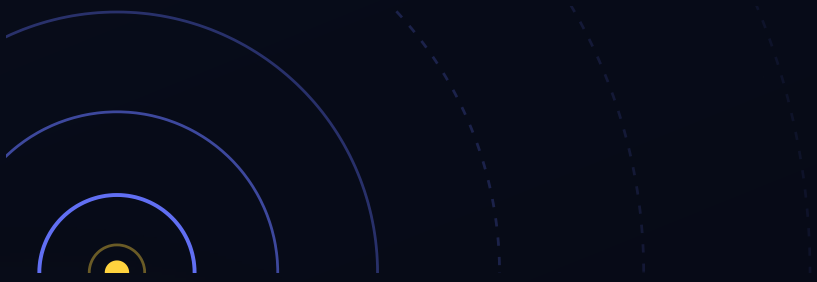
Cremit

CREMIT RESEARCH · INCIDENT RESPONSE SERIES

Blast Radius

A field playbook for investigating leaked API keys and cloud credentials across AWS, GCP, Azure and the secret stores behind them, and for proving how far they could reach.



01 One key

02 One identity

03 Every role it can assume

04 Everything those roles reach: the blast radius

PUBLISHED 4 September 2026

AUDIENCE Incident responders, DevSecOps, platform & cloud security teams

CREMIT.IO

START HERE

If a key just leaked, this page is the only one that matters.

Everything past this page is reference material you consult during and after the incident. **This page is what you act on in the first five minutes.**

1. **Preserve evidence before you contain.**
   Screenshot or export the exposure and record the exact discovery time in UTC. Revocation destroys the evidence that tells you what the key could reach and whether anyone used it.
2. **Establish the owning principal without authenticating.**
   The prefix and structure already tell you the provider, the credential class, and often the owning account, all of it offline.
3. **Assume it was used.**
   A key exposed on the public internet can be exploited within **one to five minutes**. "We found no evidence of use" is an unfinished investigation, not an all-clear. In two of the three major clouds, secret reads are not even logged by default.

⚠ COMMON MISTAKES WHILE PRESERVING EVIDENCE

Deleting the key, force-pushing the commit away, or rotating the secret before you have captured scope. Each erases part of the record, and none shortens the attacker's head start.

ⓘ REPORTING DEADLINES RUN FROM AWARENESS

Most compliance regimes require notification within a fixed time of the moment you become aware, not the moment you understand. Under NIS2 the first filing is due in 24 hours.

✓ COLLECTING EVIDENCE RIGHT NOW?

Go straight to **Appendix A** on pages 28–29: every log source worth pulling, ordered by how fast it decays, plus a collection script that seals its output into one hashed case folder.

The six phases this playbook is organised around

Every chapter maps to one phase. The running header tells you where you are.

01	02	03	04	05	06
Identify	**Scope exposure**	**Map blast radius**	**Investigate use**	**Contain & rotate**	**Prove closure**
What kind of credential is this, and whose is it?	Where did it leak, and for exactly how long?	Everything this credential could reach or become.	Did anyone use it, and what did they do?	Revoke in an order that does not break production.	Evidence that it is actually over.

Contents

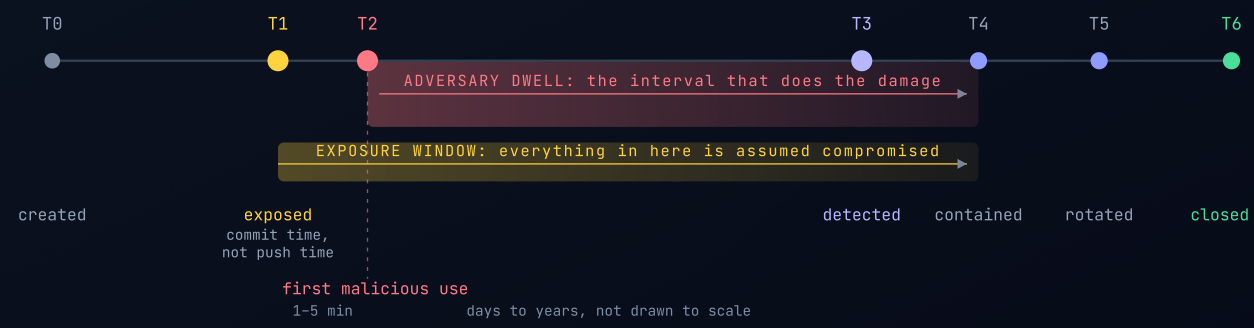
WHY RESPONSE ORDER MATTERS MORE THAN RESPONSE SPEED

Three clocks start at once, and they run at different speeds.

The attacker's clock is measured in **minutes**. Your regulatory clock is measured in **hours**. Your detection clock is, empirically, often measured in **years**. A playbook that optimises only for revocation speed loses on all three.

FIGURE 1 · THE INCIDENT CLOCK

T0 → T6 Cremit



The interval you control is T1→T3, and you control it only with detection. T4→T6 is process discipline; T1→T2 is not yours to influence. Optimising T3→T4 while ignoring T1→T3 is the most common failure in this class of incident.

How fast T1 → T2 actually is

1 min
Fastest observed abuse of an AWS key planted in a public repo (Comparitech honeypot).

4 min
From AWS applying its quarantine policy to attacker reconnaissance beginning (Unit 42).

400+
API calls issued by that actor within a seven-minute window.

AWS's own automation applied `AWSCompromisedKeyQuarantine` within two minutes of the key hitting GitHub. The attacker still got in. Automated exploitation outruns automated defence, which is why assuming use is the correct default rather than a pessimistic one.

THE REGULATORY CLOCK

REGIME	DEADLINE
GDPR Art. 33	`72 h from awareness`
EU NIS2 Art. 23	`24 h early warning, 72 h notification, 1 month final report`
US SEC (public cos.)	`Item 1.05 8-K, 4 business days from materiality`
PCI DSS	`Immediate, per brand rules`

NIS2 is the sharpest of these: the early warning is due within **24 hours of becoming aware**, long before you know the blast radius. The first filing is made from an unfinished investigation, by design.

The exact start of “awareness”

Every deadline above runs from a determination you make yourself: awareness for most, materiality for the SEC. None of the regimes defines it as “when the security team got round to the ticket”. A position taken before it is needed is the one that holds up under time pressure.

THE DEFENSIBLE READING

Awareness begins when **anyone at the organisation** has enough information to conclude a breach is possible. A scanner alert delivered to a developer's inbox is the organisation being told.

WHY THIS IS NOT ACADEMIC

In the xAI case the notification landed on **2 March** and the key was still live on **30 April**. On the reading above, the clock had been running for eight weeks before the security team was involved at all.

WHAT TO DO ABOUT IT

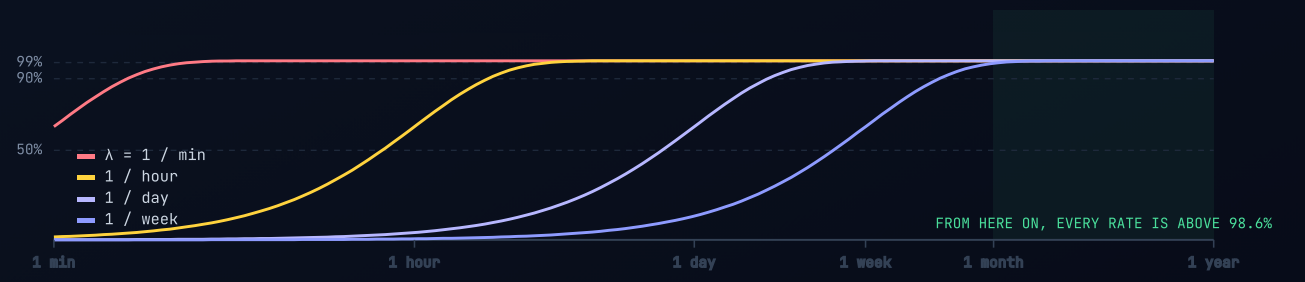
Route secret findings to a team with a duty to act, not to the person who committed the secret. That single change moves the start of your regulatory clock and your detection clock to the same moment.

TURNING AN INSTINCT INTO A NUMBER

“Assume it was found” is arithmetic, not caution.

Responders get asked to justify treating an exposed credential as compromised when the logs show nothing. The justification is short. Model discovery as a Poisson process with rate λ; the chance of at least one discovery in a window W is $1 - e^{-\lambda W}$. **The useful part is how quickly the answer stops depending on λ at all.**

FIGURE 2 · PROBABILITY OF DISCOVERY AGAINST EXPOSURE WINDOW



Four rates spanning **four orders of magnitude**. They disagree sharply for the first hour and agree completely after a month. The shaded region is where the choice of λ stops mattering.

EXPOSURE WINDOW	λ=1/MIN	1/HOUR	1/DAY	1/WEEK
1 minute	63.2%	1.7%	0.1%	0.0%
1 hour	>99.99%	63.2%	4.1%	0.6%
1 day	>99.99%	>99.99%	63.2%	13.3%
1 week	>99.99%	>99.99%	99.9%	63.2%
1 month	>99.99%	>99.99%	>99.99%	98.6%
1 year	>99.99%	>99.99%	>99.99%	>99.99%

One scan a week is a deliberately pessimistic rate for anything reachable from the internet.

Now read it against the incidents. Toyota was exposed for five years, Microsoft AI for three, Mercedes-Benz for three and a half months, xAI for two. **Every one of them sits where all four rates return the same answer.** There was never a defensible reading in which those keys went unnoticed, whatever the logs did or did not record.

THE TOYOTA WINDOW, AT THE MOST PESSIMISTIC RATE

```
# W = Dec 2017 to Sep 2022 = 1,749 days.
# Take the slowest rate in the table, one discovery a week.
lam = 1/7          # per day
W   = 1749
P   = 1 - math.exp(-lam*W)
#  P = 1 - e^-249.9  →  1.0 to 108 decimal places.
#
# There is no rate slow enough, and no rounding generous
# enough, to make that key plausibly unfound.
```

WHEN YOU CANNOT PIN THE WINDOW

Run the arithmetic twice, once on the shortest defensible window and once on the longest, and report both. **If the lower bound already clears 99%, the disagreement about the window does not change any decision**, and arguing it further is time spent on a number that cannot move the answer.

✓ THE DECISION RULE

Exposure measured in **minutes**: the question is genuinely open, investigate hard. Measured in **days or more**: discovery is settled, and the only open questions are what was done with it and how far it reached. Spend the first hour accordingly.

ⓘ WHAT THIS DOES NOT SAY

It models discovery, not damage. Not every discovery is an adversary, and not every adversary acts. **The finding is “we must assume it was found”, not “we know it was used”.**

WHERE λ COMES FROM

Only the top row is measured. Honeypot work puts first use of a public AWS key at one to five minutes, on the most heavily farmed surface that exists. Nobody has published a rate for a key sitting in a private repo, a CI log or a chat thread, which is exactly why the table varies the rate rather than asserting one.

