Scott's Lab

Complete Archive

54 articles

Generated Tue, Sep 1, 5:22 AM EDT

Contents

Stop Feeding Your Secrets to ChatGPT 🔀

https://scottslab.io/posts/scrambler-anonymize-data-llms

TL;DR

Pasting a log full of SSNs into ChatGPT is a compliance problem, not a shortcut. Scrambler swaps PII for realistic fake data before it leaves your browser, then unmasks the AI's reply on the way back: regex matching, no accounts, nothing uploaded. Text mode runs entirely client-side; PDF mode auto-redacts and hands back a clean document.

> 🔀 **Try it now:** scramble.scottslab.io. No signup required.

Stop Feeding Your Secrets to ChatGPT

We've all done it.

You're debugging a customer issue at 11pm. You've got a log file full of names, emails, SSNs, and that one guy's phone number who keeps calling about his account. You need AI help. So you...

...paste the whole thing into ChatGPT.

Congratulations. You just sent someone's social security number to OpenAI's training data. HR would like a word.

The Problem

Public LLMs are incredible tools. But they have a dirty secret: **your data might stick around**. Training data. Logging. That intern at OpenAI who's definitely not reading your prompts (they are).

HIPAA doesn't care that you "really needed help with that regex." Neither does your compliance officer. Or whoever's about to lawyer up.

The Solution: Scrambler

I built Scrambler because I was tired of manually find-replacing "Acme Corp" with "REDACTED" like some kind of caveman.

Scrambler has two modes:

- **Text Mode**: Paste text, mask PII, copy to AI, unmask response (100% browser-side)
- **PDF Mode**: Upload PDFs, auto-redact PII, download clean document

---

📝 Text Masking (Browser-Side)

Here's the deal:

1. Paste your sensitive text
2. Click "Mask"
3. It auto-detects PII and replaces it with fake data
4. Copy the safe version to any AI
5. Paste the AI's response back
6. Click "Unmask" - originals restored

That's it. No accounts. No logins. No data leaving your browser.

What It Catches

The tool uses regex pattern matching (no AI, ironically) to detect:

- **SSNs** - `123-45-6789` → `XXX-XX-1000`
- **Emails** - `john.smith@acme.com` → `marlow.quintrell@bexley-harrow.example`
- **Phone numbers** - `(317) 555-1234` → `(555) 100-1000`
- **IP addresses** - `192.168.1.50` → `192.0.2.1`
- **Dates of birth** - `DOB: 03/15/1985` → `DOB: XX/XX/1950`
- **Medical record numbers** - `MRN: 12345678` → `MRN-100000`
- **Credit card numbers** - `4532-1234-5678-9012` → `XXXX-XXXX-XXXX-1000`

- **Driver's license** - `DL# A1234567` → `DL# DL-100000`
- **Account numbers** - `Account: 9876543` → `ACCT-100000`

Yeah, the fake values don't move much on their own. It returns the same name and the same account number every time you mask. That's on purpose, and it's not laziness. Read on.

Why the Fake Names Look Like That

The obvious move is the names you'd expect: Contoso, Fabrikam, Alex, Smith, private IPs like `10.0.45.12`. Standard placeholder stuff. There's a failure mode hiding in that choice, though, and I didn't love it once I traced it through.

You paste the masked version into an LLM. The LLM writes a response. That response might, on its own, contain the word "Contoso": it's a famous Microsoft sample company, and language models say famous sample company names sometimes, for no reason related to you at all. Now unmask that response. Scrambler finds "Contoso" in the reply, assumes it's the placeholder it planted, and swaps in your real company name. Silently. At the very last step. The tool built to hide your data just leaked it, and there'd be nothing in the output to tell you.

So the replacement values changed. Names now come from pools of invented first names, surnames, and companies that don't exist anywhere (think "Marlow Quintrell" and "Bexley Harrow Ltd") paired with email domains on `.example`, which is a domain reserved by spec for exactly this and will never resolve to a real site. IP addresses use `192.0.2.x`, the documentation range set aside for the same reason. All of it is chosen to read like a normal name or address to a model, while being close to impossible for that model to produce on its own by coincidence.

One more piece worth knowing about: when you paste the AI's response back in and click Unmask, Scrambler tells you exactly how many values it found and restored, and lists any it couldn't locate. If the model reformatted a value, or a collision genuinely happens, you'll see it. The tool won't quietly get it wrong.

Deciding What Gets Masked

Names are the hard case. The tool can't automatically know that "John Smith" is a person and "Main Street" is not (well, it could, but that would require... AI. The irony is not lost on me).

So there's a configuration panel for this:

- **Per-type toggles**: flip SSN, email, phone, IP, DOB, MRN, account, credit card, and driver's license detection on or off individually.
- **Always mask**: a list of your own terms, such as names, company names, codenames, internal hostnames, and project names. Type one in, pick whether it should read back as a Name or a

Company, click Add. Nothing here is pattern-matched: it's an exact list, because no pattern is ever going to catch "Project Nightshade."
- **Never mask**: terms that should stay exactly as-is no matter what. This wins over everything else, including your own always-mask list, in case the two ever conflict.
- **Presets**: *Everything* turns every pattern on. *Technical documentation* drops DOB and IP addresses, because a config file is full of IPs you actually want to keep readable. *Medical records* turns everything on and puts extra emphasis on DOB and MRN.

One deliberate choice: your Always-mask and Never-mask lists are **not** saved to your browser by default. Those lists usually contain the exact names and codenames you're trying to keep quiet, and auto-saving them would just create a second, unprotected copy of the secret sitting in your browser's storage. There's a "Remember my custom terms" checkbox if you want them to persist across visits. It's off unless you turn it on. The toggles and your chosen preset save automatically either way, since there's nothing sensitive about knowing you like the Medical Records preset.

📄 PDF Redaction

Sometimes you have an entire document that needs to be sanitized before sharing. Maybe it's:

- A medical record you need to send to a consultant
- A police report for a case study
- Financial statements for an audit
- Any document with scattered PII

Here's how it works:

1. Click the **"PDF Redact"** tab
2. **Choose your redaction style:**
   - `[REDACTED]`: Clean text labels
   - `[_____]`: A redaction bar, if you want it to look like the real thing
3. **Drag & drop your PDF** (or click to upload)
4. **Review what was found**: See exactly what PII was detected
5. **Download the clean PDF**: All PII replaced, document structure preserved

How It's Different From Text Mode

- **Server-side processing**: PDFs are too complex for browser-only
- **No disk, ever**: the file never touches a hard drive at any point, not even briefly
- **No storage**: no database, no logs, no traces
- **Layout preserved**: headers, paragraphs, structure stays intact

- **Not identical detection**: the two engines are separate code and don't catch exactly the same things (below)

What Gets Redacted

Mostly the same patterns as text mode: SSNs, emails, phone numbers, IP addresses, credit cards, DOB (with context like "Date of Birth:"), medical record numbers, and account/patient IDs. One gap worth knowing: driver's license numbers are only caught in Text Mode. The PDF engine doesn't have that pattern yet. Don't assume the two modes are interchangeable. Audit whichever one you're trusting.

Example

Original PDF text:

```
PATIENT INTAKE FORM

Patient: John Smith
SSN: 123-45-6789
DOB: 03/15/1985
Phone: 317-555-8421
Email: john.smith@acmecorp.com
```

After redaction (text style):

```
PATIENT INTAKE FORM

Patient: John Smith
SSN: [REDACTED]
[DOB REDACTED]
Phone: [REDACTED]
Email: [REDACTED]
```

Note the date line: the PDF side swallows the "DOB:" label along with the date, where the browser side leaves it standing. Same category, two engines, two behaviours.

Notice "John Smith" is still sitting right there. That's not a bug. It's the same limitation as text mode. A name isn't a pattern the PDF engine can search for on its own. If you're redacting documents that always have the same patient or client name in them, add that name to the always-mask list before you upload, and it'll get caught with everything else.

The Privacy Part (It's Actually Private)

Text Mode

Everything runs in your browser.

- No server processing
- No data transmitted anywhere
- No accounts or cookies
- No logging

Open DevTools. Watch the Network tab. Nothing gets sent. Your HIPAA officer can sleep soundly.

PDF Mode

Ephemeral server processing, and I mean actually ephemeral.

- The file is held in memory only: read into RAM, streamed to the redaction process over its input and output pipes, streamed back. It is never written to a disk anywhere in the pipeline.
- No permanent storage
- No database entries
- Max 5 concurrent sessions (DoS protection)
- 10MB file size limit

The PDF has to hit a server (PDFs are too complex for a browser to redact on its own), but "hit a server" doesn't mean "gets saved." It means the bytes pass through memory and come back out the other side.

Scanned PDFs Are the One Place You Can Get Fooled

Here's the honest gap, and it matters more than any of the others: a scanned document is a picture of text, not text. There's no text layer for the pattern matching to search: it's pixels. So Scrambler finds nothing, because there's nothing there *to* find, and hands the file back looking processed.

It will tell you which pages it couldn't read. But a fully scanned document can come back reporting zero detections and still have every SSN in it sitting untouched in the image data. If a document might be scanned, don't trust a clean report. Check it yourself first.

What This Doesn't Catch

Pattern matching has edges, and I'd rather list them than have you find out the hard way:

- No IPv6 addresses
- No international or non-U.S.-style phone numbers

- No passport numbers or national ID numbers, from any country
- Day-first dates (`31/12/1990`) and standalone dates without a "DOB" or "born" nearby are missed. The tool only catches a birth date when it's flagged by a keyword
- The driver's license pattern is loose enough that it also grabs invoice numbers, ticket IDs, and part numbers that happen to look similar
- Any 16-digit number in groups of four gets flagged as a credit card, whether or not it actually is one. There's no real validation, just shape-matching

None of this is a reason not to use the tool. It's a reason to look at what it found before you hit send.

Open Source, So You Don't Have to Take My Word for It

The whole thing is on GitHub: schlangens/scrambler, MIT licensed. Every claim above about what runs where and what gets stored is something you can go verify yourself instead of trusting a blog post.

There are 34 automated tests, all passing. Two of them do the paranoid thing directly: one snapshots the filesystem before and after a PDF redaction run and asserts nothing new showed up anywhere, and another pulls the text back out of a redacted PDF and confirms the sensitive strings are actually gone: not covered by a black rectangle sitting on top of readable text underneath, which is a mistake other redaction tools have made.

Run It With Zero Network Access

There's also a single-file offline build: scrambler-offline.html. Save it, disconnect from the internet, open it straight from your own machine. It does text masking only. PDF redaction needs the server, so that part isn't in the offline build.

The build process refuses to produce this file if it contains any network call at all: a fetch, an XMLHttpRequest, a script tag or stylesheet pointing somewhere else, even a stray `http://` in a comment. And it's not just checked once: the automated pipeline regenerates the file on every change and fails the build if the committed copy has drifted from what the source actually produces. I downloaded the live file and checked it myself: zero occurrences of any of those.

Real World Example

You have (with "John Smith" already sitting in your always-mask list, because the patterns will never find him on their own):

```
Patient John Smith (DOB: 03/15/1985, SSN: 123-45-6789)
called from (317) 555-0199 regarding prescription refill.
Contact: john.smith@acme.com
Account: 98765432
```

Scrambler gives you:

```
Patient Marlow Quintrell (DOB: XX/XX/1950, SSN: XXX-XX-1000)
called from (555) 100-1000 regarding prescription refill.
Contact: marlow.quintrell@bexley-harrow.example
ACCT-100000
```

Note the account line: the pattern eats the "Account:" label along with the number, because the label is what tells it the digits are an account number in the first place.

Paste that into ChatGPT. Ask your question. Get your answer.

Paste the answer back into Scrambler. Click Unmask. "Marlow Quintrell" becomes "John Smith" again, and you get a count of exactly what came back.

When You Need This

- Healthcare workers using AI for documentation help
- Support teams debugging customer issues
- Developers with production logs
- Legal teams sanitizing documents for case studies
- Anyone handling financial data
- Literally anyone who values not getting fired

When You Don't Need This

- Your grocery list (unless you're buying... suspicious groceries?)
- That fanfic you're writing (no judgment)
- Anything already public

Try It

scramble.scottslab.io

No signup. No payment. No tracking. Just paste and go.

Built with JavaScript, paranoia, and an unreasonable number of regex patterns.

Infrastructure Security Hardening: Firewall Audit, IDS Tuning, and SIEM Alert Management

Wed, Feb 11, 9:20 AM EST · https://scottslab.io/posts/infrastructure-security-hardening-firewall-ids-siem

TL;DR

Audited 67 pfSense rules and every NAT forward, then killed 21 WAN pass rules and 6 forwards that had no business being open. Rewrote the Zeek C2 whitelists and cut false positives by roughly 90%, then fixed a batch of Wazuh rules that were quietly broken by XML errors and colliding rule IDs. Also chased down a DNS redirect loop on the IoT VLAN and trimmed the agent inventory from 21 to 16.

Published: February 2026 **Category:** Security Operations **Reading Time:** 15 minutes

---

Executive Summary

- Audited 67 firewall rules and all NAT port forwards; disabled 21 dangerous WAN rules and 6 risky NAT forwards
- Reduced WAN attack surface from 21+ open pass rules to only necessary NAT-associated rules
- Rewrote Zeek C2 detection whitelists, reducing false positive alerts by ~90%
- Fixed broken Wazuh SIEM rules including XML syntax errors, rule ID collisions, and 10 new suppression rules for known infrastructure traffic
- Resolved a DNS redirect loop on IoT VLAN that was generating persistent firewall errors
- Cleaned up SIEM agent inventory from 21 to 16 active agents

Goal

Problem: Over time, my perimeter firewall had accumulated dangerous rules through quick-fix troubleshooting sessions. "Pass any to any" rules, catch-all NAT forwards, and exposed management interfaces were sitting in the ruleset alongside legitimate traffic rules. Meanwhile, my network IDS was flooding alerts from legitimate infrastructure traffic (cloud tunnels, VPN relays, CDN connections), and the SIEM had broken rules that either weren't firing or were generating noise from known-good sources.

Why it mattered: A firewall with 21 overly permissive rules isn't a firewall. It's a suggestion. When your IDS generates hundreds of false positives daily, real threats get lost in the noise. Security tools that cry wolf eventually get ignored, which defeats their entire purpose.

Scope and Constraints

In Scope

- Perimeter firewall rule audit and hardening
- NAT port forward review and cleanup
- Zeek C2 detection script whitelist tuning
- Wazuh SIEM rule syntax fixes and alert suppression
- SIEM agent cleanup (decommissioned hosts)

Out of Scope

- Internal VLAN firewall rules (LAN-to-LAN traffic)
- Endpoint detection and response (EDR) tuning
- Application-layer security (WAF rules)
- Backup and disaster recovery validation

Key Constraints

- **Zero downtime requirement** - Production services must remain accessible throughout changes
- **No service exposure changes** - Cloud tunnel architecture handles most service exposure; I'm hardening, not redesigning
- **Preserve forensic capability** - Suppression rules must not hide actual threats, only known-good infrastructure traffic

---

Tools and References

Tool	Role in This Project
pfSense	Perimeter firewall - rule audit, NAT cleanup, DNS redirect fixes
Zeek	Network IDS - C2 detection via long-lived connections, DNS tunneling detection, certificate anomalies
Wazuh	SIEM - alert aggregation, rule management, agent inventory
Cloudflare Tunnels	Service exposure - outbound-only tunnels eliminate need for inbound NAT
Tailscale	Mesh VPN - secure internal access, DERP relay traffic appears in IDS
Pi-hole	DNS filtering - dedicated appliance replaced non-functional firewall package

References:

- pfSense Firewall Rule Best Practices
- Zeek Intelligence Framework
- Wazuh Custom Rules Syntax
- MITRE ATT&CK Network Defense

Approach

Phase 1: Firewall Rule Audit

What I did: Exported all 67 firewall rules and NAT entries. Reviewed each rule against three criteria: (1) Is there a documented business purpose? (2) Is the scope as narrow as possible? (3) Is the rule still needed?

Why: Firewall rules accumulate like sediment. Every "temporary" troubleshooting rule that never gets removed expands your attack surface. You can't secure what you haven't inventoried.

Phase 2: Dangerous Rule Remediation

What I did: Disabled 21 WAN firewall rules including pass-any rules, catch-all NAT passes, exposed management interfaces (SSH, remote console), and stale EasyRule entries from years-old troubleshooting sessions. Disabled 6 NAT forwards that were either catch-all rules or exposed sensitive services.

Why: A single "pass any to any" rule negates every other rule in your firewall. Exposed management interfaces are prime targets for credential stuffing and exploitation.

Phase 3: NAT Cleanup and Fixes

What I did: Replaced a dangerous "forward everything" NAT rule with a specific rule for the media server on its correct port. Fixed a DNS redirect NAT on the IoT VLAN where source/destination logic was inverted, creating a DNS resolution loop. Corrected a host alias pointing to the wrong IP.

Why: Catch-all NAT rules are the "sudo rm -rf" of network security: they solve the immediate problem by creating a much bigger one. The DNS loop was generating persistent `pf` errors on every firewall reload.

Phase 4: Zeek C2 Detection Tuning

What I did: Rewrote the C2 detection script with comprehensive whitelists covering: cloud tunnel provider IP ranges, VPN relay server IPs, VPN coordination server addresses, legitimate long-domain services (CDNs, analytics, streaming services), and reverse DNS lookup sources.

Why: Zeek was flagging every Cloudflare tunnel connection as a potential C2 channel (long-lived encrypted connection to external IP). VPN relay traffic triggered the same alerts. Without whitelists, the legitimate traffic volume made the alerts useless.

Phase 5: Wazuh SIEM Rule Fixes

What I did: Fixed XML syntax errors in custom rules, resolved a rule ID collision where a duplicate ID was silently overriding another rule, added 6 suppression rules for pfSense alerts from known infrastructure, and added 4 suppression rules for Zeek alerts from legitimate tunnel/DNS traffic.

Why: Wazuh won't load rules with XML errors, but duplicate rule IDs fail silently: the second definition just overwrites the first. Alert fatigue from known-good traffic trains analysts to ignore the console.

Phase 6: Agent Cleanup

What I did: Identified and removed 5 decommissioned SIEM agents (hosts no longer in service). Reconnected 1 agent that had lost connectivity.

Why: Stale agents clutter the inventory and can mask coverage gaps. If you think you have 21 monitored hosts but only 16 are actually reporting, you have blind spots.

Implementation Notes

Firewall Rule Disable Pattern (Sanitized)

```
# Export current rules for backup
<FIREWALL_CLI> config backup > /backup/fw-rules-$(date +%Y%m%d).xml

# Disable rule by tracker ID (pfSense uses tracker IDs, not rule numbers)
# This is done via GUI or config.xml edit - CLI shown for illustration
<FIREWALL_CLI> rule disable --tracker <RULE_TRACKER_ID>

# Reload firewall
<FIREWALL_CLI> filter reload
```

Assumption: Your firewall supports rule disable (not just delete) to allow easy rollback.

Zeek Whitelist Structure (Sanitized)

```
# c2-whitelist.zeek - Legitimate long-connection sources

module C2Detection;

export {
    # Cloud tunnel provider subnets
    const cloud_tunnel_nets: set[subnet] = {
        <CLOUD_PROVIDER_CIDR_1>,
        <CLOUD_PROVIDER_CIDR_2>,
        <CLOUD_PROVIDER_CIDR_3>,
    } &redef;

    # VPN relay IPs (mesh VPN DERP servers)
    const vpn_relay_ips: set[addr] = {
        <VPN_RELAY_IP_1>,
        <VPN_RELAY_IP_2>,
        <VPN_RELAY_IP_3>,
    } &redef;

    # Legitimate long-domain services (CDNs, analytics)
    const legit_long_domains: pattern = /\.
(cloudfront|akamai|fastly|analytics|telemetry)\./;
}

# Skip alerting if destination matches whitelist
event connection_state_remove(c: connection) {
    if (c$id$resp_h in cloud_tunnel_nets) return;
    if (c$id$resp_h in vpn_relay_ips) return;
    # ... existing C2 detection logic
}
```

Wazuh Suppression Rule Pattern (Sanitized)

```
<!-- Suppress pfSense alerts from known cloud provider subnet -->
<rule id="100201" level="0">
    <if_sid>87701</if_sid>
    <srcip><CLOUD_PROVIDER_CIDR></srcip>
    <description>Suppress: Known cloud tunnel provider traffic</description>
</rule>

<!-- WRONG: Comma-separated CIDRs not supported -->
<!-- <srcip><CIDR_1>,<CIDR_2></srcip> -->

<!-- CORRECT: One rule per subnet -->
<rule id="100202" level="0">
    <if_sid>87701</if_sid>
    <srcip><CLOUD_PROVIDER_CIDR_2></srcip>
    <description>Suppress: Known cloud tunnel provider traffic (range 2)
</description>
</rule>

<!-- Field matching requires explicit pcre2 type for regex -->
<rule id="100210" level="0">
    <if_sid>87900</if_sid>
    <field name="dns.query" type="pcre2">.*\.
(cloudprovider|cdnprovider)\.com$</field>
    <description>Suppress: Known CDN DNS queries</description>
</rule>
```

DNS Redirect Fix (Sanitized)

```
# BEFORE (inverted logic - caused DNS loop)
NAT Rule: Redirect DNS
  Source: <IOT_VLAN_SUBNET>
  Destination: <DNS_SERVER_IP>  # WRONG - was catching replies
  Redirect to: <DNS_SINKHOLE_IP>

# AFTER (correct logic)
NAT Rule: Redirect DNS
  Source: <IOT_VLAN_SUBNET>
  Destination: !<DNS_SINKHOLE_IP>  # NOT the sinkhole
  Destination Port: 53
  Redirect to: <DNS_SINKHOLE_IP>
```

Validation and Evidence

Signals That Proved It Worked

Check	Before	After
WAN pass rules	21 overly permissive	3 NAT-associated only
NAT port forwards	9 (including catch-all)	4 specific rules
Zeek C2 alerts/day	~150 (mostly false positives)	~15 (legitimate investigation targets)
Wazuh rule validation	3 XML errors, 1 ID collision	Clean load, no errors
Active SIEM agents	21 (5 stale)	16 (all reporting)
pfSense filter reload	DNS redirect errors	Clean reload

Validation Commands (Sanitized)

```
# Verify Wazuh rules load cleanly
<WAZUH_BIN>/wazuh-analysisd -t
# Expected: No errors

# Check Zeek script syntax
<ZEEK_BIN>/zeek -a <POLICY_DIR>/c2-whitelist.zeek
# Expected: No errors

# Verify firewall rule count
<FIREWALL_CLI> rule count --interface WAN --action pass
# Expected: Reduced count

# Test service accessibility through cloud tunnel
curl -I https://<SERVICE_DOMAIN>
# Expected: 200 OK (services still accessible)
```

Results

Metric	Outcome
WAN Attack Surface	Reduced from 21+ open pass rules to 3 NAT-associated rules
Dangerous NAT Forwards	Eliminated 6 (catch-all, SSH, SMTP, remote console)
Zeek False Positives	Reduced ~90% (cloud tunnel, VPN, CDN traffic no longer triggers)
Wazuh Alert Noise	Infrastructure IP alerts suppressed (10 new rules)
DNS Loop	Fixed - IoT VLAN DNS redirect now functions correctly
SIEM Coverage	Cleaned from 21 to 16 verified active agents
Service Availability	100% - all services remained accessible throughout

---

What I Learned

1. **Cloud tunnels obsolete most inbound NAT.** Services connect outbound through encrypted tunnels. You don't need to punch holes in your firewall when your services are dialing out to the tunnel provider.

2. **Catch-all NAT rules are technical debt with interest.** Every "forward everything to this host" rule added during troubleshooting becomes a permanent attack surface expansion. They're easy to add and easy to forget.
3. **Quarterly firewall audits are non-negotiable.** Rules from decommissioned services persist forever. If you're not actively pruning, you're passively expanding your attack surface.
4. **Wazuh's `<srcip>` tag is single-value only.** No comma-separated CIDR lists, no pipe alternation. One rule per subnet. The documentation doesn't make this obvious.
5. **Wazuh's `<field>` tag defaults to OS_Match, not regex.** If you want regex matching, you MUST specify `type="pcre2"`. Your patterns will silently fail otherwise.
6. **Duplicate Wazuh rule IDs fail silently.** The second rule just overwrites the first. No error, no warning. You can have detection gaps and never know it.
7. **Zeek whitelists need both IP and domain exclusions.** VPN relays trigger long-connection alerts (IP-based). CDN/cloud services trigger DNS tunneling alerts (domain-based). You need both.
8. **VPN mesh relay IPs are dynamic and distributed.** Services like Tailscale host DERP relays across various cloud providers. You can't whitelist them with a single broad subnet. You need to track individual relay IPs or resolve them dynamically.
9. **FreeBSD uses `sed -i ''` not `sed -i .`** Small difference, critical when scripting changes on pfSense. The GNU vs BSD sed distinction bites everyone eventually.
10. **Orphaned flags create invalid firewall syntax.** Removing a NAT source address without also removing the "not" flag creates `from !` with no address. The firewall won't load the ruleset.

What I Would Improve Next

P0 (Do This Week)

- **Automated firewall rule audit** - Monthly script that flags unused rules (no hits in 30 days) and dangerous patterns (any-any, wide open ports)
- **Media server investigation** - Port not listening on expected port; service may need restart or troubleshooting (pre-existing issue, not caused by these changes)

P1 (Do This Month)

- **Wazuh active response for port scans** - Auto-block IPs at firewall when Zeek detects repeated scanning behavior

- **Firewall block trends dashboard** - Wazuh visualization of top blocked sources, ports, frequency over time
- **Dynamic Tailscale DERP whitelist** - Cron job to resolve `derp*.tailscale.com` and update Zeek whitelist automatically

P2 (Do This Quarter)

- **Split DNS implementation** - Separate internal/external resolution to avoid hairpin NAT issues
- **GeoIP firewall rules** - Block inbound from countries with no business purpose (reduces noise and attack surface)
- **Alert escalation playbook** - Document which Zeek/Wazuh alerts require immediate action vs. weekly review
- **Firewall rule documentation** - Spreadsheet with owner, purpose, and last-reviewed date for every rule

---

Common Failure Modes

1. **"I disabled a rule and now X is broken"** - Cloud tunnel services should handle most access. If something breaks, check if it was relying on a NAT forward that should have been migrated to tunnel exposure.

2. **"Zeek alerts came back after whitelist update"** - The whitelist file may not have been deployed to all Zeek workers, or the script has a syntax error preventing load. Check `zeekctl status` and script validation.

3. **"Wazuh rules aren't suppressing alerts"** - Verify rule ID is unique, XML syntax is valid, and `<srcip>` contains a single value (not a list). Run `wazuh-analysisd -t` to validate.

4. **"pfSense shows filter reload errors"** - Look for orphaned flags in NAT rules (negation without address), invalid alias references, or circular DNS redirect rules. Check `/tmp/rules.debug`.

5. **"SIEM agent shows disconnected but host is running"** - Agent may have lost connectivity after IP change or firewall rule change. Restart the agent service and verify it can reach the Wazuh manager.

Security Considerations

Attack Surface Reduction

- Eliminated direct WAN exposure for SSH, SMTP, and remote console services

- Removed catch-all NAT rules that effectively bypassed firewall policy
- Reduced permissive pass rules from 21 to 3 (NAT-associated only)

Detection Integrity

- Suppression rules are scoped to specific source IPs/subnets, not blanket disables
- Rule ID collision fix restored detection coverage that was silently missing
- Agent cleanup ensures SIEM coverage matches actual infrastructure

Defense in Depth

- Cloud tunnels provide application-layer authentication even if network layer is compromised
- VPN mesh ensures internal traffic doesn't traverse untrusted networks
- DNS filtering at dedicated appliance provides redundant protection if firewall DNS package fails

Least Privilege

- NAT forwards now specify exact ports, not port ranges or "all"
- Firewall rules use specific host aliases, not subnet-wide permissions where possible

---

Runbook

If Services Become Inaccessible

1. **Check cloud tunnel status** - Most services use outbound tunnels; verify tunnel daemon is running
2. **Review recent firewall changes** - Check if a required NAT rule was disabled; re-enable if needed
3. **Verify DNS resolution** - IoT VLAN DNS redirect issue could recur if misconfigured
4. **Check firewall filter log** - Look for blocks on the affected traffic in pfSense logs
5. **Test from multiple networks** - Issue may be client-side or ISP-related, not firewall

If Alert Volume Spikes

1. **Check for new infrastructure** - New service deployments may not be whitelisted yet
2. **Verify whitelist deployment** - Zeek whitelists must be on all workers; Wazuh rules must pass validation
3. **Look for actual attack** - Not all spikes are false positives; correlate with firewall blocks
4. **Check for rule ID collision** - New custom rules may have duplicated an existing ID
5. **Review suppression rule scope** - Ensure rules match on correct fields (srcip, dstip, field name)

Rollback Procedure

```
# Restore firewall config from backup
<FIREWALL_CLI> config restore /backup/fw-rules-<DATE>.xml
<FIREWALL_CLI> filter reload

# Restore previous Zeek whitelist
cp /backup/c2-whitelist-<DATE>.zeek <POLICY_DIR>/c2-whitelist.zeek
<ZEEK_CTL> deploy

# Restore previous Wazuh rules
cp /backup/local_rules-<DATE>.xml <WAZUH_DIR>/etc/rules/local_rules.xml
<WAZUH_CTL> -R
```

Appendix

Glossary

Term	Definition
NAT (Network Address Translation)	Translates private IPs to public IPs; port forwarding is a type of NAT
IDS (Intrusion Detection System)	Monitors network traffic for suspicious patterns
SIEM (Security Information and Event Management)	Aggregates and correlates security logs from multiple sources
C2 (Command and Control)	Communication channel between malware and attacker infrastructure
DERP (Designated Encrypted Relay for Packets)	Tailscale's relay servers for NAT traversal
DGA (Domain Generation Algorithm)	Malware technique that generates pseudo-random domains to evade blocking
Hairpin NAT	Traffic that exits and re-enters the same interface; often causes routing issues

MITRE ATT&CK Mapping

Technique ID	Name	How I Addressed It
T1190	Exploit Public-Facing Application	Disabled catch-all NAT rules that exposed internal services to WAN
T1133	External Remote Services	Removed SSH and remote console exposure on WAN interface
T1071	Application Layer Protocol	Zeek C2 detection via long-lived connection analysis (with whitelist tuning)
T1071.004	DNS	DNS tunneling and exfiltration detection via Zeek (with CDN/cloud whitelist)
T1046	Network Service Discovery	Port scan detection via Zeek connection analysis
T1568.002	Domain Generation Algorithms	DGA detection in Zeek with false positive tuning for legitimate long domains
T1573.002	Encrypted Channel: Asymmetric Cryptography	Self-signed certificate detection for potential C2 identification
T1021	Remote Services	Lateral movement detection via honeypot monitoring in Wazuh

Detection Logic Summary

Detection	Tool	Logic	Whitelist Applied
Long-lived connections	Zeek	Connections >30min to external IPs	Cloud tunnel subnets, VPN relay IPs
DNS tunneling	Zeek	High query volume, long subdomains, TXT record abuse	CDN domains, cloud API domains
DGA domains	Zeek	High entropy domain names, rapid NXDOMAIN responses	Analytics/telemetry services
Self-signed certs	Zeek	Certificates not in known CA list	Internal PKI, development domains
Firewall blocks	Wazuh	pfSense block events aggregated	Known scanner IPs, cloud provider ranges

Detection	Tool	Logic	Whitelist Applied
Authentication failures	Wazuh	Repeated failed logins across services	None (all auth failures logged)
Port scans	Zeek + Wazuh	Connection attempts to multiple ports from single source	Internal vulnerability scanners

---

Artifacts Produced

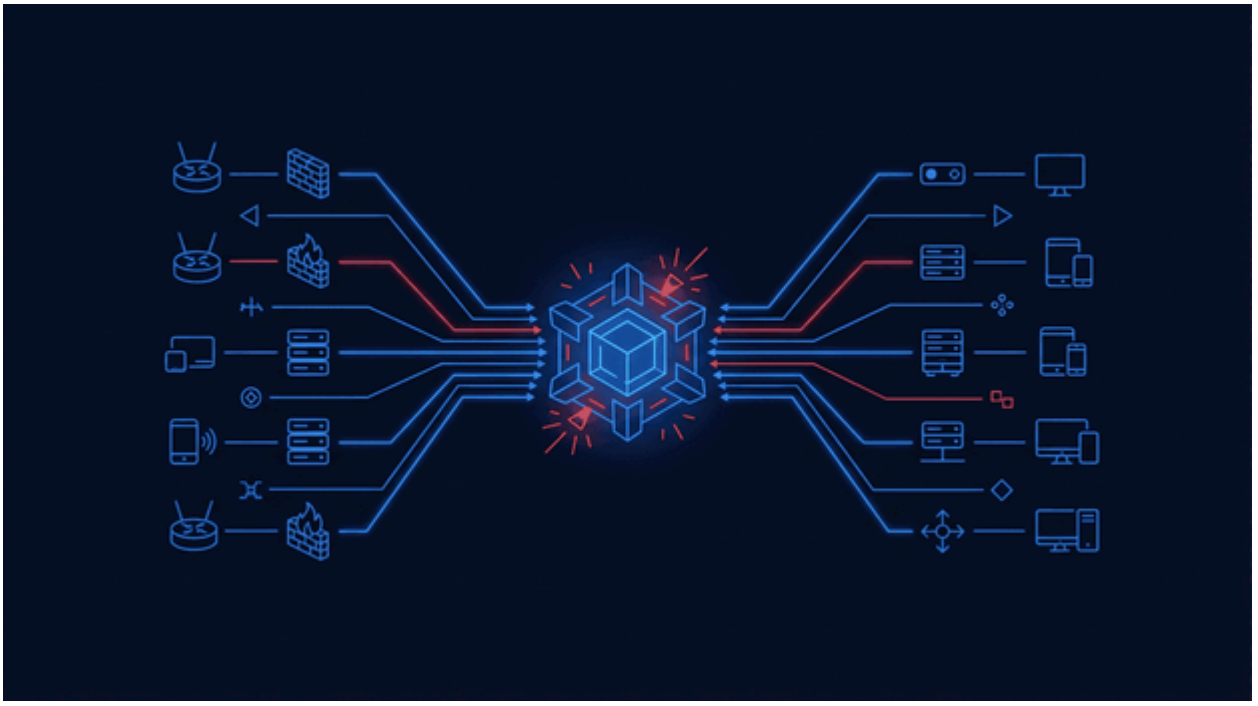
- **Config: Firewall Backup** - Pre-change backup of all firewall rules and NAT entries
- **Script: Zeek C2 Whitelist** - Comprehensive whitelist for cloud tunnels, VPN relays, and CDN traffic
- **Rules: Wazuh pfSense Suppressions** - 6 rules suppressing known infrastructure firewall alerts
- **Rules: Wazuh Zeek Suppressions** - 4 rules suppressing known-good Zeek alerts
- **Documentation: Disabled Rule Inventory** - List of all disabled rules with original purpose and disable rationale
- **Documentation: NAT Cleanup Log** - Before/after state of all NAT port forwards

---

Questions or war stories about your own firewall archaeology expeditions? Find me in the usual places.

Building a Home Lab SIEM: Wazuh Deployment with Custom Detection Rules

Tue, Mar 10, 9:45 AM EDT · https://scottslab.io/posts/home-lab-siem-wazuh-custom-detection

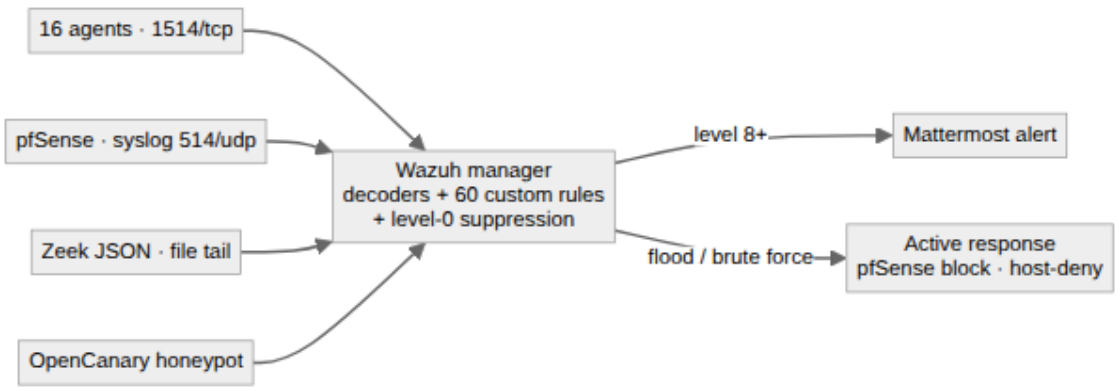


TL;DR

Sixteen hosts were each logging into the void, so everything now lands in one Wazuh 4.14.2 manager: agents, pfSense syslog, Zeek JSON, and a honeypot. About 60 custom rules do the detecting, but the level-0 suppression rules are what made the console usable: hundreds of alerts a day down to under twenty. Two rules hit back through active response, and anything level 8 or above reaches my phone through Mattermost.

Sixteen hosts, and until last winter every one of them logged into the void. The firewall kept its own block log, Zeek wrote connection records nobody read, the honeypot fired into a file, and each server rotated its auth log on a schedule I'd half forgotten. Individually, fine. Together, useless. Attackers don't announce themselves on one box. They leave a smudge on the firewall, a strange DNS query at the resolver, a failed login on the mail server, and unless something is stitching those together, the smudge is all you ever see.

So I stood up Wazuh 4.14.2 on a dedicated VM and pointed everything at it. The goal was never a pretty dashboard. It was correlation and a response loop: see the flood, block the flood, tell me about it, before I've noticed anything is wrong.



How the logs get in

Three ingestion paths, because the sources don't agree on anything. The sixteen agents talk to the manager on 1514/tcp and hand over the richest data: file integrity, command history, auth events. pfSense can't run an agent, so it ships its filter log over syslog on 514/udp. And Zeek writes JSON to disk, which the manager tails as a local file. One SIEM, three front doors.

The firewall log is the awkward one. It arrives as a comma-separated line, not syslog, not JSON, so nothing pulls fields out of it by default. That needs a custom decoder:

```
<decoder name="pfsense-filterlog">
  <prematch>filterlog:</prematch>
</decoder>

<decoder name="pfsense-filterlog-fields">
  <parent>pfsense-filterlog</parent>
  <regex type="pcre2">filterlog:\s*(\d+),.*?,.*?,.*?,(\w+),(\w+),
(\w+),.*?,.*?,.*?,.*?,.*?,(\d+\.\d+\.\d+\.\d+),(\d+\.\d+\.\d+\.\d+),(\d+),(\d+)
</regex>

<order>rule_number,interface,action,direction,srcip,dstip,srcport,dstport</order
>
</decoder>
```

Zeek is the opposite problem. Its logs are already JSON, but `conn.log`, `dns.log`, and `ssl.log` all look identical to Wazuh's built-in JSON decoder, and you can't override that decoder: it grabs anything starting with `{`. So instead of decoding by source, I tell the log types apart in the rules, by which field is

present: `conn_state` means a connection record, `qtype_name` means DNS, `validation_status` means a certificate.

The rules that earn their keep

Sixty-some custom rules ended up spread across three files (Zeek, pfSense, and the honeypot), but they all do one of two things: spot a bad shape once, or spot a bad rhythm over time.

The rhythm rules are the ones I lean on. Port scans, brute force, and floods all have a frequency signature, and Wazuh's frequency/timeframe counters catch them cleanly. The firewall flood rule is the clearest example: count blocks from a single source, and when fifty land in sixty seconds, escalate and hand it to active response:

```
<rule id="100206" level="12" frequency="50" timeframe="60">
  <if_matched_sid>100200</if_matched_sid>
  <same_source_ip />
  <description>pfSense: Flood pattern (50 blocks in 60s) - triggering active
response</description>
  <group>firewall_flood,active_response</group>
</rule>
```

The honeypot rules are the simplest and the highest severity, because the logic is trivial: nothing has a legitimate reason to touch the OpenCanary box. Any connection is level 12, SSH to it is level 13, and three services probed inside five minutes is a level 14. That's not a stray packet, that's someone walking the room.

Cutting the false positives

The first day live, the console was unusable. Hundreds of alerts, nearly all of them my own infrastructure being itself. Cloudflare tunnels look exactly like a long-lived C2 channel. Tailscale's DERP relays trip the same wire. CDN subdomains read like DGA output. None of it is an attack, and all of it drowns the things that are.

The fix is unglamorous: level-0 suppression rules for the known-good. A DNS-tunneling rule fires on any query over fifty characters; a paired level-0 rule right behind it drops the ones that end in a real CDN:

```
<rule id="100115" level="8">
  <if_sid>100110</if_sid>
  <field name="query" type="pcre2">^[a-zA-Z0-9-]{50,}\.</field>
  <description>Zeek: Possible DNS tunneling - query over 50 chars</description>
</rule>

<rule id="100116" level="0">
  <if_sid>100115</if_sid>
  <field name="query" type="pcre2">\.(cloudfront|akamaized|fastly|azureedge)\.
</field>
  <description>Suppress: known CDN long subdomain</description>
</rule>
```

Repeated for each noisy source, that took the console from hundreds of alerts a day to under twenty. The suppression rules are as much of the detection engineering as the detections themselves. A console full of false alarms is a console you stop reading, and then the real alert scrolls past unseen.

Letting it hit back

Detection without response is just note-taking, so two rules don't stop at alerting. The flood rule above calls a firewall-block script that drops the source into pfSense's block table for a day. And Wazuh's built-in SSH brute-force rule (5763) triggers host-deny on the internet-facing agent, banning the attacker for forty-five days:

```
<active-response>
  <command>firewall-block</command>
  <location>local</location>
  <rules_id>100206</rules_id>
  <timeout>86400</timeout>
</active-response>
```

The one thing I'd stress: the active-response script receives the alert as JSON on stdin, and you have to validate the source IP before you act on it. A malformed log that slips a bad value into that field is a self-inflicted block of the wrong address. Parse defensively. Everything at level 8 and up also goes to a Mattermost webhook, so the critical events reach my phone in a few seconds whether or not I'm looking at the dashboard.

What bit me, and what's next

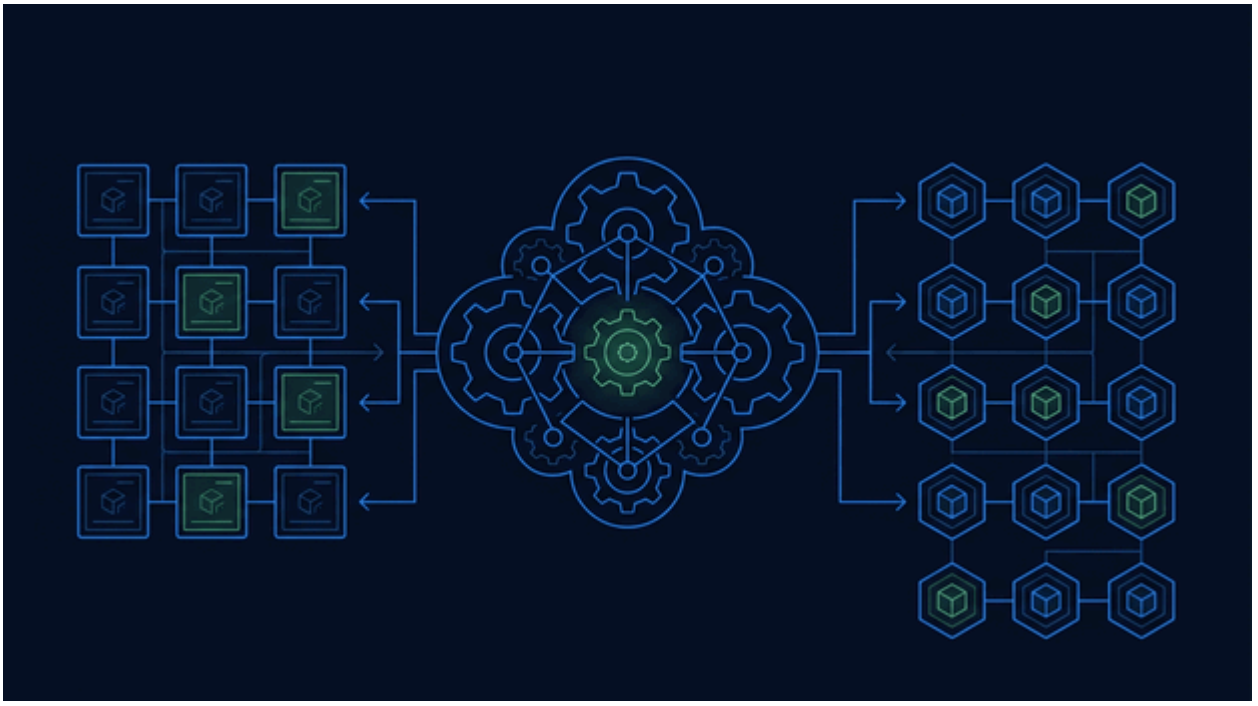
Most of my lost hours came from Wazuh behaving reasonably in ways I didn't expect. `<field>` defaults to OS_Match, not regex: your pattern silently never matches until you add `type="pcre2"`. `<srcip>` takes exactly one value, so a suppression list of five subnets is five rules, not one. And duplicate rule IDs don't error; the second definition just quietly wins, which means you can carry a detection gap and never be warned about it. `wazuh-logtest` and `wazuh-analysisd -t` are the antidote: test every decoder and rule before you reload, and the guessing goes away.

