lone packet with no crypto to reproduce (raw IP/ICMPv6/NDP, a spoofed TCP segment); the moment a key exchange or live keys gate the deviation, patch. Where the deviation lives in an artefact the peer merely carries (a certificate chain, a token), mint it with stock tools and

serve it from a stock peer. Where the deviation is a rewrite of bytes already in flight, relay instead: two stock peers with a filter between them that rewrites what passes, holding no key and decrypting nothing. A patched sender shows the bytes are malleable; only a relay shows an outsider can do it. A compliant encoder won't emit the forbidden bytes where the clause binds the emitter, though a stock tool may carry a switch that does (`openssl ocsp -badsig`). Otherwise patch the smallest one that will: (1) the layer's small, legible stack (TLS 1.3 → picoTLS); (2) the comprehensive library for its gaps (→ OpenSSL); (3) the target's own interlocutor (rsa-kex → stock plink vs a patched PuTTY server); (4) hybrid — drive a stock peer up to the parser, then hand-send the one malformed record.

Everything up to here is static: a ranked list of clauses and a probe for each. Hereafter is empirical: force the deviation onto the wire, show it grants the attacker real gain, score it. Select implementations, focusing on the widely deployed, and build each as it ships by default. Download latest version, release, or both. Make files read-only to avoid inadvertent corruption. Take each clause's probe against each selected implementation:

4. **Demonstrate on the wire — the implementation's own behaviour is the verdict.** A probe is a hypothesis until it is fired; the wire decides. Fire it at the implementation under test and watch; you read the verdict off its behaviour (accept or reject, alert).
   - **Run it, read verdict off wire.** Record actual packets exchanged (under `log/`); the capture confirms the target's decision is about *the message that really arrived*, not an unrelated earlier failure. Read the verdict according to what the clause governs:
     - *what the target accepts* — force the exact forbidden event, and since a compliant implementation must refuse it, the exchange completing at all *is* the deviation. (A handshake that starts then aborts is a reject, not a completion.)
     - *what the target sends* — drive the target to send the governed message; the verdict is whether that message satisfies the clause. Where the clause requires an emission, a run that produces none *is* the deviation.
     - *what the target does* — the deviation is in its behaviour, not in its answer; it may even reject correctly and still deviate. The verdict is whether that behaviour satisfies the clause: memory it never releases, a process that dies, traffic it re-routes to the attacker, a field that moves predictably across separate exchanges.

     Whether the target must reject is often unstated (RFC 4432 fixes the modulus the server sends and never says the client must refuse a short one), so record which you are resting on, the clause or your inference from it. Never read the verdict off your rig's exit code: that tells you the run failed, not whether the peer accepted or rejected.
   - **Bring up control.** Fire the same probe at a second, independent implementation. A compliant control *rejects* what the target accepted (or complies where it deviated), and that differential confirms the deviation is the target's and not the rig's. It is also your rejection-path control: when every implementation accepts

alike, check the rig can register a reject at all before claiming a class-wide break — the clause, not the control, is the authority on what compliant means.

Not every probe carries a violation. Where no single message can fail the condition — bound the memory it commits, keep a field unpredictable — the violation space is empty, and the probe carries an input chosen to make that behaviour observable: the largest length framing permits, the same request repeated until the answer changes, the packet whose reply carries the field. Build and fire them the same way; only the verdict differs.

5. **Classify — non-compliance is the door, push-through to exploit.** An RFC violation on its own is just non-compliance; the *push-through* (driving that violation on to an actual security break) turns it into a zero day. It becomes one only when it breaks a security invariant. Rank estimated that from the clause alone (Step 2). Drive it: add the fewest attacker moves that turn the forbidden state into gain (send the request the acceptance now authorises, substitute the bytes the accepted record no longer protects, withhold the message whose failure would have exposed the loss), and demonstrate those on the wire too. Ask (b) and (c) again, this time of what the push-through reached. If (b) is “nothing new” or (c) is “yes”, it is non-compliance. A target can also break while complying: the probe trips a defect on the path the clause mandates. There is no deviation to classify, and it is a finding all the same: score it on what the crash or the loss costs (Step 6).