WHEN λ JUMPS

The model holds λ constant; in practice it steps. A star or a fork on the repository, an index by a search engine, a publish to a package registry, a push to a public image registry. **If the window is days but contains one of those events, read the lower rows of the table.**

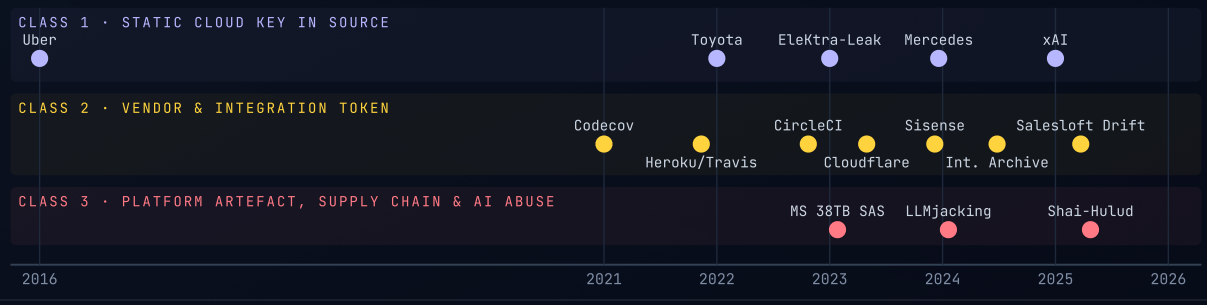
THE EVIDENCE BASE FOR EVERYTHING THAT FOLLOWS

Fifteen incidents define this problem.

All public, all documented, all caused by a credential rather than by malware or a software flaw. The credential class keeps changing; the **failure modes do not**.

FIGURE 3 · TEN YEARS, THREE CLASSES

2016 → 2026 Cremit



Horizontal position is the year of the incident. The three classes appeared in that order and all three still have a 2025 entry. **The centre of gravity has moved from "a developer committed a key" to "a credential you never issued, held by someone else, on your behalf".** Your inventory has to cover both.

YEAR	INCIDENT	CREDENTIAL AND WHERE IT LIVED	IMPACT	WHAT IT PROVES
2016	**Uber**	AWS key in a private GitHub repo, reached with stolen GitHub credentials	57M records; USD 148M settlement; CSO convicted	A private repo is not a boundary. GitHub → AWS → S3 is the blast radius
2021	**Codecov**	GCS credential extractable from a Docker image layer	Uploader altered ~2 months; CI variables of 23,000+ customers taken	Image layers leak too; one vendor key yields every customer secret
2022	**Toyota**	T-Connect DB access key pushed publicly by a subcontractor	296,019 records; public Dec 2017 → Sep 2022	Exposure windows run to years; third parties commit your secrets
2022	**Heroku / Travis**	Stolen GitHub OAuth integration tokens	Private repos of dozens of organisations cloned	An integration token is a standing key to all it was ever granted
2023	**CircleCI**	Malware stole an SSO session; that identity could mint production tokens	Customer variables, tokens and keys taken; all customers told to rotate	Breach the store and the blast radius is every secret it ever held
2023	**Microsoft AI**	Azure SAS token in a repo: account-scoped, full control, expiry set to **2051**	38 TB: workstation backups, service passwords, 30,000+ Teams messages	SAS is minted client-side; the platform never sees it created or used
2023	**EleKtra-Leak**	IAM key harvested from GitHub by automation	Quarantine at 2 min; attacker active 4 min after that; 400+ calls in 7 min	Automated abuse now outruns automated defence
2023	**Cloudflare**	1 token + 3 service accounts left unrotated, "believed unused"	Actor returned 27 days later, into Atlassian and source control	Rotation is a completeness problem, not a speed problem
2024	**Mercedes-Benz**	Unrestricted GitHub token in a public repo	Internal GitHub Enterprise source, plus more keys and strings	Source access is credential access; one token compounds into many
2024	**Sisense**	Credentials in a self-hosted GitLab instance, including S3 access	CISA advisory: all customers to reset every Sisense credential	Your vendor's git server is your credential attack surface
2024	**Internet Archive**	GitLab config token public since Dec 2022, then an unrotated Zendesk token	31M accounts, then 800,000+ tickets, twice in one month	An incomplete rotation is an invitation to a second incident
2024	**LLMjacking**	Stolen cloud credentials pointed at managed model endpoints	Victim inference bills modelled at tens of thousands of dollars a day	Impact is no longer only data; inference spend is now the payload
2025	**xAI**	x.ai API key in a staff member's public repo	~60 models incl. unreleased; notified 2 Mar, live until 30 Apr	Detection is not remediation; someone must own the clock
2025	**Salesloft Drift**	Stolen OAuth tokens for a Salesforce chat integration	700+ organisations' Salesforce data exported over ten days	Actors mined tickets for AWS keys and Snowflake tokens; second-order reach was the goal
2025	**Shai-Hulud**	npm maintainer tokens; worm ran `TruffleHog` on build hosts	Wave 1: 500+ versions. Wave 2: 796 packages, 20M+ downloads/wk	Adversaries run your scanners too, and without change control

WHAT THE FIFTEEN HAVE IN COMMON

Six laws govern the response to a leaked credential.

The fifteen differ in sector, in size and in credential class. The failure modes collapse into six.

LAW 01

Exposure does not expire.

A secret public for ten seconds is public forever. Force-pushing does not remove it: GitHub retains unreachable commits, and the push event, carrying that SHA, stays in public event archives. Researchers mine these "oops commits" at scale and still find live credentials.

EVIDENCE **Toyota**, public five years. **Microsoft AI**, a SAS token whose expiry was extended to 2051 and which sat in a repo for three years.

Consequence: rotation is mandatory; there is no closure without it. "We removed the commit" is not a remediation step; it is a tidying step.

LAW 02

Detection is not remediation.

Most organisations already get the alert. What is missing is an owner, a deadline and a verification step. The gap between "a tool told someone" and "the credential stopped working" is where damage accumulates.

EVIDENCE **xAI**: the developer was notified on 2 March. The key was still valid on 30 April, roughly **60 days**, and only died when the vendor escalated to the security team directly.

Consequence: every finding needs a named owner and a revocation SLA whose clock starts at the credential.

LAW 03

A credential is a graph, not a string.

The question is what the key can read, write, assume, impersonate or mint, and what those reach in turn. Role chaining, service-account impersonation and delegated app permissions all extend the graph past the identity you started with.

EVIDENCE **Uber**: GitHub credential → private repo → AWS key → S3 → 57M records. **Mercedes-Benz**: one token → entire internal source → further keys and connection strings.

Consequence: scope with the policy graph for each provider rather than the developer's memory of what the key was for.

LAW 04

Rotation is a completeness problem.

Partial rotation is indistinguishable from none, because the adversary only needs the one you missed. "We believed it was unused" is the most expensive sentence in this field, and it recurs across unrelated organisations.

EVIDENCE **Cloudflare**: 1 token and 3 service accounts out of thousands, believed unused, used by the actor 27 days later. **Internet Archive**: breached a second time within a month through a token left unrotated after the first.

Consequence: rotation needs an inventory and a reconciliation report, not a checklist someone worked through from memory.

LAW 05

Your logs probably cannot answer "was it read?"

The decisive question in a secret-store incident, whether anyone actually retrieved the secret, is unanswerable by default on two of the three major clouds. GCP's `AccessSecretVersion` is a Data Access log and is off unless enabled; Azure Key Vault records nothing without a diagnostic setting; Azure SAS usage is not platform-tracked at all.

EVIDENCE **Microsoft AI** is the archetype: the token was usable and invisible simultaneously.

Consequence: the logging posture is a decision made before the incident. Mid-incident is too late; the evidence you need was never recorded.

LAW 06

Revoking is not the same as revoked.

Every provider has a gap between the action you take and the moment access stops, and often the obvious action does not close it. Deleting a GCP service-account key does not invalidate tokens already issued. Deactivating an AWS IAM key does nothing to an active `ASIA…` session. Deleting a SAS token is not possible.

PROVIDER BEHAVIOUR GCP: up to **60 minutes** of continued access. AWS: needs an `aws:TokenIssueTime` deny. Azure: rotate the account key or the stored access policy.

Consequence: containment has a verification step.

HOW THESE COMBINE

Laws 1–2: the exposure window is longer than you think. Law 3: the scope is wider. Laws 4–6: the containment is less complete. **One corrective covers all of them: work from an inventory and from provider evidence, never from intuition about what a key was "for".**

WHERE THE THREE DIVERGE

Same questions, three answers

The next ten pages take AWS, GCP and Azure in turn, each in the same order: where the evidence is, how far the blast radius reaches, how to cut it off. This page collects the points where the answers differ. Every value here returns later with its source.

WHAT THE INVESTIGATION ASKS	AWS	GCP	AZURE
How much the credential alone tells you offline, no calls	Down to the owning account number `AKIA` decode	Project and service-account address `service_account` JSON	The full grant `si` · `skoid` parameters
Is a secret read logged by default Law 05	✓ On **CloudTrail management event**	× Off **nothing without `DATA_READ`**	× Off **nothing without a diagnostic setting**
Window the default logs can answer Law 01	90 days event history, per region	400 / 30 days Admin Activity / Data Access	7 / 30 days Entra Free / P1·P2
Where non-human usage is recorded	One place, CloudTrail the same trail as human calls	Two audit log classes defaults differ by class	A separate table `AADServicePrincipal…`
Where blast radius leaks most often Law 03	Resource-based policies never enumerated from the principal	Inheritance · impersonation · DWD invisible in the binding alone	RBAC and Graph, two systems enumerate one and you miss half
Does revocation take effect at once Law 06	◑ Disabling the key is not enough **an active `ASIA…` session survives**	× Access token lives up to 60 min **deleting the key only starts the wait**	× A SAS cannot be revoked **no issued list, no revocation endpoint**
What actually cuts sessions and tokens	`aws:TokenIssueTime` deny denies by issue time	Disable the account 60 min, or strip bindings authorization is evaluated per request	Rotate the account key or stored access policy everything signed with it stops
The first command you run	`cloudtrail lookup-events`	`gcloud logging read`	KQL · `AADServicePrincipal…`

The wrong conclusion each one invites

AWS

The quarantine is attached, so it is contained.

AWSCompromisedKeyQuarantineV3 targets the actions that generate a bill. The data-access actions stay open.

GCP

The key is deleted, so it is over.

Deleting a service-account key does not invalidate tokens already issued. They keep working for **up to 60 minutes**.

AZURE

Nothing in the sign-in log, so it was never used.

Non-human sign-ins land in a different table from human ones. You were reading the wrong table.

What does not vary by provider

LAW 01

Exposure does not expire.

Whoever issued it, a credential public for ten seconds is public forever. No provider has a procedure that takes it back.

LAW 02

Detection is not remediation.

An owner and a deadline set the gap between the alert and the revocation. That gap sizes the incident.

LAW 04

Partial rotation is none.

That the adversary only needs the one you missed has no per-provider exception. Only a reconciliation answers it.