What's left is mostly coverage: file-integrity monitoring on the config files that matter, MITRE tags on the custom rules so the dashboard maps to something real, and a Windows rule set (PowerShell abuse, WMI persistence) that I keep putting off. The bones are good. The rest is maintenance, which is the part nobody writes blog posts about but everybody actually lives in.

Building your own SIEM? The field-matching and srcip gotchas will bite you eventually. Hope this saves you the debugging time.

Ansible & Ludus: Automating a Home Lab with Infrastructure as Code

Tue, Apr 7, 7:30 PM EDT · https://scottslab.io/posts/ansible-ludus-homelab-infrastructure-as-code



TL;DR

An Ansible 9.3.0 control node running 14 hosts across 9 inventory groups, with fact caching, SSH pipelining, and ControlMaster enabled because the defaults are slow enough to notice. One deployment role handles the whole stack: Docker, UFW, git clone, systemd unit, health check. Ludus builds range VMs from Packer templates on top of it. Eight collections and three roles cover Linux, Windows, Proxmox, and Samba.

Published: April 2026 **Category:** Infrastructure Automation **Reading Time:** 16 minutes

---

Executive Summary

- Deployed Ansible 9.3.0 control node managing 14 infrastructure hosts across 9 inventory groups with SSH key authentication
- Configured performance optimizations: smart fact caching (1 hour TTL), SSH pipelining, ControlMaster connection reuse
- Integrated Ludus cyber range platform for automated VM provisioning with Packer templates and Ansible role execution
- Built custom application deployment role handling full stack: Docker, UFW, Git clone, systemd service, health checks

- Installed 8 Ansible collections + 3 roles for broad platform support (Linux, Windows, Proxmox, Samba)

Goal

Problem: Managing 14+ infrastructure hosts manually doesn't scale. Every system update required SSHing into each host. Every new service deployment meant repeating the same setup steps. Configuration drift crept in as I made "temporary" changes that never got documented. When I rebuilt a host, I spent hours trying to remember what packages and configs it needed.

Why it mattered: Infrastructure as Code isn't just for enterprises. A home lab with a dozen hosts benefits from the same automation principles: repeatable deployments, documented configuration, targeted updates, and the ability to rebuild any host from scratch in minutes instead of hours. Plus, I wanted to experiment with Ludus for building cyber ranges - and it's built on Ansible.

Scope and Constraints

In Scope

- Ansible control node deployment and configuration
- YAML inventory with group-based organization
- SSH key authentication with dedicated service account
- Collection and role installation
- Ludus cyber range integration
- Custom Ansible role development
- Performance optimization (fact caching, pipelining)

Out of Scope

- AWX/Ansible Tower (too heavy for home lab)
- Dynamic inventory via cloud APIs (P1 improvement)
- CI/CD pipeline integration (P2 improvement)
- Ansible Vault secrets management (P2 improvement)

Key Constraints

- **Home lab budget** - No enterprise tooling, open-source only
- **Single control node** - No HA, no distributed execution
- **Mixed environment** - Linux and Windows hosts require different approaches
- **Frequent VM rebuilds** - Host keys change often, inventory can become stale

---

Tools and References

Tool	Role in This Project
Ansible 9.3.0	Core automation engine - playbook execution, role management, inventory
Ludus	Cyber range platform - VM provisioning, Packer integration, range deployment
Proxmox VE	Hypervisor - hosts all VMs managed by Ansible and Ludus
Packer	VM template building - creates base images for Ludus deployments
Docker	Container runtime - deployed via custom Ansible role
UFW	Firewall - configured via community.general collection
systemd	Service management - templated unit files for application lifecycle

References:

- Ansible Documentation
- Ludus Cyber Range
- Proxmox VE API
- Packer Documentation

Approach

Phase 1: Control Node Deployment

What I did: Deployed Debian 12 VM on Proxmox as the Ansible control node. Installed Ansible 9.3.0 via pip. Created dedicated `ansible` service account with RSA 4096-bit SSH key pair.

Why: A dedicated control node keeps automation infrastructure separate from managed hosts. pip installation provides the latest Ansible version without waiting for distro packages. Dedicated service account follows principle of least privilege.

Phase 2: SSH Authentication Setup

What I did: Distributed the `ansible` user's public key to all managed hosts via `ssh-copy-id`. Configured passwordless sudo for the ansible user on each host. Disabled host key checking in ansible.cfg for lab flexibility.

Why: SSH key auth eliminates passwords in playbooks and command history. Passwordless sudo enables privilege escalation without interactive prompts. Host key checking is disabled because lab VMs are frequently rebuilt with new keys.

Phase 3: Inventory Organization

What I did: Built YAML inventory with 9 groups: hypervisors, infrastructure, mail, webapps, auth, network, backup, monitoring, forensics. Hosts appear in multiple groups where appropriate (e.g., mail server is in both `infrastructure` and `mail`).

Why: Group-based inventory enables targeted automation. Run updates on all webapps with one command. Deploy monitoring to all infrastructure hosts. The overlap allows flexible targeting without duplicating host definitions.

Phase 4: Performance Optimization

What I did: Configured smart fact gathering with JSON caching (1 hour TTL), SSH pipelining to avoid temp file creation, and ControlMaster with 60-second persistence for connection reuse.

Why: Default Ansible gathers facts on every play, creates temp files for each task, and opens new SSH connections constantly. These optimizations cut execution time significantly - especially noticeable on multi-host playbooks.

Phase 5: Collection and Role Installation

What I did: Installed 8 collections (ansible.posix, ansible.utils, ansible.windows, community.general, community.windows, microsoft.ad, chocolatey.chocolatey, vladgh.samba) and 3 roles (lae.proxmox, geerlingguy.packer, ansible-thoteam.nexus3-oss).

Why: Collections provide modules for specific platforms (Windows, Proxmox) and tools (UFW, Docker). Roles provide pre-built automation for common tasks (Packer installation, Nexus deployment).

Phase 6: Ludus Integration

What I did: Deployed Ludus cyber range platform on Proxmox. Configured template library with Debian, Ubuntu, Rocky, AlmaLinux, Kali, and Windows templates. Created range configs defining VMs with templates, VLANs, IPs, resources, and Ansible roles.

Why: Ludus automates the entire VM provisioning pipeline: Packer builds templates, range configs define environments, Ansible roles configure VMs post-deployment. One YAML file describes a complete lab environment.

Phase 7: Custom Role Development

What I did: Built application deployment role with full provisioning pipeline: apt update, prerequisites, Docker install, UFW config, SSH key generation, Git clone, env setup, docker compose build, systemd service creation, health check.

Why: Deploying Docker applications involves the same steps every time. A reusable role codifies that workflow with parameterized defaults for repo URL, ports, and install paths.

Implementation Notes

Ansible Configuration (Sanitized)

```
# ansible.cfg on <CONTROL_NODE>
[defaults]
inventory = ./inventory.yml
remote_user = <SERVICE_ACCOUNT>
host_key_checking = False
gathering = smart
fact_caching = jsonfile
fact_caching_connection = /tmp/ansible_facts
fact_caching_timeout = 3600

[privilege_escalation]
become = True
become_method = sudo
become_user = root

[ssh_connection]
pipelining = True
ssh_args = -o ControlMaster=auto -o ControlPersist=60s
```

Configuration explained:

- `gathering = smart` - Only gather facts when not cached
- `fact_caching = jsonfile` - Cache facts to JSON files
- `fact_caching_timeout = 3600` - 1 hour TTL
- `pipelining = True` - Execute modules without temp files
- `ControlPersist=60s` - Reuse SSH connections for 60 seconds

YAML Inventory Structure (Sanitized)

```
# inventory.yml
all:
  children:
    hypervisors:
      hosts:
        <HYPERVISOR_A>:
          ansible_host: <HYPERVISOR_A_IP>
        <HYPERVISOR_B>:
          ansible_host: <HYPERVISOR_B_IP>

    infrastructure:
      children:
        mail:
          hosts:
            <MAIL_HOST>:
              ansible_host: <MAIL_IP>

        webapps:
          hosts:
            <WEBAPP_A>:
              ansible_host: <WEBAPP_A_IP>
            <WEBAPP_B>:
              ansible_host: <WEBAPP_B_IP>
            <WEBAPP_C>:
              ansible_host: <WEBAPP_C_IP>
            <WEBAPP_D>:
              ansible_host: <WEBAPP_D_IP>

        auth:
          hosts:
            <AUTH_HOST>:
              ansible_host: <AUTH_IP>

        network:
          hosts:
            <DNS_HOST>:
              ansible_host: <DNS_IP>
```

```
        <NETBOOT_HOST>:
          ansible_host: <NETBOOT_IP>

    backup:
      hosts:
        <BACKUP_HOST>:
          ansible_host: <BACKUP_IP>

    monitoring:
      hosts:
        <SIEM_HOST>:
          ansible_host: <SIEM_IP>

    forensics:
      hosts:
        <FORENSICS_HOST>:
          ansible_host: <FORENSICS_IP>
```

Note: Hosts under `infrastructure` children inherit membership in `infrastructure` group.

Custom Role: Application Deployer (Sanitized)

```
# roles/app_deployer/tasks/main.yml
---
- name: Update apt cache
  ansible.builtin.apt:
    update_cache: yes
    cache_valid_time: 3600

- name: Install prerequisites
  ansible.builtin.apt:
    name:
      - git
      - python3-pip
      - ca-certificates
      - curl
    state: present

- name: Install Docker
  ansible.builtin.include_role:
    name: geerlingguy.docker

- name: Configure UFW for application ports
  community.general.ufw:
    rule: allow
    port: "{{ item }}"
    proto: tcp
  loop: "{{ app_ports }}"

- name: Clone application repository
  ansible.builtin.git:
    repo: "{{ app_repo_url }}"
    dest: "{{ app_install_dir }}"
    version: "{{ app_version | default('main') }}"

- name: Copy environment file
  ansible.builtin.template:
    src: env.j2
    dest: "{{ app_install_dir }}/.env"
```

```
    mode: '0600'

- name: Build and start containers
  community.docker.docker_compose_v2:
    project_src: "{{ app_install_dir }}"
    build: always
    state: present

- name: Deploy systemd service
  ansible.builtin.template:
    src: app.service.j2
    dest: "/etc/systemd/system/{{ app_name }}.service"
  notify: Reload systemd

- name: Enable and start service
  ansible.builtin.systemd:
    name: "{{ app_name }}"
    enabled: yes
    state: started

- name: Health check
  ansible.builtin.uri:
    url: "http://localhost:{{ app_health_port }}/health"
    status_code: 200
  register: health_result
  until: health_result.status == 200
  retries: 30
  delay: 10
```

Systemd Service Template (Sanitized)

```
# roles/app_deployer/templates/app.service.j2
[Unit]
Description={{ app_name }} Docker Compose Application
Requires=docker.service
After=docker.service

[Service]
Type=oneshot
RemainAfterExit=yes
WorkingDirectory={{ app_install_dir }}
ExecStart=/usr/bin/docker compose up -d
ExecStop=/usr/bin/docker compose down
TimeoutStartSec=300

[Install]
WantedBy=multi-user.target
```

Ludus Range Config Example (Sanitized)

```
# ludus-range.yml
ludus:
  - vm_name: "<RANGE_VM_A>"
    hostname: "<HOSTNAME_A>"
    template: debian-12-x64-server-template
    vlan: 10
    ip_last_octet: 11
    ram_gb: 4
    cpus: 2
    linux: true
    roles:
      - custom_role_name

  - vm_name: "<RANGE_VM_B>"
    hostname: "<HOSTNAME_B>"
    template: ubuntu-22.04-x64-server-template
    vlan: 10
    ip_last_octet: 12
    ram_gb: 8
    cpus: 4
    linux: true
    roles:
      - geerlingguy.docker
      - app_deployer
```

Validation and Evidence

Signals That Proved It Worked

Check	Expected	Actual
All hosts reachable	14 hosts OK	`ansible all -m ping` returns SUCCESS for all
Group targeting	Subset response	`ansible webapps -m ping` returns 4 hosts
Fact caching	Faster second run	45s first run → 12s second run (cached facts)
Custom role execution	Health check pass	30 retries available, typically passes in 2-3

Check	Expected	Actual
Ludus range deploy	VMs created	`ludus range deploy` provisions all VMs

Validation Commands (Sanitized)

```
# Test all host connectivity
ansible all -m ping

# Test specific group
ansible webapps -m ping

# Check fact cache
ls -la /tmp/ansible_facts/

# Verify collections installed
ansible-galaxy collection list

# Verify roles installed
ansible-galaxy role list

# Run playbook in check mode (dry run)
ansible-playbook site.yml --check

# Run with verbose output
ansible-playbook site.yml -vv
```

Results

Metric	Outcome
Managed Hosts	14 infrastructure hosts from single control node
Inventory Groups	9 groups for targeted automation
Collections Installed	8 (Linux, Windows, Proxmox, Docker, Samba support)
Roles Installed	3 (Proxmox, Packer, Nexus)
Custom Roles	1 (Application Deployer with full stack)

Metric	Outcome
Fact Cache Hit Rate	~80% on repeated playbook runs
Execution Time Reduction	~70% with pipelining + ControlMaster + caching

---

What I Learned

1. **Smart fact gathering with JSON caching dramatically reduces execution time.** Default Ansible gathers facts on every play. With a 1-hour cache TTL, repeated runs skip fact gathering entirely - 45 seconds down to 12 seconds on a 14-host inventory.
2. **SSH pipelining eliminates temp file overhead.** Default Ansible copies module code to a temp file, executes it, then deletes. Pipelining streams the module through the SSH connection directly - faster and doesn't leave artifacts.
3. **ControlMaster with 60-second persist reuses SSH connections.** Multi-task playbooks open dozens of SSH connections by default. ControlMaster keeps one connection alive and multiplexes subsequent tasks through it.
4. **Dedicated service account is cleaner than using root.** A purpose-built `ansible` user with key-only auth and passwordless sudo creates a clear audit trail. You know exactly what automation did because it all runs as one user.
5. **Group-based inventory enables flexible targeting.** Hosts can belong to multiple groups. Run `ansible webapps` for web servers, `ansible infrastructure` for everything, or `ansible mail` for just the mail server - without duplicating definitions.
6. **Ludus abstracts the VM provisioning pipeline.** One YAML config specifies templates, VLANs, IPs, resources, and Ansible roles. `ludus range deploy` creates the entire environment. No manual Proxmox clicking.
7. **Custom roles should include health checks as the final task.** Immediate feedback on deployment success. The role either completes with a passing health check or fails with a clear error - no ambiguous "maybe it worked" states.
8. **Jinja2 templates keep systemd unit files maintainable.** Hardcoding paths and service names in unit files creates drift. Templates with variables (`{{ app_name }}`, `{{ app_install_dir }}`) stay consistent across deployments.
9. **Inventory IP addresses become stale when DHCP reservations change.** I moved the forensics workstation from .131 to .112 and the auth server from .112 to .117. The inventory didn't update automatically - playbooks failed until I fixed the IPs manually.

10. **`host_key_checking = False` is necessary in lab environments.** VMs get rebuilt frequently with new SSH host keys. Strict host key checking would require updating known_hosts constantly. Trade-off: less secure, more practical for labs.

What I Would Improve Next

P0 (Do This Week)

- **Fix stale inventory IPs** - Update forensics workstation (.112) and auth server (.117) entries
- **Inventory validation playbook** - Automated ping test that reports unreachable hosts before main playbooks run

P1 (Do This Month)

- **Dynamic inventory via Proxmox API** - Auto-discover hosts and IPs instead of static YAML
- **Scheduled system updates** - Weekly apt upgrade playbook across all Linux hosts
- **Security hardening playbook** - SSH hardening, fail2ban, audit logging applied to all hosts
- **Wazuh agent deployment role** - Automatically register new hosts with SIEM

P2 (Do This Quarter)

- **Ansible Vault for secrets** - Stop hardcoding passwords in env files
- **Monitoring agent deployment** - Auto-register with SIEM on host provisioning
- **Infrastructure testing playbook** - Verify services running, ports open, DNS resolving
- **GitOps integration** - Playbooks in Gitea, webhook-triggered runs on commit

---

Common Failure Modes

1. **"Host unreachable" on previously working hosts** - Inventory IP is stale after DHCP reservation change. Check current IP via Proxmox console or DHCP lease table, update inventory.

2. **"Permission denied (publickey)" on new host** - SSH key not distributed to new host. Run `ssh-copy-id <SERVICE_ACCOUNT>@<NEW_HOST>` from control node.

3. **"Gathering facts" takes forever on second run** - Fact cache may be stale or corrupted. Clear `/tmp/ansible_facts/` directory and re-run.

4. **"Missing sudo password" errors** - Passwordless sudo not configured for ansible user on target host. Add `<SERVICE_ACCOUNT> ALL=(ALL) NOPASSWD: ALL` to sudoers.

5. **"Pipelining failed" on specific hosts** - Target host may not have writable `/tmp` or the user lacks permissions. Check `requiretty` isn't set in sudoers, verify `/tmp` permissions.

Security Considerations

Authentication

- Dedicated `ansible` service account - not root, not personal accounts
- SSH key-only authentication - no passwords in playbooks or history
- RSA 4096-bit keys - strong cryptographic foundation
- Keys stored only on control node - not distributed widely

Authorization

- Passwordless sudo limited to `ansible` user on managed hosts
- Principle of least privilege - ansible user only has necessary permissions
- No root SSH access - even with the key, root login is disabled

Secrets Management

- Current: passwords in env files (acknowledged technical debt)
- Future: Ansible Vault encryption for all secrets (P2 improvement)
- Sensitive files deployed with restricted permissions (0600)

Trade-offs in Lab vs Production

- `host_key_checking = False` - necessary for frequent VM rebuilds but would be unacceptable in production
- JSON fact caching - stores host information in plaintext on control node
- Single control node - no HA, single point of failure for automation

---

Runbook

How to Add a New Host to Inventory

```
# 1. Add host entry to appropriate group in inventory.yml
monitoring:
  hosts:
    <NEW_HOST>:
      ansible_host: <NEW_HOST_IP>

# 2. Distribute SSH key
ssh-copy-id <SERVICE_ACCOUNT>@<NEW_HOST_IP>

# 3. Configure passwordless sudo on target
echo "<SERVICE_ACCOUNT> ALL=(ALL) NOPASSWD: ALL" | sudo tee
/etc/sudoers.d/<SERVICE_ACCOUNT>

# 4. Test connectivity
ansible <NEW_HOST> -m ping
```

How to Run a Playbook Against One Group

```
# Run against specific group
ansible-playbook site.yml --limit webapps

# Run single task with ad-hoc command
ansible webapps -m apt -a "name=htop state=present" --become

# Check mode (dry run) against group
ansible-playbook site.yml --limit monitoring --check
```

How to Deploy a Ludus Range

```
# 1. Create range config (ludus-range.yml)
# 2. Set the range config
ludus range config set -f ludus-range.yml

# 3. Build any missing templates
ludus templates build

# 4. Deploy the range
ludus range deploy

# 5. Check deployment status
ludus range status
```

How to Build a New Packer Template

```
# 1. Navigate to Ludus templates directory
cd /opt/ludus/templates

# 2. Create or modify template definition
# 3. Build specific template
ludus templates build -t debian-12-x64-server-template

# 4. Verify template in Proxmox
ludus templates list
```

How to Create a Custom Ansible Role

```
# 1. Create role skeleton
ansible-galaxy role init roles/my_new_role

# 2. Edit role structure:
#    - roles/my_new_role/defaults/main.yml (default variables)
#    - roles/my_new_role/tasks/main.yml (task list)
#    - roles/my_new_role/templates/*.j2 (Jinja2 templates)
#    - roles/my_new_role/handlers/main.yml (handlers)

# 3. Test role
ansible-playbook test-role.yml --check

# 4. Run role
ansible-playbook test-role.yml
```

Appendix

Glossary

Term	Definition
Ansible	Agentless automation platform using SSH for Linux, WinRM for Windows
Ludus	Open-source cyber range platform built on Proxmox with Packer/Ansible integration
Packer	HashiCorp tool for building machine images from templates
Proxmox VE	Open-source virtualization platform (KVM + LXC)
Jinja2	Python templating engine used by Ansible for templates
YAML Inventory	Ansible inventory format using YAML syntax for host/group definitions
Ansible Role	Reusable automation unit with tasks, templates, handlers, and variables
Ansible Collection	Package format for distributing modules, plugins, and roles
Fact Caching	Storing gathered host facts locally to avoid re-collection
Pipelining	SSH optimization that streams modules instead of copying temp files

Term	Definition
ControlMaster	SSH feature that multiplexes connections through a single socket

MITRE ATT&CK Relevance

Technique ID	Name	Automation Relevance
T1059	Command and Scripting Interpreter	Ansible executes commands across hosts - legitimate automation, but same techniques used by attackers
T1072	Software Deployment Tools	Ansible deploys software at scale - powerful for defenders, attractive target for attackers
T1098	Account Manipulation	Ansible manages user accounts and sudo permissions - audit trail is critical
T1136	Create Account	Custom roles create service accounts - document all automated account creation

Infrastructure as Code Principles Applied

Principle	Implementation
Idempotency	Ansible tasks can run multiple times with same result
Version Control	Playbooks and inventory tracked in Git
Documentation	Role defaults and variable files document configuration
Repeatability	Same playbook produces same result on any host
Modularity	Roles encapsulate reusable automation units
Testability	Check mode allows dry runs before execution

Artifacts Produced

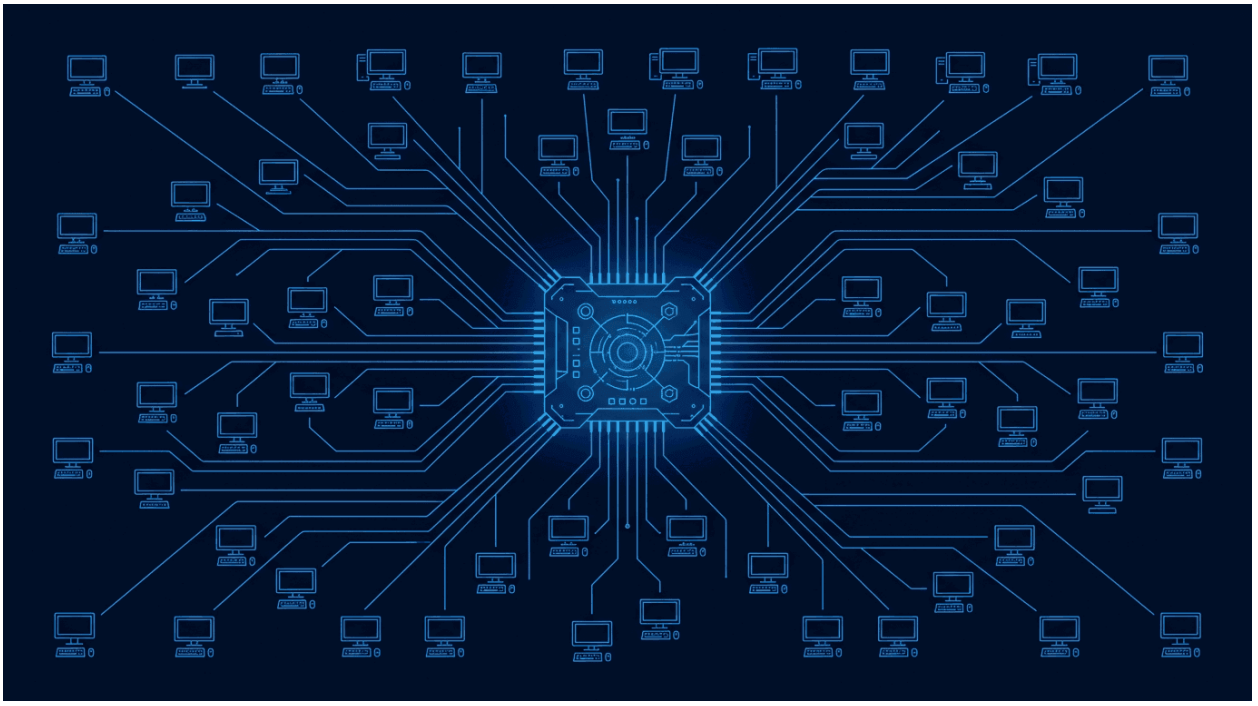
- **Ansible Configuration: ansible.cfg** - Optimized settings for fact caching, pipelining, connection reuse
- **YAML Inventory: 14 hosts / 9 groups** - Structured inventory with group-based organization
- **Custom Role: Application Deployer** - Full stack deployment (Docker, UFW, Git, systemd, health check)
- **Ludus Range Config** - VM definitions with templates, VLANs, IPs, and role assignments
- **systemd Service Template** - Jinja2 template for Docker Compose lifecycle management

- **Packer Template Library** - Multi-OS templates (Debian, Ubuntu, Rocky, AlmaLinux, Kali, Windows)

Building your own automation platform? Start with the basics: SSH keys, simple inventory, one playbook. The fancy stuff (fact caching, pipelining, Ludus) comes later. Walk before you run.

Ansible Alternative: Why I Use Action1 for Windows Management

Tue, May 5, 8:15 AM EDT · https://scottslab.io/posts/action1-windows-management-ansible-alternative



TL;DR

Ansible is good at Linux and miserable at Windows, so the Windows endpoints go to Action1 instead. Agent install runs about two minutes a box against hours of WinRM wrangling, the free tier covers 200 endpoints, and the agent dials out. No inbound rules to open. It does not replace Ansible; the two split the fleet by operating system.

Published: May 2026 **Category:** Infrastructure Automation **Reading Time:** 12 minutes

---

Executive Summary

- Deployed Action1 RMM for Windows endpoint management - free tier covers 200 endpoints (more than enough for home lab)
- Agent installation takes under 2 minutes per endpoint vs. hours of WinRM configuration for Ansible
- Cloud-based console provides patch management, software deployment, remote scripting, and real-time monitoring
- Complements Ansible rather than replacing it: Action1 handles Windows, Ansible handles Linux
- No inbound firewall rules required - agent initiates outbound connection to Action1 cloud

---

Goal

Problem: Ansible is fantastic for Linux automation, but Windows support has always been... friction-heavy. WinRM configuration, certificate management, PowerShell execution policies, firewall rules, credential delegation - every Windows host requires significant prep work before Ansible can touch it. For a home lab with a handful of Windows machines (gaming PC, media center, occasional test VMs), that overhead isn't worth it.

Why it mattered: I still need to manage Windows endpoints: push updates, deploy software, run scripts, check security posture. I wasn't going to RDP into each machine individually like it's 2005. I needed something purpose-built for Windows that "just works" - and ideally doesn't cost anything for home lab scale.

Scope and Constraints

In Scope

- Windows endpoint management (patching, software deployment, scripting)
- Agent deployment to home Windows machines
- Patch compliance monitoring
- Software inventory and deployment
- Remote PowerShell execution

Out of Scope

- Linux management (Ansible handles this)
- Enterprise features (compliance reporting, advanced RBAC)
- On-premises management server (Action1 is cloud-only)
- Integration with existing Ansible workflows

Key Constraints

- **Cost** - Must be free for home lab use (200 endpoint limit is fine)
- **Complexity** - Must be simpler than configuring WinRM for Ansible
- **Security** - Agent must not require inbound firewall rules
- **Coverage** - Must handle core RMM tasks: patching, software, scripting

---

Tools and References

Tool	Role in This Project
Action1	Cloud-based RMM platform for Windows endpoint management
Action1 Agent	Lightweight Windows service (~15MB) that connects to cloud console
PowerShell	Scripting engine for custom automation via Action1
Windows Update	Patch source managed through Action1 policies

References:

- Action1 Free Tier
- Action1 Documentation
- Why WinRM is Painful

Approach

Phase 1: Evaluate the Windows Automation Problem

What I did: Attempted to configure Ansible for Windows management. Set up WinRM listeners, configured HTTPS with self-signed certificates, adjusted PowerShell execution policies, opened firewall ports, configured CredSSP for credential delegation.

Why I stopped: After 3 hours on the first Windows host, I still had intermittent authentication failures. The juice wasn't worth the squeeze for 4 Windows endpoints.

Phase 2: Evaluate RMM Alternatives

What I did: Researched RMM (Remote Monitoring and Management) tools with free tiers: Action1, Tactical RMM, MeshCentral, PDQ (limited free). Evaluated based on: free tier limits, ease of setup, feature set, security model.

Why Action1: 200 free endpoints (way more than I need), cloud-hosted (no server to maintain), agent-initiated connections (no inbound firewall rules), and solid feature set (patching, software deployment, scripting, inventory).

Phase 3: Agent Deployment

What I did: Downloaded the Action1 agent MSI from the cloud console. Installed on Windows endpoints via simple MSI execution - no configuration required. Agent auto-registered with my Action1 organization.

Why this matters: Compare "run MSI, done" to "configure WinRM listener, generate certificate, configure firewall, set execution policy, test connectivity, debug authentication errors." The time savings are significant.

Phase 4: Configure Patch Management

What I did: Created patch policies in Action1 console: approve critical/security updates automatically, schedule weekly patch scans, configure maintenance windows for automatic installation.

Why: Unpatched Windows machines are a liability. Automated patching ensures endpoints stay current without manual intervention.

Phase 5: Software Deployment Setup

What I did: Built software deployment packages for common applications: browsers, utilities, security tools. Action1 has a built-in repository of common software plus support for custom packages.

Why: When I rebuild a Windows machine, I don't want to manually download and install 15 applications. Deploy from Action1, wait 10 minutes, done.

Phase 6: Remote Scripting Configuration

What I did: Created PowerShell script library in Action1 for common tasks: system information gathering, security configuration checks, cleanup scripts. Scripts run as SYSTEM with full privileges.

Why: Sometimes you need to run a quick command across all Windows machines. Action1's script execution is faster than RDPing into each one.

Implementation Notes

Agent Installation (Sanitized)

```
# Download agent MSI from Action1 console (URL is organization-specific)
# URL format: https://app.action1.com/agent/download/<ORG_ID>

# Silent installation
msiexec /i Action1Agent.msi /qn

# Agent auto-registers with Action1 cloud
# No additional configuration required
```

That's it. No WinRM configuration. No certificates. No firewall rules. The agent initiates an outbound HTTPS connection to Action1's cloud - works through NAT, works through firewalls, works everywhere.

Patch Policy Configuration (Sanitized)

```
Policy: Home Lab Windows Patching

Scan Schedule: Weekly (Sunday 2:00 AM)

Auto-Approve:
  - Critical Updates: Yes
  - Security Updates: Yes
  - Definition Updates: Yes
  - Feature Updates: No (manual review)
  - Driver Updates: No (manual review)

Installation Window:
  - Day: Sunday
  - Time: 3:00 AM - 6:00 AM
  - Reboot: If required, after window

Endpoints: All Windows endpoints
```

Software Deployment Example (Sanitized)

```
Package: Firefox Browser

Source: Action1 Repository (pre-built)
Version: Latest
Installation: Silent (/S flag)
Detection: Registry check for installed version

Deployment:
  - Target: All Windows endpoints
  - Schedule: Immediate
  - Retry on failure: 3 attempts
```

Custom PowerShell Script Example (Sanitized)

```
# Script: Get-SecurityStatus.ps1
# Purpose: Check Windows security configuration

$results = @{
    Hostname = $env:COMPUTERNAME
    WindowsFirewall = (Get-NetFirewallProfile | Where-Object {$_.Enabled -eq
$true}).Name
    DefenderStatus = (Get-MpComputerStatus).AntivirusEnabled
    LastUpdate = (Get-HotFix | Sort-Object InstalledOn -Descending | Select-
Object -First 1).InstalledOn
    UAC = (Get-ItemProperty
HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System).EnableLUA
}

$results | ConvertTo-Json
```

Ansible vs Action1 Comparison

Task	Ansible (Windows)	Action1
Initial setup per host	30-60 minutes (WinRM, certs, firewall)	2 minutes (run MSI)
Patch management	Custom playbook + WSUS or direct	Built-in policy engine
Software deployment	win_package module + hosting	Built-in repository
Remote scripting	win_shell module	Web console or API
Agentless	Yes (but requires WinRM)	No (lightweight agent)
Works through NAT	Requires port forwarding	Yes (outbound only)
Cost	Free	Free (200 endpoints)

Validation and Evidence

Signals That Proved It Worked

Check	Expected	Actual
Agent registration	Endpoints visible in console	All 4 Windows endpoints registered

Check	Expected	Actual
Patch scan	Compliance report generated	Shows missing patches per endpoint
Software deployment	Firefox installed silently	Deployed to all endpoints in <5 min
Remote script	JSON output returned	Security status collected from all
Patch installation	Updates installed overnight	Confirmed via compliance dashboard

Validation Steps (Sanitized)

```
1. Action1 Console → Endpoints
   - Verify all Windows machines appear
   - Check "Last Seen" timestamp (should be recent)

2. Action1 Console → Patch Management → Compliance
   - Verify scan completed
   - Review missing patches

3. Action1 Console → Apps → Managed Apps
   - Verify deployed software shows "Installed" status

4. Action1 Console → Scripts → Run Script
   - Execute test script
   - Verify output returned from all endpoints
```

Results

Metric	Outcome
Windows Endpoints Managed	4 (gaming PC, media center, 2 test VMs)
Setup Time per Endpoint	~2 minutes (MSI install)
Patch Compliance	Automated weekly scan + install
Software Packages	8 apps deployed via Action1
Custom Scripts	5 PowerShell scripts in library
Cost	$0 (free tier, 200 endpoint limit)
Ansible WinRM Configs Avoided	4 (and counting)

What I Learned

1. **The right tool for the job beats forcing a universal tool.** Ansible is excellent for Linux. Fighting with WinRM to make it work on Windows isn't worth it when purpose-built tools exist.
2. **Agent-based beats agentless for Windows in home labs.** Ansible's agentless model requires WinRM configuration on every endpoint. Action1's lightweight agent (15MB, runs as service) requires zero endpoint configuration.
3. **Outbound-only connections simplify everything.** No inbound firewall rules, no port forwarding, no NAT traversal issues. The agent calls home; the cloud console is always reachable.
4. **200 free endpoints is generous for home use.** I have 4 Windows machines. Even if I had 20, I'd still be well under the limit. Enterprise features are paid, but home lab features are free.
5. **Cloud-hosted RMM means no infrastructure to maintain.** No server to patch, no database to back up, no certificates to renew. Trade-off: you're trusting a third party with endpoint access.
6. **Built-in patch management is a massive time saver.** Writing Ansible playbooks for Windows Update is tedious. Action1's policy engine handles it with a few clicks.
7. **Software repository with common apps eliminates hosting.** I don't need to maintain an internal package repository for Firefox, Chrome, 7-Zip, etc. Action1 has them ready to deploy.
8. **PowerShell scripts run as SYSTEM.** Full privileges, no UAC prompts, consistent execution environment. More powerful than what you'd get interactively.
9. **Complements Ansible rather than replacing it.** My Linux hosts still use Ansible. My Windows hosts use Action1. Different tools, same goal: automated management.
10. **Time saved on setup is time spent on actual work.** The hours I didn't spend debugging WinRM went into building detection rules, writing playbooks, and improving actual security posture.

What I Would Improve Next

P0 (Do This Week)

- **Endpoint naming standardization** - Rename endpoints in Action1 to match inventory naming convention
- **Script library expansion** - Add scripts for common troubleshooting tasks

P1 (Do This Month)

- **Automated onboarding** - Create deployment script that installs Action1 agent as part of Windows setup
- **Security baseline policy** - Build policy that checks/enforces security settings (firewall, UAC, Defender)
- **Alert configuration** - Set up alerts for offline endpoints and failed patch installations

P2 (Do This Quarter)

- **Integration with monitoring** - Export Action1 events to SIEM for unified visibility
- **Custom package repository** - Add internal applications to Action1 for deployment
- **Endpoint groups** - Organize endpoints by function (workstations, servers, test VMs)
- **Reporting automation** - Schedule weekly compliance reports via email

---

Common Failure Modes

1. **"Endpoint shows offline in console"** - Check if Windows machine is powered on and has internet connectivity. Agent requires outbound HTTPS to Action1 cloud.

2. **"Patch installation failed"** - Check available disk space (Windows Update needs room). Review Action1 logs for specific error codes. May need to clear Windows Update cache.

3. **"Software deployment stuck at 'Pending'"** - Endpoint may be offline or agent service may have stopped. Restart Action1 Agent service on the endpoint.

4. **"Script execution returned no output"** - Script may have errored. Check script syntax in PowerShell ISE locally first. Action1 captures stderr but display can be confusing.

5. **"Can't find endpoint after reinstalling Windows"** - Old endpoint entry is orphaned. Delete the old entry in Action1 console, reinstall agent on fresh Windows install.

Security Considerations

Agent Security Model

- Agent runs as SYSTEM service (high privilege, but contained to endpoint)
- Communication is outbound HTTPS only (no inbound attack surface)
- Agent authenticated to Action1 org via unique endpoint ID
- All management traffic encrypted in transit

Cloud Trust Model

- Action1 is a third-party SaaS - you're trusting them with endpoint access
- They can execute scripts, install software, access file systems on managed endpoints
- Appropriate for home lab; evaluate carefully for business use
- SOC 2 Type II certified (for those who care about compliance)

Access Control

- Action1 console protected by username/password + optional MFA
- Role-based access available (not needed for single-user home lab)
- Audit log tracks all actions taken through console

Comparison to Ansible Security

- Ansible: credentials stored on control node, WinRM requires authentication config
- Action1: credentials managed by SaaS, agent pre-authenticated at install
- Both require trust in the management platform; different trust models

---

Runbook

How to Add a New Windows Endpoint

1. Log into Action1 console
2. Go to Endpoints → Add Endpoints
3. Download agent MSI (or use existing downloaded installer)
4. On Windows endpoint: run `msiexec /i Action1Agent.msi /qn`
5. Wait 1-2 minutes for endpoint to appear in console
6. Verify endpoint shows "Online" status

How to Deploy Software to All Windows Endpoints

1. Action1 Console → Apps → Deploy App
2. Select from repository or upload custom package
3. Choose target: All Endpoints (or specific group)
4. Configure installation options (silent, reboot behavior)
5. Click Deploy
6. Monitor progress in deployment status view

How to Run a Script Across All Endpoints

1. Action1 Console → Scripts → Run Script
2. Select existing script or create new
3. Choose target endpoints
4. Click Run
5. View output in script execution results

How to Check Patch Compliance

1. Action1 Console → Patch Management → Overview
2. Review compliance percentage per endpoint
3. Click endpoint for detailed missing patch list
4. Approve/deploy patches manually or wait for policy

How to Troubleshoot Offline Endpoint

1. Verify Windows machine is powered on
2. Check internet connectivity on the endpoint
3. Open Services (services.msc), find "Action1 Agent"
4. If stopped, start the service
5. If running, restart the service
6. Check Action1 console in 1-2 minutes for online status

Appendix

Glossary

Term	Definition
RMM	Remote Monitoring and Management - tools for managing endpoints remotely
Action1	Cloud-based RMM platform with free tier for up to 200 endpoints
WinRM	Windows Remote Management - Microsoft's implementation of WS-Management protocol
Agent	Software installed on endpoint that communicates with management platform
Agentless	Management approach that doesn't require software on managed endpoints
SYSTEM	Windows built-in account with highest local privileges
MSI	Microsoft Installer package format for Windows software deployment

Term	Definition
Patch Policy	Rules defining how and when updates are approved and installed

Why Not Just Use Ansible for Windows?

Ansible Windows Challenge	Impact
WinRM listener configuration	15-30 min per host
Certificate management (HTTPS)	Ongoing maintenance
PowerShell execution policy	Must be configured permissively
Firewall rules	Inbound ports required
CredSSP for delegation	Security implications, complex setup
Authentication debugging	"It should work but doesn't" syndrome
NAT traversal	Requires port forwarding or VPN

Assumption: For enterprise environments with hundreds of Windows servers and existing WinRM infrastructure, Ansible makes sense. For home labs with a handful of Windows endpoints, the setup cost isn't justified.

Action1 Free Tier Limitations

Feature	Free Tier	Paid Tier
Endpoints	200	Unlimited
Patch Management	Yes	Yes
Software Deployment	Yes	Yes
Remote Scripting	Yes	Yes
Reporting	Basic	Advanced
RBAC	Limited	Full
API Access	Limited	Full
Support	Community	Priority

Artifacts Produced

- **Action1 Organization** - Cloud-hosted management console for Windows endpoints
- **Endpoint Agents** - Action1 agent deployed to 4 Windows machines
- **Patch Policy: Home Lab Windows** - Weekly scan, auto-approve security updates, Sunday maintenance window
- **Software Packages** - 8 applications configured for deployment (browsers, utilities, tools)
- **Script Library** - 5 PowerShell scripts for common management tasks
- **Documentation: Ansible vs Action1** - Decision record for tool selection

---

Still fighting with Ansible and Windows? Give Action1 a try. The free tier is genuinely free, and you might actually finish setting it up.

Why I Use CSI Linux as My Digital Forensics Workstation

Tue, Jun 9, 2:05 PM EDT · https://scottslab.io/posts/csi-linux-digital-forensics-workstation

TL;DR

CSI Linux as the DFIR workstation, mostly because 50-odd forensic tools are already installed and an investigation should not start with dependency hunting. Four independent ways back onto the box (browser desktop over SSO, SSH, hypervisor guest agent, VPN), with evidence on a central SMB share that systemd automount keeps alive across reboots. Snapshot before analysis and roll back after, so one case never contaminates the next.

Published: June 2026 **Category:** Digital Forensics **Reading Time:** 14 minutes

---

Executive Summary

- Deployed CSI Linux as the primary DFIR workstation - 50+ forensic tools pre-installed across 9 categories with zero dependency hunting
- Integrated with centralized evidence management: SMB share on desktop, web upload portal with zero-trust access for external investigators
- Configured 4 access methods: web-based remote desktop (SSO), SSH, hypervisor guest agent, and VPN backup
- Solved persistent network mount issues using systemd automount for reliable evidence share access across reboots

- Established snapshot-based investigation workflow - snapshot before analysis, rollback after to prevent workstation contamination

---

Goal

Problem: When an incident happens, you need to start investigating immediately - not spend three hours installing Autopsy, hunting down Volatility dependencies, and figuring out why your disk imaging tool won't compile. I needed a forensic workstation that was ready to go the moment I needed it, with tools for disk forensics, memory analysis, network capture, OSINT, and malware triage all in one place.

Why it mattered: Incident response is time-sensitive. Every hour spent setting up tools is an hour the attacker has to cover tracks, exfiltrate data, or move laterally. A pre-configured forensic workstation means I can start analyzing evidence within minutes of receiving it, not days. Plus, running forensics on the same machine as production services risks contaminating both the investigation and the infrastructure.