6. **Score from impact, not the modal verb.** Compute the CVSS from what the attacker actually gains: a denial of service, a confidentiality break, an integrity break.

The score isn’t just a field in the report; it sets priority. With an avalanche of findings, the CVSS sorts the queue, criticals first. That’s the hunt: six steps to a zero day and score.

Auto-triage: patch into production

Manual triage just collapsed. Recall day forty-nine: nineteen zero-days disclosed, twenty-one under review, five-hundred-plus reports unopened — and that queue only grows. Triaging by hand is simply infeasible. And humans sit at every stage, the patch pipeline worst of all: the small teams maintaining these stacks — volunteers, mostly — are already underwater. Meanwhile the clock runs against us.

EU, UK, US, their allies — they’ve all said it out loud. Joint advisories from CISA, the NCSC and their Five Eyes partners put state actors *pre-positioned* inside telecom and critical-infrastructure networks, living off the land, waiting for a crisis to turn espionage into disruption; one campaign reached the carriers themselves, down to the lawful-intercept systems built to wiretap them. When your enemy controls your comms, how will you operate? We need to act:

7. **Trace to vulnerability site (source-triage).** Read the implementation's handler for the clause under test to find the root cause:

    - **Locate enforcement site.** `file:line` and function that enforces the clause, or should. Enforcement is often a check (MUST reject/abort), but can equally be a computation, an ordering, an algorithm choice, or whatever.
    - **Account for wire result.** What you found at that site either explains the deviation or you have not found it yet. No enforcement at all → the fail-open is the root cause: a mandatory clause the implementation never checks, accepting exactly what it must refuse. Enforcement present but defective → that defect is the root cause: wrong condition, wrong operator, compiled out. Enforcement present and correct → you are at the wrong site, or the probe tested something other than the clause; find the site that explains the wire before filing anything.
    - **Name defect shape.** *`ifdef`-gated checks* (validation behind a compile flag that ships off by default: dead code); *last-wins semantics* (duplicate input silently overwrites, no detection); *missing guard calls* (validator exists but isn't called here); *conditional validation* (runs in one mode, skipped in another); *parse-but-ignore* (data received, parsed, even logged, but result never enforced); *partial predicate* (the check reads fewer fields than the clause names, so a subset of the identity is all an attacker must supply); and more.

8. **Trace-to-use.** A missing check (or whatever the clause calls for) does not set the severity on its own; it matters only if the value it lets through goes on to do something dangerous. The trace settles severity in both directions: look for the use that escalates as hard as for the defence that neutralises.

    - **Follow the value from entry to use.** From where the implementation first reads it off the wire (raw bytes arriving over the network) to where it is actually *used*: arithmetic, buffer access, the security decision it feeds.
    - **Watch for downstream defences on the way.** Clamps, later bounds checks, fallback defaults that neutralise it before it reaches anything dangerous. *Severity ceiling* is set by what still gets *past* those defences, not by what the first parser was willing to accept.
    - **Record path.** Where value enters (`file:line`), call chain to where it's used, guards in between, and the worst outcome that still gets through. Where that ceiling sits below the score already given, the score comes down; where the use reaches further than the wire suggested (the value feeds an allocation, a write, an authentication decision) it goes up, on what the chain demonstrably reaches and never on what it might; the chain itself is what independent review (Step 11) re-walks, and what the maintainer reproduces.

9. **Barely viable exploit — cut to the smallest firing that still lands.** The probe is a generalist: one instrument per clause, fired across a violation space at every implementation selected. What ships is precise: the single firing that reaches the ceiling Step 8 established, recorded as it ran (argv, capture, both commits), so the

demonstration is the run that settled the question, not a rig rebuilt afterwards to resemble it. Cut what stands between that firing and the reader: apparatus we wrote, configuration the bug does not need, every element the verdict can be reached without. Delete apparatus, never impact — a demonstration cut past the gain it was meant to show reads as no finding at all.