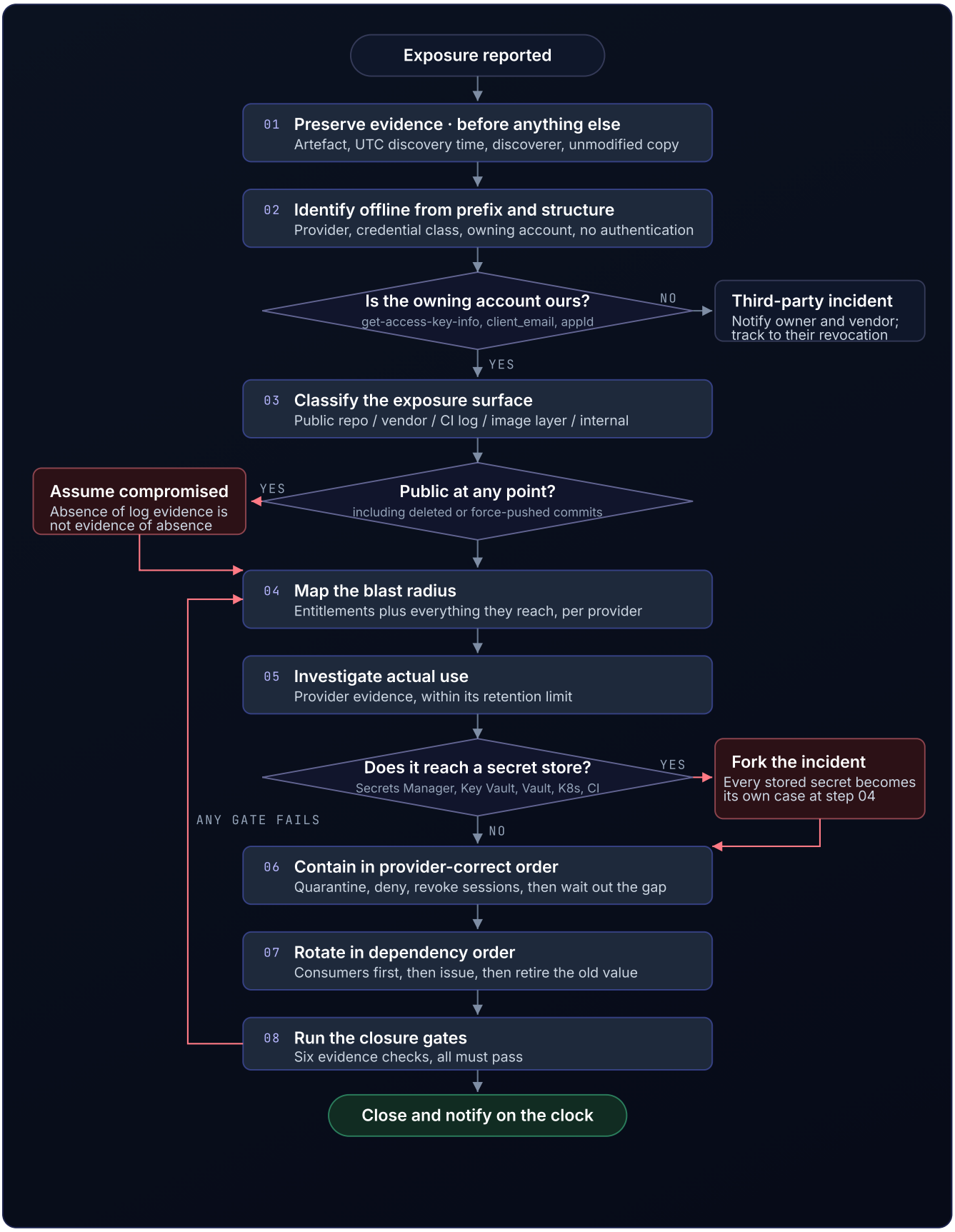
ⓘ WHAT THIS TABLE DOES NOT SETTLE

Three different values in a row do not make one provider safer than another. **What differs is what you can prove after the fact.** Almost all of that difference was decided by a logging posture set long before the incident.

THE WHOLE PLAYBOOK ON ONE PAGE

The whole playbook fits in one decision tree.

Each step is a chapter of this playbook, in order. Two branches are deliberately one-way: once a credential has been public you never return to “probably fine”, and once it reaches a secret store the incident forks.



WHAT A CREDENTIAL TELLS YOU BEFORE YOU AUTHENTICATE WITH IT

You read the key first. You use it second.

Credential formats are self-describing. Prefix, length and structure give you the provider, the credential class, and often the owning account, entirely offline, with no API call, no log entry, and no risk of destroying evidence.

FIGURE 4 · ANATOMY OF THREE CREDENTIALS

offline reading · Cremit

AWS · access key ID

```
AKIAIOSFODNN7EXAMPLE
```

- PREFIX = CREDENTIAL CLASS
- 20 CHARS · THE ACCOUNT NUMBER IS ENCODED HERE
- OR CONFIRM IT WITH sts get-access-key-info

GCP · service account key (JSON)

```
"type": "service_account",
"project_id": "prod-analytics-01",
"private_key_id": "a1b2c3d4e5...",
"client_email": "etl@prod-...iam..."
```

- OWNER, PROJECT AND KEY ID ARE ALL IN PLAINTEXT
- FULL ATTRIBUTION WITH ZERO API CALLS

Azure · shared access signature (SAS) URL

```
https://acct.blob.core.windows.net/?sv=2024-11-04&srt=sco&sp=rwdlacupx&se=2051-10-06&sig=Xy9...
```

- srt=sco → SERVICE + CONTAINER + OBJECT: THE WHOLE ACCOUNT
- sp=rwdlacupx → EVERY PERMISSION · se= IS THE ONLY LIMIT, HERE 2051

READ IT BEFORE YOU PANIC · THE SCOPE IS WRITTEN ON THE TOKEN ITSELF

The Azure example is the Microsoft AI incident's token shape: account-scoped, every permission, an expiry of **2051**. All of that is legible from the query string without a single API call, which is exactly why SAS deserves its own containment path.

AWS unique ID prefixes

The first four characters are authoritative. `ASIA` changes your entire containment plan, so check it first.

PREFIX	IDENTIFIES
AKIA	Long-term access key for an IAM user. Deactivate and delete.
ASIA	Temporary STS key. **Cannot be deleted.** Needs a session deny
AIDA	IAM user unique ID
AROA	Role: appears in `aws:userId` as `AROA…:session`
AIPA	EC2 instance profile
ANPA / ANVA	Managed policy / policy version
AGPA	User group
ABIA / ACCA	STS bearer token / context-specific credential

Provider token shapes

Read the shape, pick the chapter.

SHAPE	CREDENTIAL
`ghp_` · `github_pat_`	GitHub PAT, classic or fine-grained. Introspection differs sharply
`gho_` · `ghu_` · `ghs_` · `ghr_`	GitHub OAuth / user-to-server / App installation / refresh
`xoxb-` · `xoxp-` · `xapp-`	Slack bot / user / app-level token
`AIza…`	Google API key. No owner encoded; restrictions are the only control
`ya29.…`	Google OAuth2 access token. Short-lived, not revocable
`sk-` · `sk-ant-`	Model provider API key. Billing is often the only signal
`sk_live_` · `rk_live_`	Stripe secret / restricted key
`npm_` · `pypi-`	Package registry token. Treat as a supply-chain incident
`eyJ…`	JWT. Decode the header and claims; `exp` and `aud` bound the damage

EVERYTHING YOU CAN ESTABLISH BEFORE YOU TOUCH THE NETWORK

Most of the first hour needs no API call at all.

Every triage call costs something: a log entry that pollutes the timeline you are about to build, a `last-used` you overwrite, an alert to a third party. **Offline analysis has none of those costs, and answers more than most responders realise.**

The AWS account ID inside the key

An access key ID is structured. After the four-character prefix, the remaining sixteen characters are base32, and the first six decoded bytes carry the **owning account number, shifted left by seven bits**. You can attribute a leaked key to an account without credentials, without touching the network, and without leaving a trace.

ATTRIBUTION WITH ZERO BLAST RADIUS

```
import base64

def account_from_key(key_id):
    body = key_id[4:].upper()   # drop the prefix
    raw  = base64.b32decode(body)  # 16 chars → 10 bytes
    n    = int.from_bytes(raw[:6], 'big')
    return (n & 0x7fffffffff80) >> 7

>>> account_from_key("ASIAY34FZKBOKMUTVV7A")
609629065308
```

That is as far as it goes. It does not tell you whether the key is live, and a fabricated key still decodes to a plausible number. Use it to route the incident; confirm with `get-access-key-info` only when routing is not enough.

Tokens that prove they are real

GitHub's token formats end in six characters that are a base62-encoded **CRC32 of the random body**. Recompute it and you know, with certainty and without authenticating, whether a candidate string is a well-formed GitHub token. GitHub's own engineering write-up puts it plainly: a checksum "virtually eliminates false positives for secret scanning offline".

WHY THIS MATTERS DURING TRIAGE

```
# A dump of 400 candidate strings from a repo scan.
# Without a checksum you authenticate each one to find out
# which are real: 400 log entries, 400 chances to trip an
# alert, and last-used destroyed on every one that is live.
#
# With a checksum you sort them offline, first, and only
# ever touch the network for the ones that could be real.
ghp_ gho_ ghu_ ghs_ ghr_ github_pat_
#  prefix        30-char body          6-char checksum
#  \______/  \________________/  \____________/
```

This is also why a `ghp_` hit is nearly always a true positive and an `AIza…` hit often a false one: one format was designed to be checkable, the other was not.

What each format gives up without a single request

FORMAT	PROVABLE OFFLINE	WHAT YOU LEARN FROM THE STRING	WHAT YOU STILL CANNOT KNOW
`AKIA · ASIA · ABIA`	Partly	**Credential class from the prefix, and the owning account number by decoding**	Validity, the principal's name, and any permission at all
`ghp_ · gho_ · ghs_ · ghr_ github_pat_`	✓ Yes	**Token class, and well-formedness with certainty from the CRC32**	Scopes for fine-grained tokens; those have no introspection at all
`service_account JSON`	✓ Yes	**Project, service account address, key ID, and a parseable RSA private key**	Which roles it holds, and whether the key was already deleted
`eyJ… (JWT)`	Structurally	**Issuer, audience, subject, `exp`, `iat` from the first two segments**	Whether the signature is valid, without the issuer's key
`SAS query string`	✓ Yes	**The full grant: services, resource types, permissions, expiry, and revocability**	Whether it was ever used; SAS creation is not platform-tracked
`xoxb- · sk- · sk_live_ npm_ · AIza…`	× No	Provider, and environment where the prefix says so. Nothing more	**Owner, scope and validity all require a call. These formats force you onto the network**

ORDER THE FIRST HOUR BY COST, NOT BY CURIOSITY

Decoding, parsing and checksum verification are free and invisible. Reading provider metadata from your own account is cheap and logged as yours. Authenticating as the leaked credential is expensive, destroys `last-used`, and may notify someone outside your organisation. **In that order, the evidence is still there when the step that needs it arrives.**

WHOSE IS IT, IS IT ALIVE, AND WHAT DID YOU JUST DESTROY FINDING OUT

Testing the key overwrites the evidence.

⚠ READ THIS BEFORE YOU RUN ANYTHING

On AWS, `iam get-access-key-last-used` returns the most recent use of a key, often the best single indicator that an adversary touched it. **Authenticate with that key yourself and the last-used date becomes your own timestamp; the original is gone.** Query it first, from your own identity, and save the output.