Scope and Constraints

In Scope

- DFIR workstation deployment and configuration
- Evidence management infrastructure (centralized share, upload portal)
- Remote access methods (web desktop, SSH, guest agent)
- Tool inventory and verification
- Investigation workflow design

Out of Scope

- Mobile forensics (P1 improvement - MVT, Andriller)
- Cloud forensics (P2 improvement - AWS IR, Azure tools)
- Malware sandboxing (P2 improvement - Cuckoo/CAPE)
- Custom tool development
- Forensic report automation

Key Constraints

- **VM resources** - 4 CPU cores, 8GB RAM, 50GB SSD (home lab budget)
- **Evidence integrity** - Must prevent accidental modification of evidence
- **Investigation isolation** - Forensic work must be isolated from production infrastructure
- **Accessibility** - Investigators need GUI access for tools like Autopsy and Wireshark

Tools and References

CSI Linux Tool Categories (50+ Tools)

Category	Tools	Purpose
Disk & File Forensics	Autopsy, Sleuth Kit, Foremost, Scalpel, TestDisk, PhotoRec, Binwalk	Disk imaging, file carving, partition recovery, firmware extraction
Memory Forensics	Volatility 3	RAM analysis, process enumeration, malware artifact detection
Network Forensics	Wireshark, Nmap, Kismet, Bettercap	Packet capture, network discovery, wireless analysis
OSINT	Maltego, Recon-ng, Sherlock, theHarvester, Amass, SpiderFoot	Username lookup, subdomain enumeration, link analysis
Password Analysis	Hashcat, John the Ripper, Hydra, Medusa	Hash cracking, credential testing
Malware Analysis	ClamAV, Radare2, YARA	AV scanning, reverse engineering, pattern matching
Steganography	Steghide, ExifTool	Hidden data extraction, metadata analysis
Web Testing	GoBuster, Nikto, SQLMap	Directory enumeration, vulnerability scanning
Wireless Security	Aircrack-ng, Kismet	WiFi security testing, multi-protocol sniffing

Infrastructure Tools

Tool	Role
CSI Linux	Ubuntu 22.04 LTS-based DFIR distribution
Proxmox	VM hypervisor hosting the forensic workstation
Apache Guacamole	Web-based remote desktop access
Samba/CIFS	Evidence share file system
Cloudflare Access	Zero-trust authentication for upload portal

References:

- CSI Linux Official Documentation
- Autopsy Digital Forensics
- Volatility 3 Documentation
- SANS DFIR Poster

Approach

Phase 1: Distribution Selection

What I did: Evaluated forensic Linux distributions (SIFT, REMnux, Kali, CSI Linux) against requirements: pre-installed DFIR tools, OSINT capability, VM-friendly design, active maintenance.

Why: Building a forensic workstation from scratch on bare Ubuntu takes days and creates dependency nightmares. A purpose-built distro provides a tested toolchain where everything works together.

Phase 2: VM Deployment

What I did: Created a VM on Proxmox with 4 CPU cores, 8GB RAM, and 50GB SSD. Installed CSI Linux from ISO, enabled QEMU guest agent for remote management.

Why: Running as a VM provides isolation from production, enables snapshots before investigations, and allows resource scaling if needed. Guest agent enables shutdown/restart from hypervisor without console access.

Phase 3: Access Configuration

What I did: Configured four access methods: web-based remote desktop via Guacamole (behind SSO), SSH with password auth enabled via drop-in config, hypervisor guest agent for remote management, VPN mesh as backup path.

Why: Forensic work often requires GUI tools (Autopsy, Wireshark, Maltego). Web desktop provides full GUI access from any browser without VPN. Multiple access methods ensure I can always reach the workstation.

Phase 4: Evidence Share Integration

What I did: Mounted centralized evidence share to the forensic workstation desktop using CIFS with systemd automount. Configured credentials file, automount options, and network wait service.

Why: Evidence needs to be accessible directly from the forensic tools. Dragging files between the desktop evidence folder and Autopsy is more intuitive than transferring via SCP. Automount ensures the share survives reboots.

Phase 5: Investigation Workflow Design

What I did: Established case management workflow using CSI Linux's built-in case creation, integrated with evidence share directory structure. Documented snapshot-before-investigation procedure.

Why: Consistent workflow prevents mistakes under pressure. Snapshots before investigation allow rollback if a tool corrupts the workstation environment.

Implementation Notes

VM Configuration (Sanitized)

```
VM Name: <FORENSIC_VM>
Hypervisor: Proxmox
Resources:
  CPU: 4 cores
  RAM: 8GB
  Disk: 50GB SSD (VirtIO)
Network: Bridge to lab VLAN
Guest Agent: QEMU (enabled for remote management)
```

Evidence Share Mount (Sanitized)

```
# /etc/fstab entry with systemd automount
//<EVIDENCE_IP>/evidence  /home/csi/Desktop/Evidence  cifs
credentials=/root/.smbcreds,uid=1000,gid=1000,vers=3.0,x-systemd.automount,x-
systemd.mount-timeout=30,_netdev,nofail  0  0
```

Mount options explained:

- `x-systemd.automount` - Creates automount point, actual mount on first access
- `x-systemd.mount-timeout=30` - Don't hang boot if share unavailable
- `_netdev` - Wait for network before attempting mount
- `nofail` - Don't fail boot if mount fails
- `vers=3.0` - Explicit SMB protocol version

Credentials File (Sanitized)

```
# /root/.smbcreds (chmod 600)
username=<SMB_USER>
password=<SMB_PASSWORD>
domain=<SMB_DOMAIN>
```

Enable Network Wait Service

```
# Required for boot-time network dependencies
systemctl enable NetworkManager-wait-online.service
```

SSH Drop-in Configuration (Sanitized)

```
# /etc/ssh/sshd_config.d/99-local.conf
PasswordAuthentication yes
PermitRootLogin prohibit-password
```

Snapshot Before Investigation

```
# From Proxmox CLI or GUI
qm snapshot <VMID> pre-investigation-$(date +%Y%m%d)

# After investigation, rollback if needed
qm rollback <VMID> pre-investigation-<DATE>
```

Validation and Evidence

Signals That Proved It Worked

Check	Expected	Actual
Forensic tools installed	50+	50+ verified
Evidence share mount	Auto on boot	Working with automount
Web desktop access	GUI via browser	Functional via Guacamole
SSH access	Remote CLI	Enabled and working

Check	Expected	Actual
Guest agent	Remote management	Responding to Proxmox
Snapshot/rollback	Clean state recovery	Tested successfully

Tool Verification Commands (Sanitized)

```
# Verify key forensic tools
which autopsy && autopsy --version
which volatility3 && vol -h | head -1
which wireshark && wireshark --version
which maltego && echo "Maltego installed"
which hashcat && hashcat --version

# Verify evidence share mount
ls /home/csi/Desktop/Evidence/
mount | grep Evidence

# Verify access methods
systemctl status sshd
systemctl status qemu-guest-agent
```

Case Management Workflow Test

1. Created new case via CSI Linux case manager
2. Copied sample evidence from share to case directory
3. Opened evidence in Autopsy
4. Verified file carving results
5. Exported findings to case notes

Results

Metric	Outcome
Forensic Tools	50+ tools across 9 categories, ready to use
Evidence Access	Centralized share mounted on desktop + web upload portal
Access Methods	4 paths (web desktop, SSH, guest agent, VPN)

Metric	Outcome
Time to Ready	Minutes (vs. days for manual build)
Investigation Isolation	VM separation from production
State Recovery	Snapshot-based rollback available

What I Learned

1. **CSI Linux dramatically accelerates forensic workstation setup.** What would take days of manual installation and dependency resolution is ready in under an hour. The toolchain is pre-tested and integrated.

2. **systemd automount (`x-systemd.automount`) is essential for network mounts.** Standard fstab mounts fail when the network isn't ready at boot. Automount defers the actual mount until first directory access, completely avoiding race conditions.

3. **`_netdev` and `nofail` flags prevent boot hangs.** Without these, an unavailable network share can stall your entire boot process. Forensic workstations need to boot reliably even when infrastructure is down.

4. **The `NetworkManager-wait-online.service` must be enabled.** If you have any boot-time network dependencies, this service ensures the network is actually up before systemd continues. It's often disabled by default.

5. **CIFS mounts may require explicit protocol version.** Some SMB servers reject version negotiation. Adding `vers=3.0` explicitly can fix mysterious mount failures.

6. **Evidence on the desktop creates intuitive workflow.** Drag-and-drop between the evidence folder and tools like Autopsy feels natural. SCP transfers for every file would be tedious.

7. **VM snapshots before investigations are critical.** One wrong tool execution - accidentally writing to a disk image, corrupting a memory dump - can contaminate your workstation. Snapshots provide a clean rollback path.

8. **Web-based remote desktop works better than expected.** I was skeptical about running GUI forensic tools through a browser, but Guacamole handles Autopsy, Wireshark, and even Maltego smoothly.

9. **Running forensics on a separate VM prevents cross-contamination.** The forensic workstation never touches production services directly. Evidence comes in through the share; findings go out through reports.

10. **Zero-trust access controls on the evidence portal prevent unauthorized submissions.** External investigators can upload evidence without VPN access, but only after email-based identity verification.

What I Would Improve Next

P0 (Do This Week)

- **Automated evidence hashing** - SHA-256 hash every file on upload for chain of custody verification
- **Write-once evidence storage** - Implement append-only storage for original evidence to prevent tampering

P1 (Do This Month)

- **Additional memory forensics tools** - Add MemProcFS and standardize on Volatility 3 alongside the base Volatility (skip Rekall, it's archived and unmaintained)
- **Automated triage scripts** - Build scripts that run common tools (strings, file, exiftool, clamscan) on new evidence automatically
- **Mobile forensics capability** - Add MVT (Mobile Verification Toolkit) and Andriller for mobile device analysis

P2 (Do This Quarter)

- **SIEM integration** - Log forensic workstation activity to Wazuh for audit trail
- **Malware sandbox** - Add Cuckoo or CAPE for dynamic malware analysis
- **Report templates** - Build forensic report templates that auto-populate from tool outputs
- **Cloud forensics tools** - Add AWS IR tools and Azure forensics capabilities

---

Common Failure Modes

1. **"Evidence folder is empty after reboot"** - Network mount failed. Check that `x-systemd.automount` is in fstab, `NetworkManager-wait-online.service` is enabled, and credentials file exists with correct permissions (600).

2. **"SSH connection refused"** - CSI Linux ships with SSH disabled by default. Create drop-in config at `/etc/ssh/sshd_config.d/` to enable password auth, then restart sshd.

3. **"Wrong IP address / can't reach workstation"** - IP conflict with another device. Check DHCP leases and static configurations on other hosts. Create DHCP reservation with MAC binding to

prevent future conflicts.

4. **"QEMU guest agent not responding"** - Agent may not be running inside the VM. Must enable from within the desktop GUI first - can't bootstrap remotely without it.
5. **"Mount fails with protocol error"** - SMB version mismatch. Add `vers=3.0` (or `vers=2.1`) explicitly to the fstab mount options to force a specific protocol version.

Security Considerations

Investigation Isolation

- Forensic workstation runs on separate VM from production services
- No direct network path between forensic tools and production infrastructure
- Evidence enters through controlled share, not direct disk access

Evidence Integrity

- Centralized evidence share provides single source of truth
- Automount ensures consistent access without manual intervention
- Future: write-once storage and automated hashing for chain of custody

Access Control

- Web desktop behind SSO authentication
- SSH requires explicit enablement (disabled by default)
- Evidence upload portal uses zero-trust access control (email verification)
- Multiple access methods provide redundancy without sacrificing security

State Recovery

- Snapshot before investigation preserves clean workstation state
- Rollback capability ensures contaminated environments can be restored
- VM isolation means workstation corruption doesn't affect production

---

Runbook

If Evidence Share Is Empty

1. **Check mount status -** `mount | grep Evidence`

2. **Try manual access** - `ls /home/csi/Desktop/Evidence/` (triggers automount)
3. **Verify automount unit** - `systemctl status home-csi-Desktop-Evidence.automount`
4. **Test network connectivity** - `ping <EVIDENCE_IP>`
5. **Check credentials file** - Verify `/root/.smbcreds` exists and has 600 permissions

If Web Desktop Won't Connect

1. **Verify Guacamole service** - Check tunnel/proxy status
2. **Test direct SSH** - If SSH works, tunnel is the issue
3. **Check SSO authentication** - Verify identity provider is responding
4. **Review browser console** - WebSocket errors indicate connection issues
5. **Try alternate access** - Use VPN + direct RDP as fallback

How to Snapshot Before Investigation

```
# From Proxmox host
qm snapshot <VMID> pre-case-$(date +%Y%m%d-%H%M)

# Verify snapshot exists
qm listsnapshot <VMID>

# After investigation, rollback if needed
qm rollback <VMID> <SNAPSHOT_NAME>
```

How to Upload Evidence via Web Portal

1. Navigate to `<UPLOAD_PORTAL>` in browser
2. Authenticate via email verification (Cloudflare Access)
3. Select files for upload (drag-and-drop supported)
4. Verify upload completion
5. Files appear in evidence share within seconds

How to Start a New Case

1. Open CSI Linux case manager from desktop
2. Enter case name and investigator information
3. Case directory created with standard structure
4. Copy relevant evidence from share to case directory
5. Begin analysis with appropriate tools
6. Document findings in case notes

Appendix

Glossary

Term	Definition
CSI Linux	Ubuntu-based distribution purpose-built for digital forensics and OSINT
DFIR	Digital Forensics and Incident Response
OSINT	Open Source Intelligence - gathering information from public sources
Volatility	Memory forensics framework for analyzing RAM dumps
Autopsy	GUI forensic platform built on Sleuth Kit for disk analysis
Sleuth Kit	CLI tools for disk forensics (file system analysis, timeline creation)
Guacamole	Clientless remote desktop gateway (access via web browser)
Steganography	Hiding data within other files (images, audio, video)
YARA	Pattern matching language for malware identification
File Carving	Recovering files from disk images based on file signatures

MITRE ATT&CK Mapping (Forensic Tool Coverage)

Technique ID	Name	Tools That Detect/Analyze
T1005	Data from Local System	Autopsy, Sleuth Kit, Foremost, Scalpel
T1039	Data from Network Shared Drive	Wireshark, network log analysis
T1083	File and Directory Discovery	Autopsy timeline, Sleuth Kit fls
T1057	Process Discovery	Volatility pslist, pstree, psxview
T1049	System Network Connections Discovery	Volatility netscan, Wireshark
T1003	OS Credential Dumping	Volatility hashdump, mimikatz plugin
T1027	Obfuscated Files or Information	YARA, Binwalk, strings analysis
T1071	Application Layer Protocol	Wireshark, Zeek log analysis
T1059	Command and Scripting Interpreter	Volatility cmdline, consoles

Tool-to-Technique Coverage Matrix

Tool	Primary Techniques Covered
Autopsy	T1005, T1083, T1027 (disk-level data, files, hidden content)
Volatility	T1057, T1049, T1003, T1059 (process, network, credentials, commands)
Wireshark	T1039, T1071, T1049 (network traffic, protocols, connections)
YARA	T1027 (obfuscated/packed malware identification)
Hashcat/John	T1003 (credential hash analysis)
Binwalk	T1027 (embedded files, firmware extraction)
ExifTool	T1005 (metadata extraction from files)

Artifacts Produced

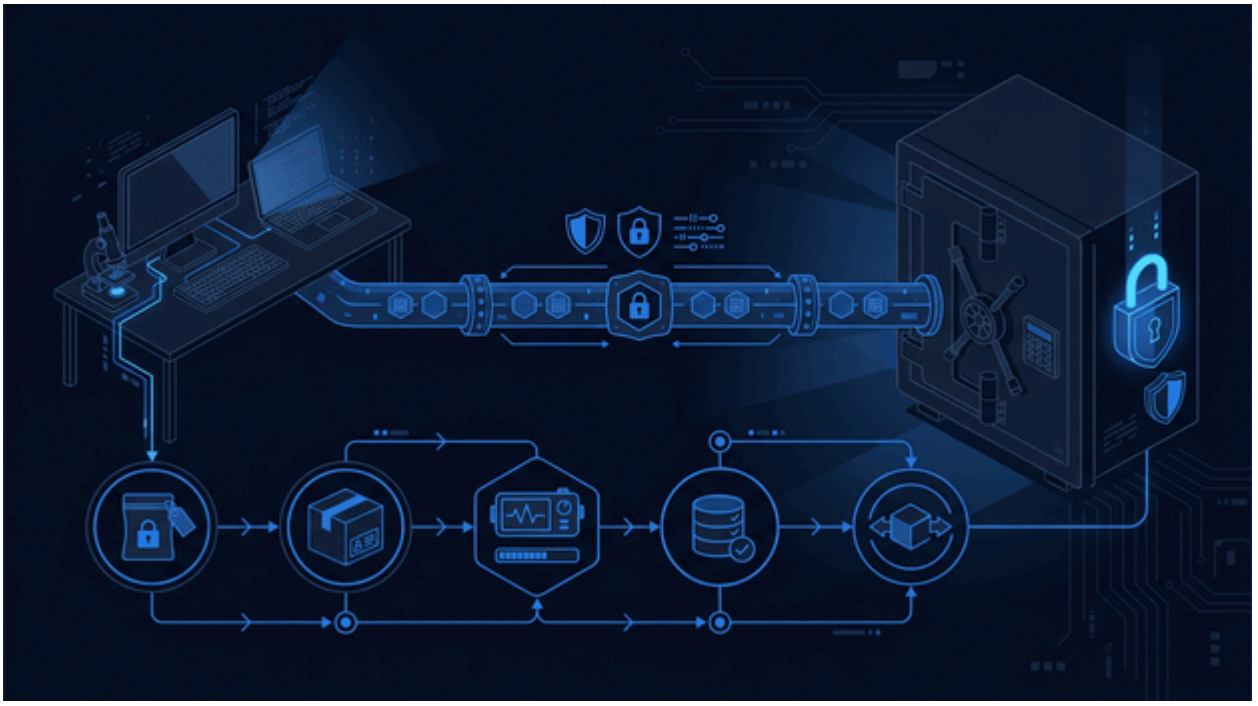
- **Forensic Workstation VM** - CSI Linux deployment with 50+ tools configured
- **Evidence Share Infrastructure** - Centralized SMB share mounted to workstation desktop
- **Web Upload Portal** - Zero-trust evidence submission for external investigators
- **Systemd Automount Configuration** - Reliable network mount across reboots
- **SSH Access Configuration** - Drop-in config enabling remote CLI access
- **Case Management Workflow** - Documented procedure for investigation lifecycle

---

Building your own forensic lab? CSI Linux won't give you everything, but it gives you a running start. The network mount gotchas will still bite you though - hope the automount notes help.

Building a Forensic Evidence Pipeline: Workstation to Evidence Locker

Tue, Jul 7, 9:00 AM EDT · https://scottslab.io/posts/forensic-evidence-pipeline-workstation-to-locker

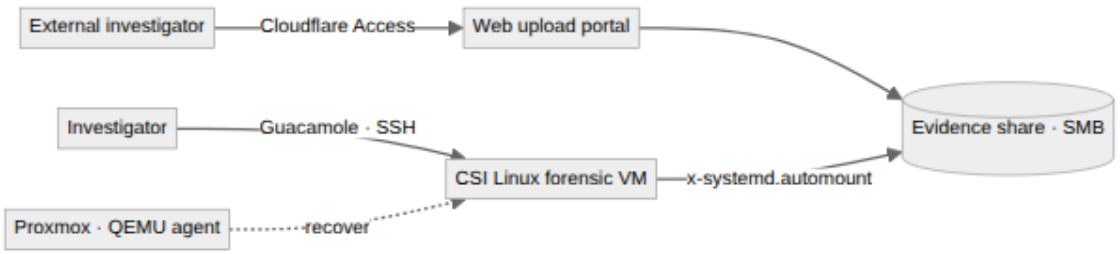


TL;DR

The evidence share came back empty after every reboot, which turned out to be a boot race rather than anything to do with forensics. x-systemd.automount with nofail and an explicit vers=3.0 fixed it and cut boot time roughly in half. The rest of the work was getting reliably back onto the box: an sshd drop-in, an IP collision with an identity container, and a Cloudflare Access portal so outside investigators can hand over files without touching the network.

Every time the forensic workstation rebooted, the evidence folder came back empty. Not an error, not a prompt, just an empty directory where a mounted SMB share used to be. The mount was firing at boot before the network was up, failing silently, and staying failed until I noticed and remounted it by hand. Fine on a lazy Tuesday. Not fine at the start of an investigation, when the first thing an investigator sees is a workstation that can't reach the evidence. That's the moment people start copying case files to the local disk and breaking chain of custody to save five minutes.

That's the whole point of an evidence pipeline: it has to be invisible. Open the folder, the case files are there. Close the laptop, come back tomorrow, they're still there. No ceremony, no "did you try remounting the share." Getting to that meant fixing four things that had nothing to do with forensics and everything to do with the workstation being reachable and reliable.



The mount that wouldn't stay

The empty-folder problem was a boot race, and the fix is a mount option I now reach for by default. Instead of a hard CIFS mount at boot, `x-systemd.automount` registers a lightweight automount point immediately and defers the real mount until something actually touches the directory:

```
//<EVIDENCE_HOST>/evidence  /mnt/evidence  cifs
credentials=/root/.smbcreds,uid=1000,gid=1000,vers=3.0,x-systemd.automount,x-
systemd.mount-timeout=30,nofail  0  0
```

`nofail` keeps a missing share from stalling the boot, `mount-timeout=30` gives up instead of hanging, and (the part I missed the first time) `NetworkManager-wait-online.service` actually has to be enabled, or nothing waits for the network at all. The explicit `vers=3.0` cleared a separate silent failure where the server refused version negotiation. Net effect: the share mounts on first access every time, and boot dropped from about ninety seconds to forty-five, because it's no longer sitting through a mount timeout that was always going to fail.

Getting back in

Before I could fix any of that, I had to get onto the box, and CSI Linux ships with password SSH disabled. Rather than edit the shipped `sshd_config`, a drop-in does it without touching the original file:

```
cat > /etc/ssh/sshd_config.d/99-local.conf << 'EOF'
PasswordAuthentication yes
PermitRootLogin prohibit-password
EOF
systemctl restart sshd
```

Then the workstation started answering on the wrong address: .131 instead of the .112 I expected. Something else had claimed .112: an identity-provider container with a static config living entirely

outside DHCP. Two systems, one address, intermittent failures that are miserable to chase. I moved the container to .117 and pinned the workstation to .112 with a DHCP reservation on the firewall, so the binding lives in one place instead of scattered across VM configs. The lesson I keep relearning: before you assign an IP, check every source that can hand one out (leases, reservations, and static container configs), not just the DHCP table.

With SSH back and the address stable, the rest of the management chain fell into place: the QEMU guest agent reporting to Proxmox, and Guacamole serving the full desktop in a browser for the tools that need one: Autopsy, Wireshark, Maltego. Three independent ways back onto the workstation, which matters when the thing you're recovering is the machine you do recovery from. One caveat worth stating plainly: you have to enable the QEMU agent from inside the VM first. You can't remotely bootstrap an agent that isn't running.

Evidence in, findings out

External investigators don't get network access; they get a web upload portal behind Cloudflare Access. Email-based identity verification, no VPN, no inbound firewall ports opened: the tunnel dials out. Files they upload land in the same evidence share the workstation automounts, so there's one path in and one source of truth, and the workstation itself never has to be exposed to make that work.

What I'd add next

The pipeline is reliable now, but it isn't yet self-proving. The next piece is chain of custody: SHA-256 every file on upload and log access with timestamps, because evidence that can't demonstrate its own integrity is just data. After that, a Wazuh rule to alert on mount failures, so I hear about a broken share from the SIEM instead of from an investigator, and an Ansible playbook to rebuild the whole workstation from scratch in minutes instead of hours.

Automount was the real unlock, though. It's a patient mount option: it waits until you actually need the files, then connects quietly when you're not looking. That's exactly the behavior I want from forensic infrastructure. Do the job, don't make a fuss about it.

Building your own DFIR lab? The persistent mount problem bites everyone eventually. Hope this saves you a few hours of boot-timeout debugging.

Hacking with AI: What Security Engineers Get Wrong (and What Moltbot Proved)

Tue, Jul 28, 6:40 PM EDT · https://scottslab.io/posts/hacking-with-ai-security-engineers-guide

TL;DR

Most AI red teaming vendors are selling stock jailbreak demos and cannot tell a chatbot from an agent holding root. Moltbot/OpenClaw is the case study that makes the point: 85K GitHub stars, hundreds of exposed instances, plaintext credentials, and a poisoned supply chain inside eight hours. OWASP's Vendor Evaluation Criteria (v1.0, January 2026) is the first framework that separates real testing from theater, and the lethal trifecta (private data, untrusted input, external comms) is the model worth carrying into any agentic AI review.

Published: July 2026 **Category:** Security Operations **Reading Time:** 20 minutes

---

Executive Summary

- The AI red teaming vendor landscape is mostly noise: stock jailbreak demos, "full coverage" claims, and tools that can't distinguish a chatbot from an agentic system with root access
- Moltbot/OpenClaw is the case study that proves why this matters: 85K GitHub stars, hundreds of exposed instances, plaintext credentials, supply-chain poisoning in 8 hours, and Google's VP of security engineering calling it "info stealer malware in disguise"
- OWASP's Vendor Evaluation Criteria for AI Red Teaming (v1.0, January 2026) finally gives us a framework to separate real security testing from security theater

- The "lethal trifecta" (private data access + untrusted input + external communication) is the mental model every security engineer needs for evaluating agentic AI
- This post provides practical dos and don'ts grounded in both the OWASP framework and Moltbot's real-world failures

---

Context: Why This Matters Now

AI adoption in security operations has outpaced security's ability to evaluate it. Every vendor claims AI-powered threat detection, AI-assisted red teaming, AI-driven vulnerability assessment. Most of it is marketing wrapped around an API call to a foundation model.

Meanwhile, the systems we're supposed to be securing have gotten dramatically more complex. We're not just testing chatbots anymore. We're testing agentic systems with tool-calling capabilities, persistent memory, multi-agent orchestration, and integration with everything from calendars to messaging apps to file systems.

The gap between "AI security testing" as sold and "AI security testing" as needed has become a chasm.

Three things happened in January 2026 that crystallized this problem:

1. **OWASP published the Vendor Evaluation Criteria for AI Red Teaming Providers & Tooling v1.0**: finally giving the industry a framework to distinguish real adversarial evaluation from jailbreak demos

2. **Moltbot/OpenClaw went viral**: an open-source agentic AI assistant with 85K+ GitHub stars that could browse the web, manage calendars, send messages, execute scheduled tasks, and maintain persistent memory. It became the poster child for what happens when agentic AI meets reality.

3. **Security researchers found hundreds of Moltbot instances exposed to the internet**: with at least 8 completely open, leaking API keys, credentials, and conversation history. A supply-chain attack via the official plugin registry compromised 16 developers in 7 countries within 8 hours.

This post is my attempt to synthesize what we should have learned: how to use AI effectively in security operations, what the red flags are, and why the OWASP framework matters. All of it is grounded in the Moltbot debacle as the practical example that makes the theory concrete.

Case Study: Moltbot/OpenClaw (When Agentic AI Meets Reality)

Background

Moltbot (also known as Clawdbot and OpenClaw across various forks) is an open-source agentic AI personal assistant created by Peter Steinberger. It went viral in January 2026, accumulating 85K+

GitHub stars. The appeal was obvious: a personal AI assistant that could actually *do things*, from browsing the web and managing your calendar to reading and writing files, sending messages via WhatsApp and Telegram, executing scheduled tasks, and maintaining persistent memory across sessions.

It's the kind of tool that makes you feel like you're living in the future. It's also a security nightmare.

The "Lethal Trifecta"

Simon Willison coined the term, and Palo Alto Networks highlighted it in their analysis: the **lethal trifecta** is what happens when an AI system has:

1. **Access to private data** (files, credentials, conversation history)
2. **Exposure to untrusted content** (web browsing, message ingestion, plugin execution)
3. **Ability to communicate externally** (send messages, make API calls, exfiltrate data)

Moltbot has all three. By design.

Any AI system with this combination should be treated as a high-risk deployment. It's not a chatbot. It's an agent with the capability profile of malware.

The Security Failures

The discoveries came fast once security researchers started looking:

Exposed Instances (Jamieson O'Reilly, DVULN) Shodan scans found hundreds of Moltbot instances exposed to the internet. At least 8 were completely open: no authentication whatsoever. API keys, Telegram bot tokens, Signal configurations, and full conversation histories were accessible to anyone who found them.

Localhost Trust Bypass Moltbot's gateway authentication trusted all localhost connections by default. This sounds reasonable until you realize the common deployment pattern: reverse proxy in front of Moltbot. The reverse proxy connects from localhost. Every request through the proxy got full unauthenticated access to credentials, conversation history, and command execution.

This is the same class of vulnerability we've been finding in traditional web applications for twenty years. Localhost ≠ secure.

Plaintext Credential Storage Secrets were stored in plaintext Markdown and JSON files on the local filesystem. No encryption, no secure credential storage, just files. Commodity infostealers (RedLine, Lumma, Vidar) already target browser credential stores. Moltbot's files are equally trivial to exfiltrate. Hudson Rock reported infostealers specifically adding Moltbot paths to their collection routines.

Supply-Chain Poisoning (8 Hours to Compromise) A security researcher demonstrated the attack: upload a malicious "skill" to the official MoltHub/ClawdHub registry, artificially inflate the download count, and wait. Within 8 hours, 16 developers in 7 countries had installed the poisoned skill.

No code review. No signing. No sandbox. The official plugin ecosystem was a supply-chain attack waiting to happen.

Persistent Memory as Attack Vector Moltbot maintains persistent memory across sessions: context about the user, previous conversations, learned preferences. This creates a novel attack pattern: time-shifted prompt injection.

A malicious payload can be written to memory during content ingestion (processing a webpage, reading a message, executing a plugin). The payload sits dormant until agent state and available tools align for detonation. It's a logic bomb, but for AI agents.

Moltbook: Agent-to-Agent Contamination Moltbook is an "agent social network" where Moltbot instances can share data, coordinate behaviors, and create shared fictional contexts. From a security perspective, it's an inter-agent contamination channel. Compromise one agent, potentially influence others through shared context.

Fake VSCode Extension Threat actors distributed a trojanized VSCode extension called "ClawBot Agent – AI Coding Assistant." Users installing what they thought was a coding assistant got malware instead. Classic supply-chain attack, AI-flavored.

Shadow IT at Scale Token Security reports that 22% of enterprise customers have employees actively using Moltbot without IT approval. This isn't a small-scale experiment. It's shadow IT with root-level system access, deployed across nearly a quarter of enterprises surveyed.

Why This Matters for Red Teaming

Every OWASP red flag for advanced system testing applies to Moltbot:

- Tool-calling misuse? Moltbot's entire value proposition is tool calls.
- Capability escalation? Plugins can request additional permissions.
- Supply-chain poisoning? Demonstrated in 8 hours.
- Persistent memory attacks? The architecture enables them by design.
- Agent-to-agent contamination? Moltbook exists.
- Missing human-in-the-loop guardrails? It's designed to operate autonomously.

Any vendor that can't test for these attack classes would miss every critical vulnerability in Moltbot. Stock jailbreak libraries wouldn't find a single one of these issues. Chatbot-level testing would produce a clean report for a system that Google's VP of security engineering described as "info stealer malware in disguise."

That's not hyperbole. That's an accurate assessment of the permissions and attack surface.

> **If you only read one section, read the Don'ts below.** The Dos are useful, but the Don'ts are where security programs fail.

The Dos: How to Use AI Effectively in Security

DO use AI for automating repetitive reconnaissance

OSINT gathering at scale, subdomain enumeration, service fingerprinting: these are perfect AI use cases. The work is repetitive, the patterns are learnable, and human creativity isn't required for the collection phase.

DO use AI-assisted tools for generating diverse adversarial test cases

Humans are pattern-matchers. We write the test cases we can imagine, which means we miss the ones we can't. AI can generate test case diversity beyond what a single human would produce, including edge cases and input mutations that wouldn't occur to a manual tester.

DO use AI for log analysis and anomaly detection

Security teams drown in logs. AI excels at pattern recognition across large datasets: finding the needle in the haystack, or more accurately, finding the haystack that shouldn't exist.

DO use AI to draft initial threat models, then validate with human expertise

AI can enumerate attack surfaces, identify common vulnerability patterns, and structure threat models faster than manual documentation. The key word is "draft." Human validation is non-negotiable.

DO use AI for regression testing

Once you've identified a vulnerability and remediated it, AI can efficiently test for regression across builds, configurations, and related systems. This is automation at its best: repeatable verification of known issues.

DO pair automated AI tools with human adversarial creativity

OWASP explicitly states: automation scales, humans provide novelty. The winning combination is AI handling breadth (volume, consistency, repetition) while humans handle depth (creativity, context, business logic).

DO demand quantitative, reproducible metrics

Any AI security tool should provide metrics you can reproduce: pass@k, average turns to jailbreak, retrieval success rate, unsafe tool-call rate. If a vendor can only offer vibes-based scoring ("our AI thinks the system is 87% secure"), walk away.

DO evaluate vendors against OWASP's criteria

The framework exists now. Use it. Technical competence, methodology coverage, adversarial creativity, threat modeling realism, evaluation rigor, tooling quality, data governance, transparency: each criterion has specific indicators and red flags.

DO evaluate agentic AI tools like you'd evaluate any privileged software

Moltbot has system-level permissions. Treat its deployment like you'd treat deploying any other privileged software: threat model the permissions, audit the authentication, map the attack surface. The fact that it's "AI" doesn't change the security engineering fundamentals.

DO treat persistent memory as a security-critical data store

Agent memory is essentially a credential cache that can be poisoned. It contains context about the user, learned preferences, and potentially sensitive information from previous sessions. Protect it accordingly.

The Don'ts: Where AI Falls Short or Creates Risk

DON'T treat AI-generated findings as ground truth

AI hallucinates. AI misclassifies. AI lacks the context to understand business impact. Every AI-generated finding requires human verification before it becomes actionable. This isn't a criticism of AI. It's a recognition of current capabilities.

DON'T assume stock jailbreak libraries constitute comprehensive red teaming

OWASP explicitly identifies this as a red flag. Jailbreak libraries test whether a model can be manipulated into generating restricted content. They don't test tool-calling misuse, capability escalation, supply-chain integrity, or any of the attack classes that matter for agentic systems.

Stock jailbreaks wouldn't have found a single Moltbot vulnerability.

DON'T use AI tools that can't explain their methodology

If a vendor can't articulate how their tool works, what it tests, and how findings are generated, you're buying a black box. Black boxes don't survive incident response. When the finding is wrong (and some will be), you need to understand why.

DON'T ignore the difference between simple and advanced GenAI systems

A chatbot and an agentic system with tool-calling, MCP, and multi-agent orchestration require fundamentally different testing approaches. OWASP categorizes these as "simple GenAI systems" vs. "advanced GenAI systems" for a reason.

Moltbot proves this: chatbot-level testing would produce a clean report for a system with critical vulnerabilities across every attack surface.

DON'T trust vendors claiming "full coverage" or "one-click red teaming"

Red flag per OWASP criteria. Full coverage doesn't exist. One-click anything in security is marketing, not methodology.

DON'T skip threat modeling and jump straight to automated scanning

Tools find what they're designed to find. Without a threat model, you don't know what to look for, what matters, or whether the tool's coverage matches your risk profile.

DON'T feed sensitive production data into third-party AI tools without understanding retention

Data governance matters. Where does your data go? How long is it retained? Who can access it? Can it be used for training? These questions apply to any third-party AI tool, including security testing platforms.

DON'T assume tool calls and MCP outputs are inherently safe

Moltbot's entire attack surface is tool calls and API integrations. The agent makes decisions about what tools to invoke, with what parameters, based on potentially untrusted input. Every tool call is an execution decision that could be manipulated.

DON'T rely on AI-as-judge scoring without human oversight

For complex emergent behaviors (especially in multi-agent systems), AI scoring isn't sufficient. OWASP notes that multi-agent system testing with automation-only is rated Low-Medium effectiveness. Humans remain essential for evaluating novel attack patterns.

DON'T confuse scanning volume with risk reduction

Running more scans doesn't inherently improve security posture. Finding the same low-severity issues 1,000 times doesn't matter if you're missing the one critical vulnerability that leads to breach.

DON'T install agentic AI assistants without IT/security review

22% of enterprises have employees running Moltbot without approval (Token Security). This is shadow IT with root access. The permissions these tools request are the permissions malware requests: file system access, credential storage, network communication, scheduled execution.

DON'T trust community plugin marketplaces without supply-chain validation

The MoltHub poisoning attack took 8 hours to compromise 16 developers across 7 countries. No code signing, no review process, no sandbox. If your agentic AI tool supports plugins, its plugin ecosystem is part of your attack surface.

DON'T assume localhost equals secure

Moltbot's default trust of localhost connections behind reverse proxies is a vulnerability class we've been documenting for two decades. "It only listens on localhost" is not a security control when reverse proxies, containers, and service meshes all connect from localhost.

What Good AI Red Teaming Actually Looks Like

Per OWASP's criteria, quality AI red teaming:

Covers both simple and advanced systems Chatbots, RAG applications, and copilots are simple systems. Tool-calling agents, MCP-enabled systems, and multi-agent orchestrations are advanced systems. Testing approaches must match system complexity.

Tests interactions and workflows, not just individual outputs Single-turn jailbreaks are table stakes. Real adversarial evaluation tests multi-step workflows, interaction patterns across sessions, and emergent behaviors from component interactions.

Includes multi-step adversarial workflows Real threat actors don't send one malicious prompt. They build context, establish trust, manipulate state, and then exploit. Testing should reflect this reality.

Maps findings to business risk A vulnerability without business impact is just a curiosity. Quality red teaming quantifies risk and provides actionable remediation, not just a list of theoretical issues.

Uses established frameworks OWASP Top 10 for LLM Applications, MITRE ATLAS, NIST AI RMF. These frameworks exist to provide common vocabulary and coverage verification.

Provides reproducible artifacts Full message traces, tool-call provenance, input sequences that trigger the vulnerability. If you can't reproduce it, you can't verify the fix.

For agentic systems specifically:

- Tests supply-chain integrity of plugin/skill ecosystems
- Evaluates persistent memory poisoning attacks
- Covers cross-session attack patterns
- Maps privilege escalation through tool integrations
- Verifies authentication under realistic deployment patterns (including reverse proxy scenarios)
- Tests for the lethal trifecta explicitly: private data + untrusted input + external communication

Moltbot vs. OWASP Criteria: A Mapping

OWASP Criterion	What It Means	Moltbot Relevance
Technical Competence	Vendor understands tool-calling semantics, MCP internals, privilege escalation	Moltbot's localhost bypass, plaintext credentials, and MoltHub poisoning all require this understanding to test
Methodology & Coverage	Testing covers tool-calling misuse, supply-chain, memory attacks	Stock jailbreaks find zero of Moltbot's actual vulnerabilities
Adversarial Creativity	Domain-specific attack strategies beyond canned tests	Time-shifted prompt injection via persistent memory requires creative threat modeling
Threat Modeling	Extends beyond prompt injection to systemic failures	Moltbot's failures are architectural, not just input validation
Evaluation Rigor	Quantitative metrics for unsafe tool-call rate, capability misuse	Moltbook contamination requires specific measurement frameworks
Tooling Quality	Supports tool-call replay and introspection	Testing Moltbot's API integrations requires tracing tool-call chains across sessions
Data Governance	Clear retention and handling policies	Testing Moltbot means accessing user data, conversation history, credentials
Transparency	Methodology and findings are explainable	Black-box testing of Moltbot would miss the architectural issues entirely

Vendor Evaluation Quick Reference

Condensed from OWASP's Vendor Evaluation Criteria v1.0:

Technical Competence

- ☐ Demonstrates understanding of LLM architecture, fine-tuning, inference
- ☐ For advanced systems: understands tool-calling, MCP, multi-agent orchestration
- ☐ Can explain how their testing identifies capability escalation and unsafe execution

Methodology

- ☐ Goes beyond stock jailbreak libraries

- ☐ Covers multi-turn adversarial interactions
- ☐ Tests tool-calling misuse, not just content generation
- ☐ For agentic systems: includes supply-chain, memory, and cross-session attacks

Evaluation Quality

- ☐ Provides quantitative metrics (pass@k, turns to jailbreak, retrieval success)
- ☐ Findings are reproducible with full message traces
- ☐ Maps vulnerabilities to business risk with remediation guidance

Red Flags

- ☐ "Full coverage" claims
- ☐ "One-click red teaming"
- ☐ AI-generated evaluations with no human oversight
- ☐ Can't explain methodology or reproduce findings
- ☐ No differentiation between simple and advanced systems

MITRE ATT&CK Relevance

Agentic AI doesn't create new attack techniques. It changes the economics of existing ones.

Technique ID	Name	AI Relevance
T1595	Active Scanning	AI automates reconnaissance at scale; Moltbot can browse/scan autonomously
T1592	Gather Victim Host Information	Persistent memory stores host context; memory poisoning enables exfiltration
T1589	Gather Victim Identity Information	Conversation history, calendar access, contact lists available to agents
T1190	Exploit Public-Facing Application	Hundreds of Moltbot instances exposed via Shodan; localhost trust bypass
T1059	Command and Scripting Interpreter	Agents execute commands; tool-calling is command execution
T1195	Supply Chain Compromise	MoltHub skill poisoning: 16 developers, 7 countries, 8 hours

Technique ID	Name	AI Relevance
T1556	Modify Authentication Process	Localhost trust bypass is authentication modification
T1552	Unsecured Credentials	Plaintext credential storage in Markdown/JSON files

The pattern: techniques that previously required manual effort or custom tooling are now automatable through legitimate AI agent functionality. The agent doesn't know it's being malicious. It's executing instructions from poisoned memory or malicious plugins.

What I Learned

1. **The gap between "AI red teaming" as marketed and as needed is enormous.** Most vendors are selling jailbreak demos for chatbots while the industry is deploying agentic systems with root access. OWASP's framework finally gives us criteria to distinguish the two.

2. **Moltbot is a preview of enterprise AI security problems.** 22% of enterprises already have shadow deployments. The same architectural patterns (persistent memory, tool-calling, plugin ecosystems) are coming to enterprise AI whether security teams are ready or not.

3. **The "lethal trifecta" is the mental model for agentic AI risk.** Private data access + untrusted input + external communication = high-risk system. If all three are present, treat it with the same rigor you'd treat any privileged software deployment.

4. **Localhost trust is not a security control.** Moltbot made this mistake; so do many traditional applications. In a world of reverse proxies, containers, and service meshes, localhost origin means nothing.

5. **Supply-chain attacks on AI ecosystems are trivially easy.** The MoltHub poisoning attack required no special access, no zero-days, no sophisticated techniques. Upload, inflate downloads, wait. 8 hours to 16 compromises.

6. **Human-AI teaming beats either alone.** OWASP's comparison matrix confirms what practitioners know: automation provides scale, humans provide creativity. For advanced systems, automation-only is insufficient.

7. **The vendors who can test agentic AI are the vendors who understand agentic AI.** Technical competence isn't just about LLM internals: it's about understanding tool-calling semantics, MCP, privilege boundaries, and multi-agent coordination. If a vendor can't explain how Moltbot's architecture creates risk, they can't test systems like it.

Security Considerations

Responsible AI Use in Security

AI tools in security operations should enhance human capability, not replace human judgment. Every AI-generated finding requires validation. Every AI-assisted decision requires oversight for high-stakes actions.

Data Handling

Production data in AI tools creates risk. Understand retention policies, access controls, and training data usage for any third-party AI platform. For internal tools, apply the same data classification and handling requirements as any sensitive system.

Legal Boundaries

AI-assisted testing doesn't change the rules of engagement. Scope, authorization, and legal boundaries apply to AI-generated actions the same as human-initiated ones. "The AI did it" is not a defense for out-of-scope testing.

Shadow AI Governance

With 22% of enterprises reporting unauthorized Moltbot usage, shadow AI is a material risk. Policy recommendations:

- Inventory agentic AI tools with the same rigor as shadow IT
- Require security review for any tool with file system access, credential storage, or network communication
- Block known-risky plugin repositories at the network level
- Monitor for indicators of agentic AI deployment (process names, network patterns, file paths)

Enterprise Policy for Agentic AI

Assumption: Organizations will increasingly adopt agentic AI tools. Proactive policy should address:

- Approval workflow for agentic AI deployment
- Minimum security requirements (authentication, credential storage, plugin sources)
- Monitoring and logging requirements
- Incident response procedures for AI agent compromise
- Employee training on agentic AI risks

References and Further Reading

Frameworks & Standards

- OWASP Top 10 for LLM Applications
- OWASP Vendor Evaluation Criteria for AI Red Teaming v1.0 (January 2026)
- MITRE ATLAS
- NIST AI Risk Management Framework
- EU AI Act

Moltbot/OpenClaw Research

- Palo Alto Networks: "Lethal Trifecta" analysis and risk assessment
- Jamieson O'Reilly / DVULN: Exposed instance discovery via Shodan
- Hudson Rock: Infostealer targeting of Moltbot credential stores
- Token Security: Enterprise shadow deployment statistics (22%)
- BleepingComputer: Moltbot security exposure reporting
- The Register: Technical analysis of Moltbot vulnerabilities
- SOCPrime: Detection content for Moltbot-related threats
- RedTeamWorld: Supply-chain attack demonstration

Conceptual Framework

- Simon Willison: "Lethal Trifecta" framework for evaluating AI agent risk

---

Building AI into your security program? The OWASP criteria are the starting point. Moltbot is the case study. The rest is up to you.