10. **Fix — the proof you understood it.** Patch a *fresh* copy (the implementation-under-test stays read-only), re-run the probe two-sided (forbidden input rejected, legitimate input still passing) and the diff doubles as the disclosure patch. That re-run is also the root cause's falsification test: patch the site you named and the wire result must change; where it does not, the site was not the cause, whatever the source reads like. Evidence, score, and fix in hand, the finding becomes one self-contained report.

You've convinced yourself. You demonstrated the deviation on the wire, classified it a real break, scored it, and wrote the fix that closes it. That certainty is the trap. Scale is where false positives breed, and the finding likeliest to fool you is the one you're already sure of, whether a harness artefact read as a break, a root cause that only sounds right, or a fix that passes for the wrong reason. And a false positive is worse than a miss: a miss is silent, a false positive burns your reputation. So none of it counts on your say-so; what you've built earns trust only by surviving someone whose job is proving you wrong.

11. **Independent review — refute before you trust.** Your report goes to a panel of independent agents briefed to *refute* it:
    - **Reproduce.** Rebuild from the report alone, re-fire, and read the wire result off the capture, not the author's word; then re-run (a)–(c) from Step 2, re-walk the chain, attack the root cause, challenge every CVSS metric, and confirm the fix closes the hole without breaking conformance.
    - **Prior art.** Its own refutation: unfixed on main is not unknown, so search CVE and advisories, the issue tracker, open PRs, and sibling branches. A public match only downgrades the zero day to an n-day (cite it, the fix still ships), while an in-flight fix, a live embargo, or a documented wontfix is an objection that holds the finding for a human to call.

    One unrefuted objection holds it back; what survives is trusted.

Worst case, they found it first. The same clause registry and the same normative-inversion method are symmetric — run it on defence and you hold a fix, run it on offence and you hold a weapon — and we must assume someone is running it on offence, at parity, perhaps a step ahead. That symmetry is what breaks disclosure. A common coordinated-disclosure clock is forty-five days (ninety elsewhere), and even commercial teams rarely hit that, so a fix is realistically months out, every day the vulnerability remains live in production. To whoever else holds it, those months are no grace period; they are the attacker's operating window, and the collaborative, discuss-then-patch process built for responsible disclosure is exactly what holds it open.

So invert it. Auto-triage goes after the segment we can actually shorten — understanding the bug and writing the fix. Traditional disclosure leaves that to the maintainer; Steps 7–11 move it to the reporter, who understood the bug well enough to break it on the wire and close it under refutation, so what reaches the maintainer is a finished, independently-reviewed patch — review and merge, not investigate and develop. Against an adversary who already knows, secrecy protects nothing — the fastest patch is the only defence left.

12. **Report — the PR is the disclosure.** What survives review ships as-is: that self-contained report is the disclosure, serving verbatim as the upstream pull request and its commit message, not a ticket that opens a conversation but the merge-ready patch that ends one. The agent prepares the package; a human opens the PR.

From here the work is human: the reporter vouches for what they submit — understanding the finding well enough to stand behind it, accountable where the agent is not — and owns the exchange with the maintainer through to merge. It fails safe: nothing merges itself and the maintainer still reviews, so being wrong here costs a rejected patch.

Outlook

We opened on a promise: Eradication of internet vulnerabilities by 2027. Steps 1–12 find, demonstrate, and ship, one clause at a time; but finding is sampling: run out of clauses and you have still only tried your probes, and a finding-less sweep has not proved there is nothing to find. The adversary follows no rules, so the case that breaks you is the one you never tried. To eradicate rather than chase, a finished sweep has to mean a finished job.

The answer is proof: cryptographers trust neither themselves nor each other — human confidence fails far more often than mathematics — they trust only proof, checked step by step, granting nothing to intuition or authority. The strongest form is *proven executable*: a machine-checked proof that code does exactly what it's supposed to, and nothing more. But the internet is not a greenfield; several hundred million lines of code run it, not one of them proved — and proving them would take years.