Attribution without authenticating

AWS · CONFIRM, THEN PRESERVE

```
# You already have the account offline. This confirms it
# under YOUR credentials, and is logged as yours, not theirs.
aws sts get-access-key-info --access-key-id AKIA...
#  { "Account": "111122223333" }
# Then, in that account, BEFORE any validity test:
aws iam get-access-key-last-used --access-key-id
AKIA...
#  LastUsedDate / ServiceName / Region  ⟵ preserve
```

GCP & AZURE · READ THE ARTEFACT, NOT THE API

```
# GCP: everything you need is plaintext in the JSON.
jq '{project_id, client_email, private_key_id}'
key.json
# Azure: a client secret is anonymous alone. Attribution comes
# from the appId that shipped with it.
az ad sp show --id <appId> --query displayName
```

Rules for a safe validity test

- **Preserve first.** Capture `last-used`, sign-in logs and usage dashboards before the credential sees any traffic from you.
- **Non-mutating calls only.** `sts get-caller-identity`, `auth.test`, `GET /user`. Never a write and never an enumeration sweep, which is indistinguishable from the attacker's.
- **From one recorded host.** A fixed jump host with a known egress IP and a distinctive user agent, so your calls are separable in the logs later.
- **Write it down as you go.** UTC time, source IP, user agent, exact command. This becomes the exclusion list when you hunt in Phase 04.
- **Assume the owner is alerted.** For a third-party key your probe may be the vendor's first sign of an incident.

ⓘ A DEAD KEY STILL NEEDS INVESTIGATING

A validity test answers one question: whether containment is still needed. It says nothing about use while the key was alive. Laws 1 and 5 still apply.

Exposure window worksheet

Establish T1 from evidence, never recollection. Take the **earliest** value any row produces. One reliable early source overrides several convenient later ones.

EVIDENCE SOURCE	WHAT IT ESTABLISHES	HOW TO GET IT	RELIABILITY
Commit author date	Earliest possible exposure of the value	`git log --format=%aI -S'<secret>' --all`	**High, but see push date below**
Commit push date	When it became visible to others	Events API / audit log. **Can be years after the author date**	**Use the earlier of the two**
Public event archives	Proof a deleted commit was once public	Search public GitHub event archives by repo and SHA	**High, independent of the repo**
Credential creation date	Absolute floor for T1	`iam list-access-keys` · `gcloud iam service-accounts keys list`	**High**
Last-used metadata	Latest confirmed use	`iam get-access-key-last-used`	**Coarse; overwritten by your own test**
Artefacts & CI logs	Exposure via a published image or build output	`docker history` · per-layer diff · job-log search	**Retention-bound; pull immediately**
Provider notification	Third-party discovery time	Vendor alert, AWS Health event, secret-scanning alert	**High, but always a ceiling not a floor**

If two sources disagree, the exposure window is the union of both, not the intersection.

■ WHEN AUTHOR AND PUSH DATES DISAGREE

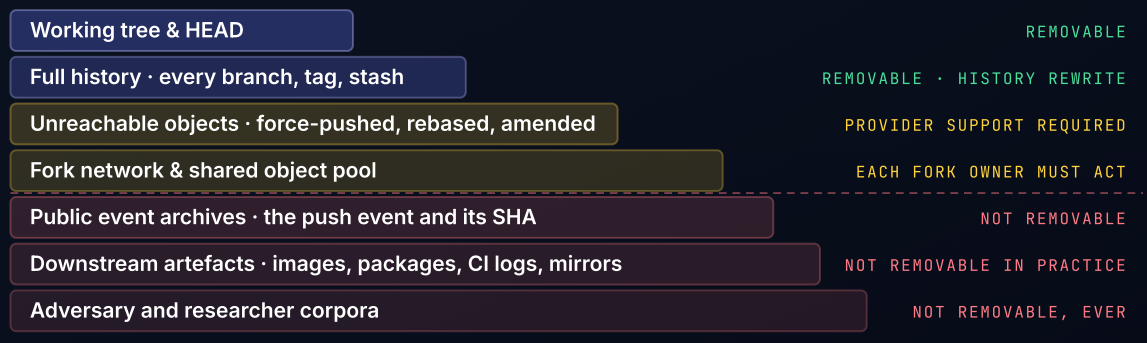
The author date is when the commit was made locally; the push date is when it became visible to anyone else. Push an old branch late and they are years apart. Squash the commits and the author date moves forward, making the window look shorter than it was. **Pull both and take the earlier as T1.** Where neither can be established, the credential's creation date is the floor.

WHERE IT LEAKED, FOR HOW LONG, AND WHAT YOU CAN NEVER TAKE BACK

Exposure is wider than the commit.

Deleting the file removes one copy. The value survives in history, in unreachable objects, in every fork, in public event archives, and in whatever pulled it first. **Ask of each surface: can I actually remove this?** Mostly you cannot.

FIGURE 5 · EXPOSURE SURFACES AND WHAT YOU CAN REMOVE



Everything below the dashed line is outside your control. If the secret crossed it, rotation is the only remediation that exists.

Git archaeology

FIND EVERY OCCURRENCE, INCLUDING IN HISTORY

```
# All refs, all history, by content, not by filename.
git log --all --format='%H %aI %cI %an' -S'AKIA...'
# %aI = author date, %cI = commit date. Neither is the PUSH
# date. Take that from the events or audit log.

# Objects no branch points at any more. Still fetchable by
# SHA on the host, and that SHA is in the public push event.
git fsck --unreachable --dangling --no-reflogs
git cat-file -p <dangling-sha>
```

THE SURFACES GIT WILL NOT SHOW YOU

```
# Image layers. The value can sit in a layer even when it is
# absent from the final filesystem.
docker history --no-trunc <image>
docker save <image> | tar -xO | grep -a 'AKIA'

# Build logs, for the value and for its masked form: a tool
# that masks in the UI often does not mask in the raw log.
gh run view <run-id> --log | grep -nE 'AKIA|\*{3,}'

# IaC state. Written by apply, read by nobody.
terraform state pull | jq -r '..|strings' | grep -i secret
```

Toyota

2022 Cremit

EXPOSURE WINDOW

A subcontractor pushed T-Connect source including a customer-database access key to a **public** repo in December 2017. Found September 2022. 296,019 records in scope.

TAKEAWAY

Nobody there could answer “how long has this been public” from memory. The commit metadata could, in one command.

Codecov

2021 Cremit

A SURFACE NOBODY SCANS

The credential was never in a repository. It was extractable from the **Docker image creation process**, and it let an attacker alter the Bash Uploader for two months and collect CI environment variables from **23,000+** customers.

TAKEAWAY

Layers, packages and build logs are exposure surfaces just as much as source is.

⚠ FORCE-PUSH DOES NOT DELETE

Git providers retain unreachable commits, and they stay fetchable by SHA. The original push event carrying that SHA is preserved in public archives independently of your repository. 2025 research mined exactly these “oops commits” at scale and recovered live cloud keys and platform tokens. **“We rewrote history” is housekeeping done after rotation, not containment.**

AWS · 1 OF 4

What AWS recorded sets the limit on what you can ask.

Every AWS investigation is bounded by decisions made months earlier. Establish those bounds first; an investigation that assumes coverage it does not have produces a confident, wrong answer.

SOURCE	ON BY DEFAULT	RETENTION	WHAT IT ANSWERS, AND THE TRAP
CloudTrail Event history	✓ Yes	90 days	Management-plane calls, per Region. **Does not contain data events**, so it will not tell you whether an object was read.
CloudTrail trail → S3	× No	You choose	The only durable copy. Confirm it is multi-Region and organisation-wide, or your attacker's Region is simply missing.
CloudTrail data events	× No	Trail-bound	**S3 `GetObject`, Lambda `Invoke`, DynamoDB item access.** Without these, "did they read the bucket" is unanswerable.
CloudTrail Lake	× No	up to 7–10 yr	SQL over historical events. Worth creating during a long-window incident; it can ingest existing trail data.
GuardDuty	◑ Per account	90 days	Behavioural findings on the credential. Check `CredentialAccess`, `Discovery`, `PrivilegeEscalation`, `Impact` and `InstanceCredentialExfiltration`.
AWS Health	✓ Yes	90 days	`AWS_RISK_CREDENTIALS_EXPOSED`. AWS found your key in public. Its timestamp is an upper bound on T1, never the actual T1.
IAM last-used / credential report	✓ Yes	Current state	Latest use per key. **Destroyed by your own validity test**: capture it first.
Access Advisor	✓ Yes	365 days	Services this identity actually used. The gap against its entitlements is your over-permission.

The bounds first, then the record

STEP 1 · WHAT COVERAGE DO YOU ACTUALLY HAVE

```
# Is there a durable, multi-Region, org-wide trail at all?
aws cloudtrail describe-trails --query 'trailList[].{n:Name,
  multiRegion:IsMultiRegionTrail, org:IsOrganizationTrail,
bucket:S3BucketName}'

# Are data events being captured? Empty output = no object-level
# visibility, and no way to answer "was the data read".
aws cloudtrail get-event-selectors --trail-name <name>

# Logging could have been stopped by the attacker. Check.
aws cloudtrail get-trail-status --name <name> \
  --query '{logging:IsLogging, stopped:LatestDeliveryError}'
```

STEP 2 · PULL THE SLICE BEFORE IT AGES OUT

```
# Event history is 90 days and per Region, so loop. This is
# the copy you keep; the console view is not evidence.
for r in $(aws ec2 describe-regions \
          --query 'Regions[].RegionName' --output text); do
  aws cloudtrail lookup-events --region "$r" \
    --start-time "$FROM" --max-results 50 \
    --lookup-attributes
AttributeKey=AccessKeyId,AttributeValue="$KEY" \
    > "ct-$r.json"
done

# If the window is longer than 90 days, create a CloudTrail
# Lake event data store now and ingest the S3 trail into it.
# You cannot recover what Event history has already dropped.
```

The fields that matter

- `userIdentity.accessKeyId`: your primary selector. For a role session the key is `ASIA…`, so pivot on `sessionContext` too.
- `userIdentity.sessionContext.attributes.creationDate`: when the session began; matches `aws:TokenIssueTime` used for revocation.
- `sourceIPAddress` + `userAgent`: separates the adversary from your own responders. Exclude your jump host.
- `errorCode`: a burst of `AccessDenied` across many distinct actions is the signature of enumeration.
- `readOnly`: split the timeline into reconnaissance and action.

ⓘ CHECK EVERY REGION

Event history is per-Region and attackers deliberately work in Regions you do not watch. EleKtra-Leak's very first call was `DescribeRegions`.

⚠ WRITE THE COVERAGE GAP INTO THE REPORT

If data events were off, the correct finding is "object access cannot be confirmed or excluded", not "no evidence of data access". The second phrasing reads as an all-clear to every non-technical reader, and what it reports is a gap in the log.