Identification, Authentication, Authorization — and the Accounting Nobody Talks About

Mon, Aug 3, 9:00 AM EDT · https://scottslab.io/posts/identification-authentication-authorization-accounting

> **TL;DR**
>
> Identification is the claim, authentication is the proof, and authorization is the yes/no on what you're allowed to do. Mix them up and you'll debug the wrong system. The AAA framework adds accounting, the logging that lets you reconstruct who did what after the fact, and it's the piece that quietly matters most once something goes wrong.

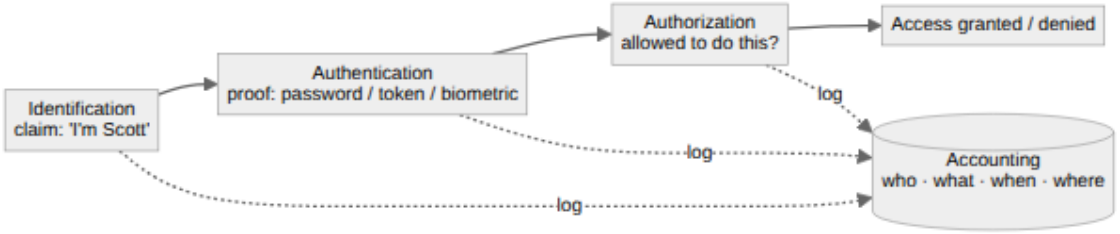
Every access problem I've ever debugged comes down to the same three questions asked in order, and most of the confusion I see comes from smearing them together: it shows up in tickets, in postmortems, in people cramming for the exam. Someone reports "authentication is broken" when the account simply wasn't authorized. That's not pedantry: the fix lives in a completely different system depending on which step actually failed.

Identification is just the claim. I type a username, tap a badge, say "I'm Scott." No proof yet. A username was never meant to be secret, which is exactly why it's usually something as guessable as first-initial-last-name. It's the label everything else hangs on, nothing more.

Authentication is the proof behind the claim. This is where the password, the token, the fingerprint shows up. That's the evidence that I actually am the identity I claimed. The clean way to keep the first

two straight is to look at the mechanism: a username identifies, a password authenticates. One names you, the other proves it.

Authorization is the separate question of whether that proven identity is allowed to do the thing. You can authenticate perfectly and still be told no. A valid login to a system you have no permissions on is authentication succeeding and authorization denying. Conflate the two and you'll spend an hour "fixing" the wrong one.



The framework people quote is AAA (authentication, authorization, accounting), and it's the word that gets dropped that matters most after an incident. Accounting is the logging: who did what, when, from where. The first three happen in the moment; accounting is what lets you reconstruct the moment later. Every detection rule I've written and every investigation I've run leans on that trail existing. The first three decide access. The fourth is how you answer "what happened" when access gets abused. If it isn't being recorded, the honest answer is that you don't know.

Proving Identity, Part 1: Usernames and Access Cards

Mon, Aug 3, 4:00 PM EDT · https://scottslab.io/posts/usernames-and-access-cards

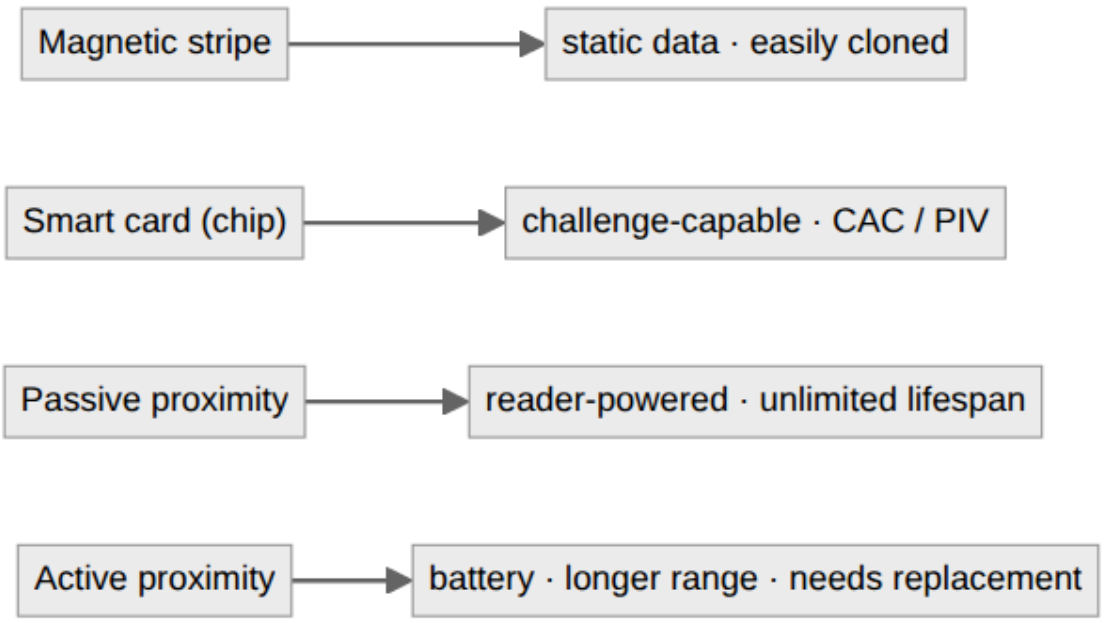
> **TL;DR**
>
> A username is a label, not a secret, so stop hardening it. The secret belongs in the authentication step. Access cards run from trivially cloneable magnetic stripes to challenge-capable smart cards, with passive and active proximity cards trading read range against a battery you'll eventually have to replace. Pick the mechanism whose failure mode you can actually live with.

Identification is the cheap step, but the mechanisms people use for it have wildly different threat models, and it's worth knowing which is which before you trust any of them. Start with the two most common: the username you type and the card you carry.

The username is the baseline, and the thing to internalize is that it was never meant to be a secret. First-initial-last-name is fine. The username is a label, not a control. Hiding it or rotating it is effort spent on the wrong layer. The secret belongs in the authentication step. If your security depends on nobody guessing the username, you don't have a security control, you have a naming convention.

Access cards are where physical and logical security meet, and the technology matters more than the plastic suggests. A magnetic stripe is trivial to clone with gear off the internet. It's static data on a strip, so anything that can read it can copy it. I wouldn't trust one to guard anything that matters. A smart card is a real step up: an embedded chip that participates in a challenge instead of just handing over static

data, which is why the DoD's Common Access Card is a chip, not a stripe. You can't clone what won't reveal its secret.



Proximity cards split one more level down, and it's a convenience-versus-maintenance trade. A passive prox card draws its power from the reader, so it has no battery and effectively lasts forever, at the cost of short range. An active prox card carries its own battery for a longer read range, which helps with gates and vehicles. But that battery eventually dies, so now you've signed up for a replacement cycle across every badge you issued.

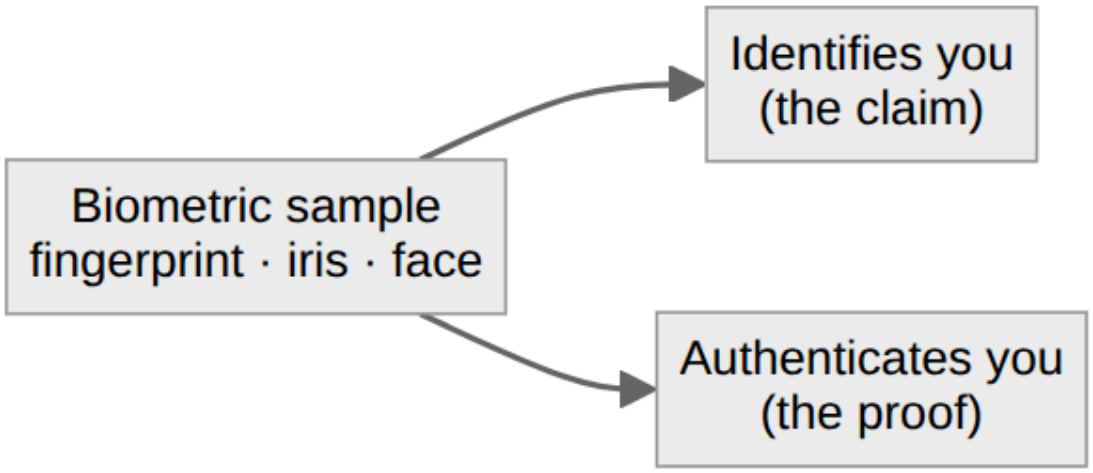
None of these are interchangeable. A username you can shout across the room, a magstripe you can clone in a parking lot, a smart card you can't easily forge. Pick the mechanism whose failure mode you can actually live with, and don't let the fact that a badge "feels" secure stand in for knowing how it works.

Biometrics: When You Are the Credential

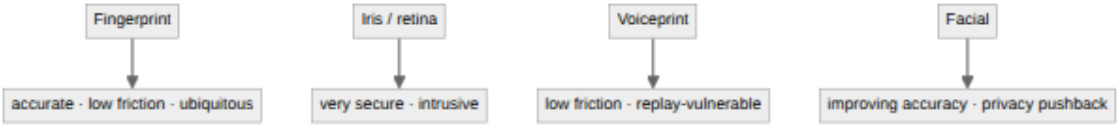
Tue, Aug 4, 9:00 AM EDT · https://scottslab.io/posts/biometrics-something-you-are

> **TL;DR**
>
> A biometric does identification and authentication in a single gesture, which is why it counts as "something you are." Every modality trades security against how much users hate it: fingerprints won on convenience, iris scans on security, voiceprints lose to replay. And unlike a password, a leaked biometric is permanent, so treat it as one strong factor, never the only one.

Biometrics are the odd one out among identification mechanisms, because a single fingerprint does two jobs at once: it identifies you and it authenticates you. That's why they file under "something you are." A username needs a password behind it; a biometric is the claim and the proof in the same gesture.



The engineering reality is that every modality trades security against how much users hate using it, and that trade decides where each one actually gets deployed. Fingerprints won the consumer world because they're accurate enough and nobody minds touching a sensor: low friction, good-enough security, everywhere. Iris and retina scans are far harder to fool and far more intrusive, so they live where the security bar justifies making people lean into a scanner. Voiceprints are easy to accept and easy to defeat with a recording, which is why they rarely stand on their own. Facial recognition keeps getting more accurate and keeps making people uneasy, and that discomfort is its own deployment cost. There's a long tail beyond these (vein patterns, hand geometry, even gait analysis) that mostly shows up in niches where the mainstream options don't fit.



Two failure modes are worth holding onto. The first is that a biometric measurement is fuzzy in a way a password never is. It's a probabilistic match, so the system is always balancing letting the wrong person in against locking the right person out. (That balance has real metrics behind it, which is a topic on its own.) The second is the one people forget in the excitement: you can't change your fingerprint. A leaked password is a reset; a leaked biometric template is permanent. That's the whole reason a biometric is best treated as one strong factor among several, not the single thing standing between an attacker and the account.

Onboarding an Identity: Registration and Identity Proofing

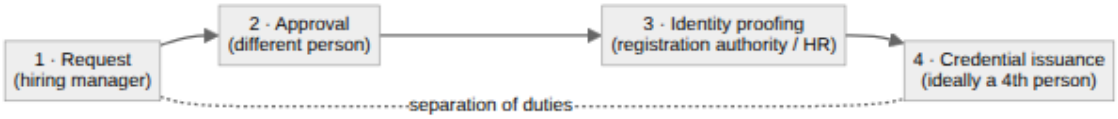
Tue, Aug 4, 4:00 PM EDT · https://scottslab.io/posts/registration-and-identity-proofing

TL;DR

Registration should pass through four different people: request, approval, identity proofing, and credential issuance. That way no single insider can conjure an account out of nothing. Identity proofing (NIST's 800-63A grades ID evidence by strength, so lean on strong, independent government photo IDs) is where you earn the right to trust everything downstream. A flawless MFA setup on an account issued to the wrong person is a strong lock on a door you already opened.

Before anyone gets a username to type, someone has to decide they're real and that they belong. That's registration, and it's the step where a surprising amount of fraud gets stopped, or waved straight through, depending on how many different people actually touch it.

The clean version is four steps, and the whole point is that they're four *different* people. Someone requests the account, usually a hiring manager. Someone else approves it. It has to be someone else, because a request that approves itself isn't a control, it's a formality. A registration authority, often HR, does the identity proofing: actually confirming this person is who they claim to be. And then, ideally a fourth person, issues the credential. Spread across four roles, no single insider can conjure an account out of nothing. Collapse it down to one busy admin doing all four, and you've rebuilt the exact gap fraudsters look for.



Identity proofing is the part that decides how much you should actually trust the claim. NIST's 800-63A framework (revised in 2025) grades identity evidence by *strength* (FAIR, STRONG, SUPERIOR) rather than simply counting IDs: the higher the assurance level you need, the stronger, and often the more numerous, the documents have to be. In practice that still means leaning on strong, independent government photo IDs, and "independent" is doing real work: two documents from the same source prove far less than one each from two. In the US that's the usual set: driver's license, passport, state ID, military ID. Higher-assurance environments layer on more, and fingerprinting is routine in government, with a criminal background check where the role warrants it.

None of this is glamorous, and it's tempting to file it under HR paperwork and move on. But every authentication control downstream is only as trustworthy as this step. A flawless MFA setup on an account that was issued to the wrong person is a very strong lock on a door you already let the intruder walk through. Proofing is where you earn the right to trust everything that comes after it.

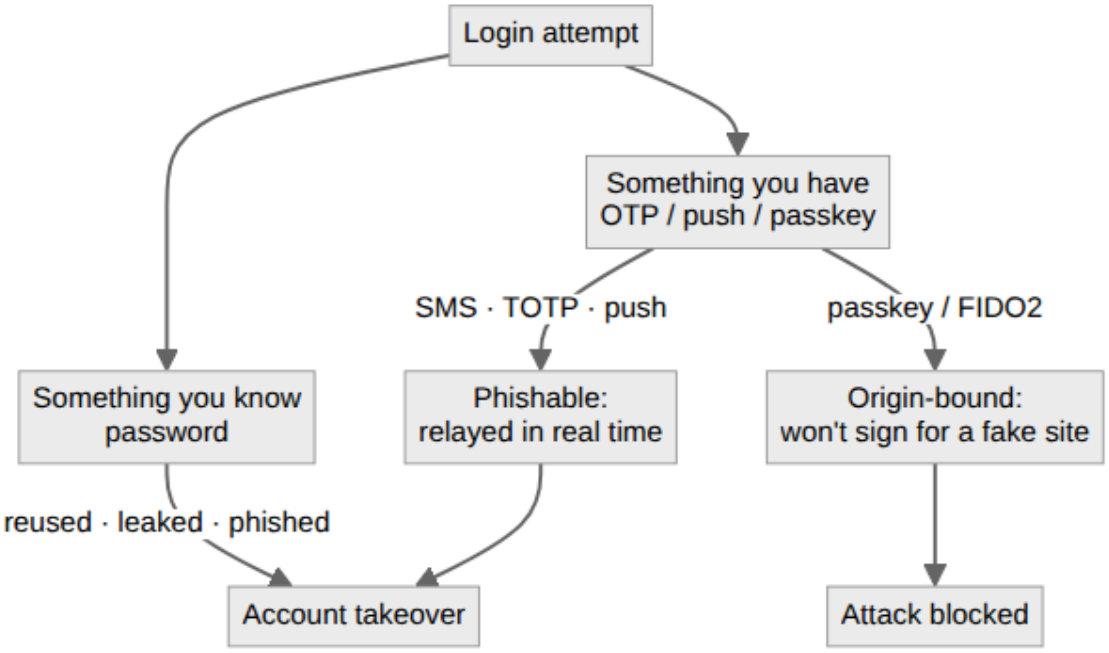
Why Passwords Keep Losing: MFA, Passkeys, and Account Takeover

Wed, Aug 5, 9:00 AM EDT · https://scottslab.io/posts/why-passwords-keep-losing-mfa-passkeys

> **TL;DR**
>
> Most account takeovers don't crack anything; they reuse a leaked password or phish it on a convincing fake page. MFA helps, but not all factors are equal. SMS and TOTP are phishable, push invites fatigue attacks, and only passkeys/FIDO2 fix phishing structurally by binding the credential to the site's origin. The next place worth spending attention is making account recovery as phishing-resistant as the login itself.

Most of the account takeovers I've worked never involved cracking anything. Nobody brute-forced a strong password. Someone reused a password that leaked in an unrelated breach, or typed it into a convincing login page, and that was the whole attack. The password worked exactly as designed. It proved knowledge of a secret. The secret just hadn't been secret for months.

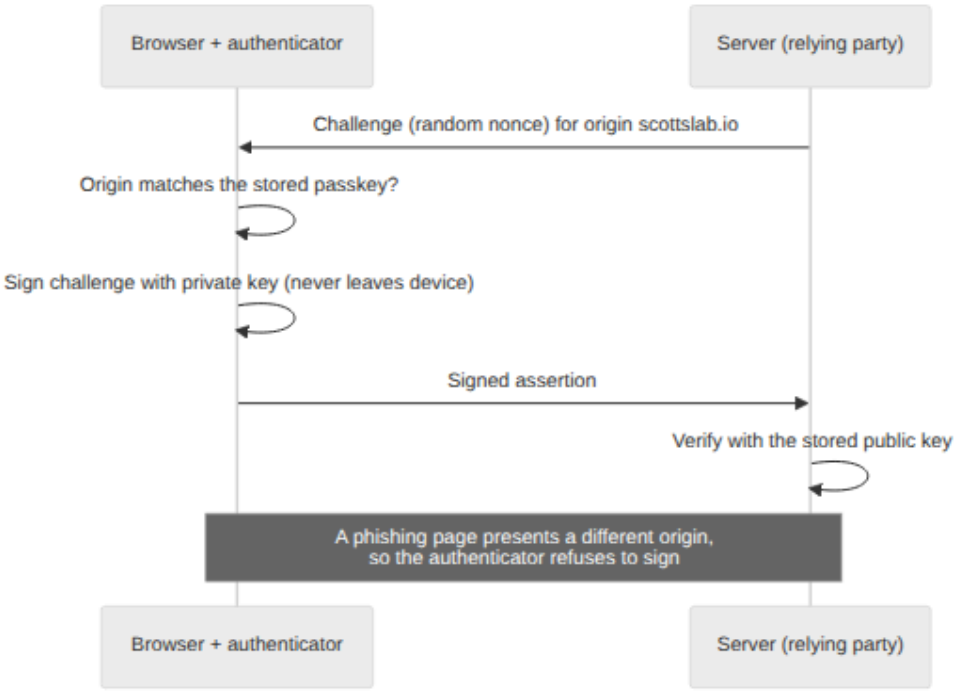
That's the thing worth internalizing before you argue about password length policies: a password is a shared secret, and shared secrets leak. Rotation, complexity rules, the little strength meter: they're all patching a model that breaks the moment the secret is copied. So the useful question isn't "how do I make the password stronger," it's "how few things break when the password is already in the attacker's hands."



MFA is the first real answer, because it stops you standing on a single leaked secret. The catch is that not all second factors are equal, and the industry spent a decade pretending they were. SMS codes and TOTP apps both help against pure credential reuse, but both are phishable. A fake login page just asks for the code too and relays it in real time. I've watched exactly that in incident timelines: password and OTP harvested on the same fake page, replayed within seconds. The second factor was real; it just wasn't resistant to the attack that mattered.

Push approvals have their own failure mode: the plain "approve?" prompt trained people to tap yes, and attackers learned to spam prompts at 3am until someone did. Number-matching helped, but any factor that leans on a tired human making a judgment call is a control with a snooze button.

Passkeys are the first factor that fixes the phishing problem structurally instead of asking the user to be careful. WebAuthn/FIDO2 binds the credential to the site's origin and never sends a reusable secret. The browser signs a challenge with a private key that never leaves the device, and it simply won't sign for the wrong origin. That last part is the whole game: a pixel-perfect phishing page can't get a valid assertion, because the domain doesn't match. It isn't "a better password." It's a different model: proof of possession instead of proof of knowledge.



None of this makes passwords vanish tomorrow. Recovery flows, legacy systems, and the account-bootstrap problem keep them around, and a passkey is only as strong as the recovery path behind it. An attacker who can reset his way back to an SMS code has routed around the good control entirely. That's where I'd spend attention now: not on password complexity, but on making the fallbacks as phishing-resistant as the happy path. The front door is finally getting good locks. The attacks are already moving to the windows.

OAuth 2.0 and OIDC, Explained by Wiring Up My Own Admin Login

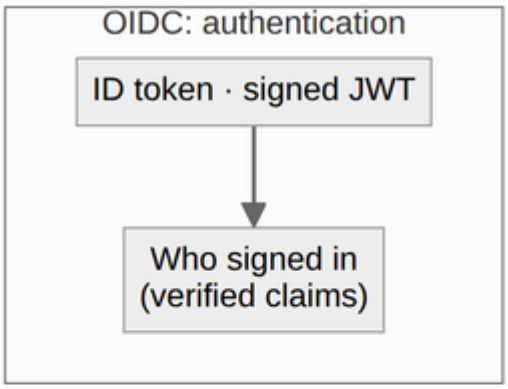
Wed, Aug 5, 4:00 PM EDT · https://scottslab.io/posts/oauth-oidc-explained-admin-login

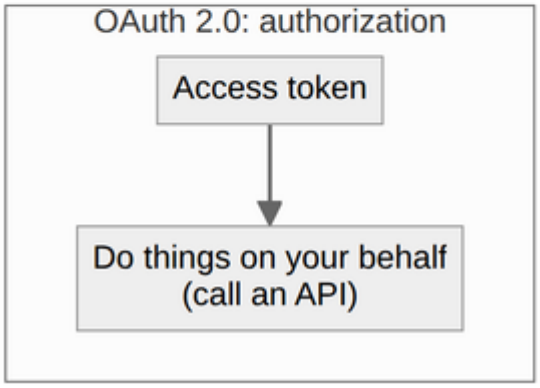
> **TL;DR**
>
> OAuth 2.0 is authorization (a token to do things on your behalf); OIDC is the identity layer on top that actually tells you who signed in. Using this site's Google login as the example: the authorization-code flow keeps the password between you and Google and does the sensitive token exchange server-to-server, and pinning the verified email claim to one address is what makes "only I can log in" true. Because these are bearer tokens, the redirect allow-list and client secret are controls, not paperwork.

I put Google sign-in on the admin panel of this site, and it's the smallest possible version of a system people find intimidating. The whole feature is: only I can log in, and I never want to store a password to make that true. OAuth 2.0 and OIDC are what turn that into a two-paragraph problem instead of a security project.

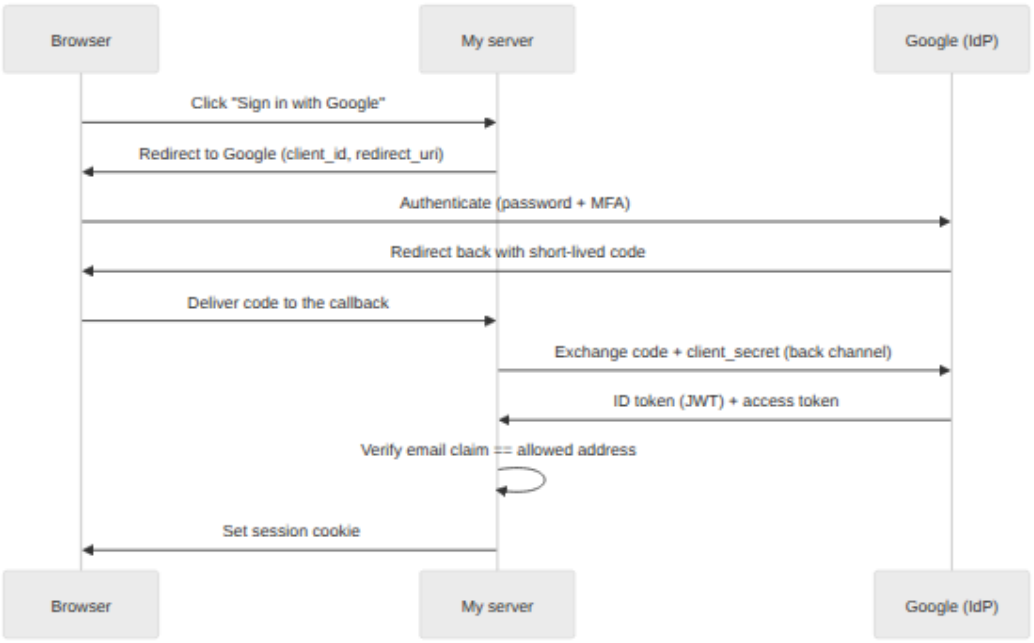
The confusion usually starts with conflating the two. OAuth 2.0 is about authorization: getting a token that lets an app do something on your behalf. It was never designed to tell an app *who you are*, and the early years of people bolting identity onto it by hand produced a lot of subtle bugs. OIDC is the thin, standardized identity layer on top: same flow, but you also get an ID token, a signed JWT that actually

asserts "this is who signed in." For a login button you want OIDC, because you're authenticating, not just authorizing.





What actually happens when I click "Sign in with Google" is the authorization code flow. My server sends the browser to Google with my client ID and a callback URL. Google authenticates me. That part is entirely between me and Google, my server never sees the password. Then it redirects back to the callback with a short-lived code. My server then exchanges that code, plus its client secret, directly with Google for tokens. The reason for the two-step dance instead of just handing a token to the browser is that the sensitive exchange happens server-to-server, where the client secret lives and the code can't be lifted out of a URL or someone's browser history.



The part I care about as the person running the thing: I pinned it to exactly one identity. Google hands back a verified email claim, and my callback checks it against a single allowed address before it establishes a session. Anyone else authenticates to Google perfectly well and still gets bounced at my door. And because these are bearer tokens, where whoever holds the token can use it, the redirect-URI allow-list and the client secret aren't paperwork; they're the controls that stop the code from being redeemed by someone else. One modern addition worth calling out: the current OAuth security best practice (RFC 9700, 2025) makes PKCE the baseline. The client commits to a random secret hashed into the authorization request and only reveals the original at the token exchange, so an intercepted code is worthless without the matching verifier. It's mandatory for public clients and recommended even for confidential ones like this that already hold a secret.

I also kept a local password login as a break-glass path for when Google is unreachable, which nicely illustrates the trade-off. OIDC moves the hard parts (credential storage, MFA, breach monitoring) onto an identity provider who does it full-time, at the cost of a dependency on them being up. For a one-person admin panel that's an easy trade. For a product with millions of users it's the same flow and a much longer list of things to get right, but the shape never changes: redirect, authenticate somewhere trusted, exchange a code in the back channel, verify the claims, start a session.

Sessions vs. Tokens: Cookies, JWTs, and Where Each One Bites

Thu, Aug 6, 9:00 AM EDT · https://scottslab.io/posts/sessions-vs-tokens-cookies-jwt

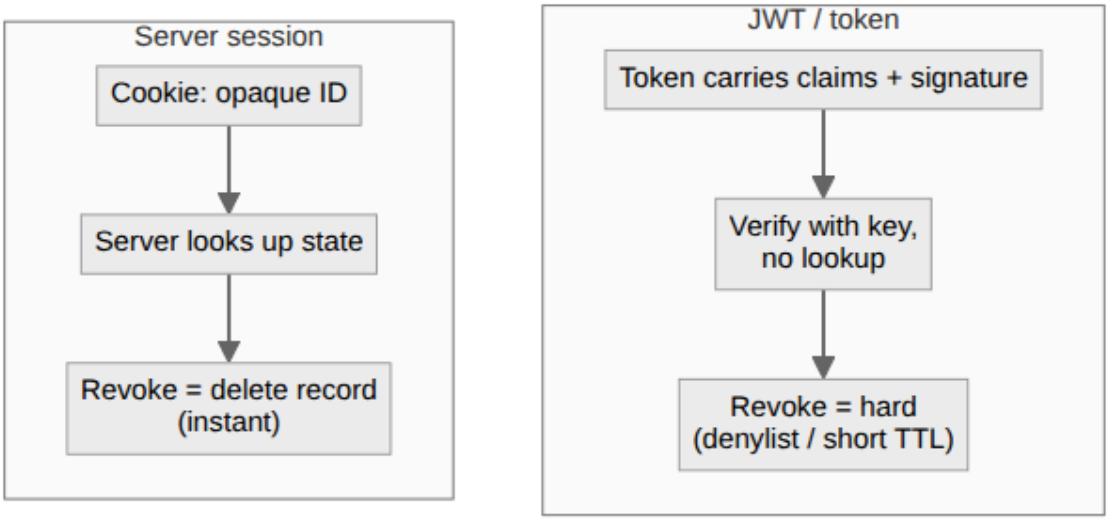
> **TL;DR**
>
> Server sessions are remembered state you can revoke instantly; JWTs are self-contained and stateless but hard to take back before they expire. Where you store the credential is its own trap. HttpOnly cookies dodge XSS but invite CSRF, localStorage does the reverse. For a normal web app, boring server sessions in HttpOnly cookies win; reach for tokens when you actually have the service-to-service problem they solve.

Once someone has authenticated, you have a smaller and more permanent problem: proving it's still them on the next request without asking them to log in every time. There are two honest answers: a server-side session or a self-contained token. Most of the auth bugs I see come from picking one without understanding what it costs.

Sessions: state the server remembers

A session is server-remembered state. You log in, the server stores a record and hands the browser an opaque ID in a cookie; every request presents the cookie, the server looks it up. The nice property is control: logging someone out, killing a compromised session, or forcing re-auth is a single delete on the server, effective immediately. The cost is that the server has to keep and check that state. That's a non-

issue for one app, and a real design question once you have a fleet of services that all need to agree on who's logged in.



Tokens: proof that travels with the request

A token, usually a JWT, flips it around. Instead of a lookup key, the token itself carries the claims (who you are, when it expires) and a signature the server can verify without storing anything. That statelessness is genuinely useful: any service holding the public key can validate the token, no shared session store required, which is why it took over APIs and microservices. But the same property is the trap. Because nothing is looked up, you can't easily take a valid token back. It's good until it expires, and "just revoke it" means rebuilding the very server-side state you adopted JWTs to avoid: a denylist, or short lifetimes plus refresh tokens, which is most of a session with extra steps.

Where you store it, and which bug you accept

Then there's where you put it, which is its own footgun. A cookie can be `HttpOnly`, so JavaScript can't read it, which takes token theft via XSS off the table. But cookies ride along automatically on every request, which is exactly what CSRF abuses, so now you need `SameSite` and anti-CSRF tokens. Put a JWT in `localStorage` to dodge CSRF and you've made it readable by any script that runs on your page, so one XSS and the token walks out the door. There's no placement that dodges both; you're choosing which class of bug you'd rather defend against.

What I actually use

My rule of thumb: for a normal web app with a browser and a login, server sessions in `HttpOnly` cookies are the boring correct answer, and it's what I use here. Instant revocation matters more than saving a lookup. Reach for tokens when you actually have the problem they solve: service-to-service calls, or enough backends that they can't share a session store. Both are fine tools. The mistakes come from choosing the stateless one because it sounds modern, then reinventing state the first time you need to log somebody out.

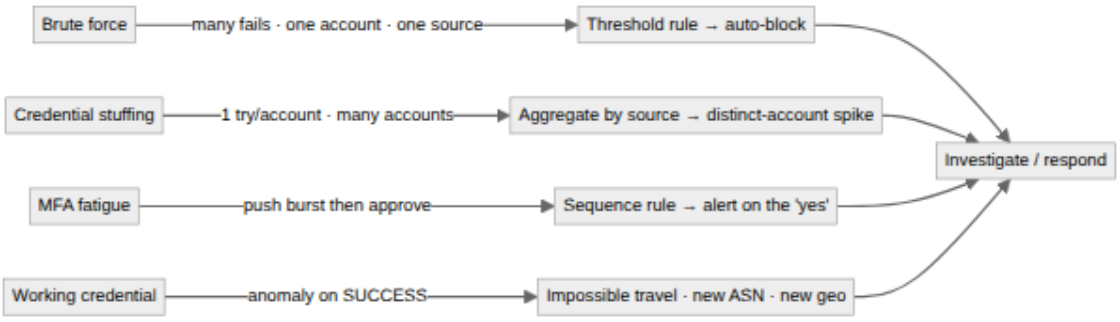
Catching Auth Attacks: Brute Force, Credential Stuffing, MFA Fatigue

Thu, Aug 6, 4:00 PM EDT · https://scottslab.io/posts/detecting-auth-attacks-brute-force-credential-stuffing

> **TL;DR**
>
> Brute force is loud, easy to catch, and mostly not what gets you; credential stuffing and MFA fatigue hide in low-and-slow attempts and, crucially, inside successful logins. The highest-signal detections watch successful auth: impossible travel, a new ASN, a burst of push prompts followed by a yes, not just failure counts. The attacker's whole goal is to look like a normal login, so that's where your detections have to be looking.

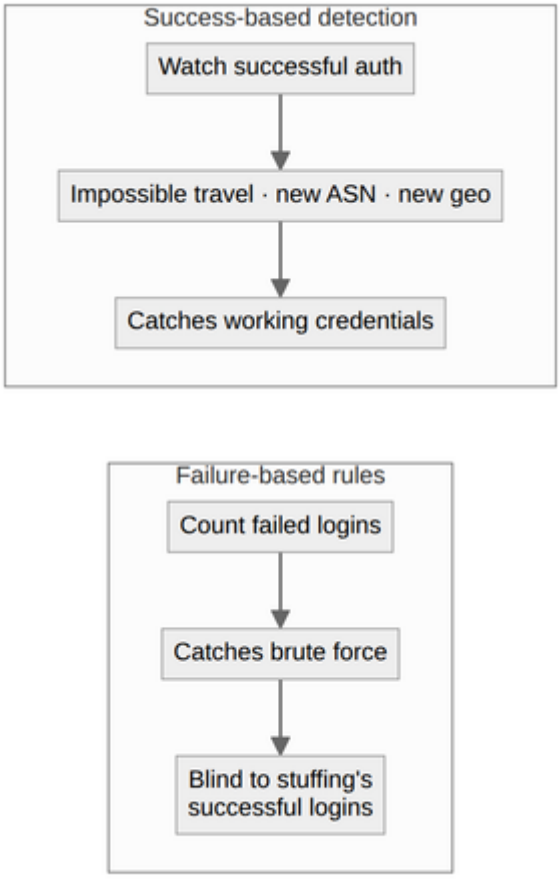
Good authentication decides who gets in. Detection decides whether you notice the people trying who shouldn't. The two failure modes look nothing alike in the logs, which is the part that trips up most home-grown alerting. If your only auth alert is "too many failed logins," you're catching the loudest, dumbest attack and missing the ones that actually work.

Brute force is the easy one, and the one everyone builds first. Many failures against one account, from one source, in a short window: it has a clean frequency signature, and a threshold rule catches it. I run exactly that in my SIEM and it auto-blocks the source. The problem is that serious attackers stopped doing this years ago precisely because it's trivial to detect. It's still worth alerting on, but if it's your whole program you're guarding the one door nobody serious uses.



Credential stuffing is the same tool turned sideways, and it's the one that quietly works. Attackers take username/password pairs from someone else's breach and try each pair *once* against you. From a single account's point of view there's one failed login. Nothing. The signal isn't per-account failures, it's the shape across accounts: one source, or a botnet spread across many, touching hundreds of different usernames with a low per-account failure rate and the occasional success. You detect it by aggregating the other way: distinct accounts per source, failures spread across the tenant, not by watching any single login. And the successes are what matter, because a stuffed password that works produces a *valid* login that no failure-based rule will ever flag.

That's why the highest-signal detections watch successful auth, not failed auth. Impossible travel, the same account authenticating from two places too far apart to be real, catches a working credential no matter how it was obtained. A first-seen login from a new country, a new ASN, a headless user-agent: none of these are failures, they're anomalies on success, and they're where takeovers actually surface. MFA fatigue lives here too. The tell is a burst of push challenges to one user followed by an approval, so the thing to alert on isn't a failed factor. It's many prompts then a yes. A rhythm, not an error.



The through-line is the same as the rest of my detection work: the loud attacks are easy and mostly harmless, and the quiet ones hide inside successful logins. So I spend the alerting budget accordingly: a cheap threshold rule for brute force, and the real attention on cross-account patterns and anomalies on the logins that *worked*. The attacker's entire goal is to become a normal successful login. Your detections have to be looking there when he arrives.

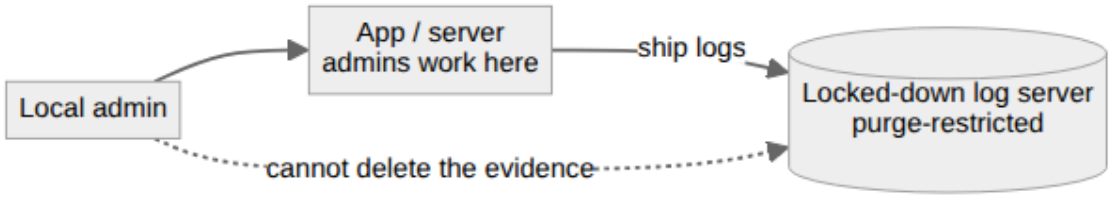
Accountability: Making Every Action Traceable

Fri, Aug 7, 9:00 AM EDT · https://scottslab.io/posts/accountability-and-session-management

> **TL;DR**
>
> Accountability means every action ties back to one real person, and it collapses the moment you allow shared accounts: everyone can blame everyone. It rests on unique identification plus strong authentication, logs shipped to a separate locked-down server the local admins can't purge, and session controls (idle timeouts, re-auth for sensitive actions, lock screens) so an abandoned session can't be inherited.

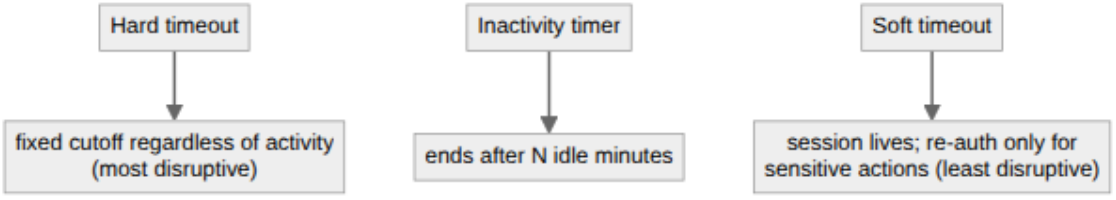
Accountability is the quiet fourth 'A', the one that only matters after something has already gone wrong, and it has a single requirement: every action has to trace back to exactly one human. The instant that chain breaks, your logs stop being evidence and become an argument. And the classic way it breaks isn't a hacker. It's a shared "ops" login three people use, so when something destructive happens at 2am, each of them can point at the other two. Plausible deniability is the enemy here, and shared accounts manufacture it for free.

So accountability stands on the two steps before it: unique identification (real per-person usernames, never a generic account) and authentication strong enough that a logged action really was that person. Without both, "the logs say Scott did it" means nothing; maybe it was whoever knew Scott's reused password.



Then there's a subtler failure: who can edit the evidence. If the logs live on the same box as the activity, an admin who does something they shouldn't can also delete the record of it: the fox auditing the henhouse. That's why logs get shipped off to a separate, locked-down log server whose database restricts purging, so even a local admin can't quietly rewrite history. Most orgs centralize logging for exactly this reason; it isn't about storage, it's about tamper-resistance.

Sessions are the other place accountability leaks. An authenticated session left open is an unlocked door with your name on it. Whatever happens next is attributed to you. The blunt tool is a hard timeout: a fixed cutoff regardless of activity, like a two-hour VPN disconnect. Effective, and genuinely annoying. Gentler is an inactivity timer that ends the session after, say, ten idle minutes. Gentlest, and my favorite, is the soft timeout: the session stays alive but sensitive actions demand a fresh re-authentication, so routine work is never interrupted while the dangerous stuff stays gated.



And the smallest habit that carries the most weight: locking the screen. A screensaver lock doesn't log you out. It just requires re-authentication to resume, but it closes the "walked away from my desk" gap that no server-side timeout can see. It's cheap, and it's the whole difference between a session being yours and a session being whoever sits down next.

Account Types, and Why You Don't Do Daily Work as Admin

Fri, Aug 7, 4:00 PM EDT · https://scottslab.io/posts/account-types-and-privilege-elevation

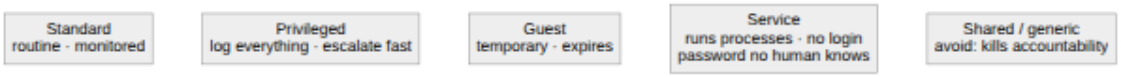
> **TL;DR**
>
> Account types matter because the blast radius differs: standard accounts for daily work, privileged accounts that log every action and only get elevated to when needed, guest accounts that must expire, and service accounts that run processes with no interactive login and a password no human knows. Shared/generic accounts are the one type to avoid entirely, because they destroy accountability.

The account types are worth keeping straight because the damage each can do is wildly different, and the single most important habit in all of security operations falls right out of that: you do not do your daily work in a privileged account.

A standard user account is the default: routine access, monitored like everything else, subject to the usual lifecycle. A privileged account is the one that can actually hurt you, so it gets treated differently: every action logged, and anything suspicious escalated immediately rather than triaged next week. The discipline that keeps privileged accounts safe is separating your everyday and elevated identities. You live in a standard account and elevate only for the task that needs it, whether that's signing into a separate admin account, assuming a role, or a scoped `sudo`. An admin who reads email and browses the

web from a domain-admin account is one phishing link away from handing that account to someone else.

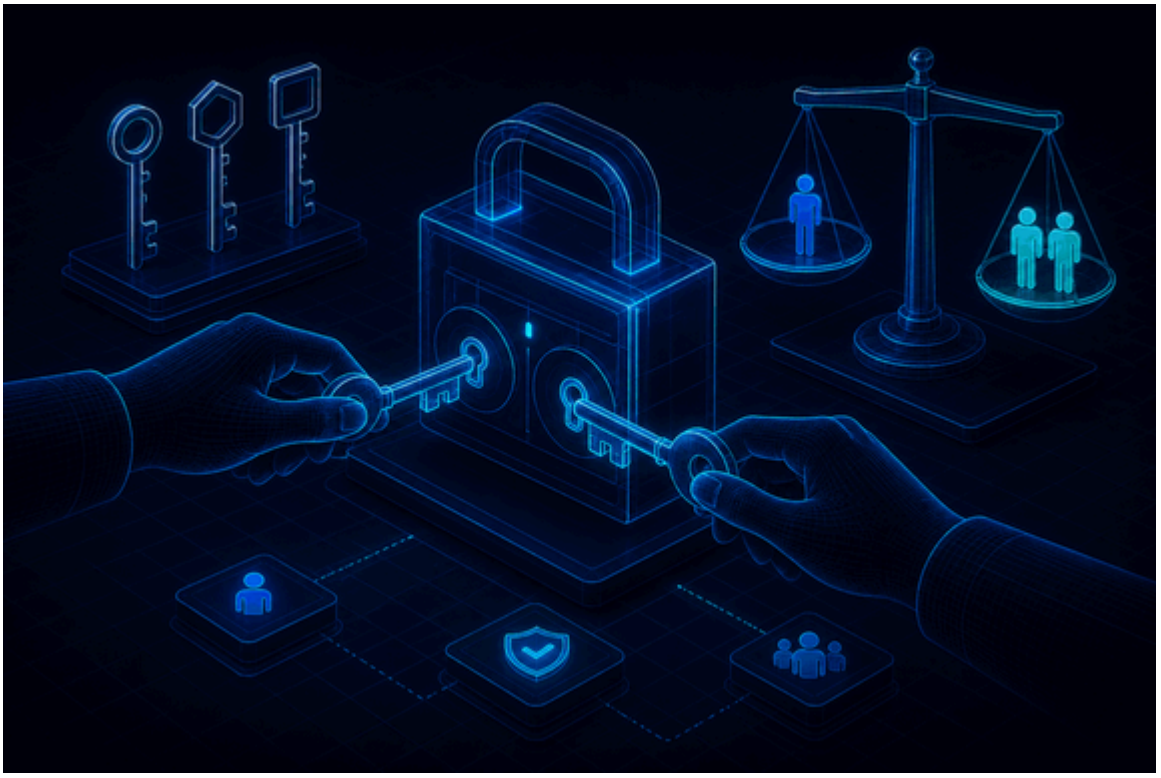
Guest accounts are temporary by definition and should be tied to a specific person and set to expire on their own. A guest account that outlives its reason is just an unowned door standing open. Service accounts are the quiet ones: they run processes and daemons, never log in interactively, and ideally carry a password no human actually knows, system-managed and rotated automatically. A service credential living in someone's head or a config file is a favorite target.