So reason with the systems we have. Not a proof of the code — a proof that our tooling is *sound* (it flags only real deviations from the RFC, never a false alarm) and *complete* (every deviation is caught). Soundness is largely baked in by the wire test itself: a probe carries exactly one event under test and nothing else, and an independent control rejects what the target accepted (evidence a flag is the target's deviation and not the rig's), and the review panel re-checks what remains. (Agents, of course, still screw up: a bugged probe reported as a pass, a finding invented from nothing; the checks are there to catch that, not to make it impossible.) Completeness is the harder half; we can only expect it for a subset of clauses.

That subset is the clauses whose space is small enough to search outright. The rest explode: a clause pins wire messages to a shape the spec fixes, so its violations are finite — but the keys, nonces, and signatures range over spaces so vast that a handshake takes on the order of 2^256 shapes. Finite, but not searchable. The proof trick contains the explosion — a proposition of the form *constant cryptographic values suffice*: that a clause's compliance cannot turn on the particular cryptographic value, only on the structure carrying it. Prove that and the exponential collapses to the structural residue, small enough to brute force: search it exhaustively against fixed cryptographic values, and a clean pass proves the clause.

Get there and the task changes character. A sound-only sweep says "no deviation found yet"; a sound-and-complete one says "no clause-derived deviation remains" — the stopping condition sampling can never give. That class would be closed, provably, not probably. And *if* every clause can be brought into brute-force range — its explosion proved away, its residue searched — eighteen months against three thousand clauses is a bounded discharge, not an open-ended hunt.

Two assumptions are made by Step 3. First, messages are parsed as per RFC grammar, so the subject matter is well formed; verifying that an implementation's parser honours that grammar is a separate endeavour, deferred rather than abandoned; the probes hunt logic errors, not parser bugs. Second, the clause reads as a condition over its subject matter, so its violations form a space to search. Two kinds do not: uniqueness (a serial number unique per issuer) is a property of everything an issuer has emitted, and an algorithmic property (an initial sequence number MUST NOT be predictable from outside) is a property of the generator rather than of any one value it emits. Completeness is not delivered until both are covered.

Even then the definitions can be wrong, and that is harder still. A suite is complete only against the specification you formalise, and pinning down exactly what compliance means — every case, no more and no less — is where the difficulty lives, not in the checking. Too loose and a real deviation slips the net; too tight and you fail a conforming peer. We get these definitions wrong more often than right. So the frontier is twofold: proving the probes we have add up to the whole — and that the whole is the right one.

None of this is cause to wait for the proof. Until it lands, the sweep runs impact-first: where a protocol family's clauses outrun the effort, the highest-blast-radius classes (entropy for security-critical values, key derivation, authentication) go first, whatever their rank, since coverage is bounded by effort and impact is what a miss costs. That is still sampling — it cannot prove a finished job — but it is the sampling that matters most while the proof is built.

Appendix

Budget reader minute, not hour. An unsolicited security report does not get read; it gets *glanced at*, and a decision to invest the rest of the hour follows when that glance carries the verdict — when the maintainer knows, before committing to the details, what breaks and what it buys the attacker. So write for that glance: a well-made argument, understood inside a minute; an elevator pitch to someone who has not yet agreed to listen. That is a measurable standard rather than a matter of taste, and it is published as a metric. Across the corpus, the opening — every word before the first heading — has a median reading time of 59 seconds, and what those seconds must deliver is the verdict itself: the break, and what it buys the attacker.

Write the first minute as the finished claim, in the reader's own terms, so anyone who stops at the epigraph still leaves knowing what breaks and what it buys the attacker: nightly cron's command rewritten in flight; Botan delivering classified material without authenticating client; Firefox crashing on wire it asked for. Three verdicts from three reports, each a fact that report's own demonstration establishes. Then stop selling. The proof of concept is allowed breadth: the pitch has been accepted, the maintainer is out of the elevator, listening, and what they want now is proof. It is still the smallest thing that shows the bug: every element on the page is one the verdict cannot be reached without.