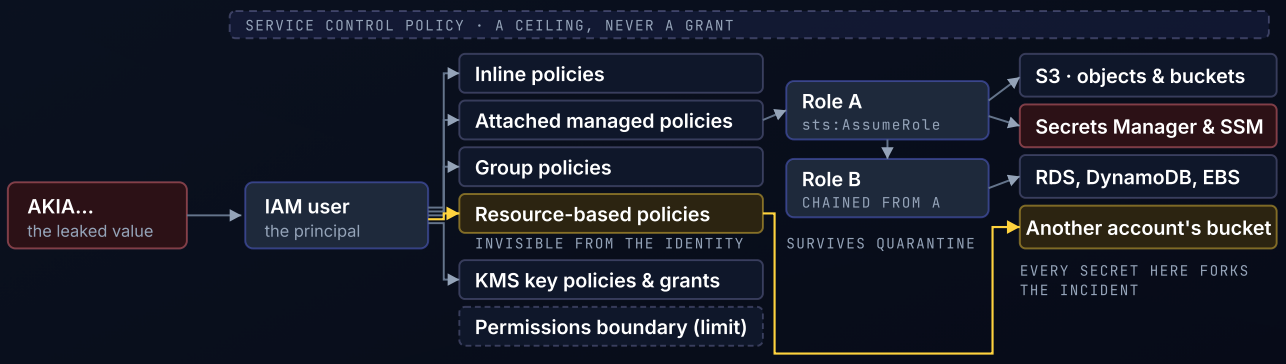
AWS · 2 OF 4

The key is a starting node, not the answer.

Blast radius is six grant mechanisms minus two constraints, closed transitively over every role the principal can assume. Miss one and you under-report scope, usually the resource-based policies, because they live on the resource, not the identity.

FIGURE 6 · WHAT ONE AWS ACCESS KEY REACHES

permission graph Cremit



Orange path: a grant on the **resource**, not the identity. Enumerating the user's policies never surfaces it, which is why a blast-radius call comes back too small.

COMPUTE THE RADIUS · ENTITLEMENT, THEN REALITY

```
# 1. Everything the account grants, in one document.
aws iam get-account-authorization-details --filter User Role \
  Group LocalManagedPolicy > auth.json

# 2. Which roles may this principal become? Read the TRUST
#    policies, not the permission policies.
jq '.RoleDetailList[] | select(.AssumeRolePolicyDocument|tostring
   |contains("user/leaked-user")) | .Arn' auth.json

# 3. Ask specific questions instead of reading policy prose.
aws iam simulate-principal-policy --policy-source-arn <arn> \
  --action-names secretsmanager:GetSecretValue kms:Decrypt \
  --resource-arns '*'

# 4. Cross-account exposure via RESOURCE policies.
aws accessanalyzer list-findings --analyzer-arn <arn>
# 5. What it used in 365d. Entitlement minus this = the
#    over-permission you now own.
aws iam generate-service-last-accessed-details --arn <arn>
```

Uber 2016 Cremit

THE GRAPH, NOT THE STRING

Stolen GitHub credentials reached a **private** repository. Inside was an AWS key. The key reached S3. S3 held rider and driver records.

RECORDS	57 million
SETTLEMENT	USD 148M across 50 states and DC
PERSONAL	The CSO was criminally convicted over the concealment

TAKEAWAY

Nobody scoped "a GitHub credential" as an S3 data-breach risk, because they scoped the credential rather than everything it reached.

The mechanisms, the constraints, and how to enumerate each

	MECHANISM	ENUMERATE WITH	MOST COMMON MISS
G1	Inline user policies	`iam list-user-policies · get-user-policy`	Never surfaced in the console summary view
G2	Attached managed policies	`iam list-attached-user-policies`	Only the default policy version is evaluated
G3	Group membership	`iam list-groups-for-user → group policies`	Inherited admin nobody remembers granting
G4	Resource-based policies	`accessanalyzer list-findings`	**Invisible from the identity. The usual gap.**
G5	KMS key policies & grants	`kms get-key-policy · kms list-grants`	Grants outlive the policy that made them
G6	Assumable roles (transitive)	`trust policies in get-account-authorization-details`	Chains of depth 2+ are rarely walked
C1	Permissions boundary	`iam get-user → PermissionsBoundary`	Present but scoped so wide it constrains nothing
C2	Service control policy	`organizations describe-effective-policy`	Assumed to apply; often not on that OU

Effective permission = (G1∪…∪G6) ∩ C1 ∩ C2, for every role the key can reach.

AWS · 3 OF 4

An attacker works an AWS key in a predictable order.

Automated credential abuse is highly formulaic. Work the table below as a checklist against your timeline: presence of any row moves the incident from **exposed** to **used**, and the Impact rows are what generate the invoice.

STAGE	CLOUDTRAIL EVENTNAME TO HUNT	WHY IT MATTERS
Discovery	`GetCallerIdentity` · `DescribeRegions` · `DescribeInstances` · `ListBuckets` · `ListRoles` · `GetAccountAuthorizationDetails` · `DescribeDBInstances`	Almost always first. `DescribeRegions` was EleKtra-Leak's opening call.
Persistence	`CreateAccessKey` · `CreateUser` · `CreateLoginProfile` · `UpdateLoginProfile` · `AttachUserPolicy` · `ImportKeyPair`	A second credential the attacker owns. **Rotating the leaked key does not remove it.**
Priv. escalation	`PutUserPolicy` · `AttachRolePolicy` · `UpdateAssumeRolePolicy` · `PassRole` · `AssumeRole`	Changes the blast radius after you measured it. Re-measure at the end.
Defence evasion	`StopLogging` · `DeleteTrail` · `UpdateTrail` · `DeleteDetector` · `DeleteLogGroup` · `PutEventSelectors`	Any of these means your later timeline is incomplete by construction. That belongs in the report.
Credential access	`GetSecretValue` · `GetParameter(s)` · `Decrypt` · `GetPasswordData`	Fork the incident. **Note: not blocked by AWS's quarantine policy.**
Collection & exfil	`GetObject` DATA EVENT · `CreateDBSnapshot` + `ModifyDBSnapshotAttribute` · `CreateSnapshot` + `ModifySnapshotAttribute` · `PutBucketPolicy`	The `Modify…Attribute` pair shares a snapshot with an **external account**: exfiltration that leaves no data-event trail at all.
Impact	`RunInstances` · `InvokeModel` · `InvokeModelWithResponseStream` · `CreateEndpointConfig` · `SendEmail` · `VerifyEmailIdentity` · `ScheduleKeyDeletion`	Cryptomining, LLMjacking, spam, destruction. Cross-check the billing console for the same window.

Two queries that do most of the work

ATHENA · THE FULL TIMELINE FOR ONE KEY

```
SELECT eventtime, awsregion, eventsource, eventname,
       sourceipaddress, useragent, errorcode, readonly
FROM   cloudtrail_logs
WHERE  useridentity.accesskeyid = 'AKIA...'
  AND  eventtime BETWEEN '2026-06-01T00:00:00Z' AND '2026-08-
21T00:00:00Z'
ORDER BY eventtime;
-- For a role session, pivot on
--   useridentity.sessioncontext.attributes.creationdate
```

ATHENA · THE ENUMERATION FINGERPRINT

```
SELECT sourceipaddress, useragent,
       count(*)                          AS calls,
       count_if(errorcode IS NOT NULL)   AS denied,
       count(DISTINCT eventname)         AS distinct_actions
FROM   cloudtrail_logs
WHERE  useridentity.accesskeyid = 'AKIA...'
GROUP BY 1, 2 ORDER BY denied DESC;
-- High denied + high distinct_actions from one IP is an
-- adversary mapping your permissions. Legitimate workloads
-- call few actions and rarely get denied.
```

EleKtra-Leak — 2023 — Cremit

SPEED, MEASURED

A key hit a public repository. AWS applied its quarantine policy **within two minutes**. The actor began reconnaissance **four minutes after that** and issued **400+ API calls in seven minutes**, starting with `DescribeRegions` and moving to security-group changes and multi-Region `RunInstances` for mining.

TAKEAWAY

Two minutes of automated defence was not enough. Design your timeline around the assumption that Discovery already completed.

EXCLUDE YOURSELF FIRST

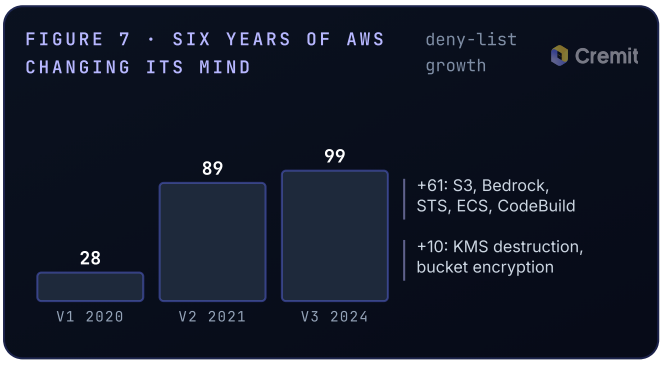
Before drawing conclusions, remove your responders' source IPs and user agents from the result set, and the `get-caller-identity` you ran during the validity test. Misattributing your own probe as adversary activity means rebuilding the whole timeline.

AWS · 4 OF 4

The quarantine policy protects AWS's invoice, not your data.

When AWS finds your key in public it attaches `AWSCompromisedKeyQuarantineV3`, whose stated purpose is limiting damage **"caused by fraud-related activity leading to unauthorized charges, while not impacting the existing resources."** Read that as written: it is a cost-abuse control, and the actions that matter most in a data-breach investigation are left open.

FIGURE 7 · SIX YEARS OF AWS CHANGING ITS MIND



Each version restates what AWS thinks a stolen key is for. **2020**: compute fraud, so EC2 and Lightsail. **2021**: destruction and resale, so fifteen S3 actions, then `bedrock:InvokeModel` once LLMjacking arrived. **2024–26**: ransomware, so `kms:ScheduleKeyDeletion` and `s3:PutEncryptionConfiguration`. **Diffing the three versions shows no action has ever been removed**, and that beyond S3 reads, not one of the twenty-one data-access actions we checked has ever been denied in any of them.

STILL PERMITTED: CLOSE THESE YOURSELF

Everything a data-theft operator needs:

- `sts:AssumeRole`: **role chaining still works.** The quarantine applies to the user, not to roles it becomes
- `secretsmanager:GetSecretValue`: every secret it can reach is still readable
- `ssm:GetParameter(s)`: parameters still come back decrypted with `WithDecryption`
- `kms:Decrypt`: only key-policy and deletion actions are denied
- `rds:CreateDBSnapshot` + `ModifyDBSnapshotAttribute` and the ec2 equivalents: **share a snapshot to an external account, exfiltrating around the S3 denies**
- Read paths into DynamoDB and RDS

Containment order

1. **Capture first.**
   `get-access-key-last-used`, trail status, GuardDuty findings, billing. Once you act, these change.
2. **Leave AWS's policy attached.**
   Removing it before remediation is complete can affect your eligibility for AWS concessions.
3. **Close the gaps above with your own deny.**
   Quarantine is a floor, not a ceiling.
4. **Deactivate the key; do not delete it.**
   `iam update-access-key --status Inactive`. A deleted key ID cannot be resolved against your logs, and `last-used` goes with it. Delete once the investigation closes.
5. **Revoke role sessions**
   if any `ASIA…` credential or assumed role is in scope. Deactivating a user key does nothing to a live session.
6. **Hunt persistence before declaring containment.**
   Any `CreateUser`, `CreateAccessKey` or `CreateLoginProfile` in the window is a second credential that survives rotation.