Then the anti-pattern that ties this series together: shared or generic accounts. Everything above exists to make actions attributable; a shared account throws that away and hands plausible deniability to everyone who holds it. If you take one rule from account management, it's that the right number of shared accounts is zero. Role-based security groups give you the exact convenience people reach for shared accounts to get: everyone on the team can get in. You just don't pay the cost of nobody knowing who did what.

Least Privilege, Segregation of Duties, and the Slow Rot of Privilege Creep

Sat, Aug 8, 10:00 AM EDT · https://scottslab.io/posts/least-privilege-segregation-privilege-creep



TL;DR

Least privilege gives people only the access their role needs; segregation of duties splits sensitive actions so no one person can complete them alone. Job rotation and mandatory vacation exist to surface fraud someone's been quietly covering. And privilege creep (the permissions people accumulate as they change roles) is the slow leak that undoes all of it, fixed only by regular access reviews with the managers who actually know who should have what.

Least privilege is the whole game in four words: the minimum access a role needs, and nothing extra "just in case." Spare permissions aren't convenience, they're standing risk. Every right an account holds is a right an attacker inherits the moment they land as that account.

Segregation of duties is least privilege applied to a single dangerous action: split it so it takes two people. Whoever requests a payment can't also approve it; whoever writes a change can't be its only approver. It isn't about distrust. Requiring collusion between two people is a far higher bar than requiring one person to be careless or compromised.

Two related controls are really fraud-detectors wearing an HR costume. Job rotation moves people through roles periodically, which spreads knowledge but mostly means a scheme someone's been running gets inherited by a person with no reason to keep hiding it. Mandatory vacation is the sharper

version: force someone out with no system access for a stretch, and anything they were manually holding together (a cooked reconciliation, a suppressed alert) surfaces while they can't cover for it. Fraud that needs daily babysitting doesn't survive two weeks of enforced absence.

The thing that quietly defeats all of the above is privilege creep. People change roles, the new permissions get added, and the old ones never get taken away. A five-year employee ends up holding a sediment of access from every job they've ever had, none of it reviewed, all of it live. You don't fix creep by hoping; you fix it with regular access reviews and recertification, sitting down with managers to validate each person's permissions against what they actually do now. Internal transfers are where it bites hardest, because revoking the old role's access is the step everyone forgets: adding the new access unblocks the person, removing the old access unblocks nobody, until it's the exact thing an attacker uses.

Joiners, Movers, Leavers — and Watching the Accounts In Between

Sat, Aug 8, 2:00 PM EDT · https://scottslab.io/posts/provisioning-deprovisioning-account-monitoring

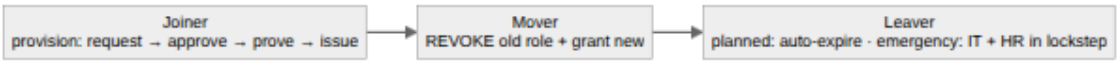
TL;DR

The account lifecycle has three moments that matter: provisioning a joiner, moving someone (where you must revoke the old role, not just add the new), and deprovisioning a leaver. Planned exits should auto-expire on the last day; emergency terminations need IT and HR timed to the minute. Too early tips the person off, too late leaves them access. Continuous monitoring watches for the tells: impossible travel, odd hours, and bulk downloads.

Every account has a beginning, a middle, and an end, and each transition is a place security quietly gets lost.

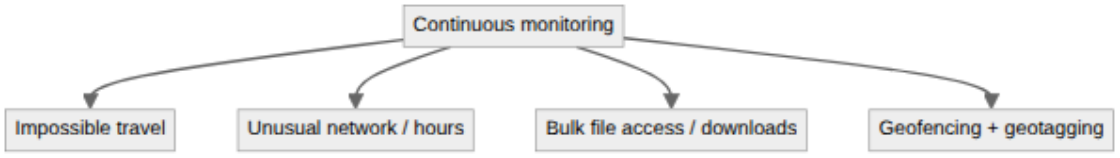
Provisioning a joiner is the easy one, and it's the registration-and-separation-of-duties process from earlier in this series: request, approve, prove, issue. The interesting failures come later.

The mover is the transition everyone botches. When someone changes roles, adding their new permissions is what unblocks them, so that always happens; revoking the permissions from their old role unblocks nobody, so it quietly doesn't. Do that a few times per employee and you've hand-built privilege creep. An internal transfer has to be handled as a revoke-and-grant, never just a grant.



The leaver is where timing becomes everything. A planned departure is clean: set the account to expire automatically on the last day and nobody has to remember. An emergency termination is the hard case, because IT and HR have to move in lockstep. Cut access too early and the person notices they're locked out before anyone has spoken to them. That's advance warning to someone who may want to retaliate. Cut it too late and a just-fired employee still holds the keys. The entire exercise is collapsing that window toward zero, which only happens when HR and IT are working off the same clock.

Once accounts are live, monitoring is how you catch the ones that have been taken over or turned malicious. The useful signals are behavioral, not failed logins. Impossible travel: one account authenticating from two places too far apart to be real. Logins from networks or at hours that don't fit the person. Bulk file access or downloads that look like someone emptying the drawers on the way out. Location sharpens all of it. Geotagging and geofencing turn "unusual place" from a hunch into a rule.



Password Policies, Group Policy, and What NIST Changed

Sat, Aug 8, 6:00 PM EDT · https://scottslab.io/posts/password-policies-and-group-policy

TL;DR

Most Active Directory domains still enforce the password rules NIST retired years ago: forced 90-day rotation and character-class complexity, both baked into the Default Domain Policy and never revisited. NIST 800-63B reversed that: length over composition, no routine expiration, and screening against known-bad passwords. The catch is that classic Group Policy only gives you *one* password policy for the whole domain; the fix almost nobody configures is Fine-Grained Password Policies (PSOs), which let you give service accounts, admins, and standard users their own rules. And before you flip anything: dropping rotation will trip your auditor, so bring the 800-63B citation to the meeting.

If you open the Default Domain Policy on a random AD domain, you can usually guess what you'll find before it loads: minimum length 8, complexity enabled, maximum password age 90 days, history of the last few passwords. Those numbers have been the reflex since before most of the domain's users were hired, and nobody has touched them since the domain was stood up. That's the actual problem here, and it's worth being precise about it. This is not a knowledge problem. NIST published the new guidance years ago and everyone's read the headlines. It's a configuration problem. The rules are sitting in a GPO that got set once and never opened again.

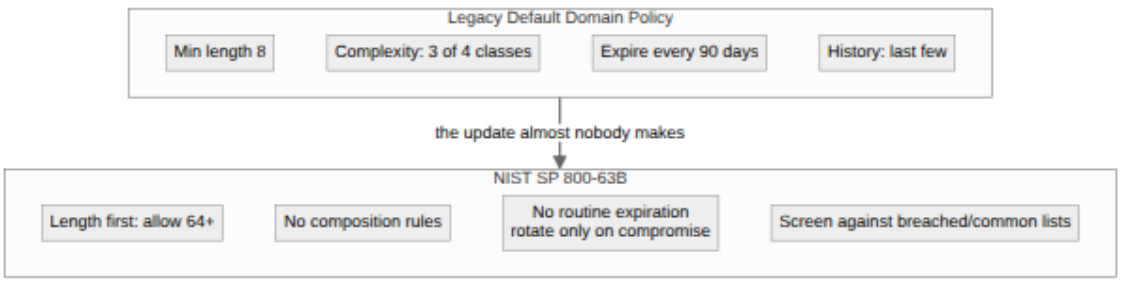
How the policy actually reaches the machine

Password policy in AD is not a document, it's Group Policy, and the delivery has one quirk that shapes everything else. Group Policy in general inherits down the tree: site, then domain, then nested OUs, with the closest object winning. That's why you can hand the Marketing OU a different desktop wallpaper than Finance. Account policy is the exception. The password and lockout settings that apply to domain users are read from the **Default Domain Policy at the domain root**, and linking a stricter password GPO to an OU does nothing for domain accounts. For decades that meant one truth per domain: every user, from the temp who started Monday to the Domain Admin, lived under the same password rule. If you wanted admins on a longer password, your options were a separate domain or a sticky note. That single constraint is why so many environments never tightened anything. There was no clean place to put the exception.

What NIST 800-63B actually says now

The reversal is broader than "make them longer," and the specifics matter because your audience will check them. NIST 800-63B drops the two rituals we defended for years: **no composition rules** (no mandated upper/lower/digit/symbol mix) and **no periodic expiration**: you rotate a password only when there's evidence it was compromised. In their place, the guidance is length-first: a minimum of 8, but verifiers *should* accept at least 64, because the whole point is to let people use a passphrase. Screen every new password against a list of known-compromised and common values, and stop fighting the user in petty ways: allow paste so password managers work, allow the full printable and Unicode range.

The reasoning is the part the old policy got exactly backwards. Forced complexity plus a 90-day clock doesn't produce strong passwords; it produces `Summer2026!`, then `Summer2026!!`, then a sticky note on the monitor. It optimized for a form validator, not for an attacker, and an attacker was never slowed by it. Length is what costs a cracker real time; a rotation treadmill just guarantees a predictable pattern.



There's a hardening layer worth stating separately so it isn't confused with NIST: privileged accounts should be held to a higher bar than everyone else. A **15-character minimum** is the common CIS-style recommendation, and it sits naturally on top of the length-first philosophy. But you can't apply "15 for

admins, 12 for everyone else" from a single Default Domain Policy. That's the whole reason the next section exists.

Fine-Grained Password Policies: the setting nobody opens

This is the part that surprised me when I first dug in, and in a decade of AD conversations I've found most admins either don't know it exists or have never configured one. **Fine-Grained Password Policies** (implemented as Password Settings Objects, PSOs) have shipped since Windows Server 2008, and they break the one-policy-per-domain rule cleanly. You define a policy object with its own length, complexity, history, age, and lockout values, and you scope it to specific users or groups, all without touching the Default Domain Policy.

They live in a container most people never browse: `CN=Password Settings Container,CN=System` in the domain. The one scoping rule to internalize is that a PSO applies to **users and global security groups only, not OUs**. If you think in OUs, the standard move is a *shadow group*: a security group whose membership mirrors the OU, which you point the PSO at. This is also the moment security groups stop being just a folder-permission tool and become the delivery mechanism for password policy. The same group you use to grant access is the one that carries the rule.

The build is short. In the Active Directory Administrative Center, go to the tree view → System → Password Settings Container → New → Password Settings, fill in the values, and add the target group as a subject. In PowerShell it's two commands:

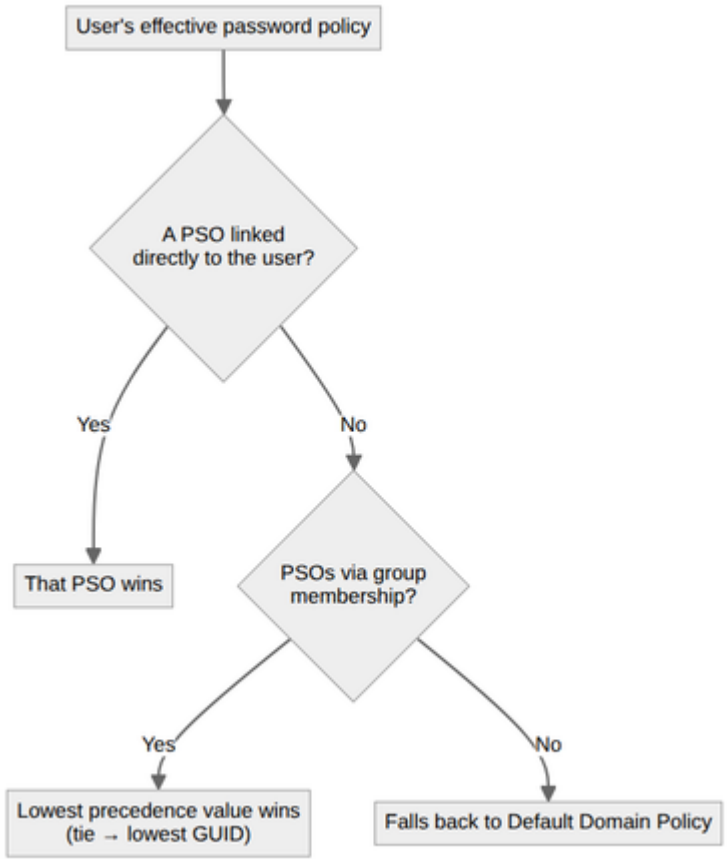
```
New-ADFineGrainedPasswordPolicy -Name "PSO-Privileged" -Precedence 10 `
  -MinPasswordLength 15 -ComplexityEnabled $false `
  -PasswordHistoryCount 24 -MaxPasswordAge "00:00:00"
Add-ADFineGrainedPasswordPolicySubject -Identity "PSO-Privileged" -Subjects
"Domain Admins"
```

`-MaxPasswordAge "00:00:00"` is how you implement NIST's no-rotation guidance. Zero means the password never expires on a schedule. So a realistic setup ends up being three PSOs: privileged accounts at 15 characters and no expiry, service accounts at 25+ (they never type it, so make it long), and standard users on a sane length-first default: each targeting its own group, none of them touching the domain policy.

When two policies both apply

The obvious question, and the one an engineer asked me the minute I posted about this: what happens when a user is in two groups that each have a PSO? PSOs resolve by the **`msDS-PasswordSettingsPrecedence`** value, and **lower wins**. A PSO linked directly to the user beats any

group-linked PSO regardless of number. Among group PSOs, the lowest precedence value applies, and if there's a tie, the lowest object GUID breaks it. That's a deterministic-but-arbitrary tiebreaker you never want to actually rely on, so keep your precedence numbers distinct and spaced out (10, 20, 30) rather than clever.



The control that actually kills Summer2026!

Dropping complexity rules only works if something is still stopping genuinely weak passwords, and that something is a banned-password list. **Entra Password Protection** ships a Microsoft-maintained global list of known-bad passwords and lets you add a custom list of your own (your company name, local sports teams, `Summer`, `Winter`, whatever your users actually pick). The piece people miss is that it extends to on-prem AD: you install a DC agent and a proxy service, and the same fuzzy, substring-aware matching that runs in the cloud now rejects `Summer2026!` and `P@ssw0rd` at the domain controller. That's the real answer to "if I drop complexity, won't people pick garbage?" You replace a rule that guessed at strength with one that checks it against reality.

Bring the citation to the audit

Here's the unglamorous part that will actually cost you time. Turning off forced rotation is technically two clicks, but it directly contradicts the questionnaire your cyber-insurance carrier and your auditor are

reading from, because those documents were written against the old advice and haven't caught up. If you flip it silently, you'll spend renewal season arguing the point. Pair the change with a one-line note in your control documentation citing NIST SP 800-63B as the basis for removing scheduled expiration, and hand it to your assessor before they ask. It converts a finding into a footnote, and it's the difference between "we're out of compliance" and "we made a deliberate, standards-backed decision."

None of this is the endgame. The endgame is passwordless, and phishing-resistant passkeys make most of this argument moot for the accounts that support them. But you have a domain full of passwords today, and "eventually passwordless" is not a policy. Fix the GPO, add the PSOs, turn on the banned list, and write the note for your auditor. It's a genuinely easy win, which is exactly why it's worth being the one who finally opens the setting nobody has touched in years.

Kerberos, NTLM, and the Ticket That Runs Your Network

Sun, Aug 9, 10:00 AM EDT · https://scottslab.io/posts/kerberos-ntlm-tgt-how-it-works

> **TL;DR**
>
> Kerberos is how Active Directory actually authenticates: you prove yourself once to the KDC, get a Ticket Granting Ticket (TGT), and from then on trade that TGT for per-service tickets without your password ever crossing the wire again. NTLM is the older challenge-response scheme it replaced: still lurking for backwards compatibility, and still enabling pass-the-hash, which is why Microsoft's standing advice is to turn it off. Understanding the TGT is also understanding where the attacks (Kerberoasting, golden tickets) live.

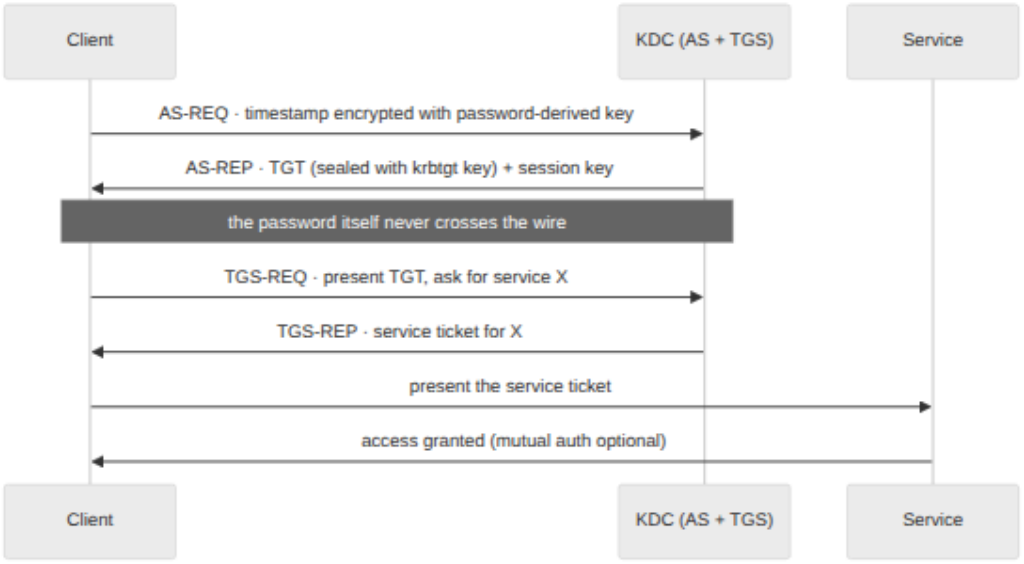
Here's the problem every network authentication protocol is really solving: you need to prove who you are to fifty different servers over the course of a day, and you absolutely do not want to hand your password to fifty different servers to do it. Kerberos answers that with tickets, and the whole system turns on one of them.

How Kerberos works, and where the TGT fits

The center of it is the KDC (the Key Distribution Center), which in Active Directory is just a role the domain controller plays. It has two halves: an Authentication Service (AS) and a Ticket Granting Service

(TGS). You reach it on port 88: UDP by default in Active Directory, with TCP as the fallback when a reply (a ticket stuffed with group memberships, say) is too big for a single UDP datagram.

When you log in, your machine sends the AS a request with a timestamp encrypted using a key derived from your password. The KDC knows your password's key too, so if it can decrypt that timestamp, you've proven you know the password, without the password itself ever being sent. In return you get two things: a **Ticket Granting Ticket**, sealed with a key only the KDC knows (the `krbtgt` account's key), and a session key. That TGT is your "I already proved who I am" token. It's time-stamped and it expires, which is why Kerberos is fussy about clock sync across the domain.



From then on you never re-enter your password. To reach a file share or a database, your machine presents the TGT back to the TGS and says "give me a ticket for this service." The TGS hands back a service ticket, which you present to the service itself. The service can validate it without ever calling back to the KDC. Password sent once, to one place; everything after is tickets. And because the tickets can carry mutual authentication, the service proves itself to you too, so you're not talking to an impostor.

Why the TGT is the crown jewel

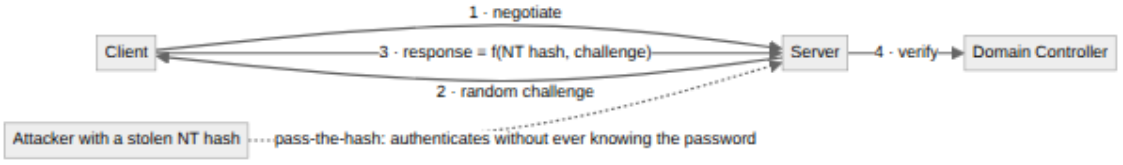
Everything convenient about the TGT is also what makes it dangerous. It's a reusable proof of identity, and the key that seals it belongs to a single account: `krbtgt`. Anyone who steals that key can forge a TGT for *any* user, valid for as long as they like. That's a **golden ticket**, and it's game over for the domain because the KDC will trust its own signature without question.

The more common attack is quieter. **Kerberoasting** abuses the fact that any authenticated user can request a service ticket for any service account, and part of that ticket is encrypted with the service

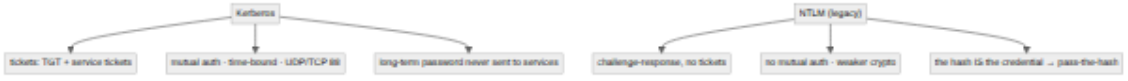
account's password hash. Request the ticket, take it offline, and crack it at your leisure: no failed logins, no alarms, just a weak service-account password turning into a foothold. This is why service accounts get long random passwords and why security teams watch for accounts requesting piles of service tickets.

NTLM, and the hash that is the password

NTLM is what came before, and it works differently. There are no tickets and no KDC ticket dance: it's a straight challenge-response. The server sends a random challenge, the client responds with a value computed from the challenge and the NT hash of the user's password, and the server (or the domain controller behind it) checks the math. Like Kerberos, the password itself isn't sent. Unlike Kerberos, there's no mutual authentication, the crypto is weaker, and there's a fatal design property: the hash *is* the credential.



That last point is **pass-the-hash**. Because the response is computed from the hash and not the plaintext, an attacker who lifts the NT hash out of a machine's memory (the classic Mimikatz move) can authenticate as that user without ever cracking or knowing the actual password. The hash isn't a step toward the credential; it *is* the credential. No amount of password complexity helps once the hash is loose.



What to actually do about it

Microsoft has recommended disabling NTLM for years, and the reason is everything above: it's a weaker protocol whose core mechanic hands attackers a reusable credential. The catch is that it lingers: legacy applications, connections made to a raw IP instead of a hostname, and old appliances all fall back to it, so ripping it out means finding what still depends on it first. Audit NTLM usage, kill it where you can, and where you can't, isolate it.

For Kerberos, the work is protecting the objects the attacks target: guard the `krbtgt` key (and rotate it if you ever suspect compromise, twice, because of how it caches), give service accounts long random passwords to defeat Kerberoasting, and monitor for the ticket-request patterns that give golden tickets and roasting away. The TGT is a genuinely elegant piece of engineering: prove yourself once, carry a ticket, keep your password off the wire. Just remember that the same ticket that saves you fifty password prompts is the thing an attacker most wants in their pocket.

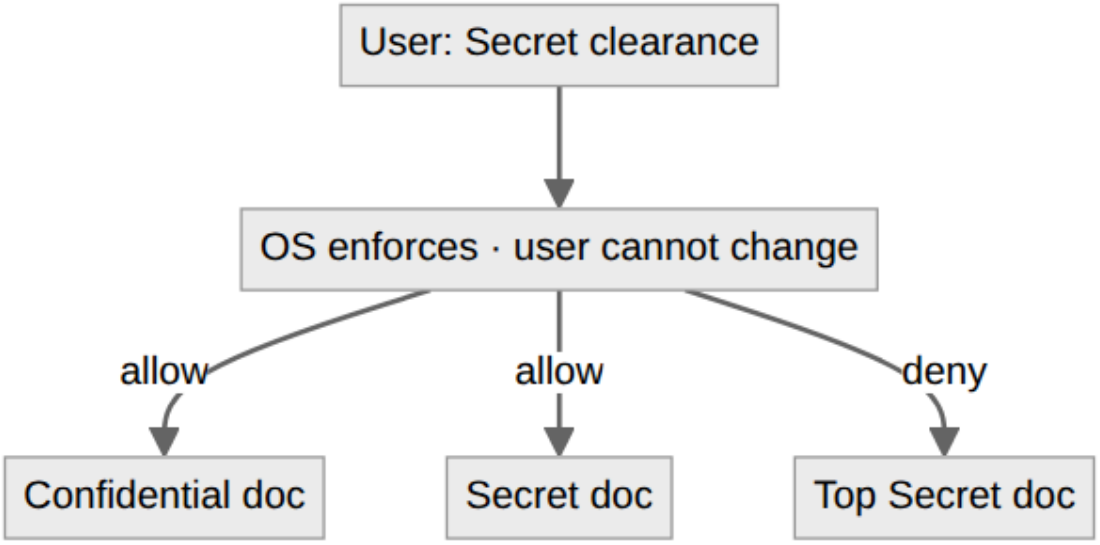
MAC and DAC: Who Actually Decides Who Gets In

Sun, Aug 9, 2:00 PM EDT · https://scottslab.io/posts/mac-and-dac-access-control-models

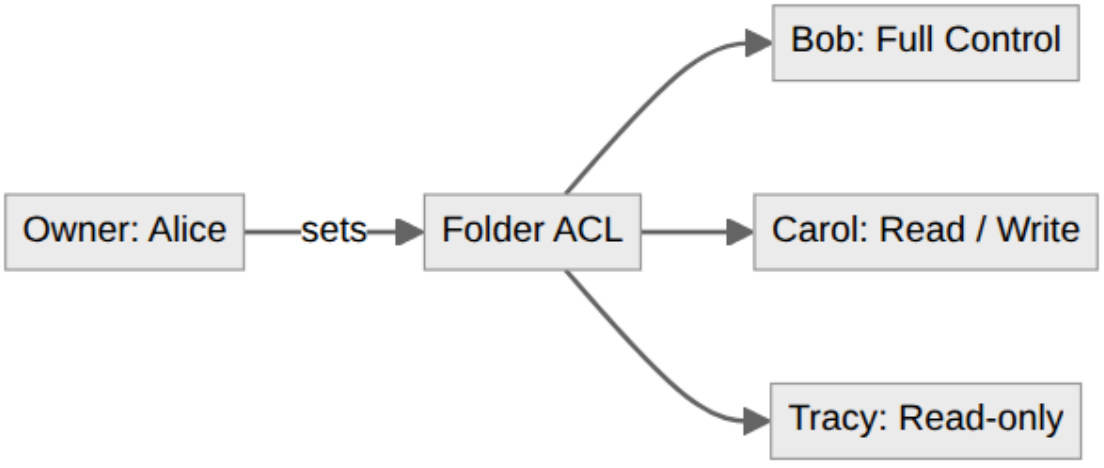
> **TL;DR**
>
> Access control models come down to who holds the pen. Under Mandatory Access Control the operating system enforces permissions from labels and users can't change them: think clearances and SELinux. Under Discretionary Access Control the resource owner grants and delegates access themselves. It's the everyday model, and exactly what NTFS permissions and ACLs on Windows are. MAC trades flexibility for rigor; DAC trades rigor for flexibility.

Every access control model is answering one question: who gets to decide what you can touch. The first split is between the *system* deciding and the *owner* deciding.

Mandatory Access Control puts the operating system in charge, and the user can't override it. It's rule-based in a precise sense: every user and every resource carries a label, and the OS compares them on each access. The canonical example is government clearances: a user cleared to Secret can open Secret and Confidential documents but not Top Secret, and nobody below can grant them that. The system enforces the lattice, full stop. On Linux the primary implementation is SELinux, originally built by the NSA and shipped in RHEL, Fedora, and the RHEL rebuilds like Rocky and AlmaLinux (CentOS lives on as CentOS Stream now that CentOS Linux is end-of-life). MAC is rigid on purpose: it's what you want when a mistake, or a compromised account, must not be able to widen its own access.



Discretionary Access Control flips it: the resource owner sets the permissions and can delegate them. It's the most common model because it's flexible: the person who made the file decides who else gets in, which is how most organizations actually operate. On Windows this is NTFS, and its permission set is worth knowing by name: Full Control, Modify, Read & Execute, Read, and Write, each layering on more capability. In practice it's an ACL hanging off the object: Alice owns a folder and grants Bob Full Control, Carol read/write, and Tracy read-only, with no administrator in the loop.



The trade-off is the whole story. DAC's flexibility is also its weakness: owners over-share, misjudge, and forget to revoke, and there's no system-level backstop when they do. MAC's rigor is also its cost: it's heavier to administer and unforgiving, which is why you see it in high-assurance environments and in

targeted enforcement (SELinux confining a specific service) rather than as the everyday default. Most shops run DAC and reach for MAC where the stakes justify taking the pen out of users' hands.

RBAC and ABAC: Roles vs. Attributes

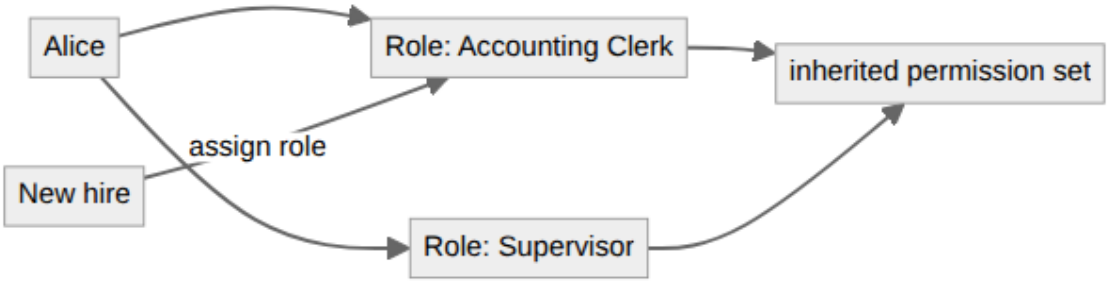
Sun, Aug 9, 6:00 PM EDT · https://scottslab.io/posts/rbac-and-abac-access-control-models

TL;DR

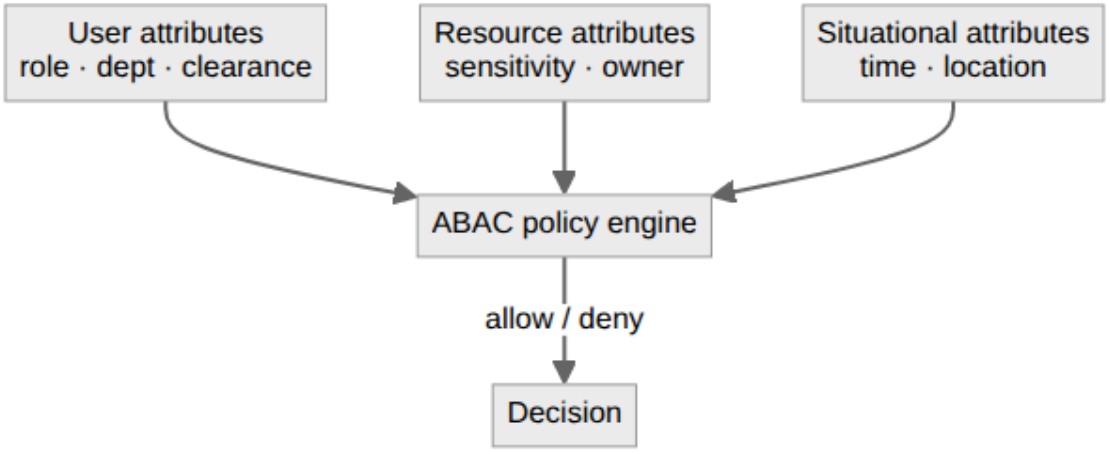
Where MAC and DAC decide *who holds the pen*, RBAC and ABAC decide *how you express the rules*. Role-Based Access Control attaches permissions to roles and users to roles, so onboarding and bulk changes become a membership edit instead of a permissions safari. Attribute-Based Access Control goes finer, writing policy over user, resource, and situational attributes. That enables conditional access like "managers see salary data, but only after merit reviews close." Underneath all of it sits implicit deny: anything not explicitly allowed is blocked.

Once you've decided who sets permissions, you still have to express them. Doing it one user at a time doesn't scale. That's the job RBAC and ABAC split between them.

Role-Based Access Control assigns permissions to roles, then assigns users to roles; you never grant rights to a person directly. The payoff is onboarding and change management. A new accounting clerk gets the "accounting clerk" role and inherits its entire permission set instantly, and when the rules change you edit the role once instead of touching everyone who holds it. Roles stack, too. Give Alice both "accounting clerk" and "supervisor" and she inherits the union, say six permissions, automatically. This is also the honest answer to the shared-account temptation from earlier in the series: roles and groups give you "everyone on the team can get in" without throwing away who-did-what.



Attribute-Based Access Control is the fine-grained evolution. Instead of "which role are you," it evaluates a policy over attributes: something about the user, something about the resource, and something about the situation. That last part is the point. ABAC does conditional access. You can write "managers may read salary data, but only after March 15, once merit increases are finalized," and the same engine treats physical location as just another attribute, so access can depend on where you actually are. It's more powerful and more work to author correctly; you're writing policy, not clicking checkboxes.



Whatever model expresses the rules, one default sits under all of them: implicit deny. Anything not explicitly permitted is blocked. That's the safe posture, because it fails closed. The clearest example is a firewall: traffic runs down the rule list, and if nothing matches, the connection is dropped. Build access that way everywhere: allow what you mean to allow, and let the absence of a rule be a no.

Database Authorization: Roles, Grants, and Locking It to Business Hours

Mon, Aug 10, 9:00 AM EDT · https://scottslab.io/posts/database-authorization-time-of-day

> **TL;DR**
>
> Database access is two questions stacked: authentication (SQL Server logins, Windows/AD accounts, or mixed mode) then authorization (broad role membership like sysadmin or read-only, or granular per-table GRANTs, revoked with REVOKE). Layer on time-of-day restrictions from Active Directory so an account only works during business hours, and scale the strictness to the data's sensitivity: risk-based access.

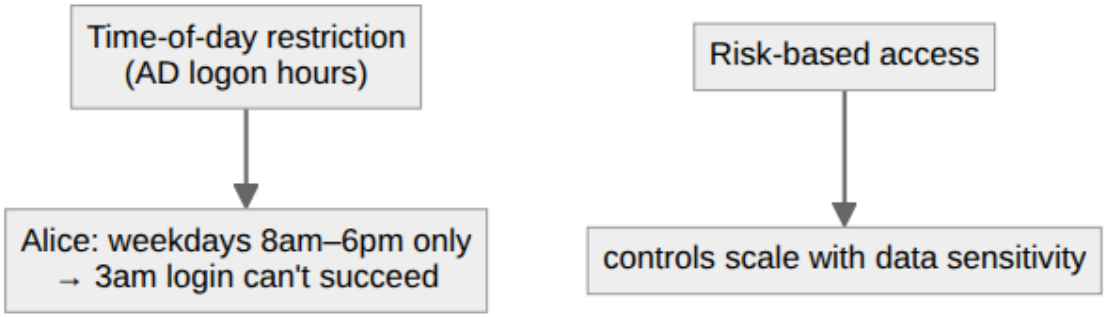
A database is where authorization gets concrete, because you can dial it from "you administer everything" down to "you may insert into this one table." And you should.

It starts with authentication, and SQL Server is the clean example. You can authenticate with SQL Server logins managed by the database itself, with Windows/Active Directory accounts managed by the domain, or with mixed mode that allows both. AD-backed auth is usually what you want, because it means database access rides your existing identity lifecycle: disable the person in Active Directory and their database access dies with the same click.

Then authorization, which comes at two granularities. Role membership is the broad brush: drop a login into `sysadmin`, or a read-only role, or a backup-admin role, and it inherits that role's whole reach. Granular grants are the scalpel: `GRANT INSERT ON orders TO rallen` hands over exactly one

privilege on exactly one table, and `REVOKE` takes it back just as precisely. Least privilege lives right here: reach for the narrowest grant that lets the job get done, not the role that happens to be convenient.

Two more levers are worth knowing. Time-of-day restrictions cap *when* an account can even log in. In Active Directory Users and Computers you can limit logon hours, so Alice's account only works weekdays from 8 to 6, and a 3am login with her credentials simply can't succeed no matter who's typing them. And risk-based access ties it together: the more sensitive the data, the more controls you stack on top of it. Not every table deserves the same ceremony, and not every table can be left wide open. Match the friction to what's actually at stake.



Social Engineering: Hacking the Human, and How to Make It Harder

Mon, Aug 10, 4:00 PM EDT · https://scottslab.io/posts/social-engineering-tactics-and-defenses

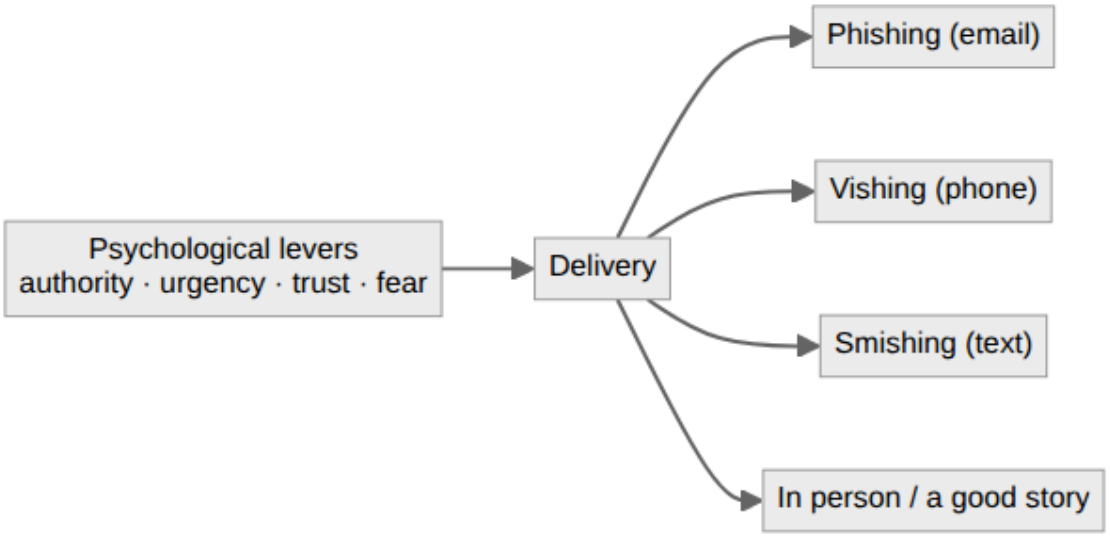
TL;DR

Social engineering skips the firewall and targets the person, because it's cheaper and it works. A human element shows up in most breaches (Verizon's DBIR puts it around 60%): a person convinced or tricked, not just something cracked. The tactics lean on a few psychological levers (authority, urgency, trust, fear) delivered by phishing, vishing, or a good story. The defense isn't a product; it's friction in the right place: verification habits, out-of-band callbacks, and a culture where "let me confirm that" is treated as competence, not obstruction.

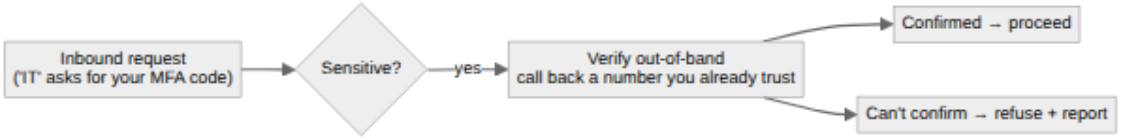
The uncomfortable truth of most incidents is that nothing was hacked in the movie sense. Someone got a convincing email, or a phone call from "IT," and did exactly what they were asked. Social engineering is attacking the human instead of the machine, and it stays popular because it's cheap, it scales, and the target, a helpful person under time pressure, is standing right there.

The tactics all pull on the same few psychological levers. Authority: people comply with someone who sounds like the boss or the bank. Urgency: "your account locks in an hour" shuts off the part of the brain that would otherwise pause and check. Trust and familiarity: a message that looks like it's from a coworker or a vendor you actually use. Fear: the threat of consequences. The delivery changes: phishing

over email, vishing over the phone, a smishing text, or just a confident story in a lobby. The lever underneath is almost always one of those four.



What makes this hard to defend is that the "vulnerability" is a feature: people are supposed to be helpful and responsive. So the defenses aren't about switching that off, they're about adding friction exactly where it matters. Verification is the core: for anything sensitive (a payment, a password reset, a data request), confirm through a second channel you already trust, not the one the request arrived on. If "IT" calls asking for your MFA code, you hang up and call IT back on the number you already have. That out-of-band callback defeats most of it, because the attacker controls the inbound channel, not your directory.



The rest is culture and reps. Training that shows real examples beats a policy nobody reads; simulated phishing that teaches instead of punishing; and, most importantly, a workplace where "let me verify that before I send it" reads as competence, not obstruction. The organizations that get burned are usually the ones where slowing down to check felt like it would get you in trouble. Make checking the easy, praised default and you've quietly removed the pressure the whole attack was counting on.

Impersonation, Pretexting, and Identity Fraud: The Story Is the Weapon

Tue, Aug 11, 9:00 AM EDT · https://scottslab.io/posts/impersonation-pretexting-identity-fraud

TL;DR

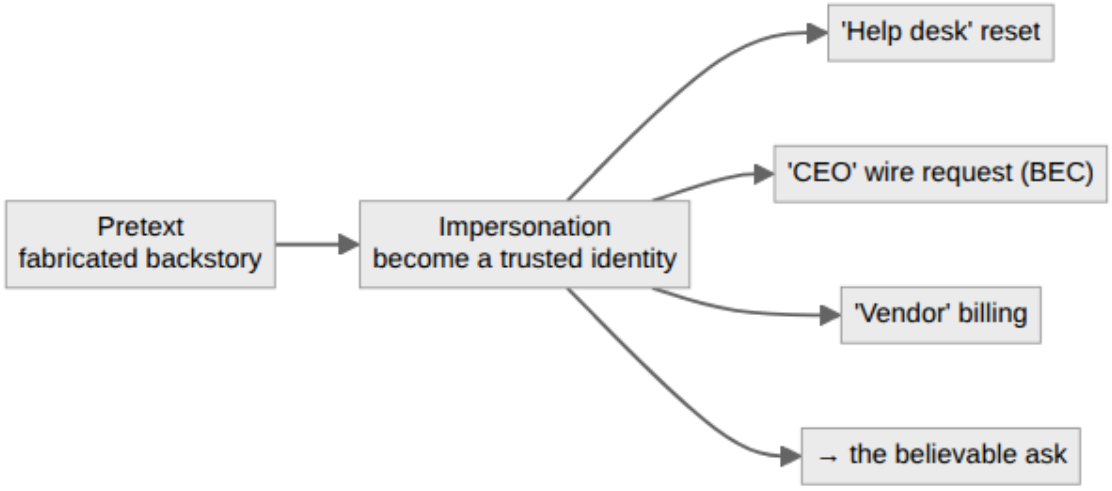
These three are the "who I'm pretending to be" family of social engineering. Pretexting is the fabricated backstory that makes the ask believable; impersonation is stepping into a specific trusted identity (the help desk, a vendor, the CEO); and identity fraud is using someone's real stolen details to become them on paper. The defense is the same across all three: verify identity through something the attacker can't easily supply, and never let a good story stand in for proof.

Where broad phishing is a wide net, this family is targeted and personal. It's about convincing you that the person on the other end is someone they're not.

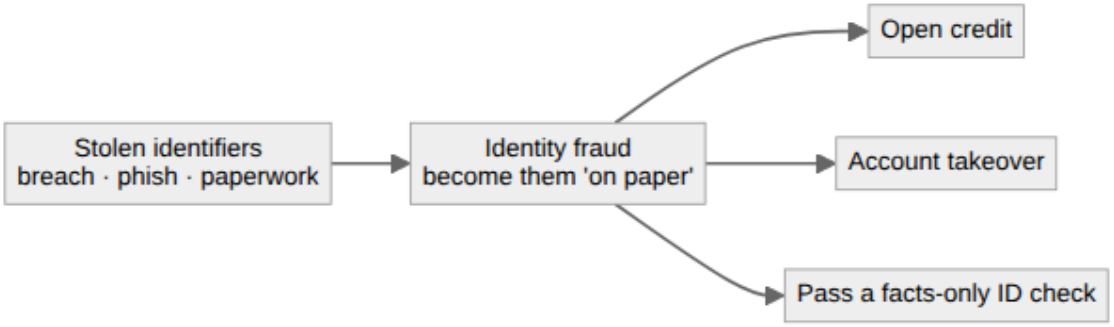
Pretexting is the foundation: the invented scenario that makes a request make sense. "I'm from the vendor's billing team, reconciling last quarter's invoices" isn't an attack on its own. It's the stage set that makes the coming ask ("can you confirm the account details?") feel routine. The better the pretext, the less the target questions the request, because people evaluate plausibility, not authenticity. A story that fits your day is one you tend not to interrogate.