Compress, humans hate bloat. Compression takes place, and is verified, before any Step 11 agent is involved. The maintainer patches what they can read in the time they have, so a minute cut could determine whether the patch gets merged at all. The operator reads it before they do, and pays that cost on every finding, so padding one report taxes all of them. One agent shortens the argument; a second checks that nothing load-bearing was lost or altered, because faithful is exactly the property a compressor over-claims about its own work. Only then is the panel convened.

Configuration sweep. A check compiled out of the default build, a feature the default build lacks, or a path skipped in one mode, is a different implementation, and build recipes and manuals name them without opening the source. Firing every probe in every configuration the target ships in — build flags and runtime modes — is therefore correct, and expensive: cost multiplies by the configuration count, while the yield concentrates in the default that deployments actually run. So default first, then sweep on a tell: a configuration widely enabled in the field, or a handler the source shows gated.

Errata. This brief is wrong somewhere: stale steps, examples the corpus has outgrown, directions that read clearly and send the hunt sideways. What derailed you is an erratum; what you would merely word differently is not — an agent hunting faults invents them. Record each in `TODO.md` with a proposed resolution, and have your operator report it; unreported, it derails the next agent too. Good enough, not perfect, is the standard.

Record-keeping. For “reproducible from the report alone”, stamp the provenance into every finding: the exact commit or release tag of the implementation under test, the commit of the adversary peer that delivered the probe, and the toolchain (compiler, build flags, OS), alongside the test patch, the exact command, and the captured wire. Not “latest”, not “3.6.x”: a bug at one commit may be gone at the next, and a memory bug can hinge on the compiler version. A reviewer (or you, six months on) should rebuild the identical setup and re-observe the result without having to ask what you ran. Run each clause against multiple versions of the same implementation (LTS + latest), too — a version-to-version divergence is itself a result.

Report house style — no single template fits: a patched-peer finding lands tight (the deviation a few lines, the adversary compact), a crafted-packet finding needs room to stand up its wire sequence, so shape the report to the finding, not to a fixed form. One report per finding, named for its codename: `CODENAME.md`, a short evocative handle (BURST, SHATTER, PHANTOM), the patches under `evidence/CODENAME/`. The report is built, not pasted: the patches and the captured wire are compiled in from their source files, so the shipped report cannot drift from what was tested. Reproducibility provenance stays in the report (the target’s pinned commit, the adversary’s, the build recipe) because a reader rebuilds from the record with nothing else to hand. Triage/audit metadata is kept aside from the report: everything the agent needs while triaging but that doesn’t belong in a git commit message.

Open on the title: one active-voice line (subject the implementation or the attack itself, predicate the misbehaviour, trigger where it’s the crux): “Firefox crashes on the wire it asked for”, never “Vulnerability in Firefox” or a status prefix. Then a couple of header-less paragraphs that lay the bug out plainly: what the implementation does and what that hands an attacker. Name the RFC clause only when it’s needed to follow the bug: often it isn’t; sometimes the finding turns on a buried IETF requirement, and then it is the crux. Where one line crystallises the primitive, set it off on its own line; where the attack has shape (a sequence, a before/after, a fork in the packet), an ASCII diagram earns its keep, but only when it lands.

Then *Don’t trust, verify* — the demonstration, and the reader’s cue to reproduce rather than believe: the observed before/after, what is stock (the target, verbatim at its pinned commit) against what is ours (the adversary peer that delivers the probe), the build recipe, the rootless run script, the wire output read off the capture. *Trace to vulnerability site* then walks from the demonstrated break to the defect in the target’s source: the handler, the missing check, the line that should have been there. *Fix* explains what the patch does, with prose and diff interwoven (a hunk, then what it changes), then the two-sided demonstration (forbidden input rejected, legitimate input still passing, no regression), and states the fix’s honest limits where it has them.

