

Log Analysis & Pattern Discovery

WARNING — Authorization Required: Only analyze logs from systems you own or have explicit written permission to access. Unauthorized access to log files is a criminal offense under laws such as the CFAA (US), CMA (UK), and equivalent statutes worldwide. In CTF environments, you are working within a sanctioned scope — never carry techniques outside that boundary without authorization.

Why Attackers Analyze Logs

Logs are the nervous system of any infrastructure. Every authentication attempt, every HTTP request, every service restart leaves a trace. For a red teamer or CTF competitor, logs are a goldmine of intelligence. Understanding *why* attackers care about logs is the first step toward thinking like one.

Finding credentials and secrets. Developers routinely make mistakes: logging API keys in debug output, recording full query strings that include tokens, or writing session IDs into access logs. A single `grep` through an exposed log file can yield database passwords, bearer tokens, or SSH keys that were accidentally dumped during an error trace.

Understanding defenses. Logs reveal what a system monitors. If you see structured JSON logs flowing into an ELK stack, you know the blue team has centralized logging. If `/var/log/auth.log` shows entries from `fail2ban`, you know brute-force attempts are being rate-limited. This intelligence shapes your attack strategy — you might throttle login attempts, rotate source IPs, or target a service that lacks monitoring.

Identifying targets. Application logs expose internal architecture. A web server log that references `/api/v2/admin/users` tells you there is an admin API. A syslog entry mentioning `mysql-backup.service` confirms MySQL is in play and hints at backup schedules. Error logs that reference internal hostnames like `db-prod-03.internal` map out the network without a single port scan.

Log Types You Will Encounter

Different systems produce different log formats. Knowing where to look and what each log contains is essential.

Web Server Logs (Apache / Nginx)

Found at `/var/log/apache2/access.log` or `/var/log/nginx/access.log`. These record every HTTP request: client IP, timestamp, request method and URI, status code, response size, referrer, and user agent. They are the single most common log type in web-focused CTFs.

Authentication Logs

On Linux: `/var/log/auth.log` (Debian/Ubuntu) or `/var/log/secure` (RHEL/CentOS). These capture SSH logins, `sudo` usage, PAM events, and user account changes. Failed login attempts are especially telling.

Syslog

The general-purpose system log at `/var/log/syslog` or `/var/log/messages`. Services that lack their own log files write here via the syslog daemon. Cron job execution, kernel messages, and service start/stop events all appear in syslog.

Windows Event Logs

Stored in `.evtx` format under `C:\Windows\System32\winevt\Logs\`. The key channels are **Security** (Event IDs 4624/4625 for logon success/failure), **System** (service events), and **Application**. Use `wevtutil` or PowerShell's `Get-WinEvent` to query them.

Application Logs

Custom application logs vary wildly. Django writes to its configured handler, Java apps emit log4j output, Node.js apps might log JSON to stdout. The format depends entirely on the developer's choices, but the intelligence value can be enormous — stack traces, database queries, internal user IDs, and session tokens all appear here.

Log Format Anatomy

Before you can parse logs, you need to understand their structure.

Common Log Format (CLF)

```
192.168.1.105 - frank [10/Oct/2025:13:55:36 -0700] "GET /index.html HTTP/1.1" 200 1234
```

Fields: `host ident authuser [date] "request" status bytes`. The `ident` field is almost always `-`. The `authuser` field is populated when HTTP basic auth is used.

Combined Log Format

```
192.168.1.105 - frank [10/Oct/2025:13:55:36 -0700] "GET /dashboard HTTP/1.1" 200 1234 "https://example.com" Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.4431.51 Safari/537.36
```

Extends CLF with **referrer** and **user agent**. The referrer reveals navigation flow; the user agent identifies the client software (or an attacker's tool).

