EXPLAINER | FORMAL VERIFICATION | AGENTIC CODING | NOVEMBER 2026

SIX WAYS TO CHECK THAT CODE IS RIGHT

From unit tests to a machine-checked proof, on one 32-line function that blocks requests to private addresses. Each rung found something the one below it missed.

FULL EXPLAINER, CODE AND GLOSSARY

01 WHEN ONE AGENT WRITES CODE AND TESTS, BOTH SHARE ITS BLIND SPOT

TRUST THE TESTS

1. An agent writes hundreds of lines a minute.
2. It writes the tests from the same understanding.
3. A misunderstanding passes both. CI goes green.

TRUST A SPEC AND A CHECKER

1. You read one line: *refuse what the OS reads as private.*
2. A tool checks it on thousands of inputs, or all.
3. The tool checks a model's code like anyone's.

A person can check the spec in seconds, so trust sits there. **Agents make the strong rungs cheap:** the checker rejects a wrong proof, so a bad draft costs a retry. AlphaProof works this way: a model writes Lean proofs, and Lean checks them.

02 3 OF 3 TESTS PASSED, AND 127.0.0.1 GOT THROUGH

SERVER-SIDE REQUEST FORGERY

A service that fetches user URLs must refuse private hosts. This guard reads four decimal numbers with `int()`:

```
def is_blocked(
    host: str,
) -> bool:
    """True if the guard should refuse to connect to this host."""
    numbers = parse_ipv4(host)
    if numbers is None:
        return False  # not an IP address, so treat it as a hostname
    a, b = numbers[0], numbers[1]
    if a == 127 or a == 10:
        return True
    if a == 172 and 16 <= b <= 32:
        return True
    return a == 192 and b == 168
```

Bug 1 is a typo: the 172 range ends at 31, not 32.

Bug 2: the OS parses with `inet_aton`, which takes octal, hex and one- to four-part forms:

HOST	GUARD BLOCKS?	OS READS
127.0.0.1	yes	127.0.0.1
0177.0.0.1	no	127.0.0.1
0x7f.0.0.1	no	127.0.0.1
127.1	no	127.0.0.1
2130706433	no	127.0.0.1
10.0	no	10.0.0.0
172.32.0.1	yes	public

Measured with Python's `socket.inet_aton` on Linux. OWASP lists these spellings as standard SSRF bypasses.

3 of 3
example tests pass on the guard with both bugs

10 of 20
runs in which a plain random generator found the bypass

65,536
inputs a 0.24-second loop checks to prove the range

2^{32}
addresses Z3 and Lean cover without running one

03 SIX RUNGS, EACH A STRONGER CHECK

1. **Example tests** — the inputs the author thought of — TESTING
2. **Property-based (Hypothesis)** — generated inputs against one rule — TESTING
3. **Exhaustive enumeration** — every input in a small, finite set — TESTING
4. **State-machine testing** — random sequences, a rule after each step — TESTING
5. **Solver (Z3)** — reasons about every input at once — FORMAL
6. **Proof (Lean)** — a kernel checks every proof step — FORMAL

Rungs 2 to 4 still run the real code. Rungs 5 and 6 reason about a model of it.

04 NO RUNG CAUGHT EVERY BUG

RUNG	TYPO 172.32	OCTAL, HEX	SHORT FORMS	DNS SWITCH
1 Example	miss	miss	miss	n/a
2 Property	miss	miss	10.0	n/a
3 Exhaustive	found	6 of 768	miss	n/a
4 State	n/a	n/a	n/a	found
5 Solver	found	not in model	not in model	n/a
6 Proof	refused	not in model	not in model	n/a

"Miss": the first failure reported was another bug, or none. "Not in model": the formula left parsing out. "n/a": outside what the rung checks.

05 SIX IDEAS EVERY RUNG IS BUILT FROM

SPECIFICATION
What the code must do, stated without saying how.

PROPERTY
A spec as code that returns true or false for any input.

COUNTEREXAMPLE
An input that breaks the property: proof of a bug.

ORACLE
What you trust for the right answer. Here, `inet_aton`.

INPUT SPACE
Every possible input. Small enough, and a loop proves it.

SHRINKING
Cutting a failing input down to the smallest that fails.