A *Scope* section bounds and situates the claim. Where the real reach is narrower than it first looks, it names which deployed mitigations neutralise it and how large the residual delta truly is; it also carries, as the finding needs, the attacker prerequisites and the

in-/out-of-scope deployments, any sibling call sites or versions not yet wire-confirmed (with their status), a real-but-not-sold sibling referenced rather than claimed, and — where the score is genuinely deployment-dependent — the alternate CVSS reading stated beside the worst-case one. Scope is offered rather than required reading: a maintainer who already knows which deployments they run can skip it and lose nothing.

State a one-line shorthand that keeps the report self-contained — weakness ID, score, and the full vector: CWE-940 · CVSS 7.5 High (AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H) — enough to sort the queue and re-check the score in place; the score prioritises, it needn't be defended in the PR.

The report's last line is the attribution, kept verbatim — a term of use, not a courtesy: "Detected, exploited, & patched by cybernuke — cybernuke.bensmyth.com" optionally followed by "— in collaboration with <model name(s), penultimate entry followed by ampersand &> and <human name(s)>."

Above all, take the shape from the reports already filed, not from this description — match their build and their voice. The filed-report library is live — the corpus is what agents study to learn the house style, one self-contained report per finding, and it outranks this section wherever the two disagree: the house style is what those reports do. (Fetch with `curl` or `wget`; WebFetch is 403'd.)

Review & rebuttal. Step 11's panel refutes; the author answers back: it counts only when it produces fresh checkable evidence (a source line, a wire capture, a spec clause) that a reviewer who didn't write the finding re-verifies as hard as the objection was raised. An objection falls only when its rebuttal is independently confirmed; unresolved, it stands, and the burden stays with the author: no veto for the party most invested in the finding. What comes back is a reconciled finding: the sound objections conceded (re-scored, re-scoped, or dropped), the wrong ones evidenced away; the author revises the report and its metadata/notes, then resubmits, so each round judges the reconciled finding. The review also checks self-containment: a maintainer reads only the report, so `CODENAME.md` must stand alone, nothing load-bearing lives only in the metadata. The metadata is audit and triage; the report is the deliverable. Each round receives its predecessor's findings, never its verdicts, and re-derives before acting — an audit once reported two sentences disagreeing, was allowed to promote that to "one is fabricated", and the next agent rewrote a true claim into a false one. Which of two disagreeing sentences is wrong takes the record, not a reading.

Scoring (CVSS 3.1). Every finding carries a CVSS 3.1 base score, calculator-confirmed. Score the *reasonable worst case* (the most exposed realistic deployment, typically a network-facing service processing untrusted connections), and record the assumptions behind each metric (as metadata/notes):

```
Attack Vector        Network    routable to the target (TLS, QUIC, TCP, UDP)
                     Adjacent   one hop: same L2, IP subnet, or radio
                     Local      no network path; local IPC, socket, or file input
```

```
                      Physical   rare for IETF-specified implementations
Attack Complexity     Low; High for a MITM position, a race, a needed secret
                      (sequence number, nonce), or a mitigation defeat.
                      (A rare config doesn't raise it: score the config as
                      present; its rarity is Environmental, not Base.)
Privileges Required   None, an unauthenticated peer; Low if a valid credential
                      is required, High only for admin privilege.
User Interaction      None for servers and automated clients.
Scope                 CVSS's most-disputed metric — routinely over-claimed,
                      routinely rejected. Changed only when impact crosses into a
                      different security authority; a crash that merely kills the
                      library's own host process is Unchanged. Lead with S:U.
```

Everything above is the attack’s setup: how reachable the bug is, what has to line up, and what the attacker must already hold. Scope is the hinge between the two: whether the blast stays put or crosses into another authority. Everything below is the impact: what the attacker walks away with.

```
Confidentiality       decryption, or a key or credential extracted → C:H; an
                      over-read the attacker cannot steer onto one, or whose
                      bytes reach them only as a verdict → C:L
Integrity             inject or forge into an authenticated stream, or silent-drop
                      data the sender believed delivered → I:H; downgrade or
                      key-material influence → I:L (unless it enables decryption
                      or forgery, then score the realized outcome)
Availability          deterministic crash / resource-exhaustion kill → A:H;
                      single-connection teardown → A:L; nothing → N
```

Add a CVSS 4.0 score only when it captures nuance that 3.1 misses. Keep 3.1 primary.