JSON Structured Logs

```
{
  "timestamp": "2025-10-10T20:55:36.042Z",
  "level": "ERROR",
  "service": "auth-api",
  "message": "Login failed for user admin",
  "client_ip": "10.0.14.88",
  "attempt_count": 47,
  "user_agent": "python-requests/2.31.0"
}
```

Modern applications increasingly use structured JSON logs. These are easier to parse programmatically but also easier for attackers to query with tools like `jq`.

Pattern Discovery

Raw logs are noise. Pattern discovery turns noise into signal. Here are the patterns that matter most in offensive security.

Failed Login Bursts

A sudden spike in authentication failures indicates brute-force activity — either yours or someone else's. In CTFs, discovering another team's brute-force logs can reveal valid

usernames.

```
Oct 10 14:22:01 target sshd[4412]: Failed password for admin from 10.0.14.88 p
Oct 10 14:22:01 target sshd[4414]: Failed password for admin from 10.0.14.88 p
Oct 10 14:22:02 target sshd[4416]: Failed password for admin from 10.0.14.88 p
Oct 10 14:22:02 target sshd[4418]: Failed password for admin from 10.0.14.88 p
Oct 10 14:22:03 target sshd[4420]: Accepted password for admin from 10.0.14.88
```

Note the final line: the brute force *succeeded*. The attacker is now inside.

Directory Traversal Attempts

```
10.0.14.88 - - [10/Oct/2025:14:30:12 -0700] "GET /files?path=../../../../etc/pass
10.0.14.88 - - [10/Oct/2025:14:30:14 -0700] "GET /files?path=../../../../etc/shad
```

The `../../../../` sequences are the signature. A 200 response on the `/etc/passwd` request means the traversal worked — the application is vulnerable.

SQL Injection Signatures

```
10.0.14.88 - - [10/Oct/2025:14:35:07 -0700] "GET /search?q=test'+OR+1%3D1--+
10.0.14.88 - - [10/Oct/2025:14:35:09 -0700] "GET /search?q=test'+UNION+SELECT+
```

URL-encoded SQL keywords (`UNION`, `SELECT`, `OR 1=1`) in query parameters are unmistakable injection attempts. The 200 status with a larger response body suggests data was exfiltrated.

Unusual User Agents

```
"sqlmap/1.7.2#stable (https://sqlmap.org)"
"Nikto/2.5.0"
"gobuster/3.6"
"python-requests/2.31.0"
"curl/8.4.0"
```

Default tool user agents are immediate red flags. Skilled attackers spoof these, but in CTFs and real-world incidents, they frequently appear unmodified.

Command-Line Log Parsing

Fast, effective log analysis starts at the command line. These tools are available on virtually every Linux system.

grep — Filter Lines

```
# Find all failed SSH logins
grep "Failed password" /var/log/auth.log

# Find requests from a specific IP
grep "10.0.14.88" /var/log/nginx/access.log

# Case-insensitive search for SQL injection keywords
grep -iE "(union|select|insert|drop|delete|update).*--" access.log
```

awk — Extract Fields

```
# Count requests per IP address (top 10)
awk '{print $1}' access.log | sort | uniq -c | sort -rn | head -10

# Extract all requested URLs
awk '{print $7}' access.log | sort -u

# Show only 4xx and 5xx responses
awk '$9 ~ /^[45]/' access.log
```

sed — Transform and Clean

```
# Strip timestamps for cleaner output
sed 's/\[.*\]//g' access.log

# Extract just the date portion from Combined Log Format
sed -n 's/.*\[([^\:]*\):.*\1/p' access.log | sort | uniq -c
```

jq — Parse JSON Logs

```
# Extract all error-level messages
cat app.log | jq 'select(.level == "ERROR") | .message'

# Find login failures with high attempt counts
cat app.log | jq 'select(.message | test("Login failed")) | select(.attempt_co
```

```
# Build a timeline of events from a specific IP
cat app.log | jq 'select(.client_ip == "10.0.14.88") | {time: .timestamp, msg:
```