CLOSE THE QUARANTINE GAPS

```
// Attach alongside quarantine, on the same user.
{ "Version": "2012-10-17", "Statement": [{
  "Sid": "IRCloseQuarantineGaps",
  "Effect": "Deny", "Resource": "*",
  "Action": [
    "sts:AssumeRole",
    "kms:Decrypt",
    "secretsmanager:GetSecretValue",
    "ssm:GetParameter*",
    "rds:CreateDBSnapshot",
    "rds:ModifyDBSnapshotAttribute",
    "ec2:CreateSnapshot",
    "ec2:ModifySnapshotAttribute",
    "ec2:ModifyImageAttribute"
  ] }] }
```

REVOKE LIVE ROLE SESSIONS (ASIA…)

```
# Console: Role → Revoke sessions. Attaches
# AWSRevokeOlderSessions, denying every session
# issued before now, plus ~30s for propagation.
{ "Effect": "Deny",
  "Action": "*", "Resource": "*",
  "Condition": { "DateLessThan": {
    "aws:TokenIssueTime":
      "2026-08-25T09:00:00Z" }}}
# Set this timestamp to your own T4.
# Service-linked roles cannot be revoked this
# way. Identity Center sessions revoke in IdC.
```

GCP · 1 OF 3

On GCP the default answer to “was the secret read?” is unknown.

GCP splits audit logging into two classes with opposite defaults. Admin Activity is always on, free, 400 days. Data Access is **off**, billable, 30 days. Reading a secret is `DATA_READ`. Which side of that line you are on decides what you can promise anyone.

LOG CLASS	DEFAULT	RETENTION	CONTAINS
Admin Activity `_Required` bucket	✓ On, free	`400 days` not configurable	Config and metadata writes: IAM bindings, key creation and deletion. **Where persistence shows up.**

Data Access
`_Default` bucket
	× Off	`30 days`
		`1-3,650`
		configurable
			`AccessSecretVersion`, `GetSecret`, `ListSecrets`, GCS object reads. **Without it, secret reads leave no trace.**

Security Command Center	◑ Tier-dependent	`Varies`	Anomalous IAM grants, exfiltration and cryptomining. Silence here is not absence.

Pivoting on the key, not the account

Audit entries name the exact key used, which separates the leaked key from every other credential the same service account holds.

CONFIRM WHAT IS BEING LOGGED, THEN HUNT

```
# 1. Is Data Access logging on for Secret Manager at all?
gcloud projects get-iam-policy PROJECT --format=json \
  | jq '.auditConfigs'
#   null  →  secret reads were never recorded. Say so.

# 2. Every call authenticated by THIS key.
gcloud logging read '
  protoPayload.authenticationInfo.serviceAccountKeyName:"KEY_ID"
  AND timestamp ≥ "2026-06-01T00:00:00Z"' \
  --project=PROJECT --format=json

# 3. Denials: the enumeration fingerprint, GCP edition.
gcloud logging read '
  protoPayload.authenticationInfo.principalEmail="etl@
...gserviceaccount.com"
  AND protoPayload.status.code ≠ 0' --project=PROJECT
```

FIELDS WORTH KNOWING

- `authenticationInfo.serviceAccountKeyName`: which key signed the request
- `authenticationInfo.principalEmail`: the identity
- `authenticationInfo.serviceAccountDelegationInfo`: **proof of impersonation**, and the chain that produced it
- `requestMetadata.callerIp` and `callerSuppliedUserAgent`
- `authorizationInfo[].granted`: per-permission allow/deny
- `status.code`: non-zero is a denial

⚠ THE SETTING FOR NEXT TIME

Enabling Data Access logging mid-incident records nothing retroactively. At the **organisation** level, `DATA_READ` for Secret Manager covers every project. The cost is real but bounded; the alternative is not knowing.

What you can still prove with Admin Activity alone

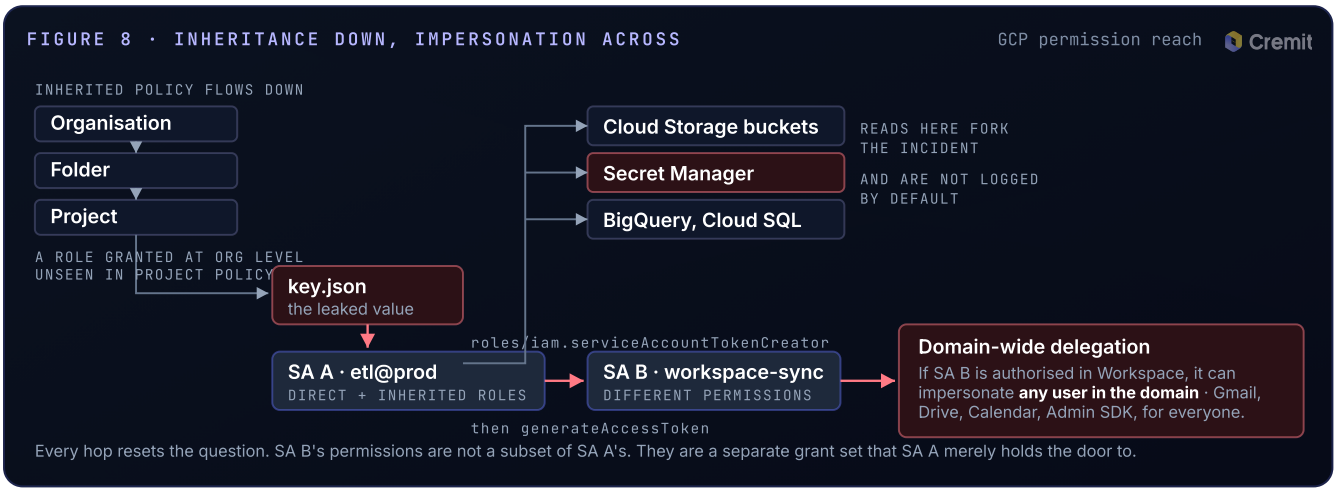
Data Access being off does not end the investigation. Persistence and escalation are configuration changes, and those are always recorded for 400 days, free.

ADVERSARY GOAL	METHOD NAME IN THE AUDIT LOG	WHAT ITS PRESENCE MEANS
Persistence	`google.iam.admin.v1.CreateServiceAccountKey`	A second key the attacker holds. Deleting the leaked one changes nothing.
Persistence	`google.iam.admin.v1.CreateServiceAccount`	A brand-new identity outside your inventory.
Escalation	`SetIamPolicy`	Compare `policyDelta` to your baseline. New `tokenCreator` bindings are the impersonation setup.
Escalation	`google.iam.credentials.v1.GenerateAccessToken`	Impersonation happened. Follow the chain.
Evasion	`UpdateSink · DeleteSink`	Log routing altered. Your timeline has a hole; document it.
Impact	`compute.instances.insert · aiplatform…predict`	Cryptomining and model abuse. Cross-check billing.

GCP · 2 OF 3

A service account key is a licence to become other identities.

GCP's blast radius grows in two directions the identity's own bindings never show. **Downward**, through policy inherited from the organisation and folder. **Sideways**, through impersonation, and if any account in that chain holds domain-wide delegation, sideways means every mailbox and every Drive file in the company.



FOLLOW THE GRAPH TO THE END

```
# 1. Effective policy INCLUDING inheritance. Project-level
#    get-iam-policy alone will understate this.
gcloud asset analyze-iam-policy --organization=ORG_ID \
  --identity="serviceAccount:etl@prod.iam\
                      .gserviceaccount.com"

# 2. Who can this SA become?
gcloud asset analyze-iam-policy --organization=ORG_ID \
  --identity="serviceAccount:etl@..." \
  --permissions="iam.serviceAccounts.getAccessToken"
#    Repeat for every SA returned. That is the full reach.

# 3. Did impersonation actually happen?
gcloud logging read 'protoPayload.methodName=
  "GenerateAccessToken"' --project=PROJECT
```

⚠ CHECK DOMAIN-WIDE DELEGATION BY HAND

DWD is not visible from GCP IAM at all. It is configured in the **Workspace Admin console** under Security → API controls → Domain-wide delegation, and the entries are OAuth client IDs. Get the service account's unique ID with `gcloud iam service-accounts describe SA --format='value(uniqueId)'` and match it against that list. A hit escalates the incident from “a project” to “the company”.

API KEYS ARE A SEPARATE PROBLEM

An `AIza…` key carries no identity and no IAM role. Its only controls are API restrictions and application restrictions, both optional. There is nothing to impersonate and nothing to inherit, but equally nothing to scope, so treat an unrestricted API key as valid for every API enabled on the project.

The bindings that turn one identity into many

Search the effective policy for these. Any of them means the reach is larger than the identity's own permissions suggest.

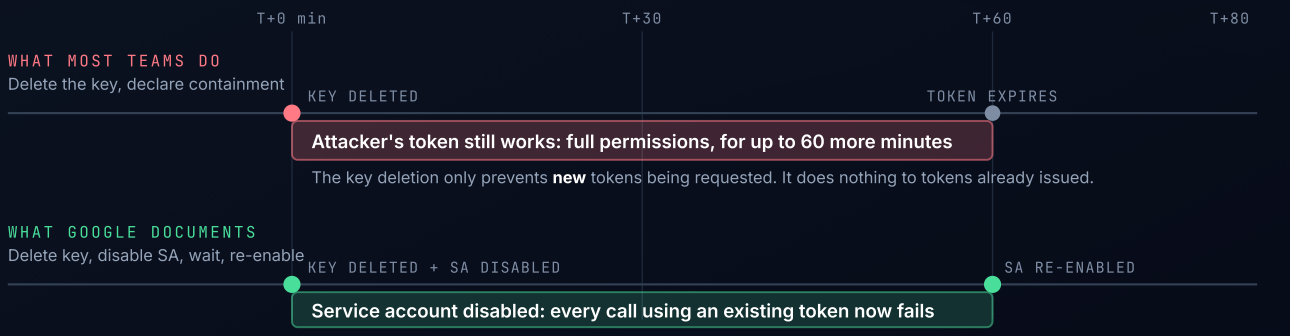
ROLE OR PERMISSION	WHAT IT GRANTS	WHY IT MATTERS IN THIS INCIDENT
`roles/iam.serviceAccountTokenCreator`	Mint access tokens as another SA	**The core impersonation primitive. Follow every edge.**
`roles/iam.serviceAccountUser`	Attach an SA to a new resource	Deploy a job that runs as a more privileged SA.
`iam.serviceAccounts.signJwt / signBlob`	Sign assertions as the SA	Token minting without `getAccessToken` ever appearing.
`roles/iam.serviceAccountKeyAdmin`	Create new SA keys	Persistence. Survives deletion of the leaked key.
`roles/resourcemanager.projectIamAdmin`	Edit the project IAM policy	Self-escalation to anything in the project.
`roles/owner · roles/editor`	Broad access across most project resources	Frequently inherited from a folder and forgotten.

GCP · 3 OF 3

Deleting the key does not stop the attacker.

This is the single most consequential provider difference in this document, and it is counter-intuitive enough that experienced responders get it wrong. **Service account access tokens cannot be revoked.** Not by deleting the key, not through the Admin console, and not with `gcloud auth revoke`. They simply expire, an hour after they were issued.