Impersonation is pretexting pointed at a specific trusted identity. Instead of a generic role, the attacker becomes the help desk resetting your password, the executive urgently wiring funds (business email compromise is exactly this), or a supplier you actually use. It works because we extend standing trust to

roles and relationships. We do what the help desk says, because the help desk is the thing that's supposed to help.



Identity fraud is the heaviest version: using someone's real, stolen identifying information, pulled from a breach, a phish, or discarded paperwork, to actually pass as them. Now it isn't just a convincing story, it's your name, your account number, your date of birth, deployed to open credit, take over an account, or clear a verification check that only ever asks for facts an attacker already bought.



The defense collapses to one habit across all three: verify the person, not the plausibility. Static facts (name, birthday, account number, the last four of a card) are not authentication anymore, because they've all leaked. Real verification means something the attacker can't easily supply on demand: a callback to a number of record, a code to an enrolled device, a challenge only the real party can answer. When someone's identity is the thing in question, a good story should raise the bar for proof, not lower it.

Watering Holes and Tailgating: Attacking Where People Already Trust

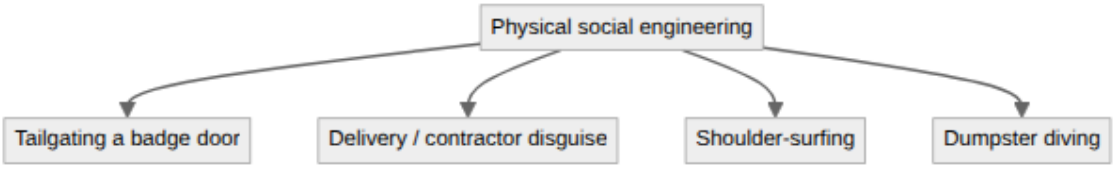
Tue, Aug 11, 4:00 PM EDT · https://scottslab.io/posts/watering-hole-and-physical-social-engineering

> **TL;DR**
>
> Two social-engineering plays that skip your inbox. A watering hole attack compromises a site the target already trusts and visits (an industry forum, a vendor portal) and lets the victims come to it. That sails right past "don't click strange links." Physical social engineering just walks in: tailgating through a badge door, posing as a delivery or contractor, shoulder-surfing, dumpster diving. Both exploit trust in a *place* rather than a message. Both are beaten by the same instinct: verify before you grant access, digital or physical.

Not every social-engineering attack comes at you head-on. Two of the sneakier ones work by planting themselves where your trust already lives.

A watering hole attack borrows its name from a predator waiting at the pond instead of chasing prey across the plain. Rather than phishing a hardened target directly, the attacker compromises a website that target is known to trust and visit (an industry association, a niche forum, a supplier's portal) and waits. The victims arrive on their own, to a site they've been to a hundred times, and get served the payload there. It's effective precisely because it defeats the advice we drill into everyone: there's no suspicious link to not-click; they went to a bookmark. That's why it's aimed at specific industries or groups, where the attacker knows exactly which watering hole everyone drinks from.

Physical social engineering drops the keyboard entirely and exploits the same helpfulness in the lobby. Tailgating is the classic: follow someone through a badge-controlled door while carrying a box, and most people hold it for you, because holding doors is polite. Add a hi-vis vest and a clipboard and you're a contractor; a delivery uniform and you're expected. Once inside, the low-tech attacks pay off: shoulder-surfing a password, walking off with an unlocked laptop, reading the sticky note on the monitor, or dumpster diving for the paperwork that feeds the pretexts and identity fraud from the last post.



The through-line is that the trust being exploited lives in a place, not a message, so the defense has to live at the place too. For watering holes that means the boring technical hygiene that catches drive-bys regardless of where they came from: patching, endpoint protection, least-privilege browsers. You can't tell people to distrust their own bookmarks. For the physical side it's culture again: one badge swipe per person, challenge the unbadged stranger without feeling rude, lock the screen (the habit that keeps coming up in this series), and shred what you throw away. Same instinct in both worlds: verify before you grant access, and never let familiarity stand in for authorization.

What Vulnerability Management Actually Is (and Why You're Doing It)

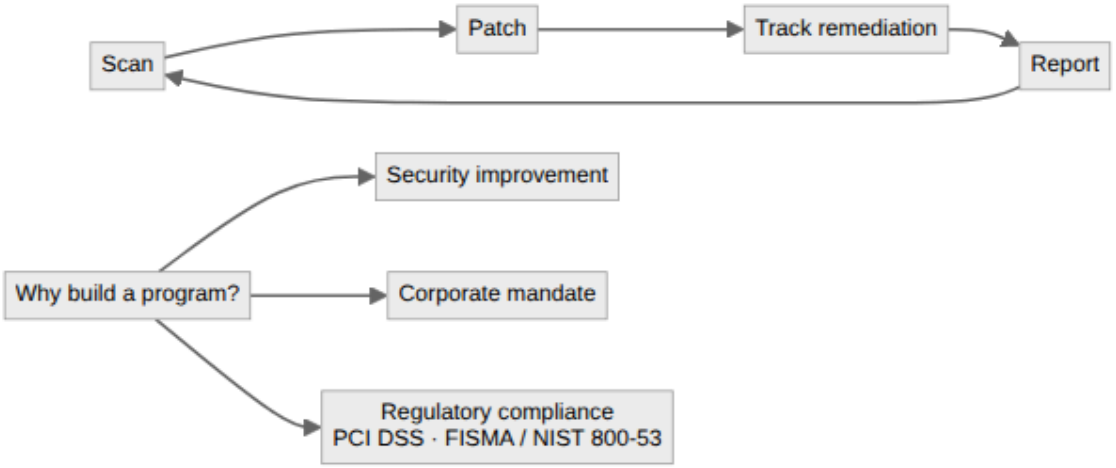
Wed, Aug 12, 9:00 AM EDT · https://scottslab.io/posts/vulnerability-management-fundamentals

TL;DR

Vulnerability management is the whole loop, not just the scan: find issues, apply patches, track remediation, and report on it. Programs get built for one of three reasons: general security improvement, a corporate mandate, or regulatory compliance. The driver shapes everything. When it's compliance, the regulation writes your requirements for you: PCI DSS demands quarterly internal and external scans (external via an approved vendor) with a rescan-until-clean rule; FISMA/NIST 800-53 wants regular scans and remediation of real findings.

People collapse "vulnerability management" down to "running Nessus," and that's the smallest part of it. The actual program is a loop: scan to find the issues, apply patches, track remediation so nothing falls through the cracks, and report on where you stand. Scanning without the tracking and the reporting is just generating anxiety on a schedule.

Before you build one, it's worth being honest about *why*, because the driver shapes the whole thing. There are three. The first is plain security improvement: you want to find and fix your weak spots. The second is a corporate policy or mandate. Leadership or a customer contract requires it, which usually comes with strings attached: specific tools, deadlines, centralized reporting. The third is regulatory compliance, and that's the one that stops being optional.



When compliance is the driver, the nice thing is the regulation writes your requirements for you. PCI DSS, for anyone touching cardholder data, is explicit: quarterly internal and external scans, external scans run by an Approved Scanning Vendor, a requirement to rescan until you get a clean result, and a fresh scan after any significant change. There's no ambiguity to argue about. The standard hands you the cadence and the bar. FISMA and NIST 800-53 do the same for federal systems: regular vulnerability scanning and risk-based remediation of legitimate findings, under control RA-5.

The reason to name your driver out loud is that it sets your floor. A security-improvement program can be pragmatic and risk-based. A compliance program has a minimum you cannot go under no matter how you feel about a particular finding. "We decided it wasn't worth patching" is a fine sentence right up until an auditor holding the PCI DSS text disagrees. Know which game you're playing before you tune the rest of the program.

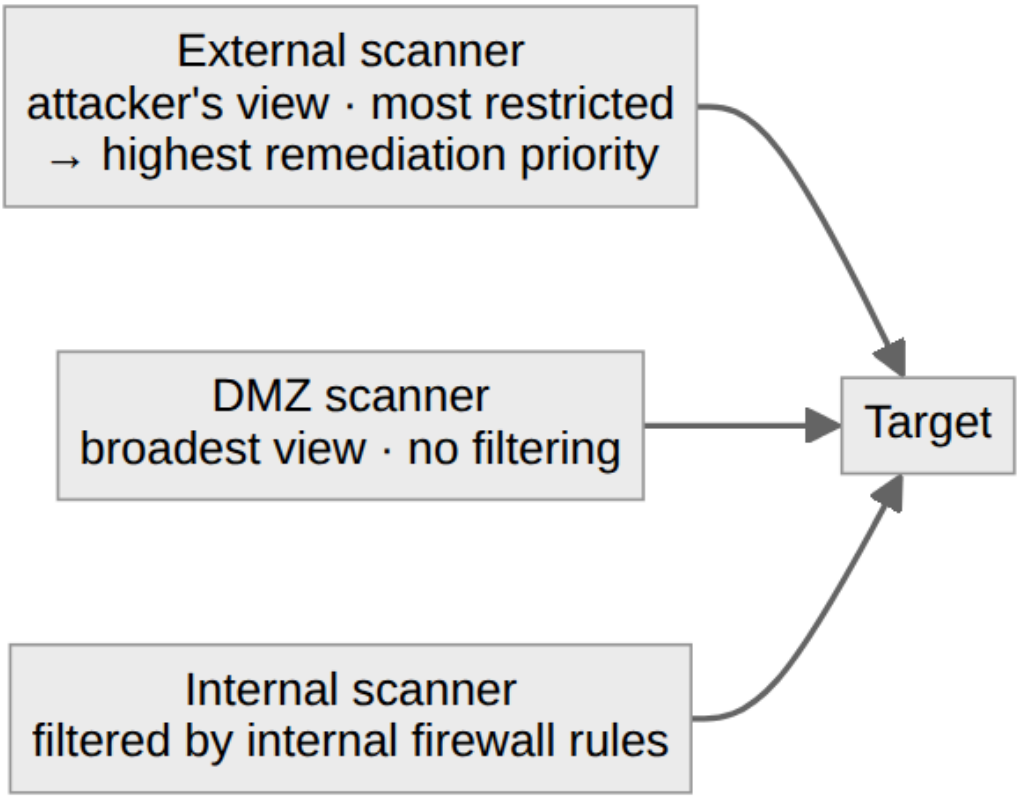
Picking and Configuring What You Scan

Wed, Aug 12, 4:00 PM EDT · https://scottslab.io/posts/vulnerability-scan-targets-and-configuration

> **TL;DR**
>
> You can't scan what you haven't inventoried, so a program starts with an asset list (from a CMDB, config management, or a discovery scan) and prioritizes it by impact, likelihood, and criticality. Where you put the scanner changes what you see: a DMZ scanner sees everything, an external scanner sees the attacker's view (most restricted, highest remediation priority), an internal one sees through your firewall rules. And prefer credentialed scans (a read-only login) over agents or admin creds: depth without the extra baggage.

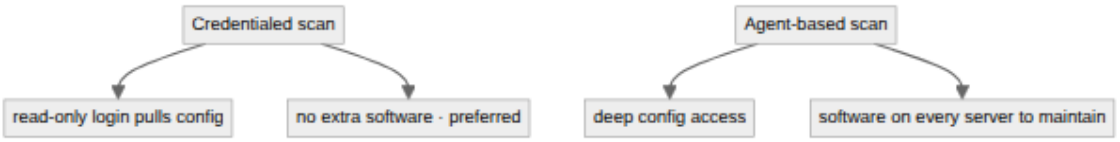
Every scan-config decision traces back to two questions: what am I scanning, and from where.

The "what" has a hard prerequisite people skip: an asset inventory. You cannot manage vulnerabilities on assets you don't know exist, so you pull the list from a CMDB, from config-management tooling, or, if you have neither, you run a lightweight discovery scan first just to build one. Then you prioritize, because scanning everything with equal weight wastes the effort. Three axes do it: impact (how sensitive is the data it stores, processes, or transmits), likelihood (how exposed is it: internet-facing or tucked behind a firewall, and which services are open), and criticality (how badly does it hurt if the system goes down). A public-facing box holding regulated data ranks above an internal wiki, and your scan schedule should say so.



The "from where" is scan perspective, and it's not a detail. The same target scanned from three places gives three different answers. A scanner in the DMZ has the broadest view, unfiltered by internal firewalls. An internal scanner sees what the internal firewall rules allow, which is the realistic picture for lateral-movement worries. And an external, internet-based scanner sees exactly what an outside attacker sees: the most restricted view, but the highest-priority findings, because those are exposed to the entire world. You want more than one perspective, because "invisible from the internet" and "invisible from the inside" are very different safety claims.

Then the mechanics. Combine scan types: network vulnerability scans for the infrastructure, application scans, and specialized web-app scans that actually probe for SQL injection and XSS. And choose credentialed over agent-based when you can: an agent gives deep config access but means running more software on every server, while a credentialed scan just logs in with a read-only account to read configuration. Read-only, not admin. Your scanner shouldn't be a domain-admin-shaped hole in the network waiting to be stolen.



A few Nessus specifics, because they're where scans quietly go wrong. Point it at IPs, hostnames, or an imported target file from your asset tool. Use the schedule tab to set frequency per system group, and scan the sensitive tiers more often. Leave assessment accuracy on normal, then decide whether potential false alarms show or hide. In production, enable safe checks and rate-limit so the scan doesn't knock things over. Disable plugin families you don't run (no point scanning for Amazon Linux bugs across a Windows fleet) to speed things up. And remember that an active IPS on the scan path will distort the results: it blocks the scanner and hands you a rosier picture than reality.

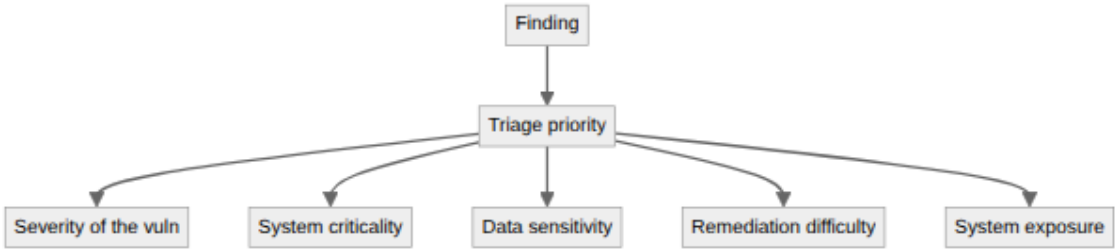
Reading the Report: Triage, False Positives, and the Analyst Who Cried Wolf

Thu, Aug 13, 9:00 AM EDT · https://scottslab.io/posts/vulnerability-scan-triage-and-correlation

> **TL;DR**
>
> A scan report is a to-do list you have to rank, because you can't fix everything at once. Prioritize each finding on five factors: severity, system criticality, data sensitivity, remediation difficulty, and exposure. Validate before you escalate. One or two false positives and people stop trusting your tickets. Know the four outcomes (true/false positive, true/false negative; the false negative is the scary one), and correlate findings against compliance standards, your own config/log data, and historical trends.

The scan finishes and hands you a pile of findings. The skill was never running the scan. It's what you do with the report, and the first job is ranking, because you can't remediate everything at once and pretending otherwise just parks the important fixes behind the trivial ones.

Five factors drive the triage. The severity of the vulnerability itself. The criticality of the affected system. The sensitivity of the data involved. The difficulty of remediation: a critical bug with a one-click patch may go ahead of a medium one that needs a maintenance window and a prayer. And the exposure of the system: an internet-facing host outranks the same flaw on an isolated internal box. Weigh those together and the pile sorts itself into an order that actually reduces risk instead of just closing tickets.



Before any of that reaches a system owner, validate. This is the step junior analysts skip and regret: escalate a false positive and you've burned someone else's time on nothing, do it twice and you're the analyst who cried wolf. After that your real findings get ignored, which is worse than not scanning at all. So confirm the finding is real by reviewing the scanner's own input/output evidence for it, and check whether there's already a compensating control or an accepted risk documented in the CMDB. A surprising share of "vulnerabilities" are things the org already knows about and has consciously chosen to live with.

It helps to hold the four outcomes clearly. A true positive is a real finding, correctly reported. A false positive is a reported finding that isn't actually there: annoying, credibility-eroding. A true negative is correctly finding nothing. And the dangerous one is the false negative: the scanner reports nothing but the vulnerability is really there. That's the risk you don't even know you're carrying, which is exactly why one scanner and one perspective are never enough.

Finally, findings mean more in context, so correlate them. Against compliance standards: PCI DSS, for one, says an external ASV scan can't pass with anything carrying a CVSS base score of 4.0 or higher (that's the numeric score, not CVSS v4.0 the version, and it's scoped to your internet-facing assets), which turns a "medium" on an exposed host into a hard fail. Against your own internal data: config-management systems and log repositories tell you whether a flagged service is even in use. And against history: the same XSS finding showing up scan after scan isn't one bug, it's a signal that a dev team needs training or an input-validation library, and you fix the pattern instead of the instance.

Vulnerability Testing vs. Penetration Testing (and Why the ROE Comes First)

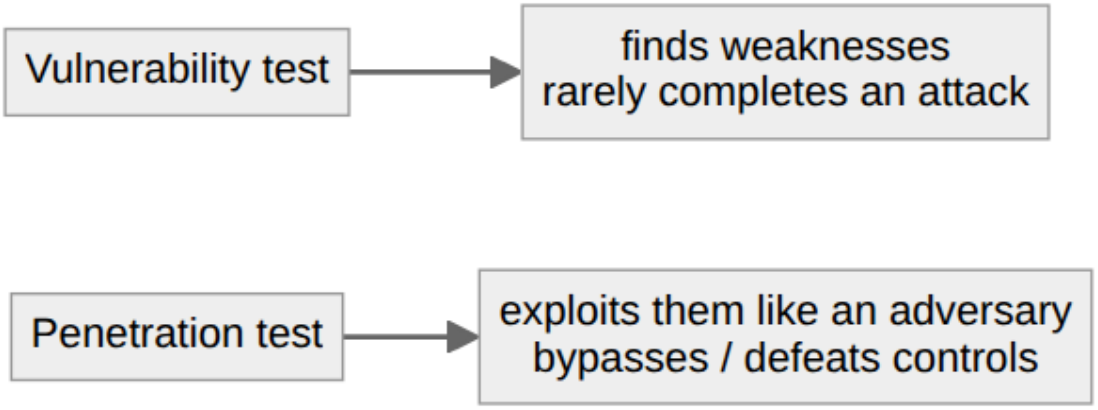
Thu, Aug 13, 4:00 PM EDT · https://scottslab.io/posts/vulnerability-testing-vs-penetration-testing

TL;DR

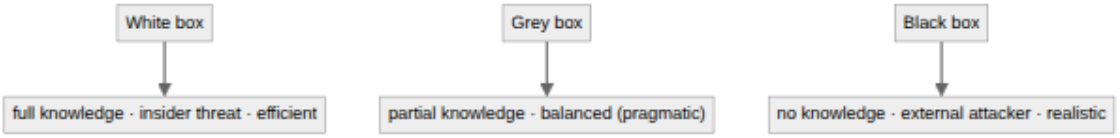
A vulnerability test finds weaknesses; a penetration test proves them by actually exploiting them like an attacker would. That difference is the point: a scanner says "port 445 looks vulnerable," a pentester walks through it, pivots, and shows you what it costs. None of it starts without documented Rules of Engagement. The knowledge level, white, grey, or black box, decides which attacker you're simulating.

These two get lumped together and they are not the same job. A vulnerability test probes for weaknesses and stops there. It tells you what looks exploitable. A penetration test simulates a real attack: the tester acts as an adversary and actually tries to bypass or defeat your controls, chaining findings into a working intrusion. The scanner hands you a list; the pentester hands you a story of how far they got with it. Both are useful, but only one tells you whether a "medium" finding is actually a path to the crown jewels.

Because a pentest is a real attack by consent, nothing happens until the Rules of Engagement are written down. The ROE defines what's in scope, which systems are fair game, and which techniques are allowed: no social engineering, no denial of service, only these IP ranges, only these hours. It's the document that keeps an authorized test from becoming an incident (or a crime), and skipping it is how well-meaning tests take down production and end careers.



Then there's how much the tester knows going in, which decides which threat you're modeling. White box means full knowledge of the network: architecture, credentials, source. It simulates an insider, or a determined attacker who's already done the homework, and it's efficient because no time is burned on discovery. Black box is the opposite: no prior knowledge, the tester starts from the outside like a stranger on the internet, which is realistic but spends real effort just mapping the attack surface. Grey box splits the difference with partial knowledge, and it's usually the pragmatic pick: an external attacker's perspective without paying full price for reconnaissance.



The takeaway is to match the engagement to the question you're actually asking. Want to know whether you'd survive a random internet attacker? Black box. Want to know how bad a compromised employee could get, fast? White box. Want a realistic result on a budget? Grey box. And whichever you pick, the ROE is written and signed before anyone touches a keyboard.

How a Pen Test Actually Runs: Discovery, Attack, and Leaving No Trace

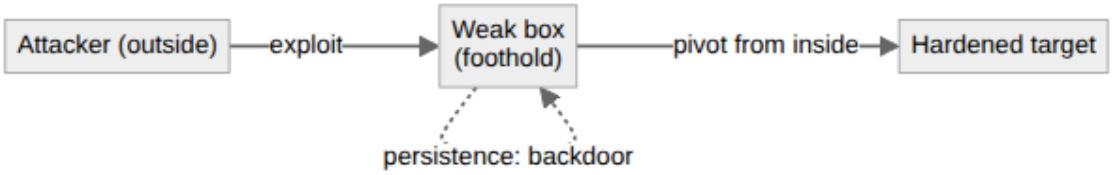
Fri, Aug 14, 9:00 AM EDT · https://scottslab.io/posts/how-a-penetration-test-runs

> **TL;DR**
>
> NIST models a pentest as an iterative loop between discovery and attack, not a straight line. You recon, you break in, and what you find inside sends you back to recon. Discovery is reconnaissance (passive and active, OSINT, footprinting, even war driving and war flying). The attack phase is gain access → escalate privileges → move laterally, with pivoting (using a weak box as a foothold to reach hard targets) and persistence (backdoors that survive the original hole). Then you clean up everything and restore what you touched.

A real engagement isn't a single shot; NIST frames it as a loop between discovery and attack that you run over and over. You learn something, you use it to get in, and getting in reveals new things to learn. So you drop back to discovery from a better vantage point. The map you draw from outside is nothing like the one you draw from a foothold inside.

Discovery is reconnaissance, and it comes in flavors. Passive recon watches without touching: OSINT from public sources, footprinting the organization's exposed surface. It leaves no trace. Active recon actually probes, which is louder but richer. It even goes physical: war driving to find wireless networks from a car, or war flying to do the same from a drone. The goal throughout is to build the attack surface before you attack it.

The attack phase is the part people picture, and it has a shape: gain access through some weakness, escalate privileges from whatever you landed as to something powerful, then move laterally to reach what you actually came for. Two techniques carry a lot of the weight. Pivoting is using a system you've already popped, often a soft, low-value box, as a foothold to reach the hardened targets you couldn't touch from outside. The weak machine becomes your inside vantage point. Persistence is installing a way back in that doesn't depend on the original exploit: a backdoor. So when the hole you came through gets patched, you're still there.



The part amateurs skip is the ending. A professional pentest cleans up after itself: remove the backdoors, delete the tools and artifacts, and restore every modified system to its pre-test state. You were simulating an attacker, not becoming a liability. Leaving persistence mechanisms or altered configs behind turns your security assessment into the very thing it was supposed to find. Test, document, and put everything back the way you found it.

Zero-Days and Responsible Disclosure: The Window Nobody Can Patch Yet

Fri, Aug 14, 4:00 PM EDT · https://scottslab.io/posts/zero-days-and-responsible-disclosure

TL;DR

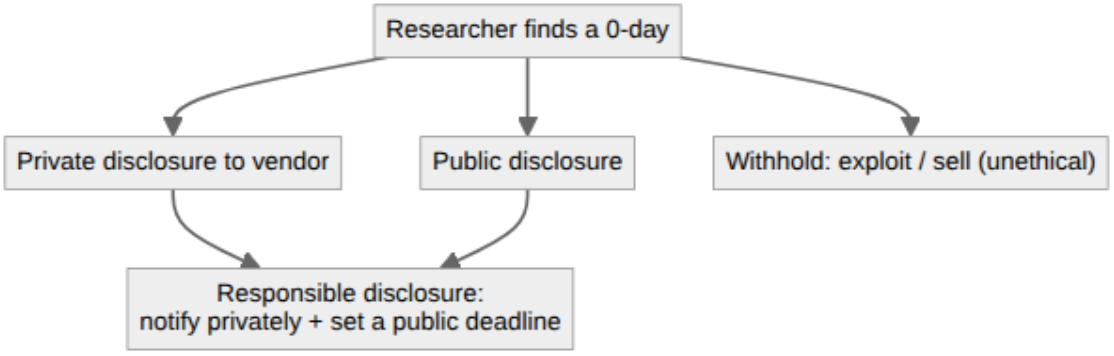
A zero-day is a vulnerability nobody's supposed to know about yet: no patch, no compensating control, defenders blind. The window of vulnerability opens the moment it's discovered and closes only when the vendor ships a fix and admins apply it. A researcher who finds one has three choices: tell the vendor privately, go public, or withhold it (exploit or sell, which is where the ethics end). Responsible disclosure threads the needle: tell the vendor first, set a public deadline, and let the clock create pressure to patch.

"Zero-day" gets used loosely, so pin it down: it's a vulnerability unknown to the people who'd defend against it, with no patch available and no compensating control in place. That's what makes it dangerous: not that the bug is clever, but that nobody's watching for it, so there's nothing standing between it and the systems it affects.

The useful mental model is the window of vulnerability. It opens the instant the flaw is discovered and stays open until two things happen: the vendor releases a patch, and administrators actually apply it. That second half is the one people forget. Once the vendor ships a fix the flaw technically stops being a zero-day and becomes an *n-day* (a known, patchable bug), but an unpatched server is just as exposed to

it, and now the patch itself hands attackers a roadmap for building the exploit. The whole game of disclosure is about closing that window fast without handing attackers a head start while it's open.

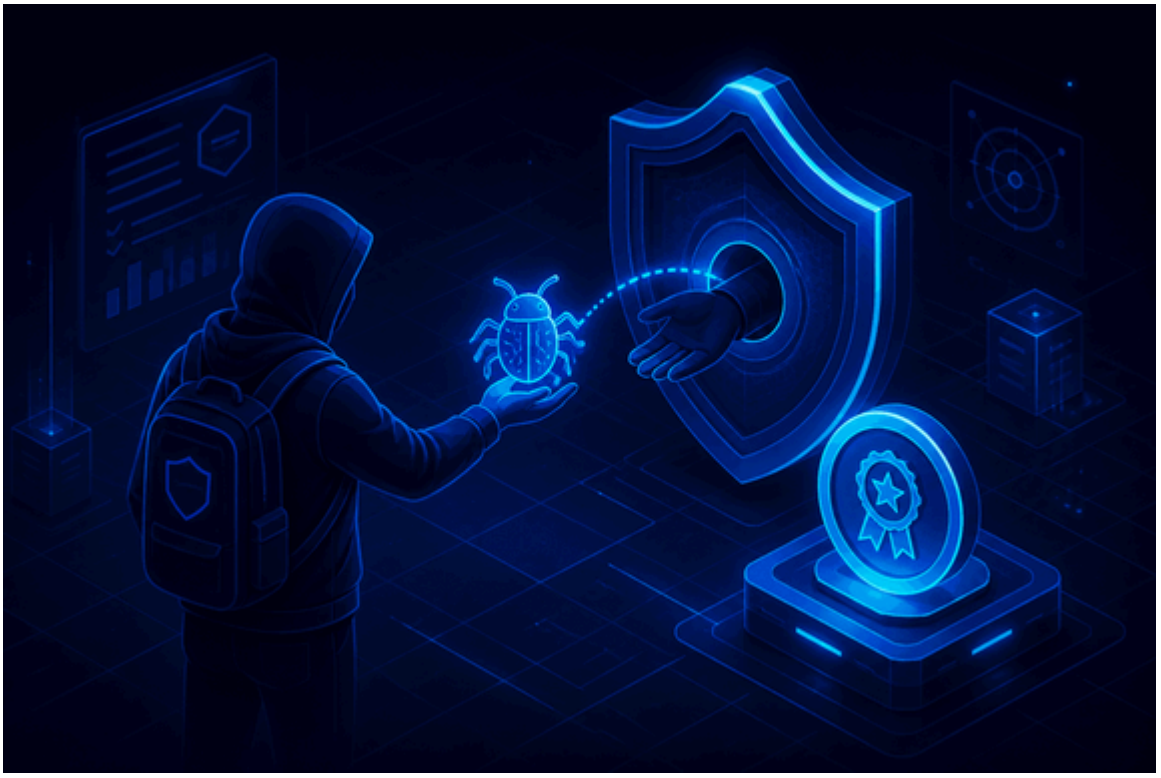
A researcher who finds one has three options, and only some are defensible. They can disclose privately to the vendor and let them fix it quietly. They can disclose publicly. That warns everyone, including attackers. Or they can withhold it: sit on it, exploit it themselves, or sell it on the black market. That's unambiguously the unethical path, because it leaves everyone exposed for someone's private gain.



The community standard is responsible disclosure, and it's a deliberate compromise. You notify the vendor privately first, giving them a real chance to fix it before anyone else knows. But you also set a public disclosure deadline: a date you'll go public whether they've patched or not. That deadline is the whole point: private notice alone lets a vendor sit on a fix indefinitely, while immediate public disclosure endangers users. The deadline balances the two: enough time to remediate, enough pressure that they actually do.

Bug Bounty Programs: Paying Attackers to Be on Your Side

Sat, Aug 15, 10:00 AM EDT · https://scottslab.io/posts/bug-bounty-programs

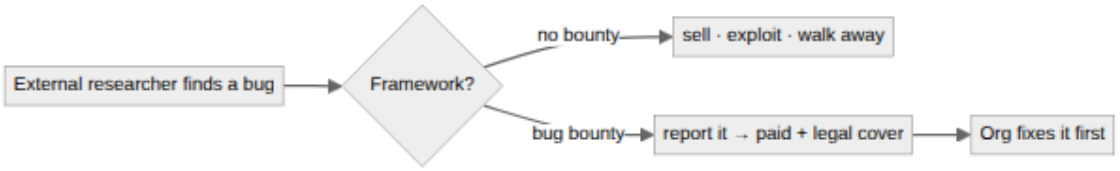


TL;DR

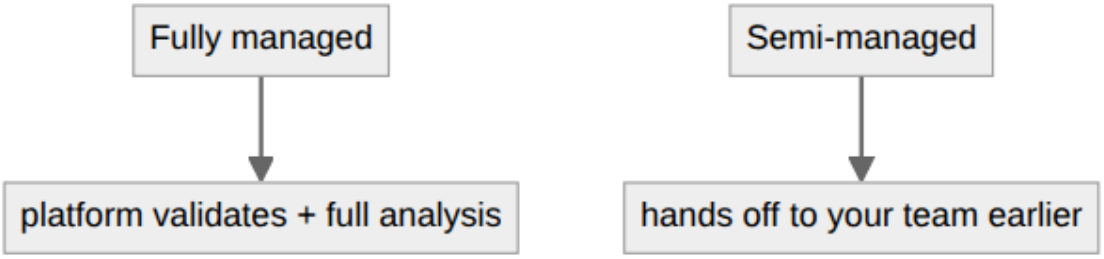
A bug bounty formalizes outside-researcher access in a legal, controlled framework, and it works by aligning incentives: a researcher who'd otherwise have no safe way to report (or a tempting reason to sell) gets paid to hand you the bug instead. Programs run fully managed (the platform validates and analyzes reports for you) or semi-managed (they hand off to you earlier). The payouts are real: Google paid $112,500 for a Pixel flaw; the DoD's "Hack the Army 2.0" surfaced 146 vulnerabilities for over $275K.

A bug bounty program is how you turn the internet's curiosity into a security asset instead of a threat. It gives external researchers a legal, controlled framework to probe your systems and report what they find: scope, rules, and a promise you won't sue them for looking. Without that framework, a well-meaning researcher who trips over a flaw has a genuinely bad set of options; the bounty gives them a safe, rewarded one.

The reason it works is incentive alignment. The same person who could sell a vulnerability on the black market, or just walk away and leave you exposed, now has a reason to bring it straight to you: money, recognition, and legal cover. You're not hoping attackers are nice. You're making "report it to the owner" the most attractive move on the board.



Operationally, programs come in two shapes. A fully managed program leans on a platform to triage incoming reports, validate them, and deliver you complete analysis. You get vetted findings, not a firehose. A semi-managed program hands off to your own team earlier in the lifecycle, which is cheaper and gives you more control but puts the triage burden back on you. The right choice comes down to whether you have the internal capacity to separate real bugs from noise.



And the numbers aren't token gestures. Google paid a researcher $112,500 for a single Pixel phone vulnerability back in 2018. The US Department of Defense's second Army bug bounty, "Hack the Army 2.0," ran in late 2019 (results announced January 2020). 52 hackers found 146 valid vulnerabilities for more than $275,000 in payouts, which is a rounding error against what any one of those bugs could have cost in a real breach. (The first Hack the Army, back in 2016, drew nearly 400 hackers and paid out around $100,000 for 118 bugs.) That's the pitch in a sentence: bounties are cheap compared to incidents.

Red, Blue, and Purple Teams: Running the Fight on Purpose

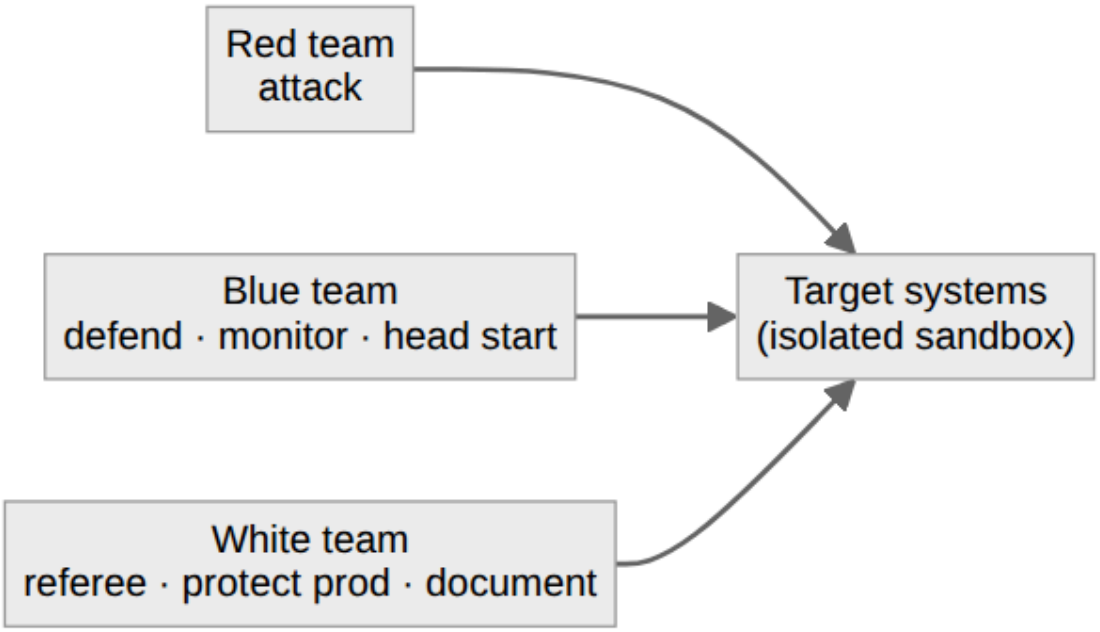
Sat, Aug 15, 2:00 PM EDT · https://scottslab.io/posts/red-blue-and-purple-teams

TL;DR

A security exercise splits into teams: red attacks, blue defends (usually with a head start), and a white team referees to keep it out of production and write up the lessons. Purple teaming is the part that actually makes you better: red and blue sharing findings afterward so the attacks teach the defenses. Capture the Flag gives red concrete objectives scored against blue's prevention, and all of it runs in an isolated sandbox, never on live systems.

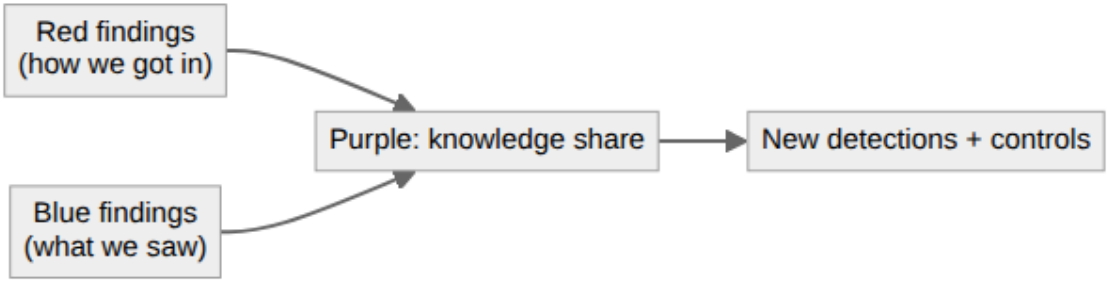
The colored teams are just roles in a staged fight, and knowing them keeps the point in view: you're not trying to win, you're trying to learn where you'd lose.

Red team is offense: they attempt to compromise systems, playing the adversary as realistically as the rules allow. Blue team is defense: securing, monitoring, and responding. They usually get a head start, because the exercise is testing whether prepared defenders can hold, not whether they can be ambushed cold. The tension between the two is the whole product: red finds the gaps, blue finds out whether they can see and stop what red is doing.



There's a third team people forget: the white team, the observers and referees. They set the boundaries, make sure the exercise doesn't spill into production and cause a real outage, adjudicate disputes, and, most importantly, document the lessons learned. Without a white team, an exercise either descends into chaos or quietly damages something real, and either way nobody writes down what happened.

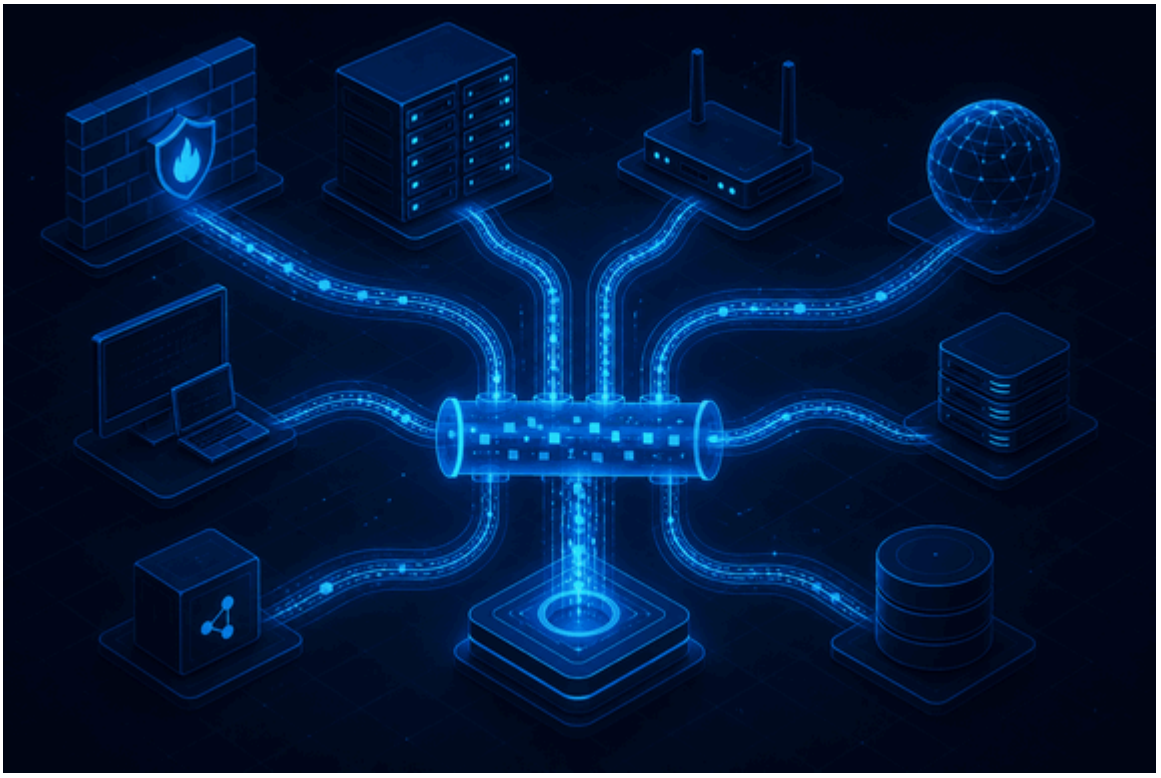
The color that matters most is purple. Purple teaming isn't a separate squad so much as what happens after the whistle: red and blue sit down together and combine findings, so every attack red pulled off becomes a detection or control blue builds next. An exercise where red "wins" and walks away taught you nothing; the value is entirely in the debrief where offense explains exactly how, and defense turns that into coverage. A common format is Capture the Flag: red pursues specific objectives, the flags, scored against blue's ability to prevent them. That keeps the whole thing concrete and measurable.



One rule runs under all of it: these exercises happen in isolated sandbox environments, not on production systems. You're deliberately trying to break things and deploy real attack techniques; doing that against live infrastructure trades a learning exercise for a self-inflicted incident. Build the range, run the fight there, and bring the lessons back.

Log Sources: What to Collect Before You Can Analyze Anything

Sat, Aug 15, 6:00 PM EDT · https://scottslab.io/posts/log-sources-and-data-types



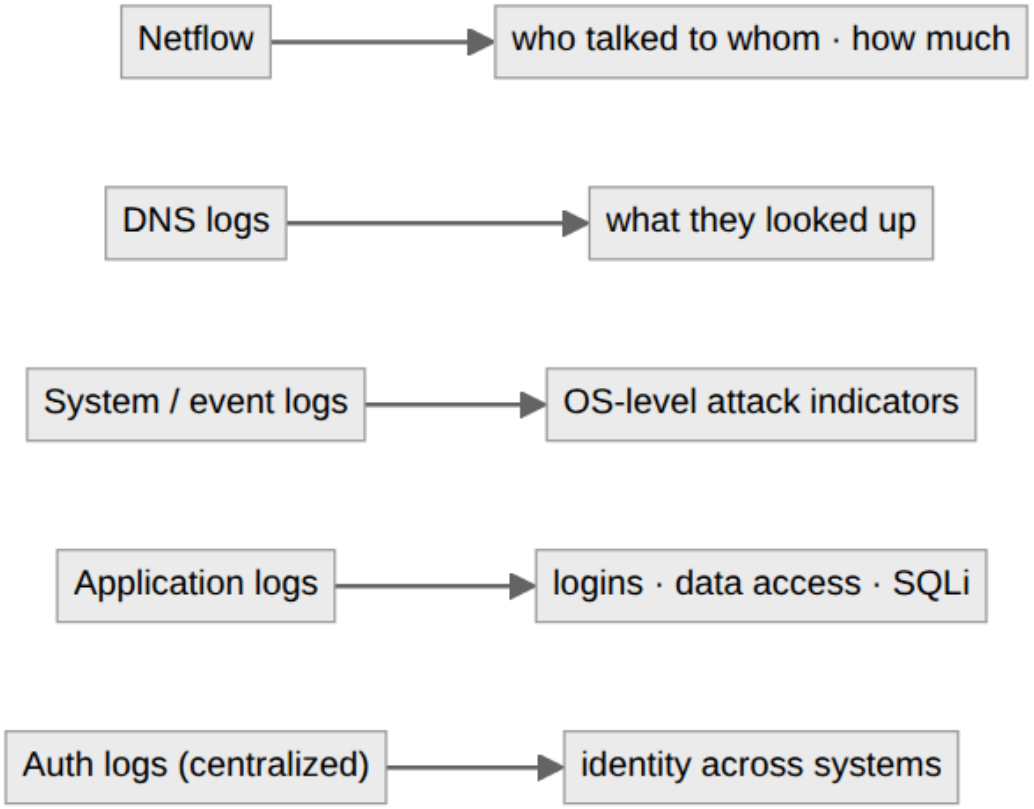
TL;DR

You can only investigate what you logged, and each source answers a different question: netflow tells you who talked to whom and how much, DNS shows what they looked up, system/event logs carry OS-level attack indicators, application logs catch logins and SQL injection, and centralized auth logs tie identity together. Beyond the staples there's a long tail you pull in when the investigation needs it: VoIP/SIP, full packet captures, memory analyzers, scan output.