Python Log Parsing

When command-line one-liners are not enough, Python provides the flexibility you need for deeper analysis.

```
import re
from collections import Counter

# Parse Apache Combined Log Format
LOG_PATTERN = re.compile(
    r'(?P<ip>\S+) \S+ \S+ '
    r'\[(?P<timestamp>[^\]]+)\] '
    r'"(?P<method>\S+) (?P<path>\S+) \S+" '
    r'(?P<status>\d{3}) (?P<size>\S+)'
)

failed_logins = Counter()
suspicious_paths = []

with open("access.log") as f:
    for line in f:
        m = LOG_PATTERN.match(line)
        if not m:
            continue

        ip = m.group("ip")
        path = m.group("path")
        status = int(m.group("status"))

        # Track IPs generating 401/403 responses
        if status in (401, 403):
```

For JSON logs, Python's `json` module makes analysis straightforward:

```
import json
from datetime import datetime

events = []
with open("app.log") as f:
```

```
for line in f:
    entry = json.loads(line.strip())
    if entry.get("level") == "ERROR":
        events.append(entry)

# Sort by timestamp and print timeline
events.sort(key=lambda e: e["timestamp"])
for e in events:
    ts = e["timestamp"][:19]
    print(f"[{ts}] {e['service']:<12} {e['message']}")
```

AI-Assisted Log Analysis

Large language models like Claude excel at log analysis tasks that are tedious or ambiguous for rule-based tools.

Anomaly detection. Paste a block of log entries and ask Claude to identify anything unusual. The model can spot subtle patterns: a single successful login sandwiched between hundreds of failures, a user agent that switches mid-session, or a request sequence that follows a known exploit chain.

Pattern extraction. Given raw, unstructured logs from an unfamiliar application, Claude can identify the format, extract fields, and even write a parser for you. This is especially valuable for proprietary application logs where no documentation exists.

Timeline reconstruction. In incident response and CTF forensics, you often have logs from multiple sources — web server, auth, application, firewall. Claude can correlate events across sources, merge them into a unified timeline, and narrate the attack sequence in plain language.

Prompt example:

```
"Here are auth.log and access.log entries from 14:00–15:00 on Oct 10.
Correlate events by IP address and reconstruct what the attacker did,
step by step. Identify which attack succeeded and what was accessed."
```

Writing parsers on the fly. Rather than spending twenty minutes crafting a regex for an unfamiliar log format, describe a few sample lines to Claude and ask for a Python parser. You get working code in seconds that you can refine as needed.

Practical tip: When feeding logs to an LLM, strip or redact sensitive information first (real IP addresses, actual credentials, PII). In CTF contexts this is less of a concern, but building the habit now prevents mistakes in real engagements.

Key Takeaways

- Logs reveal credentials, architecture, defenses, and attack history — they are a primary intelligence source in any engagement.
- Know your log types: web access logs, auth logs, syslog, Windows Event Logs, and application-specific logs each offer different intelligence.
- Master the formats: CLF, Combined, and JSON structured logs each require different parsing approaches.
- Pattern recognition — failed login bursts, traversal attempts, injection signatures, and anomalous user agents — separates useful signal from noise.
- Command-line tools (`grep`, `awk`, `sed`, `jq`) handle the vast majority of log parsing tasks quickly.
- Python fills the gap for complex, multi-source, or stateful analysis.
- AI assistants like Claude accelerate anomaly detection, format identification, and timeline reconstruction.

*Next lesson: **Lesson 03** — we move from reading logs to actively probing systems with network reconnaissance techniques.*

Challenge Resources

Download Files

These log files contain synthetic attack patterns for analysis. All files are plain text and safe to open.

↓ [apache-access.log.txt](#) ~8 KB

↓ [auth-log.txt](#) ~4 KB

↓ windows-events.txt ~3 KB

↓ app-debug.log.txt ~2 KB