FIGURE 9 · TWO CONTAINMENT SEQUENCES, ONE HOUR APART



Google's wording: "Deleting a service account key can invalidate the long-lived credential so that an attacker can't request a new access token, but it doesn't invalidate the access tokens already granted. To ensure access tokens aren't abused within their 60 minute lifetime, you must disable the service account for 60 minutes."

CONTAINMENT SEQUENCE

```
# 0. Preserve: list keys BEFORE deleting anything.
#    A key you did not create is the attacker's persistence.
gcloud iam service-accounts keys list --iam-account=SA \
  --managed-by=user --format="table(name,validAfterTime,keyOrigin)"

# 1. Stop NEW tokens.
gcloud iam service-accounts keys delete KEY_ID --iam-account=SA

# 2. Stop EXISTING tokens. This is the step that is skipped.
gcloud iam service-accounts disable SA

# 3. Wait out the full token lifetime. Do not shorten this.
#    ...60 minutes...

# 4. Restore service.
gcloud iam service-accounts enable SA

# For a USER's leaked gcloud credentials the path is different:
# remove the gcloud CLI from that user's connected applications
# in the Admin console. That does invalidate their tokens.
```

PROVE THE WAIT ACTUALLY WORKED

```
# Any call still succeeding on the old key after T+0 means the
# disable did not take. Expect status.code ≠ 0 throughout.
gcloud logging read '
  protoPayload.authenticationInfo.serviceAccountKeyName:"KEY_ID"
  AND timestamp ≥ "T0"' --project=PROJECT \
  --format="value(protoPayload.status.code)" | sort -u
```

⚠ THE THREE THAT DO NOT WORK

`gcloud auth revoke` on a service account only deletes the local credential file and prints a warning. The token stays valid server-side. Google Cloud session-control settings apply to user identities, never to service accounts. And rotating the key without disabling the account leaves the window fully open.

ⓘ IF YOU CANNOT TAKE THE OUTAGE

Disabling a production service account for an hour is disruptive, which is exactly why teams skip it. The supported alternative is to remove the account's IAM bindings, since authorisation is evaluated per request, but allow for IAM propagation, which Google documents as taking up to seven minutes. **Either way, write down which one you chose and when. The closure gates will ask.**

THEN VERIFY

- Key gone from `keys list`
- No `CreateServiceAccountKey` in the window that you did not perform
- No further calls bearing the old `serviceAccountKeyName`
- Impersonation edges re-checked after re-enabling

AZURE · 1 OF 3

Service principals sign in somewhere else entirely.

The most common Azure investigation error is searching the sign-in logs, finding nothing, and concluding the credential was unused. **Non-human sign-ins are recorded in separate logs** from user sign-ins. If you are not explicitly querying the service principal and managed identity logs, you are looking at the wrong table.

SOURCE	DEFAULT	RETENTION	WHAT IT ANSWERS
Entra service principal sign-ins `AADServicePrincipalSignInLogs`	✓ On	`7 d free` `30 d P1/P2`	**The primary source for a leaked client secret.** Carries the caller IP, the resource, the result, and which credential was used.
Entra managed identity sign-ins `AADManagedIdentitySignInLogs`	✓ On	`7 / 30 d`	Separate again. Relevant when the compromised path runs through a VM or Function identity.
Entra audit logs	✓ On	`7 / 30 d`	Directory changes: new credentials on an app, new owners, consent grants. **Persistence lives here.**
Azure Activity Log	✓ On	`90 days`	Control-plane operations against subscriptions and resources.
Key Vault `AuditEvent`	✕ Off	`Sink-bound`	`SecretGet`, `SecretList` and friends. **Nothing is recorded until a diagnostic setting exists.**
Storage logs `StorageBlobLogs`	✕ Off	`Sink-bound`	The only way to see SAS usage. Filter `AuthenticationType == "SAS"`; correlate a specific token by `AuthenticationHash`.
Defender for Cloud	◑ Plan-dependent	`Varies`	Behavioural alerts on identity and resource abuse.

KQL · THE QUERIES TO RUN FIRST

```
// 1. Every sign-in by this app, and by WHICH credential.
//    ServicePrincipalCredentialKeyId separates the leaked
//    secret from the other secrets on the same app.
AADServicePrincipalSignInLogs
| where AppId == "<appId>"
| project TimeGenerated, IPAddress, ResourceDisplayName,
          ResultType, ServicePrincipalCredentialKeyId
| order by TimeGenerated asc

// 2. Was anything read out of Key Vault?
AzureDiagnostics
| where ResourceProvider == "MICROSOFT.KEYVAULT"
| where OperationName startswith "Secret"
| project TimeGenerated, OperationName, identity_claim_appid_g,
          CallerIPAddress, requestUri_s, httpStatusCode_d

// 3. Which SAS token, and what did it touch?
StorageBlobLogs
| where AuthenticationType == "SAS"
| summarize ops=count(), bytes=sum(ResponseBodySize),
          first=min(TimeGenerated), last=max(TimeGenerated)
          by AuthenticationHash, CallerIpAddress

// 4. Persistence. A credential or an owner added to the app
//    during the window survives everything you rotate.
AuditLogs
| where OperationName has_any ("Add service principal credentials",
                               "Add owner to application",
                               "Update application")
| project TimeGenerated, OperationName, InitiatedBy,
          TargetResources
```

⚠ SEVEN DAYS IS NOT AN INVESTIGATION WINDOW

On Entra ID Free, sign-in logs are kept for **seven days**, and diagnostic settings (the mechanism for exporting them anywhere durable) require P1 or P2. Upgrading mid-incident does not recover history: only data still inside the free window comes with you. If your exposure window is longer than seven days and you are on Free, state plainly that use cannot be confirmed or excluded.

ⓘ SAS CREATION IS INVISIBLE BY DESIGN

An account-key SAS is computed on the client with an HMAC. No call is made to Azure, so there is no creation event, anywhere, ever. You can only observe a SAS being used, and only if storage logging was already on.

READ THE ResultType

A run of failures is as informative as a success. In the service principal log:

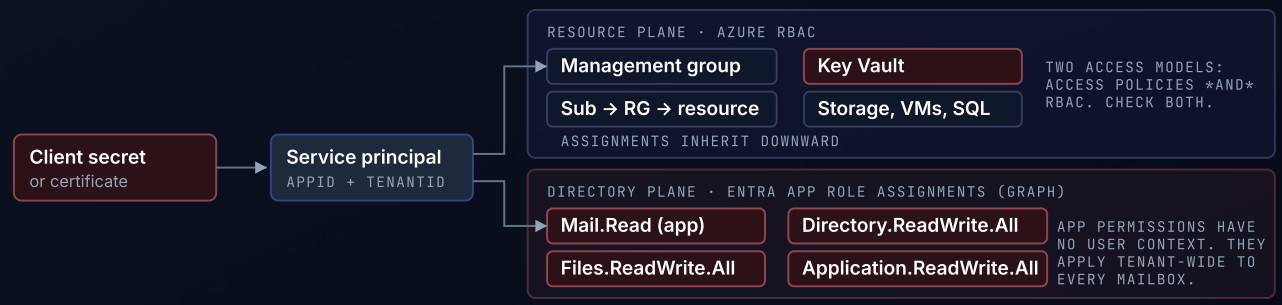
- `0`: success. The credential worked.
- `7000215`: invalid client secret. Someone tried a stale or guessed value.
- `700016`: application not found in the directory. **A leaked secret being replayed against the wrong tenant**, a strong signal that the value is circulating.
- `90002`: tenant not found. Same pattern, broader sweep.

AZURE · 2 OF 3

Azure RBAC and Graph permissions are two systems, and reading one understates the scope.

An application identity holds two unrelated permission sets. **Azure RBAC** governs resources; **Entra app role assignments** govern the directory and Microsoft 365 through Graph. Enumerating one and not the other is how a tenant-wide mailbox compromise gets reported as a storage incident.

FIGURE 10 · WHAT AN AZURE APP CREDENTIAL REACHES



Application.ReadWrite.All is an escalation primitive: it lets the app add credentials to any other app, including far more privileged ones.

ENUMERATE BOTH PLANES

```
# Resource plane: every scope, including inherited.
az role assignment list --assignee <appId> --all \
  --include-inherited --include-groups \
  -o table

# Directory plane: Graph app roles actually granted.
az ad app permission list-grants --id <appId>
az rest --method get --url "https://graph.microsoft.com/v1.0/
  servicePrincipals(appId='<appId>')/appRoleAssignments"

# Who else can add credentials to this app?
az ad app owner list --id <appId>

# Key Vault uses TWO models. Check whichever is in force.
az keyvault show --name <vault> \
  --query "{rbac:properties.enableRbacAuthorization,
            policies:properties.accessPolicies}"
```

Microsoft AI Research 2023 Cremit

THE SCOPE WAS WRITTEN ON THE TOKEN

A SAS URL published in a GitHub repository to share a training dataset was scoped to the **entire storage account** with **full control**.

EXPOSED	38 TB, including two employees' workstation backups
CONTENTS	Passwords to Microsoft services, secret keys, 30,000+ internal Teams messages from 359 employees
LIVE	Committed July 2020; expiry later extended to **October 2051**; reported June 2023

TAKEAWAY

The over-scope was legible in the query string from day one. Nobody read it, because Azure never surfaced the token for review.

Graph application permissions that change the severity

They carry no user context. If one of these is granted, the scope is the whole tenant, not one mailbox or one site.

PERMISSION	EFFECTIVE REACH	REPORT IT AS
`Application.ReadWrite.All`	Add credentials to any app registration	**Escalation to whatever the most privileged app can do**
`RoleManagement.ReadWrite.Directory`	Assign directory roles	**A path to Global Administrator**
`Directory.ReadWrite.All`	Read and write the whole directory	**Full user, group and device control**
`Mail.Read` · `Mail.ReadWrite`	Every mailbox in the tenant	**Organisation-wide mail access, notifiable in most regimes**
`Files.ReadWrite.All` · `Sites.FullControl.All`	All OneDrive and SharePoint content	**Organisation-wide document access**
`User.ReadWrite.All`	Modify user objects	**Includes attributes used for authentication decisions**

AZURE · 3 OF 3

You cannot revoke a SAS token. You can only invalidate what signed it.

App credentials are contained cleanly: delete the secret, disable the principal, done. Shared access signatures do not. A SAS is an HMAC over a set of claims, verified offline against a signing key, with no list of issued tokens and no revoke endpoint. **Your options depend entirely on which of three ways it was signed**, and the presence or absence of `si=` and `skoid=` in the query string tells you which.