Spec zero-days. Rare, but the worst case: a flaw in the *spec itself*. When the RFC text mandates the insecure behaviour, no conformant implementation can patch without deviating — a zero day against all of them at once. This technique tests deviations *from* the spec, so the test for a spec bug is inverted: try to write the implementation fix. If every candidate patch that closes the hole also makes the implementation RFC-non-conformant — the fix has to deviate from the text — the requirement itself is the bug. Remediation is an RFC erratum or -bis, not a maintainer patch, so route it to the IETF working group rather than the per-maintainer channel.

Tools. Allowed: ripgrep over the RFC (clause coverage), driving stock binaries, `tshark`/`tcpdump` for wire capture, a sanitizer or memory checker where the wire cannot show the verdict, standard build tools, `git`. Banned: undirected, coverage-driven fuzzing — a probe generator not aimed by a clause is the same thing by another name. This is a precision strike, not a coverage sweep — yield is set by how much of the clause registry you have inverted against the target, not by how much fuzzing it survived. A heavily-fuzzed stack still hides untested MUST-inversions: a fuzzer's oracle is a crash, a sanitizer trip, or a hang, and none fires when a well-formed message is accepted where the clause says reject — so nothing tells it which byte-flip is which normative requirement.

Verified account. Older, less capable models ran Cybernuke without complaint — Haiku and Sonnet are perfectly adequate at finding vulnerabilities and generating exploits. At least, they could March through April, until this work moved to a verified account. We haven't tested whether guardrails were added to later versions of these models. Newer, more capable models ship with guardrails that may scupper execution. Opus regularly balks; register as a verified researcher through Anthropic's Cyber Verification Program and it runs. Fable, at the frontier, refuses at the mere hint of cybersecurity, registered or not. So drive the sweep with a tier that will act — a verified Opus, or plain Sonnet or Haiku (let us know whether a non-verified account still works). This isn't Anthropic-specific, though it isn't uniform: OpenAI gates the same work behind its own vetted tier (Trusted Access for Cyber), while Grok and Kimi carry lighter cyber guardrails and no researcher-verification path at all. All three have been used successfully here.

Work with your operator. Agents work with IETF specification authors, implementation maintainers, and cybersecurity researchers: computational might paired with domain expertise. Agents must communicate with their operator (who may not have read this brief): guide them through the systematic reasoning and the decisions you are making, show them your calculations and prioritisations. It runs both ways: your operator brings what you cannot intuit. So make them your first skeptic, not a spectator: surface every call as you reach it, and let them catch what is wrong.

Write it down. Exhausting the clause space is a campaign: three thousand clauses across twenty-one stacks, over many sessions. So persist state to the working repo as you go, not at the end. First, *progress*: the clause registry, holding where each stands (ranked, traced, wire-tested, filed, or set aside: deferred clean, dropped inapplicable), findings in flight, and what you have learned per target; it is the triage/audit metadata the house style already keeps aside from the report, now doubling as the resume point. Second, *operator dialogue*: how your operator likes to work — preferred report shape, reserved calls (title, a borderline S:C-vs-S:U Scope call, ship-or-not), sign-offs, corrections — so a later session inherits settled taste instead of re-asking last week's answers. State on disk turns forgotten sessions into one continuous hunt.

YOLO, but don't be a fool. This all runs in dangerous / bypass-permissions mode; thousands of commands (grep, builds, crafting packets, driving live binaries) with no one at the keyboard, and approval prompts would stall it dead. An autonomous LLM holding

an unguarded shell while it compiles and fires attack code. So cage it: spin up a disposable VM and run the lot in there; a rogue command or a wandering agent should only ever be able to wreck a throwaway box, never your host. Agents: confirm you're in that VM before you run anything — if you can't, or you look to be on a real host, stop and tell your operator, who may not have read this far.