Every investigation I've run was bounded by one thing: what was actually being logged at the time. You cannot analyze a log you didn't collect, so the first job in security monitoring is deciding what to capture. The useful way to think about sources is by the question each one answers.

Network flow logs (netflow) answer "who talked to whom, and how much." They don't carry packet contents, but they show which systems communicated and the data volumes, which is exactly what you need to spot an exfiltration or a beacon. DNS logs answer "what did they try to reach": every external lookup, which is where a lot of C2 and phishing infrastructure shows itself before any payload moves. System and event logs are the OS-level record: security events and the indicators of an attack on the host itself. Application logs go a layer up: logins, data access, and the telltale signs of things like SQL

injection in the requests. And authentication logs, ideally centralized, tie identity across all of it, internal and external.



Then there's the specialized tail you reach for when a case demands it. VoIP and call-manager logs (SIP traffic) matter when the phone system is in scope. Full network traffic dumps and memory analyzers give you depth when flow logs and event logs aren't enough: the actual packets, the actual RAM. And vulnerability scan output is a log source in its own right, telling you what was weak at a point in time.

The practical lesson is that coverage is a decision you make *before* the incident, not during it. The gap you didn't fill six months ago is the blind spot you'll be staring into when it counts. Pick your sources deliberately, get them flowing to one place (which is the next problem), and remember that the cheapest log to have is the one you were already collecting.

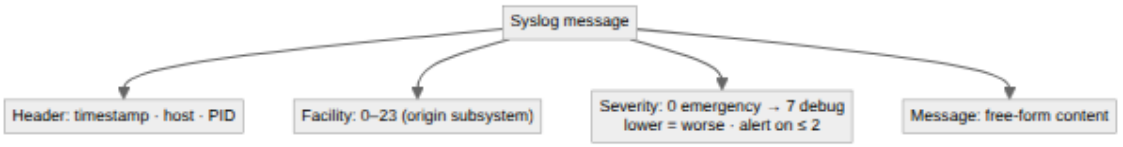
The Syslog Standard: Facilities, Severities, and Which Variant to Run

Sun, Aug 16, 10:00 AM EDT · https://scottslab.io/posts/the-syslog-standard

> **TL;DR**
>
> Syslog is the lingua franca of Unix logging, and each message has four parts: a header (timestamp, host, process ID), a facility code (0–23, where it came from), a severity (0 emergency → 7 debug, lower is worse), and the free-form message. A common alert rule fires on severity 2 or lower. The original syslog is deprecated; syslog-ng and rsyslog are what actually run on Linux, journalctl reads systemd's binary journal, and NXLog centralizes across platforms including Windows.

If logs are the raw material, syslog is the format most of them arrive in, and knowing its anatomy makes filtering and alerting far less mysterious.

A syslog message has four components. The header carries the basics: a timestamp, the sending host (a hostname, or an IP when no name is available), and the originating process ID. The facility is a code from 0 to 23 that says where the message came from (kernel, mail, auth, and so on), which is how you route or filter by subsystem. The severity is the one worth memorizing, because it's backwards from intuition: it runs 0 to 7 where 0 is emergency and 7 is debug, so a *lower* number is *more* severe. And the message is the free-form content the process wanted to record. A sane default alert rule is to page on severity 2 (critical) or lower. Here, "lower" means "worse."



The implementation has drifted over the years, which trips people up. The original syslog daemon is largely deprecated now. Syslog-NG came along in 1998 and added the things the original lacked: encryption and reliable delivery. Rsyslog arrived in 2004 with further enhancements, and both syslog-ng and rsyslog are what you'll actually find running on Linux today. Separately, systemd brought journalctl, which reads a binary journal format instead of syslog's plain text. It's more structured and queryable, but not a text file you can grep, which surprises people the first time. And when you need to pull it all together across operating systems, NXLog centralizes both syslog and Windows sources.

The reason any of this matters for security is that filtering and forwarding rules live on these fields. You alert on severity, you route by facility, and you ship it all off the host. A log that only exists on the box that generated it is a log an attacker can delete. Which is exactly what a SIEM is for.

SIEM: Turning a Pile of Logs Into a Story

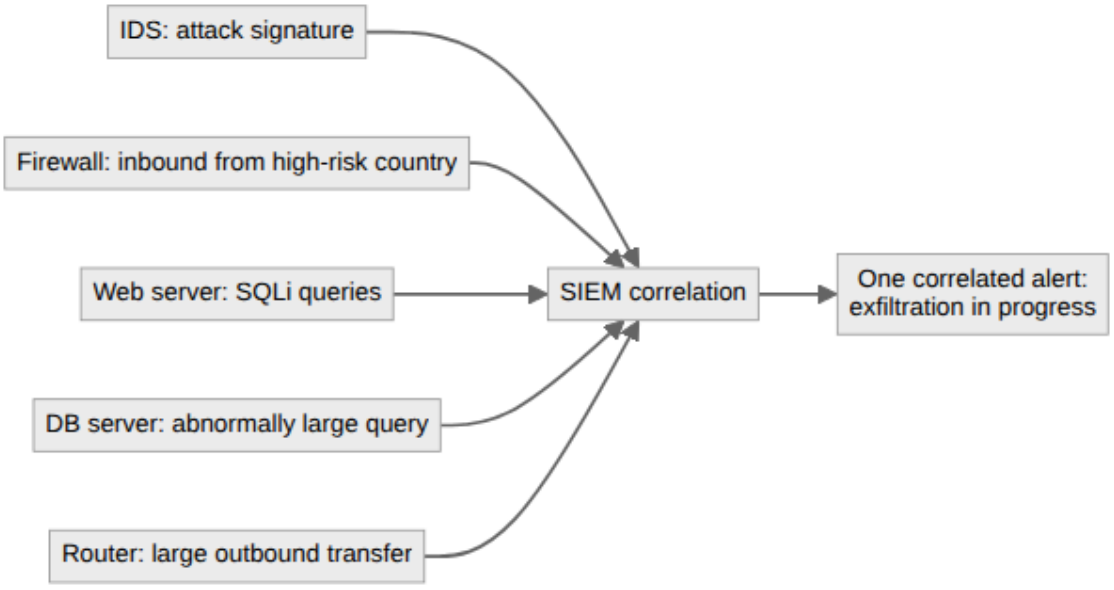
Sun, Aug 16, 2:00 PM EDT · https://scottslab.io/posts/siem-and-log-correlation

TL;DR

A SIEM does two things: collect logs securely from everything in one place, and correlate across them to spot patterns no single device can see. That correlation is the whole value. It solves the silo problem where each team only sees their own fragment. A real attack lights up in sequence across sensors (IDS signature → firewall connection from a risky country → SQLi at the web server → huge DB query → big outbound transfer), and only the SIEM sees all five as one event.

I've written before about building a SIEM in my home lab, so here's the fundamentals view of why it exists at all. A SIEM (Security Information and Event Management) does two core jobs. First, it's central, secure log collection: every sensor and device ships its logs to one place, off the hosts that generated them, where they can't be quietly tampered with. Second, and this is the part that earns its keep, it correlates (increasingly with AI/ML) across all those sources to surface patterns of malicious activity that no single log would reveal.

The problem it solves is silos. The firewall team sees firewall logs. The web team sees web logs. The DBA sees database logs. Each is looking at one frame of a movie and none of them can see the plot. The SIEM assembles the frames.

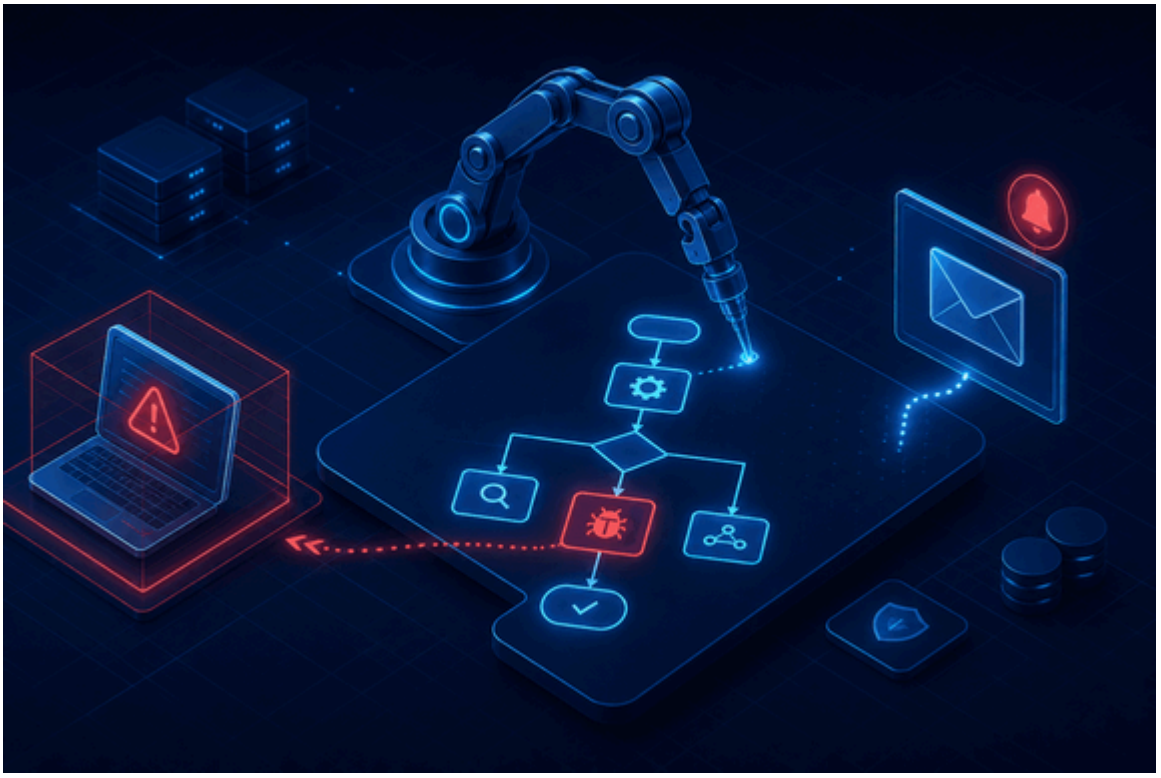


The classic example makes it concrete. On its own, each of these is a shrug; in sequence, it's an intrusion. The IDS flags an inbound attack signature. The firewall logs an inbound connection from a high-risk country. The web server reports suspicious SQL injection queries. The database server reports an abnormally large query coming from the web app. And the router reports a large outbound data flow to the internet. Five different devices, five different teams. And one attacker walks straight through, visible only when something is watching all five at once and connecting them in order.

That's what the SIEM dashboard is for: centralized alerts across every source, trend analysis over time, and sensitivity you can tune so the signal isn't buried in noise (the false-positive fight never ends). Collection gets the data into one place; correlation turns it into the sentence "someone is exfiltrating your database right now." The next question is whether it can act on that by itself. That's where SOAR comes in.

SOAR: When the SIEM Stops Watching and Starts Acting

Sun, Aug 16, 6:00 PM EDT · https://scottslab.io/posts/soar-security-orchestration-automation-response



> **TL;DR**
>
> SOAR extends a SIEM from detection into action, and the honest framing is a spectrum of how much human sits in the loop. One warning first: "playbook" reliably means the automated workflow, but "runbook" means whatever the vendor in front of you decided it means. There is no industry definition, so stop arguing about the word and say what you actually mean. The real design question isn't what to automate. It's what happens when automation fires on a false positive.

A SIEM tells you what's happening. SOAR does something about it. That's the whole relationship: Security Orchestration, Automation and Response sits on top of detection and closes the gap between "the SIEM noticed" and "something was done", which in a manual shop is measured in however long it takes an analyst to look.

First, the terminology trap

Exam-style material tends to split it this way: a **playbook** mixes human and automated steps and is tied to your incident-response policy, while a **runbook** is fully automated end to end.

That is not what the word means once you're inside a product, and it's worth knowing before a meeting goes sideways.

"Playbook" is the reliable one. In SOAR platforms it consistently means *the automated workflow*: the thing that executes, with conditional logic and API calls into your other tools. Splunk SOAR uses it exactly that way, and so does essentially every competitor.

"Runbook" is the unreliable one, and this is where I'd stop trusting any single definition:

- In Splunk's own ecosystem, the artefact an analyst *follows by hand* isn't called a runbook at all. It's a **workbook**, the case-management checklist, with playbooks attached to individual tasks where automation applies.
- Meanwhile "Azure Automation Runbooks" are *scripts*. Fully automated. The opposite of a manual document.
- Some vendors define a runbook as one narrow technical task and a playbook as the broader response framework: a completely different axis from how much automation is involved.
- And at least one major vendor's own documentation says plainly that in practice there is no difference and people use the two interchangeably.

So this isn't a clean inversion you can memorise. It's genuine industry-wide inconsistency, and the study definition is one convention among several rather than the wrong half of a neat swap.

The practical move is to stop arguing about the noun. Say **fully automated**, **automated with an approval gate**, or **documented manual procedure**. Those phrases mean the same thing in every room you'll ever stand in, and nobody has to guess which vendor's dialect you learned.

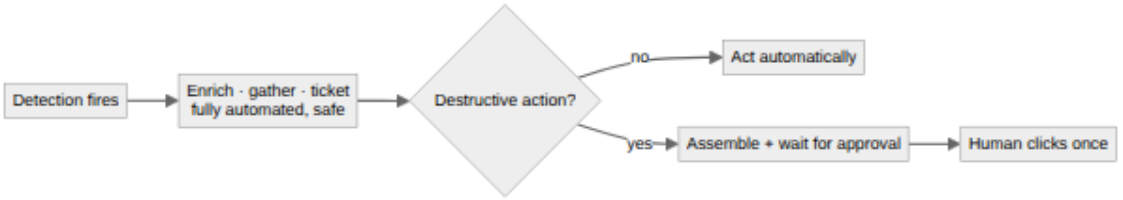
The spectrum that actually matters

Forget the labels and think about where the human sits:

Fully automated. The system acts the instant a detection fires. Right for fast, unambiguous, low-blast-radius responses: enrich an alert with threat intelligence, pull the file hash, open the ticket, gather context so the analyst opens a complete picture rather than a bare alert. Enrichment is where almost every SOAR programme should start, because it is enormously useful and **cannot hurt anything**.

Automated with an approval gate. The system assembles everything and waits for one click. This is the sweet spot for anything destructive: the analyst spends five seconds approving rather than five minutes gathering.

Documented manual. Judgment, legal exposure, or anything you can't yet trust to a rule.



The question nobody asks until it's too late

Everyone designing automation asks "what can we automate?" The better question is: **what does this do when it fires on a false positive?**

Automatically isolating a host is the textbook example, and it's the right response to confirmed ransomware. Now run it against a noisy detection at 09:00 on a Monday. If the matched host is a domain controller, a payments gateway or the database everything depends on, your automation has just caused an outage that your detection rule merely *suspected* was an attack.

So the practical rules:

Scope by blast radius, not by confidence alone. Maintain a list of assets automation may never touch unattended. Confidence in the rule is not the only variable. Cost of being wrong is the other one, and it isn't uniform across your estate.

Make every automated action reversible, and know how. If you auto-isolate, the un-isolate has to be one step and someone has to have done it in a drill. An automation you cannot undo at 3am is a liability wearing a badge.

Run new automation in dry-run first. Log what it *would* have done for a couple of weeks. The list is always surprising, and it's much cheaper to be surprised in a log file.

Automate the response, not the decision, until the decision has earned it. Start with enrichment, graduate to gated action, and promote to unattended only for detections whose false-positive rate you have actually measured.

What it's really buying you

The pitch is speed, and speed is real. An attacker moving laterally is racing your response time, and automation doesn't get tired or paged at 3am.

But the bigger win is **consistency and attention**. A tired analyst at hour seven handles the fortieth phishing report differently from the first. Automation handles it identically every time, which both

improves the floor and frees the humans for the work that actually needs a brain. Most SOC time is spent gathering context, not deciding, and gathering context is exactly what a machine is good at.

The trap is treating SOAR as a headcount substitute rather than an attention multiplier. Playbooks are code. They break when an API changes, they rot when a detection is retuned, and an unmaintained automation library fails silently in a way a human never would: it just stops doing the thing, and nobody notices, because nothing errored.

Automate the certain, gate the ambiguous, document the rest. Then test the automation the way you'd test anything else you depend on.

Continuous Monitoring: Turning Logs Into Ongoing Awareness

Mon, Aug 17, 9:00 AM EDT · https://scottslab.io/posts/continuous-monitoring-and-analytics

TL;DR

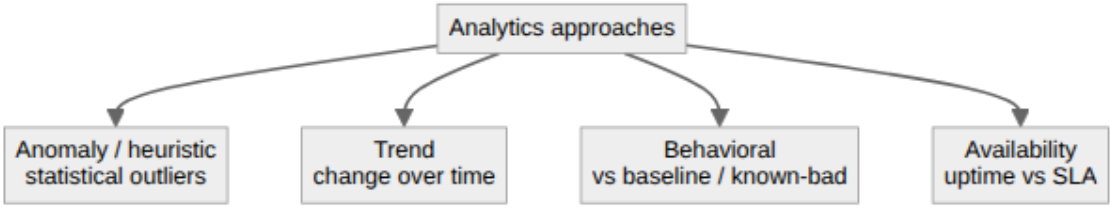
NIST defines continuous monitoring as maintaining ongoing awareness of vulnerabilities and threats to support risk decisions. Not a quarterly scan, but an always-on program aligned to your risk tolerance, adaptable, and actively managed. It runs a six-step loop (strategy → metrics → automate collection → analyze → respond → review) and leans on four analytics styles: anomaly, trend, behavioral, and availability. Endpoint monitoring and UEBA push it down to individual hosts and users.

Continuous monitoring is what all the logging and correlation is *for*. NIST's definition is worth quoting in spirit: maintaining ongoing awareness of vulnerabilities and threats so you can make informed risk decisions. The key word is ongoing. This isn't a scan you run each quarter and file, it's an always-on picture of where you stand. NIST frames three characteristics that keep it honest: it's aligned to your risk tolerance, it's adaptable as things change, and it has active management involvement, because a monitoring program nobody in charge looks at is just disk usage.

The program runs as a six-step loop. Define a strategy based on your risk tolerance. Establish metrics and how often you'll assess them. Implement automated collection and reporting. Automated, because humans can't sustain "continuous." Analyze and report the findings. Respond by mitigating, avoiding,

transferring, or accepting the risk: the only four things you can ever do with a risk. And review and update the program, because the last step feeds the first.

The analysis itself comes in four flavors, and good monitoring uses all of them. Anomaly (or heuristic) analysis flags statistical outliers: the bandwidth spike on January 13 that doesn't match any normal day. Trend analysis tracks change over time, and change is a signal in both directions: a sudden *drop* in account compromises is as worth a look as a spike, because it might mean detection broke, not that the attackers went home. Behavioral analysis compares activity against baselines or known-malicious patterns. And availability analysis watches uptime against your SLAs, because "is it even up" is a security question too.



Two things push this all the way down to the edges. Endpoint monitoring treats processor, memory, and file-system activity as security indicators: a process pinning the CPU or a sudden burst of file changes is often the first sign of ransomware. It folds malware protection, OS logs, and file-integrity monitoring back into the SIEM. And UEBA (User and Entity Behavior Analytics) builds machine-learning models of what normal looks like for each user and flags the deviations for an analyst, which is how you catch the compromised account that's authenticating perfectly but behaving like someone else. The whole point, from a single log line up to a UEBA model, is the same: know what normal is, so abnormal has somewhere to show up.

Code Review, Done Formally: The Fagan Inspection

Mon, Aug 17, 4:00 PM EDT · https://scottslab.io/posts/code-review-fagan-inspection

> **TL;DR**
>
> Fagan inspection is the most formal code review there is: six defined steps, five named roles, entry and exit criteria, and a hard ceiling on how fast you're allowed to read. Fagan's own 1976 IBM data found 38 defects per thousand lines by inspection against 8 by unit test, and 82% total detection efficiency. Almost nobody runs the full ceremony. Every shortcut your team takes maps to a specific step, and the rate limit is the one that actually decides whether review works.

Code review usually means someone skims your pull request, leaves two comments, and clicks approve. That's fine for a lot of changes. It's worth knowing what the rigorous end of the spectrum looks like, though, because everyday review is a shortcut for it. If you can name which part you cut, you can predict what leaks through.

That rigorous end is the Fagan inspection, published by Michael Fagan in the *IBM Systems Journal* in 1976. It is not a vibe. It is a defined process with roles, criteria, metrics and a stopwatch.

The six steps, and what each one is actually for

Planning. The moderator confirms the work product meets *entry criteria*: it compiles, it passes its own tests, the author isn't sending half-finished work to be finished by committee. Then materials go out and

the meeting is scheduled. Entry criteria are the step most teams don't realise exists, and they are the whole reason an inspection isn't a design session in disguise.

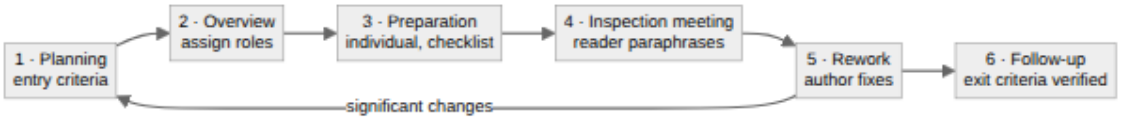
Overview. The author briefs the team on what the code is for and roles are assigned. This is the only step Fagan considered optional. If everyone already knows the subsystem, skip it.

Preparation. Each inspector reads the material *alone*, against a checklist, and logs candidate defects before anyone meets. This is where inspections actually find bugs. Skip it and the meeting becomes a group read-through, which finds a fraction as much.

Inspection meeting. The reader paraphrases the code aloud. Not the author. That distinction matters more than it looks: if the author narrates, they explain what they *meant*, and the team reviews the intention rather than the text. A second person forced to say out loud what the code does is how you find the gap between the two.

Rework. The author fixes what was logged. Significant changes loop back to planning and the thing gets inspected again.

Follow-up. The moderator verifies every logged defect was actually resolved and checks the *exit criteria* before closing. Not "the author says it's fixed."



Five roles, and why the author isn't in charge

Fagan separated the work into distinct roles, and the separation is the point. Every one of them is a check on a different failure mode.

The **moderator** runs the process: schedules, enforces entry and exit criteria, keeps the meeting on defects, and owns follow-up. The **author** wrote the thing, answers questions, and does the rework. They deliberately do not lead. The **reader** paraphrases the work product to the group. **Inspectors** find defects. The **recorder** logs every defect raised, with enough detail that follow-up can verify it.

The role most teams collapse is recorder. Nobody writes anything down, so "we discussed it" becomes the record, and three weeks later nobody can prove whether the fix landed. If you adopt exactly one thing from this article, make it a written defect log.

The rate limit is the whole ballgame

Here is the number that decides whether your review works: roughly **150 lines of code per hour**. That's the ceiling, not a target. Read faster and detection falls off a cliff.

This is the finding people skip past, and it is the most useful thing in the entire method. It reframes the common complaint. When someone says "we do code review and bugs still get through," the question isn't whether they review. It's how many lines they approved per hour. A 2,000-line pull request skimmed in twenty minutes is running at roughly 6,000 lines an hour, forty times the rate at which humans reliably find defects. That review didn't fail. It was never a review.

The practical consequence is that **pull request size is a security control**. If you want review to catch anything, cap the diff. Everything else in this process is negotiable; the rate is physics.

The numbers Fagan actually reported

Fagan's 1976 study found **38 defects per thousand lines of code caught by inspection, against 8 per thousand caught by unit testing** on the same system, with total detection efficiency of **82%** across design and code inspection on a COBOL application.

Treat those as an existence proof rather than a promise. They came from 1970s IBM, with full-time inspection discipline and waterfall-era work products, and the modern replication literature reports a wide range depending on preparation quality. The durable claim isn't "inspections find 82%." It's that reading code carefully, before it runs, finds a different and larger class of defect than executing it does. And that this was measured, not asserted, half a century ago.

Where it lives now

Inspection isn't folklore; it's standardised. **IEEE 1028** defines five distinct review types: management reviews, technical reviews, inspections, walk-throughs, and audits. They are genuinely different things. An inspection is defect-focused, led by a trained moderator, with metrics. A walk-through is author-led and educational. If you're in a regulated environment and someone asks for evidence of "formal review," those words have specific meanings and you should use them precisely.

Almost nobody runs the full ceremony today, and that's a reasonable trade. But a modern pull request is a Fagan inspection with most of the steps deleted: no entry criteria, no individual preparation against a checklist, no reader, no recorder, no verified exit. Sometimes that's fine. On the code that authenticates users, handles money, or parses untrusted input, it isn't. Those are exactly the diffs worth putting through something closer to the real process.

The value here isn't adopting 1976 wholesale. It's that when a bug sails through review, you can name the step you cut. Nobody prepared. The author narrated. Nothing was written down. Nobody checked the

fix landed. Or, most often, the diff was too big to read at human speed, and everyone approved it anyway.

SAST, DAST, IAST: Three Ways to Test Code, and Why You Need All of Them

Wed, Aug 19, 10:00 AM EDT · https://scottslab.io/posts/sast-dast-iast-code-testing

TL;DR

SAST reads code without running it, DAST attacks it while it runs, IAST instruments it from the inside, and SCA inventories what you borrowed. Each is blind to what the others see, so they're a set, not a bracket. The part nobody tells you: SAST's real cost is false positives, DAST's is authentication and coverage, IAST's is that it only sees paths your tests reach. None of the four find broken access control, which has sat at the top of the OWASP Top 10 for years.

There are three main ways to test code for security flaws, and the mistake is treating them as competitors. They're a set. Each one is structurally blind to a class of bug the others catch, and knowing *which* blindness you're accepting is the entire skill.

Static: reads the code, never runs it

SAST parses your source (often into an abstract syntax tree, then a data-flow graph) and looks for a path from a *source* (untrusted input) to a *sink* (something dangerous: a SQL string, a shell exec, a file path). It needs no running application, so it can run on a feature branch before anything is deployed, cover code paths no test exercises, and point at the exact line.

Its cost is false positives, and they're not a rounding error. SAST can see that data flows from a request parameter into a query, but it often can't prove whether the sanitiser in between actually works, so it flags the path anyway. A tool that cries wolf on 200 lines gets muted, and a muted tool is worth nothing. The practical move is to tune ruthlessly on day one, fail the build only on high-confidence rules, and report the rest without blocking. A SAST rollout that blocks builds on everything gets switched off within a month. That's the failure mode, not the tool.

SAST is also blind to anything that only exists at runtime: your actual configuration, your reverse proxy, the environment variable that turns debug mode on in production.

Dynamic: attacks the running thing

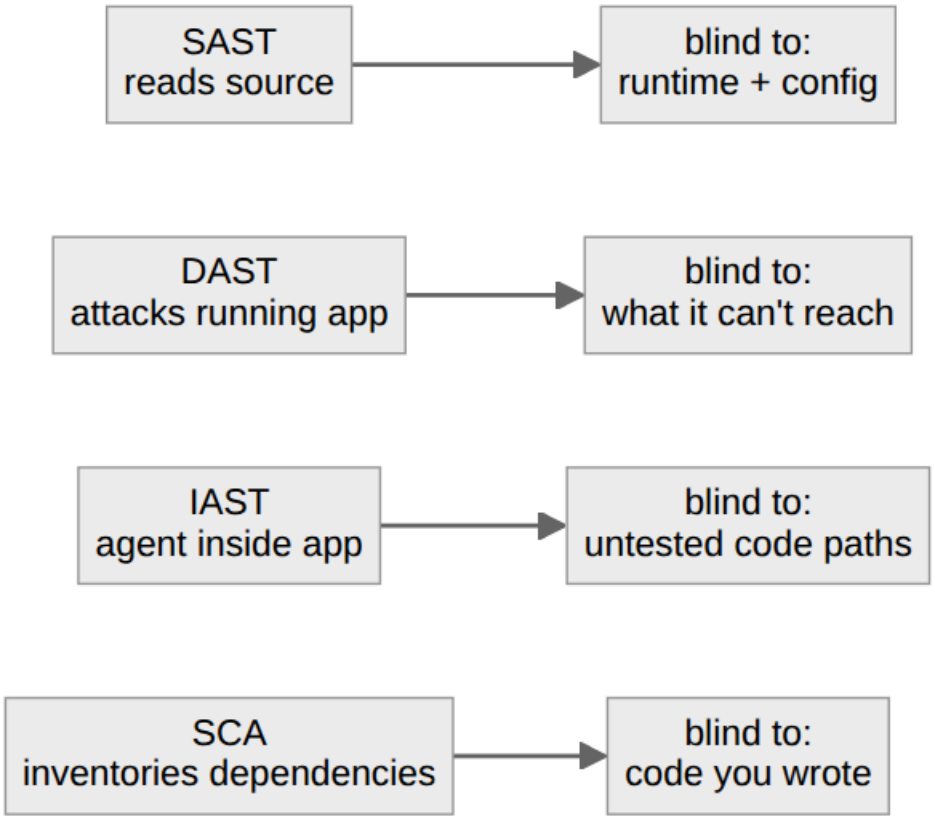
DAST runs the application, sends it input, and reads the output. It doesn't care what language you wrote it in, because it only sees what an attacker sees. That's exactly its value. It finds the misconfigured header, the verbose error page, the endpoint you forgot was exposed. A finding from DAST is almost always real, because it *did the thing*.

Its cost is coverage. DAST has to find your attack surface before it can test it, and two things break that badly: **authentication** and **client-side rendering**. If the scanner can't log in, it tests your login page and nothing else. So configuring authenticated scanning is the single highest-value thing you'll do with a DAST tool. And a single-page app that builds its UI in JavaScript is largely invisible to a crawler that only reads HTML; you point it at your API routes directly instead.

Interactive: watches from inside

IAST puts an agent inside the running application and watches data and control flow from within while the app is exercised. That's usually your existing functional tests or a DAST run. Because it sees both the request and the code path it triggered, it can say *this input reached this line*: a real vulnerability with the stack trace attached, and very few false positives.

Its cost is honest and often understated: **it only sees code your tests actually execute.** IAST inherits your test coverage. If 40% of your code is untested, IAST is blind to 40% of your code. It also needs runtime instrumentation, which is a real conversation with whoever owns production performance.



Software Composition Analysis: what you borrowed

SCA is its own category, and it answers one question: *what open-source code am I shipping, and is any of it known-vulnerable?* It reads manifests and lockfiles, builds an inventory, and matches it against known CVEs.

That sounds mundane right up to the morning a critical vulnerability drops in a library everyone uses, and the only question that matters is "am I running it?" Teams with SCA answer in minutes. Teams without it spend three days grepping.

This is also where compliance arrived. **Executive Order 14028** made a Software Bill of Materials a condition of selling software to the US federal government, and two formats matter: **CycloneDX**, an OWASP project, security-focused with built-in VEX support; and **SPDX**, a Linux Foundation standard (ISO 5962) that reaches wider into licensing and provenance. If a customer asks for an SBOM, they mean one of those two.

The interesting recent development is **reachability analysis**. Rather than alerting on every vulnerable dependency, it traces whether your code actually calls the vulnerable function. Most flagged dependencies are never reached, so this cuts the noise dramatically, and it's the difference between a dependency report someone reads and one they close.

White box, black box, and the thing none of them find

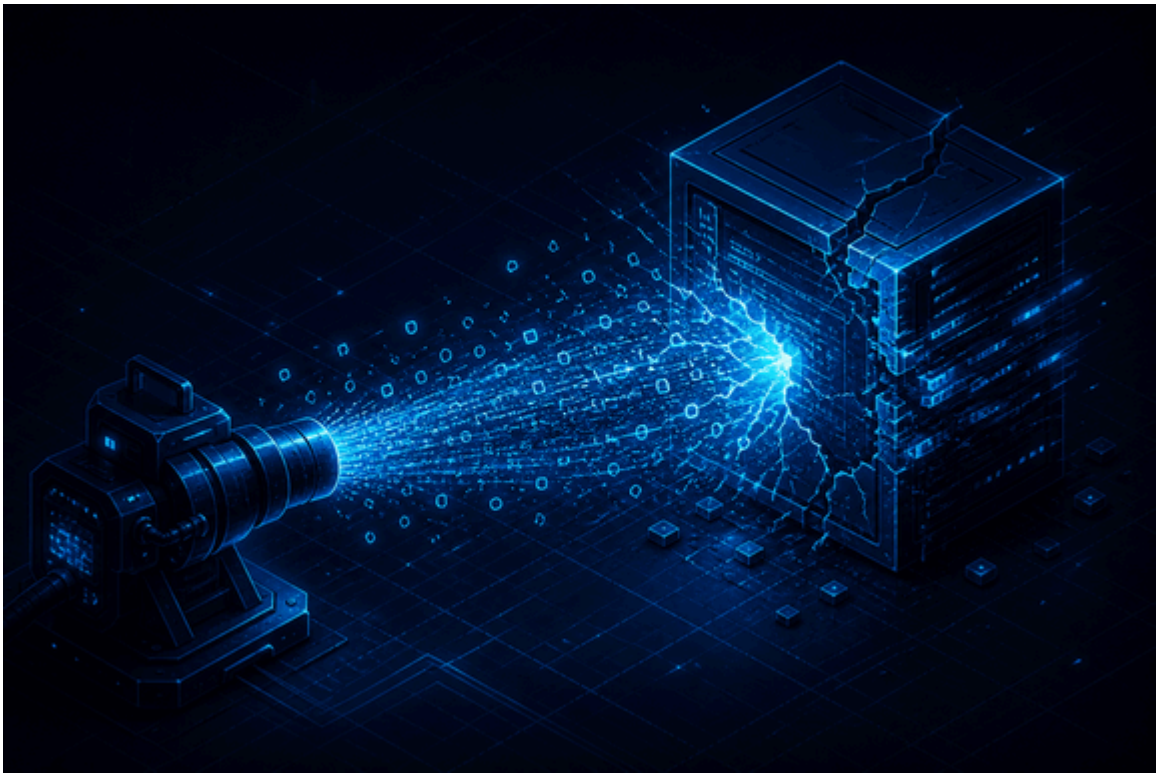
Cutting across all of it is how much the tester can see. **White box** means full source access. **Black box** means none: the attacker's view. Neither is better; they answer different questions, and a mature program does both.

Here's the part worth internalising. Every tool above finds a bug by recognising a *pattern*: a tainted path, a bad header, a known CVE. That means none of them reliably find flaws where the code does exactly what it was told and the instructions were wrong: **broken access control**, business logic abuse, an endpoint that checks you're logged in but never checks the record belongs to you. There is no pattern to match. That class has sat at the top of the OWASP Top 10 for years precisely because scanners don't catch it.

So run all four, tune them so people still read the output, and then spend your human review time where the scanners are structurally blind. The tools tell you which of your walls have holes. They cannot tell you that you built a door where a wall belonged.

Fuzzing and Misuse Cases: Testing Like Someone Who Wants It to Break

Wed, Aug 19, 6:00 PM EDT · https://scottslab.io/posts/fuzz-testing-and-misuse-case-testing



> **TL;DR**
>
> Fuzzing throws malformed input at software until something breaks. The version that works is coverage-guided: the fuzzer instruments the binary, keeps any input that reaches new code, and evolves toward the dark corners. Run it without a sanitiser and you'll miss most memory bugs, because a corruption that doesn't crash looks like a pass. Google's OSS-Fuzz has found over 13,000 vulnerabilities and 50,000 bugs this way. Misuse case testing is the human half: it finds the logic flaws no random string will ever hit.

Most testing checks that software does the right thing with the right input. These two check what it does with the *wrong* input, on purpose. That's where the security bugs actually live.

Dumb fuzzing, and why it stopped being enough

The naive version feeds a program a flood of junk and watches for a crash. Inputs come from a list of nasty strings, a script, purely random bytes (**generation fuzzing**), or by taking real valid inputs and mangling them (**mutation fuzzing**, which usually goes deeper because it starts from something the parser already accepts).

The problem is that random bytes die at the front door. If your program parses JSON, a random byte string fails validation in the first millisecond and never reaches the interesting code. You can run that for a week and test nothing but your input validator.

Coverage-guided fuzzing, which is the actual technique

The advance that made fuzzing serious is **coverage guidance**. The fuzzer instruments the binary so it can see which code paths an input reached. If a mutated input reaches a branch nothing has reached before, it's kept and becomes a parent for the next generation. The fuzzer is now hill-climbing toward code coverage instead of guessing.

This is why it finds things humans don't. It doesn't know your format. It *discovers* it, by keeping whatever gets deeper. Given a JPEG parser and one valid JPEG, a coverage-guided fuzzer will often rediscover chunks of the file format on its own, because inputs that keep the magic bytes intact reach further and survive.

The engines worth knowing are **AFL++**, **libFuzzer** and **honggfuzz**. All three are what Google's **OSS-Fuzz** runs against open source, where the results speak plainly: **over 13,000 vulnerabilities and 50,000 bugs across around a thousand projects.**

Two practical points decide whether your run works. First, the **seed corpus**: start it with real, valid, diverse inputs, because the fuzzer evolves from what you give it and a good corpus can be the difference between hours and weeks. Second, **structure-aware fuzzing** for formats with checksums or required framing. Otherwise every mutation fails an integrity check before it reaches your logic.

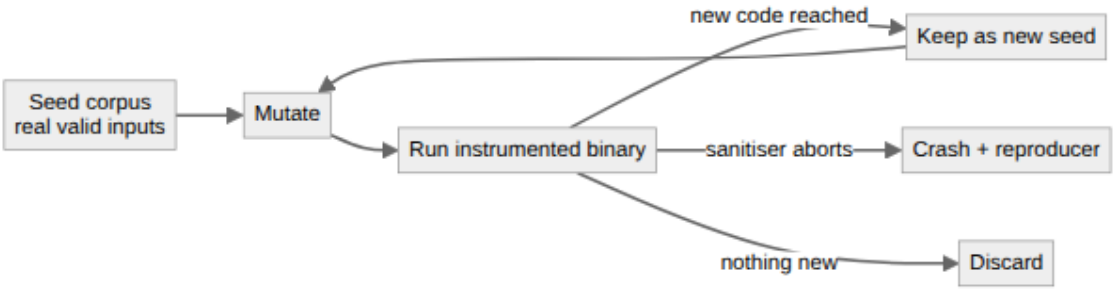
Run a sanitiser, or you're mostly wasting the electricity

This is the part that gets skipped, and it matters more than the choice of engine.

A memory bug does not reliably crash. A read a few bytes past a buffer usually returns adjacent heap memory and carries on perfectly happily. Your fuzzer sees no crash, records a pass, and moves on. The bug that leaks memory to an attacker sits right there.

Sanitisers fix that by making the failure loud. **AddressSanitizer** poisons the memory around every allocation so an out-of-bounds read aborts immediately. **UndefinedBehaviorSanitizer** catches integer overflow and the rest of the undefined-behaviour family. **MemorySanitizer** catches reads of uninitialised memory.

Fuzzing without a sanitiser finds the bugs that crash. Fuzzing with one finds the bugs that matter. Turn them on.



One serious caveat that hasn't changed: fuzzing looks like an attack because it *is* one. Get written permission from the application owner before you point it at anything you don't own. Fuzzing someone else's service is how a test becomes an incident report with your name on it.

Misuse cases: the half a fuzzer can't do

Misuse case testing comes at the same goal from the design side. Instead of throwing inputs at a running system, you deliberately think like an attacker and write test cases around it. The guiding question is literally "how would someone break this?"

The difference from a penetration test is **timing**. Misuse case testing happens during development, before it ships. A pentest happens after deployment, when fixing an architectural mistake costs twenty times more. It works best with a mix of people: the developers who built it, because they know where the bodies are buried, and outside reviewers, who aren't blinded by knowing how it's *supposed* to be used.

The recurring cases are worth keeping as a checklist, because they show up everywhere:

- **Boundary violations**: the classic being whether you can debit an account below zero. Then: can you order a *negative* quantity and get refunded? Does price times quantity overflow?
- **Missing or unexpected input**: what happens when a required field is absent rather than empty?
- **Injection**: the field that ends up in a query, a shell, a template, or a log line.
- **State-order abuse**: can you skip step two and go straight to step three? Can you replay the confirmation?

That last one is the reason humans are still required. A fuzzer will never think to complete a checkout twice, because there is no malformed *input* involved. The requests are all perfectly valid. It's the *sequence* that's wrong, and pattern-matching doesn't see sequences.

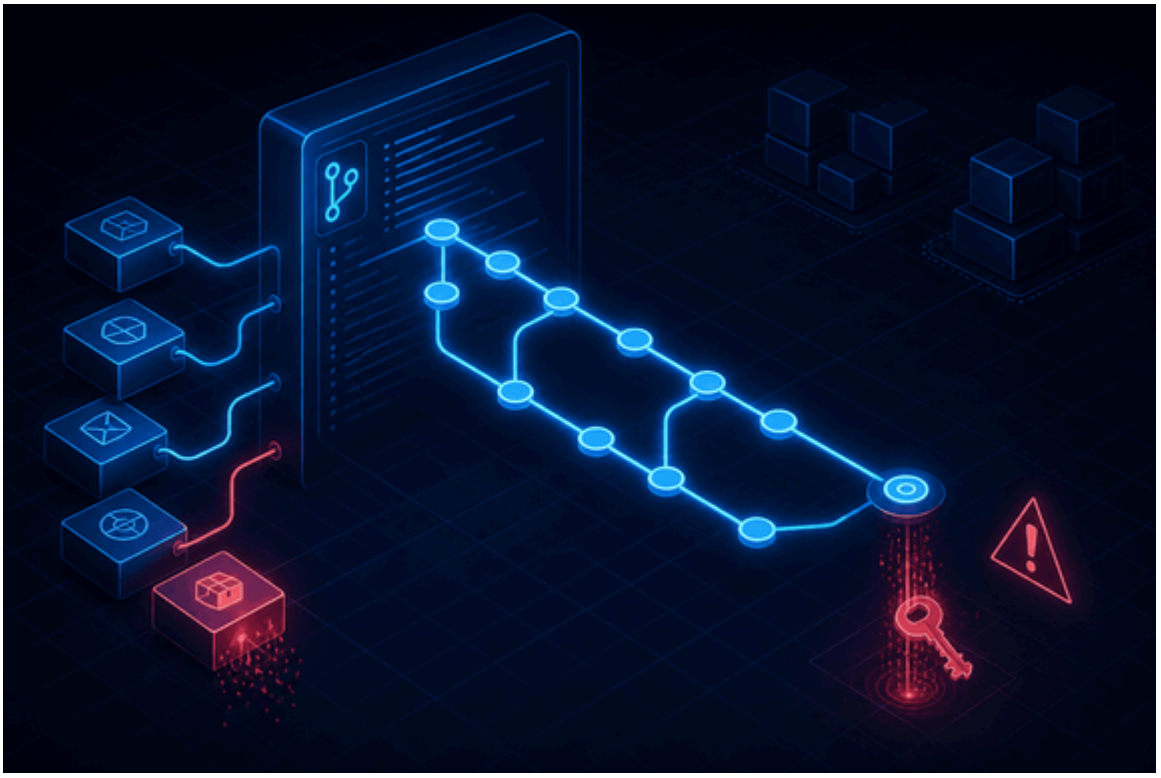
The mindset, which is the actual point

Stop testing whether it works and start testing whether it can be made to fail.

Fuzzing does that mechanically, at a scale and stupidity no human can match. It will try the four-gigabyte filename you'd never bother with. Misuse cases do it deliberately, at design time, against the logic. You want both, because the machine finds the inputs you'd never think to try, and the humans find the flaws where every input was valid and the *order* was the attack.