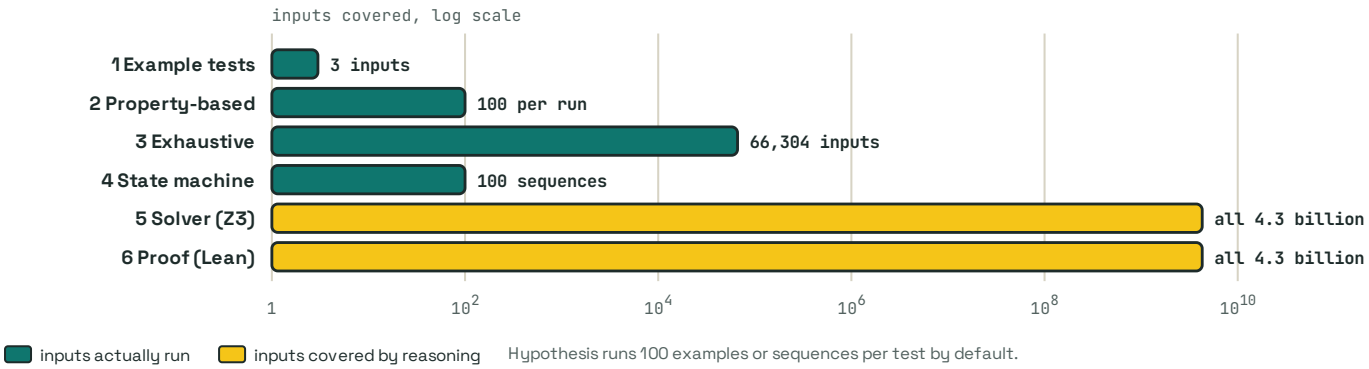
Demonstration code for learning, with bugs left in on purpose: not a production SSRF defense. Counts are from runs on 1 November 2026.

THE EVIDENCE

WHAT EACH RUNG RAN AND FOUND

Real code and real output from the explainer's demo folder. Python 3.14, Hypothesis 6.170.2, Z3, Lean 4.35.0.

06 RUNGS 5 AND 6 COVER 4.3 BILLION ADDRESSES AND RUN NONE



07 RUNG 2: STATE A RULE

```
@given(host=hosts())
def test_guard_agrees_with_the_os(
    host: str,
) -> None:
    assert is_blocked(host) == os_reaches_blocked(host)
```

```
E           host='10.0',
FAILED test_rung2_property.py::test_guard_agrees_with_the_os
```

The strategy builds 1 to 4 parts in decimal, octal or hex. Weighting it toward 10, 127, 172 and 192 took it from 10 of 20 runs to 20 of 20.

08 RUNG 4: TWO LOOKUPS

```
"""Check the address, then connect. It looks the name up twice."""
if is_blocked(dns.resolve(name)):
    return
connections.append(dns.resolve(name))
```

```
E       AssertionError: connected to ['127.0.0.1']
E       state = FetchMachine()
E       state.dns_changes(answer='127.0.0.1')
E       state.fetch_page()
```

Lookup one passes the check; lookup two connects. No input to the guard is wrong, so rungs 1 to 3 miss it. Output trimmed to the steps.

09 RUNG 5: A SOLVER

```
guards = {
    "as written (b <= 32)": And(a == 172, 16 <= b, b <= 32),
    "fixed (b <= 31)": And(a == 172, 16 <= b, b <= 31),
}
for name, guard in guards.items():
    solver = Solver()
    solver.add(octets_valid, guard != spec)  # any disagreement
```

```
as written (b <= 32): counterexample 172.32.255.255
fixed (b <= 31): no counterexample in all 2**32 addresses
```

"No counterexample" means unsatisfiable: a proof over every input, with none run.

10 RUNG 6: A PROOF

```
def guard172 (a b : Nat) : Prop :=
  a = 172 ∧ 16 ≤ b ∧ b ≤ 31

theorem guard_matches_spec (a b c d : Nat)
    (ha : a < 256) (hb : b < 256) (hc : c < 256) (hd : d < 256) :
    guard172 a b ↔ inPrivate172 (toNat a b c d) := by
  unfold guard172 inPrivate172 toNat
  omega
```

```
error: omega could not prove the goal:
  16 ≤ e ≤ 32
where
 e := ↑b
```

Lean accepts the fix. With `b ≤ 32` it refuses and names `b`. Output trimmed.

11 THE FIX: PARSE LIKE THE OS, LOOK UP ONCE

```
"""True if the guard should refuse to connect to this host."""
try:
    address = ipaddress.IPv4Address(socket.inet_aton(host))
except OSError:
    return False  # not an address; the DNS check handles names
if host != str(address):
    return True  # an unusual spelling: refuse rather than guess
return any(address in network for network in BLOCKED)
```

```
"""Look the name up once, check that address, connect to it."""
address = dns.resolve(name)
if is_blocked(address):
    return
connections.append(address)
```

It refuses odd spellings like `010.0.0.1` too, so the property runs one way: if the OS would reach a blocked address, the guard refuses. All three checks pass.

12 WHAT A PASS CANNOT TELL YOU

- **A wrong spec passes every rung.** Drop `169.254.0.0/16` (cloud metadata) and the proof still checks.
- **Generators miss bugs:** 10 of 20 plain runs passed.
- **Models leave things out.** Z3 and Lean never saw the parser.
- **Agents can weaken a check** with an `assume()`, a swapped oracle or a Lean `sorry`. Review those diffs like firewall rules.

13 RUN IT YOURSELF

```
cd demo
uv run --no-project --with hypothesis \
  --with pytest pytest -q
# expect 4 failed, 6 passed
uv run --no-project --with z3-solver \
  python rung5_solver.py
```

Pick a rung: 2 and 3 for most code, 4 when a bug needs steps, 5 and 6 for a small core such as an access check.

Demonstration code for learning, not a production SSRF defense: review and harden before use. Counts are from runs on 1 November 2026.