Code Repositories and Third-Party Code: Where Your Risk Actually Lives

Fri, Aug 21, 10:00 AM EDT · https://scottslab.io/posts/code-repositories-and-third-party-code



> **TL;DR**
>
> A secret pushed to a public repo is compromised the moment it lands, and deleting the commit does not undo it. Git keeps history, so the only fix is rotation. Borrowed code is the other half: event-stream, dependency confusion and the xz-utils backdoor were all attacks on the supply chain rather than on anyone's application. You own the risk of every line that runs in your name, whether you wrote it, borrowed it, or published it by accident.

Two of the biggest sources of risk in modern software aren't your code at all: they're where you keep it, and whose code you borrowed.

The public/private line is a one-way door

A repository is unambiguously good engineering: version control, an audit log of who changed what, easy reuse. The security wrinkle is the public/private boundary, and the thing to understand is that crossing it is **not reversible**.

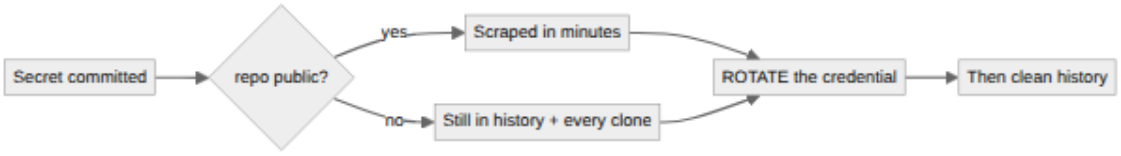
Public repositories are scraped continuously by automation that exists specifically to harvest credentials. A committed cloud key is typically found and used in **minutes, not days**. That's long before a human notices. There is no window in which you quietly delete it and get away with it.

And this is the part people get wrong: **deleting the commit does not remove the secret.** Git keeps history. The blob is still reachable by hash, still in every clone anyone made, still in every fork, and, if the repo was ever public, very likely in a third-party mirror you can't reach. Rewriting history helps future clones and does nothing about the copies already taken.

So the only correct response to a leaked credential is **rotate it**. Treat the key as burned the moment it lands, revoke it, and *then* clean the history if you like. A team that spends the first hour rewriting git history instead of revoking the key has spent the hour the attacker needed.

The controls that actually work sit *before* the push: **push protection**, which blocks a commit containing a credential pattern from ever leaving the developer's machine, and secret scanning across history so you learn about the ones already there. Both are cheap and both beat detection after the fact.

The related control is **code integrity measurement**: a cryptographic hash proving the artefact you shipped is the one that was reviewed and approved, so nothing was swapped between approval and release.



Borrowed code is code you own

Libraries and SDKs let you move fast on code you didn't write. You also inherit every flaw in it. That much is obvious. What's less obvious is that the dependency tree is now an *attack surface in its own right*, targeted deliberately. Three shapes are worth knowing because they're structurally different:

Compromising a real package. In 2018 the widely-used `event-stream` npm package was handed over to a new maintainer who added a dependency carrying obfuscated code that targeted the Copay wallet application. It harvested keys from any account holding more than 100 BTC or 1,000 Bitcoin Cash. Nobody was breached by a bug. The maintainer changed.

Dependency confusion. In 2021 Alex Birsan showed that if your build resolves package names from both a public registry and a private one, publishing a *public* package with the same name as your *internal* one, at a higher version, can make the build fetch the attacker's. It worked against Apple, Microsoft and PayPal, and it needed no vulnerability at all: just how resolution order works.

The long game. The xz-utils backdoor (CVE-2024-3094, 2024) was a multi-year effort in which a contributor built trust, gained maintainer rights, and slipped a backdoor into a compression library in a

path that reached sshd. It was found by an engineer investigating a *half-second* login slowdown. It was very nearly in every major Linux distribution.

The lesson from all three is the same: **your dependency review cannot be a one-time check at adoption**, because the package that was safe when you chose it can change hands afterwards.

What to actually do about it

Pin and use a lockfile. A lockfile records the exact resolved versions and their hashes for the whole tree, including transitive dependencies. Without one, "the same build" isn't the same build.

Verify integrity, not just version. Hashes in the lockfile mean a tampered artefact fails to install rather than silently succeeding.

Know your transitive tree. Nobody is compromised by the twelve dependencies they chose; they're compromised by the eight hundred those twelve pulled in. Software Composition Analysis is how you answer "am I running it?" in minutes rather than three days of grepping.

Configure your resolver explicitly. Dependency confusion is a *configuration* bug. Scope your internal packages and make sure private names cannot be satisfied from a public registry.

Apply the same testing rigour to borrowed code. "A vendor wrote it" is not a security assessment.

The model

You own the risk of every line that runs in your name, whether you wrote it, borrowed it, or accidentally published it.

Repository hygiene keeps your secrets from becoming someone else's. Dependency tracking keeps someone else's bug from becoming your breach. Neither is glamorous. Both are where real incidents actually start: not in clever exploits against code you wrote, but in a key someone pasted and a package someone trusted.

Building Security In: The Secure SDLC and Test Coverage

Fri, Aug 21, 6:00 PM EDT · https://scottslab.io/posts/secure-sdlc-and-test-coverage

TL;DR

Bolt-on security fails because the expensive mistakes are architectural, and no amount of scanning at the end undoes a design decision. Threat modelling is the cheapest security activity there is, because it happens before anything is built. And coverage numbers lie: 100% line coverage only proves every line ran, not that anything was checked. Mutation testing is how you find out whether your tests would actually notice a bug.

"Bolt-on security doesn't work" is one of those phrases that gets repeated until it stops meaning anything. Here's the concrete version: the expensive mistakes are **architectural**, and a scanner at the end of the pipeline cannot undo a decision made in a design meeting six months earlier.

A tool can tell you a query is built by string concatenation. No tool will tell you that you built a system where the tenant identifier arrives from the client and is trusted. The first is a bug. The second is a design, and by the time it's running it's in your data model, your API contracts, and every integration anyone built on top.

Threat modelling: the cheapest security you will ever do

If security has to move earlier, the earliest useful activity is **threat modelling**: sitting down before the code exists and asking what an attacker would do.

The reason to bother is economics. At design time, changing the answer costs a conversation. After launch it costs a migration, a customer comms plan, and probably an incident.

STRIDE is the usual scaffolding: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege, walked over each component and each boundary the data crosses. It isn't magic. Its value is that it forces the question *per trust boundary* rather than in general, and "in general" is where these conversations go to die.

You don't need a ceremony. Draw how data moves, mark where it crosses from something you control to something you don't, and ask the six questions at each crossing. An hour at a whiteboard is worth more than a year of scanning, because it is the only activity on this list that can change the *shape* of the thing.

The mitigations that keep working

Across design, development, testing and deployment, the same small set does most of the work:

Input validation, because most injection is untrusted input that got trusted. Validate on an allowlist at the trust boundary, and remember the boundary is your server. A client-side check is a usability feature, not a control.

Encrypting sensitive data, so a storage breach isn't automatically a data breach.

Least privilege, so a compromised component can't reach past its own job. This is the one that decides how bad an incident gets, and it's decided long before the incident.

Sandboxed development and test environments, so the messy, half-secure state of software under construction can't touch anything real, including production data copied into a test database "just for debugging."

None of these are clever. All of them are the difference between a contained incident and a headline.

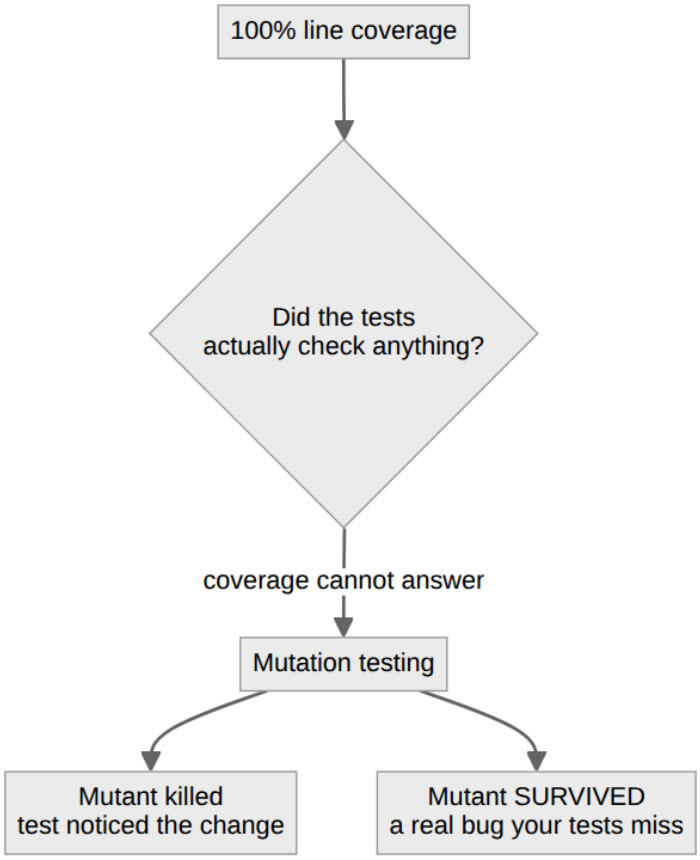
Coverage numbers lie, and here's exactly how

Test coverage analysis tells you how much of your code the tests actually exercised. The levels climb in granularity: **use case**, **function**, **line** (or statement), and **branch**. Branch coverage asks whether *both* outcomes of every decision were taken, which is why it's the honest one. Line coverage counts an `if` as covered when only the true path ever ran.

Now the part that matters. Coverage measures **execution**, not **verification**.

A test can call a function, run every line in it, assert nothing meaningful, and pass. Your report says 100%. Delete the function's entire body and the test may still pass. That number is not lying about what it measures. It's just measuring something much weaker than people believe it measures.

Mutation testing is the tool that closes the gap. It deliberately introduces small faults into your code: flip a comparison, change a boundary, remove a call. Then it re-runs the suite. If the tests still pass, that mutant "survived", and a surviving mutant is a precise, actionable statement: *there is a change to your code that your tests do not notice.* It's slower than coverage and infinitely more informative, and running it once on your most security-sensitive module is genuinely uncomfortable in a useful way.



The target isn't 100% of anything. It's knowing which paths nobody tested, because those are the paths where bugs live. It's also knowing which paths *are* tested but not actually checked, which is the larger and better-hidden category.

Where the maturity models fit

If you want to measure the programme rather than the code, **OWASP SAMM** and **BSIMM** both exist for that, and they answer different questions. SAMM is prescriptive: here are practices, here's what maturity looks like, here's your gap. BSIMM is descriptive: here is what a large sample of real firms actually do, so you can see where you sit against your peers.

Use one when someone asks "are we doing enough?", because that question can't be answered by a scanner. Don't mistake either for security itself. A maturity score describes your process, not your attack surface.

The through-line is the same as everywhere else in this field: the tools find the bugs, and the design decides how bad the bugs get. Spend accordingly.

Testing the Plan: DR Exercises and the After-Action Report

Mon, Aug 24, 10:00 AM EDT · https://scottslab.io/posts/dr-testing-and-after-action-reviews

> **TL;DR**
>
> Five test types climb in rigour and risk, from read-through to full interruption. But the honest problem is that almost every DR test is announced, scheduled in business hours, with the right people present and everything else working. A real disaster is none of those. The failures that actually bite are circular dependencies, DNS TTLs and replication lag, none of which a tabletop will surface. And an after-action item without an owner and a date is a wish.

The uncomfortable truth about disaster recovery plans is that most have never been run, which makes them documentation rather than capability. Testing is how you find out whether the plan survives contact with reality.

The five levels, and what each actually proves

A **read-through** is the lightest: key personnel read the plan and confirm they know their responsibilities. It catches "wait, that's my job?" and not much else. But that's worth catching.

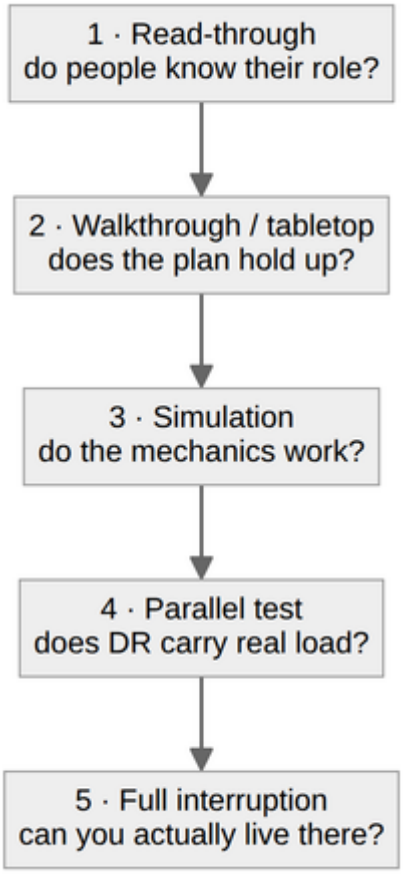
A **walkthrough**, or tabletop, has the team step through the plan together against a scenario a facilitator presents. Nobody touches a system. It surfaces assumptions and gaps a solo read never will, and it is by far the best value per hour spent.

A **simulation** actually activates components (restoring from backup, for instance) without impacting live systems. This is the first level that tests *mechanics* rather than *documents*, and it is where most plans first break.

A **parallel test** stands up the full DR environment alongside primary and runs it for real, without switching production over. High confidence, low blast radius.

A **full interruption** test shuts the primary down and runs operations entirely from DR. The most disruptive and the most honest, which is exactly why it's rare and planned carefully.

NIST's contingency planning guidance (SP 800-34) frames these as tabletop, functional and full-scale exercises, and expects at least annual testing for high-impact systems. Annual is a floor, not a target. A plan tested once a year is a plan that is eleven months stale on average.



The problem with how everyone tests

Here is the thing worth saying out loud. Almost every DR test is **announced in advance, scheduled during business hours, run with the right people available, and performed while everything else is working.**

A real disaster is none of those. It happens at 2am, the person who knows the failover is on a plane, and the thing that broke has taken three other systems with it.

So the failures that actually bite in production are the ones a comfortable test is structurally incapable of finding:

Circular dependencies. The DR plan lives in the wiki that runs on the cluster you're recovering. The credentials are in a password manager behind SSO that depends on the directory that's down. The runbook says "contact the on-call rota" and the rota tool is hosted in the failed region. This is the single most common way a technically-correct DR plan fails on the day. *Print the critical path and store it somewhere that cannot fail with you.*

DNS TTLs. You failed over in eight minutes and customers were down for an hour, because a record had a 3600-second TTL and resolvers everywhere cheerfully kept serving the old address. Nobody discovers this in a tabletop.

Replication lag versus your stated RPO. You promised fifteen minutes. Measure the actual lag under peak write load, not at 3am on a quiet Sunday.

Capacity and licensing on the DR side. DR environments are routinely smaller than production and sized for the diagram rather than the load. Some licences are node-locked and won't start on different hardware.

Recovery order. Restoring the application before the database it depends on wastes the window, and dependency order is exactly what nobody documents.

The fix isn't more testing. It's *less comfortable* testing: run one unannounced, hand it to the second-string team, and disable one thing you assumed would be available.

The after-action report

Every activation ends with one, real disaster or test. That includes the runs that went well, not just the ones that went badly, because the smooth runs teach you what to keep.

A good report has a fixed shape: executive summary; background; the detailed facts (who, what, when, where, why, how); lessons learned; and next steps.

Two things determine whether it's worth writing.

It has to be blameless. The moment a report can end someone's career, people stop volunteering what actually happened, and you lose the only information that mattered. You want the engineer who typed the wrong command to say so within the hour.

Every action item needs an owner and a date. "We should improve backups" is not an action item, it's a mood. Track them where you track real work, and review them before the *next* test. The most reliable finding in this field is that the last report's lessons were never implemented, and the next incident is the same incident.

Communication is part of the test too. Notify internal staff so nobody mistakes the drill for a real outage or the reverse, external partners who depend on you, and regulators where required. A DR test that quietly triggers someone else's incident response has failed regardless of whether the failover worked.

A plan is a hypothesis. A test is the experiment. Run the cheap experiments often, the expensive ones deliberately, and write down what you learned every single time. The next disaster won't wait for you to remember what you meant to fix.

Security Data and Documentation: If It Isn't Recorded, It Didn't Happen

Mon, Aug 24, 6:00 PM EDT · https://scottslab.io/posts/security-data-and-documentation

> **TL;DR**
>
> A security programme runs on two kinds of data: technical logs and the process records proving your controls actually ran. Three things quietly decide whether either survives contact with an auditor or a court: synchronised clocks, because unsynchronised logs cannot be correlated and are trivially challenged; tamper-evident storage, because a log an administrator can edit proves nothing about that administrator; and contemporaneous records, because version history does not lie. Backfilled evidence isn't evidence. It's a confession with extra steps.

Running a security programme means proving it works, and proof is data. It comes in two flavours people tend to conflate.

Technical data is machine exhaust: logs from servers, firewalls, the IPS, access control, pulled together and made sense of in a SIEM. **Process data** is the paperwork: the records showing your processes happened at all, that backups ran, that account reviews were done, that scans went out.

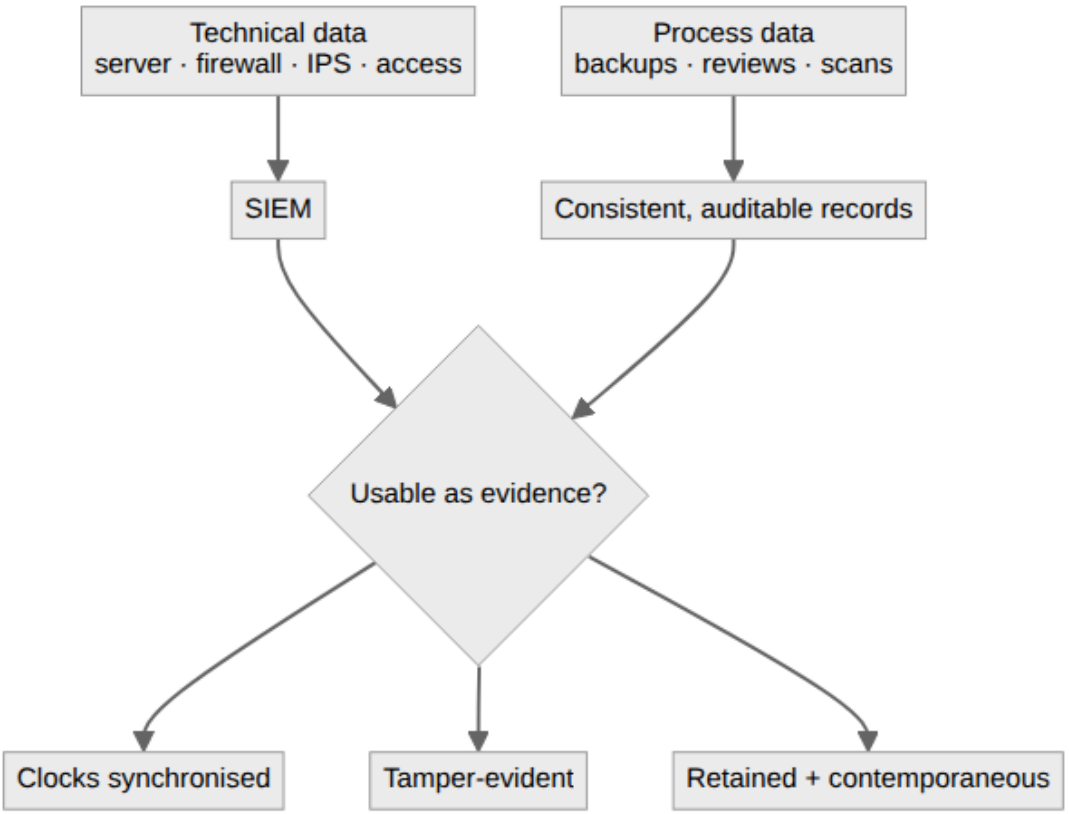
The SIEM tells you what the systems did. The process records tell you what the humans did. Auditors ask for both, and most programmes are far weaker on the second.

Three things that decide whether your logs are worth anything

Synchronised clocks. This is the boring one that ruins investigations. Correlating an event across a firewall, a server and an identity provider requires their timestamps to mean the same thing. If one host's clock drifts by four minutes, your timeline is wrong in a way that looks *plausible*. That's worse than obviously broken, because you'll believe it. Run NTP everywhere, log in UTC, monitor for drift, and record the timezone explicitly. In any proceeding, unsynchronised timestamps are the first thing a competent opponent attacks, and it works.

Tamper-evident storage. A log file an administrator can edit tells you nothing about what that administrator did. If logs matter as evidence, they need write-once storage, append-only retention, or hash-chaining so alteration is detectable, and they need to leave the originating host promptly, because the first thing a capable intruder does is clean up after themselves. Centralised logging isn't just for convenience; it means the attacker has to compromise two systems to erase their tracks instead of one.

Retention that matches your obligations. Know the number for your regime rather than guessing. PCI DSS, for instance, requires audit log history be retained for **at least 12 months, with the most recent three months immediately available for analysis**. "Immediately available" is doing real work in that sentence, because a year of logs in cold storage you can't query in an afternoon does not satisfy it.



What an auditor is actually testing

This is the part that surprises people the first time. An auditor is generally not checking whether a control *is* in place today. They're checking whether it **operated throughout the period**. That difference decides how much of your year you spend on the audit.

So they ask for a **population** and then sample it. If the control is quarterly access reviews, the population is all four reviews for the year, and you must be able to produce the complete list before they pick which to examine. A programme that can produce three of four has not passed three-quarters of the test; it has demonstrated the control does not reliably run.

Which means the useful question when designing any control is not "can we do this?" but **"what artefact will this produce, automatically, every time it runs?"** A control that leaves no trace is unauditable no matter how diligently you perform it.

Whatever holds the process data, a GRC platform or, honestly, a well-kept spreadsheet, needs to be consistent and auditable. The tooling matters less than the discipline. What matters is that records are complete, follow a consistent format, and can be handed over without a translation layer.

Contemporaneous, or it isn't real

The part that trips people up is recording things **when they happen**, not reconstructing them the night before the audit.

Modern tools quietly enforce this. A shared spreadsheet keeps version history, so "we've been reviewing accounts quarterly" collapses instantly when the history shows no edits between March 2018 and December 2020. The same is true of a document created three days before fieldwork, or twelve months of entries in identical handwriting and one ink.

Backfilled records aren't evidence. They're a confession with extra steps. Unlike a missing record, which is a finding, a fabricated one is a different category of problem entirely.

The habit is boring and non-negotiable: record the control as you perform it, in something that timestamps you honestly. Because a programme you can't prove you ran is, to an auditor, a programme you didn't run. To a court, it's worse than that.

Disaster Recovery Fundamentals: RTO, RPO, and Somewhere Else to Run

Wed, Aug 26, 10:00 AM EDT · https://scottslab.io/posts/disaster-recovery-fundamentals

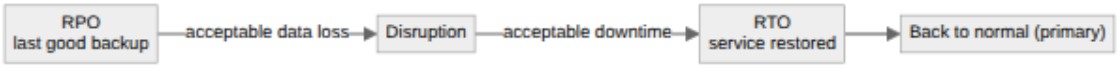
TL;DR

Disaster recovery is the subset of business continuity that gets normal operations back after a disruption: a hurricane, ransomware, or a data-center outage. It isn't "done" until you're fully back in the primary environment. Three numbers frame it: RTO (how fast you must restore a service), RPO (how much data loss you can accept), and RSL (the minimum service level to hold during the disaster). And you need somewhere to run: a hot, warm, or cold site, trading cost against activation time.

Disaster recovery is a subset of business continuity. Where continuity is keeping the business alive through a disruption, DR is the specific job of restoring normal operations after one. The trigger can be natural (a hurricane takes the building), man-made (ransomware encrypts everything), or internal (the data center just goes dark). And here's the part people declare too early: DR concludes only when you're fully back to normal in the primary environment, not when you've limped onto a backup. Running on the spare tire isn't "recovered."

Three metrics do the framing, and every DR conversation should start with them. RTO (Recovery Time Objective) is the target time to get a service back after it goes down; it answers "how long can this be offline?" RPO (Recovery Point Objective) is the maximum acceptable data-loss window; it answers

"how much recent work can we afford to lose?" and it's really a statement about how often you back up. And RSL (Recovery Service Level) is the minimum percentage of a service that must stay available during the disaster, because "degraded but up" is often the real goal, not all-or-nothing.

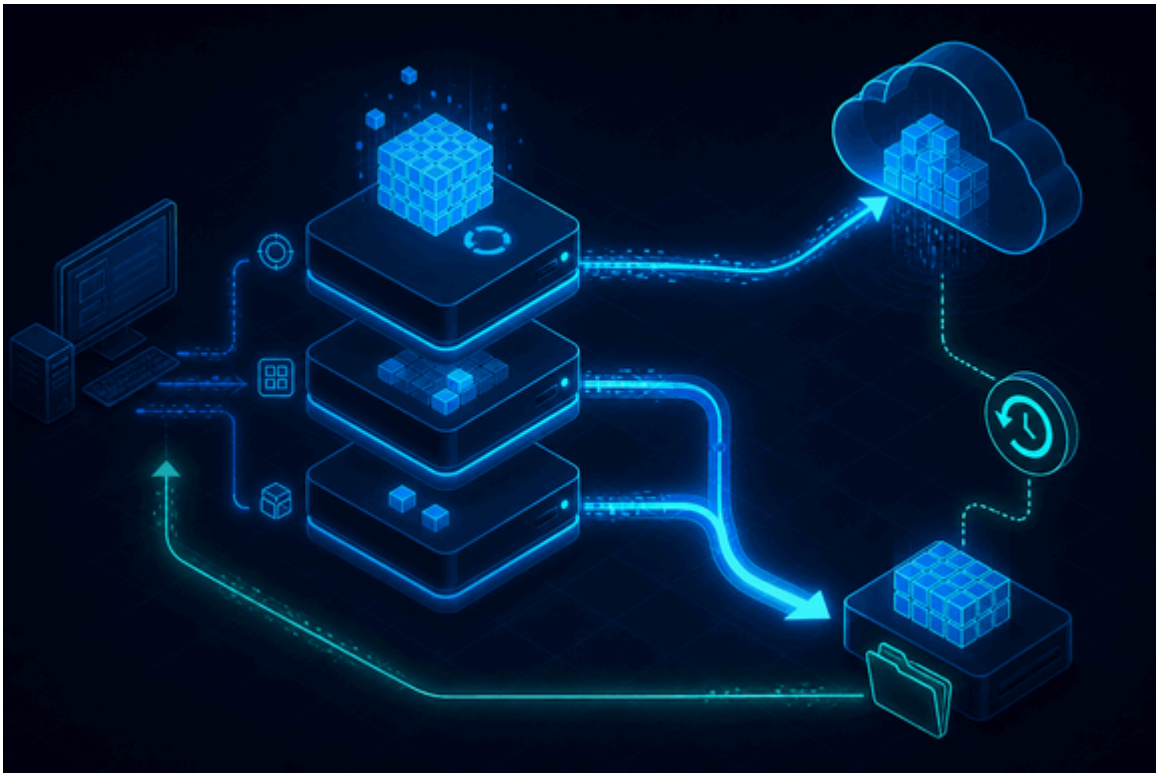


Then you need somewhere else to run, and the three options are a straight cost-versus-speed trade. A hot site is a fully operational mirror of your primary data center. It activates immediately, and it costs the most because you're paying to run a second everything. A warm site has the hardware racked and ready but not running in parallel, so activation takes hours to days while you load data and bring it up. A cold site is an empty shell of power, cooling, cabling, and floor space: cheap to hold but weeks to months to turn into a working environment. Which one you pick falls straight out of your RTO: a two-hour RTO cannot be met from a cold site, full stop.

One modern piece ties it together: Resource Capacity Agreements, contracts with providers that guarantee you'll actually get the compute, storage, and networking you need during a disaster. It's the unglamorous detail that saves you, because a disaster is exactly when everyone else is also scrambling for capacity, and "we assumed the cloud would have room" is not a recovery plan.

Backup Strategies: Full, Differential, Incremental — and Actually Getting Data Back

Wed, Aug 26, 6:00 PM EDT · https://scottslab.io/posts/backup-strategies-full-differential-incremental

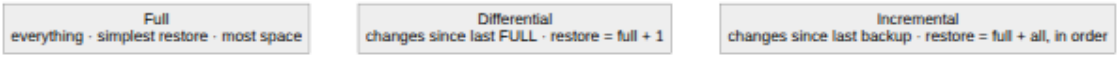


TL;DR

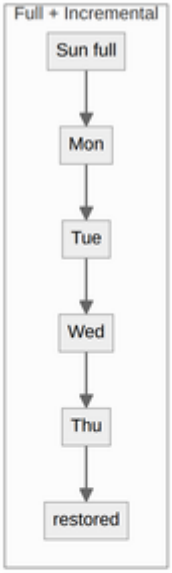
Three backup types trade space against restore effort. Full backups copy everything and are the simplest to restore. Differential backups capture all changes since the last full, so restoring means the full plus one differential. Incremental backups capture only the changes since the last backup of any kind: smallest to write, but restoring means replaying the full plus every incremental in order. The most common reason you restore isn't a disaster, it's human or technical error. Keep at least one copy offsite, and remember a backup you've never restored from is a hope, not a backup.

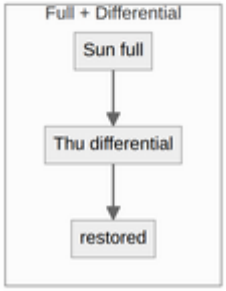
There are three primary backup types, and the whole choice is a trade between how much space they eat and how much work a restore takes.

A full backup is a complete copy of all the data. It's the simplest to restore because everything's in one place, and the most expensive in time and space to make. Snapshots are a hardware-platform variant of the same idea. A differential backup captures all changes since the last full backup; it grows each day as changes accumulate, but restoring is easy: the last full plus one differential. An incremental backup captures only the changes since the last backup of any kind, full or incremental. It's the smallest and fastest to write, but the restore is the most work, because you replay the full plus every incremental since, in order.



Make it concrete with a Sunday-full schedule. With differentials, to recover on Friday you restore Sunday's full, then Thursday's differential. Two steps, done. With incrementals, you restore Sunday's full, then apply Monday, Tuesday, Wednesday, and Thursday's incrementals in order. Five steps, and if any one is missing or corrupt, the chain breaks. That's the trade in one example: differential spends storage for a faster, more robust restore; incremental saves storage at the cost of a longer, more fragile one.





A few things the metrics never tell you. The most common reason you'll actually restore is boring: human or technical error, someone deleting the wrong thing. Not a hurricane. Optimize for the routine restore, not just the apocalypse. Keep at least one copy offsite (cloud counts); if you're cloud-only, consider two different providers, because "all my backups are in one account" is a single point of failure wearing a redundancy costume. Online backups restore instantly but cost more; offline backups are cheaper but need someone to go get them. Two patterns worth knowing. The non-persistence approach backs up only each server's unique data and rebuilds the OS from infrastructure-as-code, so you never

back up what you can regenerate. Live boot media, a bootable USB, lets you pull data off a machine's storage without trusting, or even starting, its own OS.

The one rule under all of it: a backup you have never restored from is not a backup, it's a guess. Test the restore, or you don't have one.

Watching the Watchers: Management Reviews and Privileged Access

Fri, Aug 28, 10:00 AM EDT · https://scottslab.io/posts/management-reviews-and-privileged-access

TL;DR

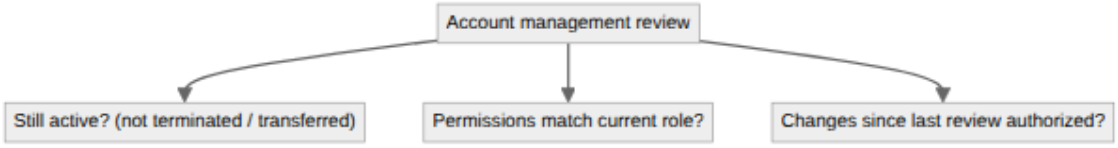
Management reviews do two jobs: catch errors in people's work, and deter fraud by making oversight normal. The sharpest scrutiny goes to privileged actions: admins and engineers who can override the controls everyone else obeys. Their exceptions should ride a formal RFC with VP approval, with a manager reviewing the logs (100% for the highest-risk, or a random sample). And account reviews confirm three things: the person still works here, their permissions match their current role, and every change since last time was authorized.

Management review is one of those controls that sounds like bureaucracy and is actually load-bearing. It does two things at once. First, it's a correctness check: a second set of eyes confirming an employee's work is accurate and complete. Second, and less obvious, it deters fraud simply by existing: when people know their work gets reviewed, the culture shifts, and most would-be shortcuts never get taken. The review that never catches anything is still working. It's why there was nothing to catch.

The scrutiny should scale with power, and nobody has more power than the privileged users. System engineers and admins can override the very controls everyone else is bound by, which makes their actions the highest-risk thing happening on your network. So privileged exceptions shouldn't happen casually. They ride a formal RFC (request for change) with VP-level approval, so there's a paper trail

and a senior name attached. And a manager reviews the privileged-action logs afterward: for the highest-risk systems, that means verifying 100% of privileged actions; for lower-risk ones, a random sample is enough to keep everyone honest without drowning the reviewer.

The other recurring review is over accounts themselves, and it's a checklist worth memorizing because it maps exactly to how access goes wrong. Is the user still active: not terminated, not transferred out? Do their permissions match their *current* role, or are they carrying access from a job they left two moves ago (the privilege creep problem)? And were the changes since the last review, every escalation, creation, and revocation, actually authorized? Done with the manager who owns those people, this is the control that catches the orphaned admin account and the quietly over-privileged veteran before an attacker does.



The theme under all of it: trust is fine, but unverified trust in the people who can override your controls is just hope. Review scales with privilege, and the highest privilege gets the closest look.

KPIs vs. KRIs: Measuring the Program You Have and the Risk You're Taking

Fri, Aug 28, 6:00 PM EDT · https://scottslab.io/posts/kpis-and-kris-security-metrics

> **TL;DR**
>
> KPIs look backward: are we hitting our security goals? (ITIL lists example KPIs: % decrease in breaches, number of major incidents, time from spotting a threat to shipping a control.) KRIs look forward: what risk could jeopardize us next? (ISACA says pick them by business impact, implementation effort, reliability, and sensitivity.) The rule that makes either one honest: define the metric before you measure, so nobody cherry-picks the number that flatters them.

Security metrics come in two directions, and mixing them up is how leadership ends up reassured about the wrong thing.

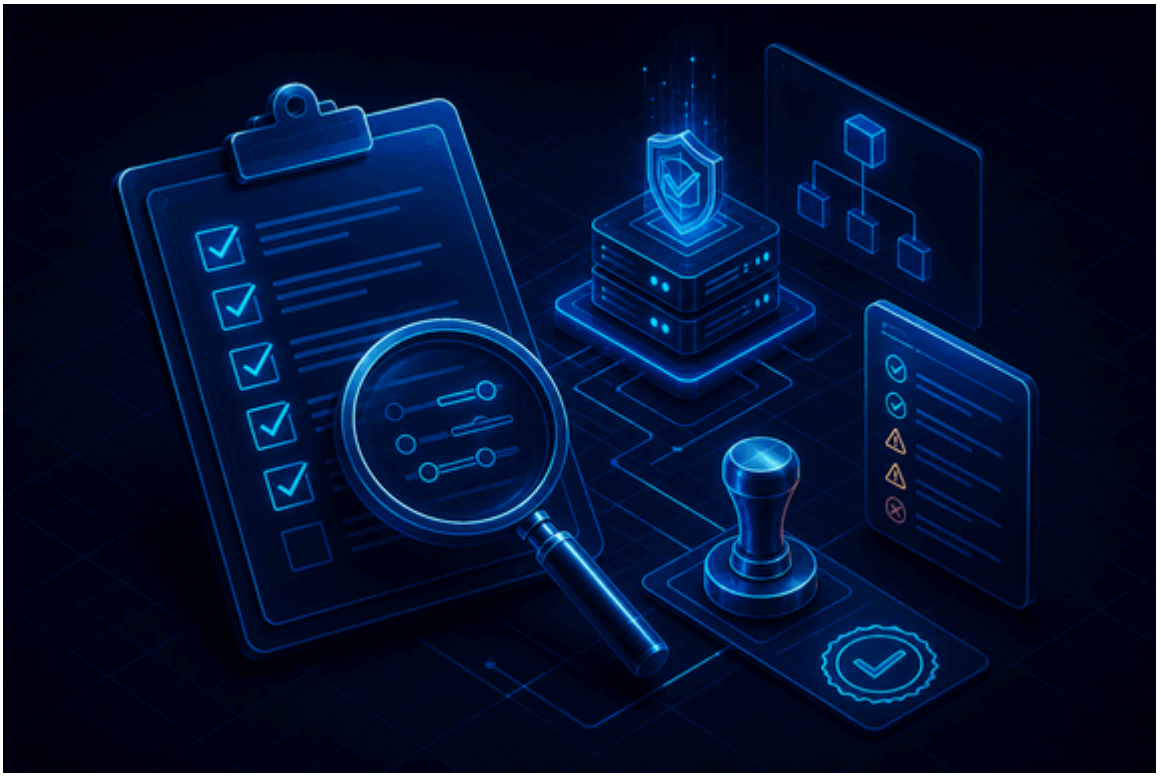
Key Performance Indicators look backward. They measure whether the security program actually met the goals you set: a report card on what already happened. ITIL offers a set of example KPIs worth stealing, among them the percentage decrease in breaches, the number of major incidents, and the time between identifying a threat and getting a control in place for it. That last one is my favorite, because it measures the thing that actually determines your exposure: not whether you eventually fixed it, but how long you sat in the window before you did.

Key Risk Indicators look forward. Instead of grading past performance, they quantify risk that could jeopardize your security going forward: early-warning gauges rather than a rear-view mirror. Because you could track an infinite number of them, ISACA gives you selection criteria so you pick the ones that earn their place: business impact (does this risk actually matter?), implementation effort (can we realistically measure it?), reliability (is the signal trustworthy?), and sensitivity (does it move early enough to warn you?). A KRI that only spikes after you're already breached is just a slow KPI.

The rule that keeps both honest is the same, and it's the whole point: define your metrics in advance. If you decide what "good" looks like *after* the numbers are in, you'll unconsciously (or consciously) pick the framing that makes the program look best. That's cherry-picking dressed up as reporting. Metrics chosen before the measurement are accountability; metrics chosen after are marketing. Write down what you'll track and what target counts as success, then let the numbers land where they land.

Audits vs. Assessments: Who's Asking, and How Formal It Gets

Mon, Aug 31, 10:00 AM EDT · https://scottslab.io/posts/audits-and-assessments



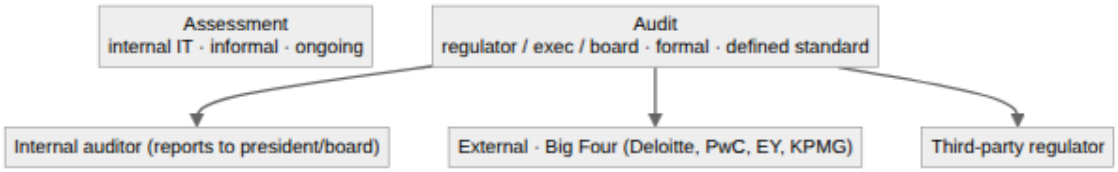
TL;DR

An assessment is the informal, ongoing self-check your own IT team runs. An audit is the formal one: demanded by a regulator, executive, or board, against a defined standard, by one of three auditor types (internal, external "Big Four," or a third-party regulator). Define the scope upfront (on-prem, cloud, hybrid) so nobody's surprised, and expect two deliverables: a gap analysis (your remediation roadmap) and an attestation (the formal "controls are adequate and working").

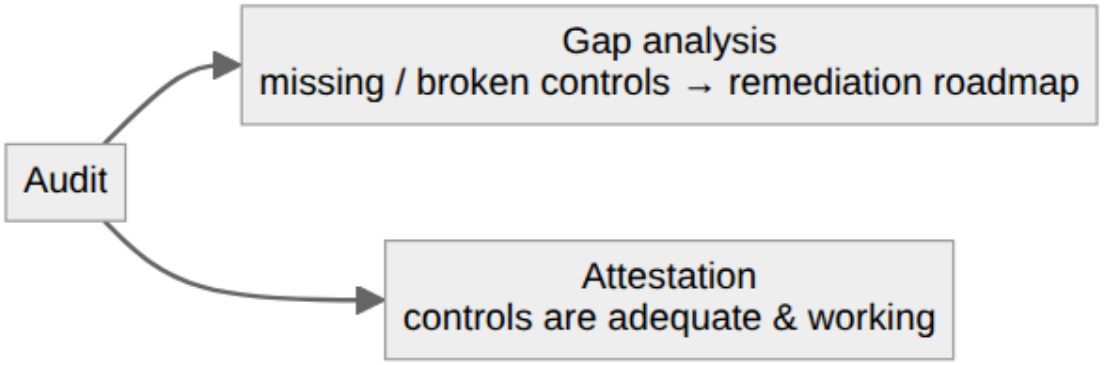
The words get used interchangeably and they shouldn't be, because the difference is who's asking and how much it binds you. An assessment is requested by your own internal IT staff. It's less formal, and it's an ongoing evaluation tool you run on yourselves to find problems before anyone official does. An audit is a formal examination requested by an outside authority: a regulator, an executive, the board. It follows a defined standard, not your own judgment. Assessments are how you stay ready; audits are how you get graded.

Audits come from three types of auditor, and the independence increases as you go down the list. Internal auditors work for the organization but report up to the president or board (not to IT), so they can call it straight. External auditors are the independent firms: the "Big Four" being Deloitte, PwC, EY, and

KPMG. And third-party regulators audit you against the rules they enforce. The pattern is deliberate: the more the result matters to outsiders, the more independent the person signing it needs to be.



Real audit standards are far more concrete than "are you secure?" PCI DSS ships a roughly 360-page audit procedures document, where each requirement carries specific test procedures the auditor must perform. Requirement 3.5.1 (rendering card numbers unreadable) alone breaks into three distinct steps the auditor walks through. Which is why scope has to be nailed down upfront: on-premises, cloud, hybrid, or all three. Defining scope before the audit starts prevents nasty surprises for everyone. The auditor doesn't wander into systems nobody meant to include, and you don't get graded on infrastructure that was never in play.



Two deliverables come out the other end, and you want to know which you're getting. A gap analysis lists the controls that are missing or not working properly. It's unglamorous and it's the most useful thing an audit produces, because it's a ready-made remediation roadmap. An attestation is the formal statement that your controls are adequate and functioning: the clean bill of health you can hand to a customer or regulator. Aim for the attestation; treasure the gap analysis, because it's the one that actually makes you better.

Control Testing, Exceptions, and Compensating Controls

Mon, Aug 31, 6:00 PM EDT · https://scottslab.io/posts/control-management-and-exceptions

TL;DR

Periodic audits aren't enough; supplement them with routine control testing: automated checks that catch, say, an unexpected firewall port change the day it happens, not at the next audit. When a control genuinely can't be met, the exception has to be written down (justification, risks, residual risk) and approved by a defined authority. It doesn't get waved through in a hallway. And a compensating control has to earn it: PCI's bar is that it meets the intent and rigor of the original requirement, provides similar defense, and isn't already being counted for something else.

An audit is a snapshot, and snapshots miss everything that happens between them. That's why routine control testing has to supplement periodic audits: continuous or automated checks that flag drift as it happens. The classic example is an automated check for unexpected firewall port changes: an audit would catch that open port next quarter; a routine test catches it the afternoon someone opens it. Audits prove the program on a schedule; routine testing keeps it honest in between.

Sometimes a control genuinely can't be met: a legacy system that can't do the required encryption, a business process that breaks under the rule. That's allowed, but only through a real exception process, and the two requirements are what keep "we made an exception" from becoming "we ignored it." First, formal written documentation: something like Washington State's exception request form, which forces you to spell out the justification, the risks, and the residual risk you're accepting. Second, a defined approval authority: a specific role empowered to accept that risk on the organization's behalf. An exception nobody signed and nobody wrote down isn't an exception, it's a gap you've decided not to look at.

When an exception is granted, you don't just accept the naked risk. You put a compensating control in its place, an alternative that covers what the original requirement would have. But a compensating control has to genuinely stand in, not just look like effort, and PCI DSS spells out the bar cleanly: it must meet the intent and rigor of the original requirement, provide a similar level of defense, and not already be counted toward some other requirement (no double-dipping one control across two obligations). Meet that bar and the exception is defensible; miss it and you've just documented, in writing, that you knew about the gap and left it open. That's worse than not documenting at all.