SAS TYPE	RECOGNISE IT BY	HOW TO INVALIDATE IT	COLLATERAL DAMAGE
Ad-hoc account / service SAS	`no si=, no skoid=`	**Rotate the storage account key that signed it. There is no narrower option.**	**Every other SAS and every client using that key stops working instantly.**
Service SAS with stored access policy	`has si=<policy-id>`	**Delete or re-date the stored access policy on the container.**	**Only tokens bound to that policy. This is why policies exist.**
User delegation SAS	`has skoid= / sktid=`	`az storage account revoke-delegation-keys`	**All user delegation SAS on that account, but account keys are untouched.**

APPLICATION CREDENTIAL · THE CLEAN CASE

```
# 1. What credentials does this app have? Preserve the list;
#    one you did not create is attacker persistence.
az ad app credential list --id <appId> -o table

# 2. Remove only the leaked one, by keyId.
az ad app credential delete --id <appId> \
  --key-id <keyId>

# 3. If use is confirmed, disable the principal outright.
az ad sp update --id <appId> --set accountEnabled=false

# 4. Existing access tokens live up to ~60-90 min. Continuous
#    Access Evaluation shortens this for supporting resources,
#    do not assume it covers every API the app was calling.
```

SAS · THE ROTATION YOU HAVE TO PLAN

```
# Account keys come in pairs precisely so you can do this
# without an outage. Move consumers to key2, then rotate key1.
az storage account keys list -n <acct> -g <rg> -o table
# ...repoint every consumer to key2 and verify...
az storage account keys renew -n <acct> -g <rg> \
  --key key1

# Stored-access-policy case: surgical, no outage elsewhere.
az storage container policy delete -c <container> \
  -n <policy-id> --account-name <acct>

# User delegation case.
az storage account revoke-delegation-keys \
  -n <acct> -g <rg>
```

⚠ ROTATING AN ACCOUNT KEY IS A PRODUCTION EVENT

It invalidates every SAS ever minted from that key, plus every connection string embedding it. In a mature environment that is dozens of consumers you have no inventory for. **This is why SAS incidents drag on for days, and why the containment decision belongs to someone who can authorise the outage.**

VERIFY CONTAINMENT

- The `keyId` is gone from `az ad app credential list`
- That `ServicePrincipalCredentialKeyId` shows no sign-ins after T4
- That `AuthenticationHash` now returns `403` in `StorageBlobLogs`
- The Entra audit log shows no new app credentials, and owners are re-verified, since an owner re-mints at will

The consumers to find before rotating an account key

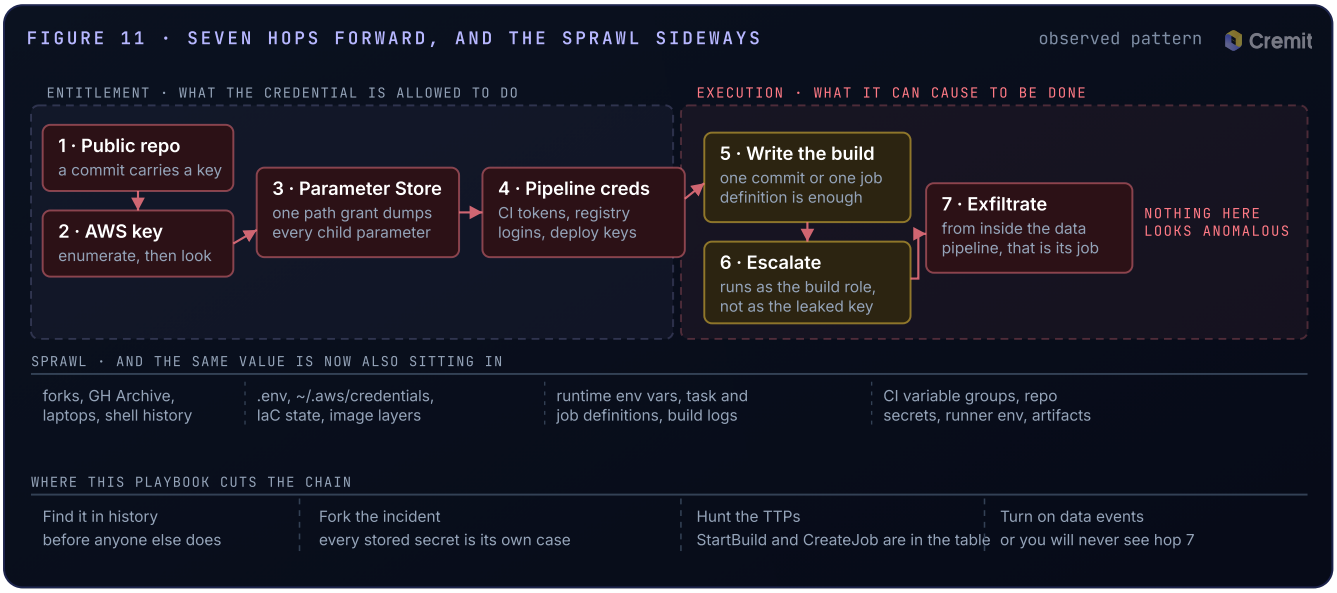
No API lists them, because a SAS leaves no server-side record. This is the search space; treat anything you cannot rule out as a consumer.

WHERE CONNECTION STRINGS HIDE	HOW TO FIND THEM	WHAT BREAKS IF YOU MISS IT
App settings, Key Vault references	`az webapp config appsettings list`	The app fails on its next storage call, not at deploy time
Data Factory / Synapse	Linked service definitions, per workspace	Pipelines fail on the next scheduled run, possibly overnight
Kubernetes secrets, CSI drivers	`kubectl get secrets -A`	Pods run until they restart, then crash-loop
CI/CD variables, IaC state	Variable groups, Terraform state and tfvars	The next deploy writes the old key straight back
Partners, and desktop tooling	Contracts and tickets; Storage Explorer profiles and analyst scripts	Your decision causes an outage for someone else, or a failure surfaces silently weeks later

WHAT HAPPENS AFTER THE PART THE PUBLIC WRITE-UPS DESCRIBE

Real attackers do not stop at the credential they found.

Published post-mortems tend to end at "a key was exposed and has been rotated." The engagements behind this playbook rarely end there. This is the path, hop by hop. **Note where it leaves IAM: from hop 5 onward the attacker stops asking for permissions and starts asking for execution**, and an entitlement model cannot see that coming.



Hop 3, the multiplier

TWO CALLS TURN ONE KEY INTO EVERY KEY

```
# Names first. Cheap, and it maps the whole tree.
aws ssm describe-parameters --query 'Parameters[].Name'

# The multiplier: --recursive walks every child of the prefix,
# --with-decryption returns SecureString plaintext. One grant
# on /prod/* is every credential under /prod/*.
aws ssm get-parameters-by-path --path /prod \
  --recursive --with-decryption

# Both are CloudTrail MANAGEMENT events, so unlike GCP and
# Azure this hop IS recorded by default. Hunt for a
# single principal reading many parameters in one minute.
```

⚠ THE GAP IN THE ENTITLEMENT MODEL

The permission graph computes what a credential is **permitted** to do. It does not compute what it can **cause** to be done. **Write access to a build definition is code execution as the build role.** An identity with no `iam:*` at all, but with a CI token in reach, ends up running with whatever that pipeline was trusted with. That is usually far more than the key itself ever had. `simulate-principal-policy` does not compute this.

Hops 5 and 6, in the providers' own verbs

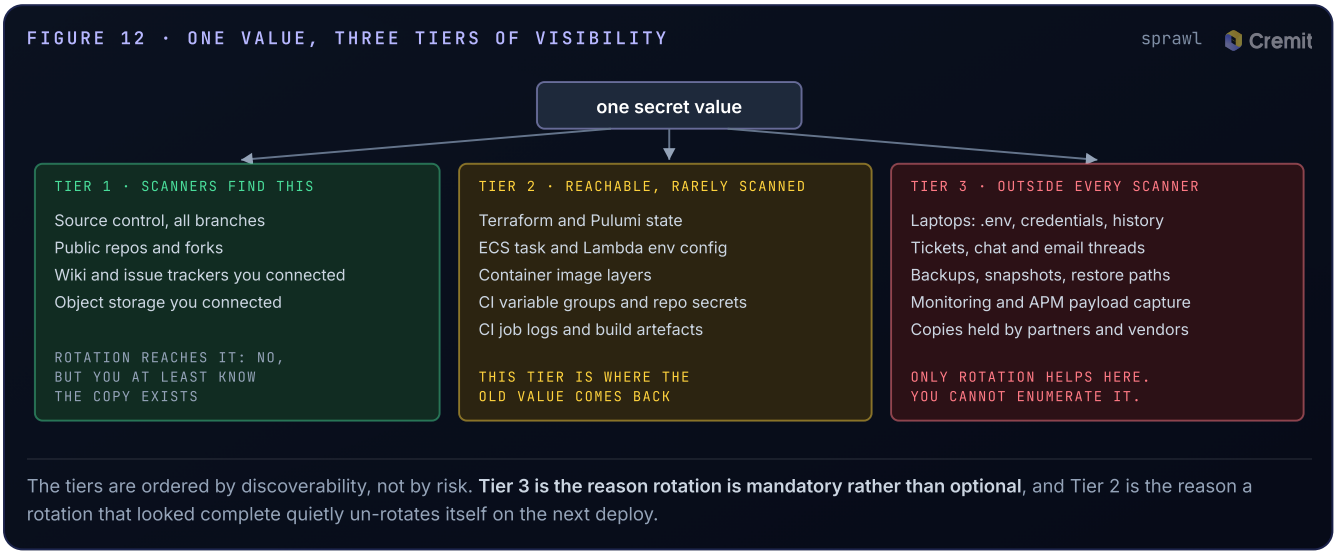
EXECUTION SURFACE	WHAT THE ATTACKER NEEDS	WHAT THEY GET, AND WHY IT IS QUIET
AWS CodeBuild	`iam:PassRole` + `codebuild:CreateProject` / `UpdateProject` + `StartBuild`	Arbitrary commands as the passed role. `StartBuild` from a build account is the most normal event there is
AWS Glue	`iam:PassRole` + `glue:CreateJob` / `UpdateJob` + `StartJobRun`	Code execution in the data platform, warehouse credentials already attached
Lambda	`iam:PassRole` + `lambda:CreateFunction` / `UpdateFunctionCode`	Persistent execution. `UpdateFunctionCode` leaves name and ARN unchanged
Source control	A write-scoped token, or a repo secret	The build runs on the next push. No cloud API call is involved at all until the build starts
CI provider	A CI API token from hop 3	Job definitions, runners and every other project's secrets

AWS's quarantine policy denies much of the middle column, but only for the quarantined user, and only until the attacker is executing as something else.

THE PART OF THE INCIDENT THAT OUTLIVES THE INTRUSION

Every hop copies the secret sideways.

The chain we just walked shows one value moving forward. What it does not show is that the same value has, at each hop, been copied into places nobody keeps an inventory of. **The attacker needs one copy. You have to find all of them.** That asymmetry is why Law 4 is a completeness problem, and why the rotation ledger keeps producing rows marked **unknown**.



The tiers are ordered by discoverability, not by risk. **Tier 3 is the reason rotation is mandatory rather than optional**, and Tier 2 is the reason a rotation that looked complete quietly un-rotates itself on the next deploy.

Where the same value ends up, and why rotation misses it

RESTING PLACE	HOW IT GOT THERE	WHY IT SURVIVES ROTATION	HOW TO FIND IT
IaC state	Written at `apply`, not by a human	**State is not code, so nobody greps it. The next `apply` can write the old value straight back**	`terraform state pull` · check the state bucket's own ACL while you are there
Task & function config	Injected at deploy as an env var	**The value lives in the definition, not the image or the repo**	`ecs describe-task-definition` · `lambda get-function-configuration`
Container image layers	`ARG`, `COPY`, or a build-time fetch	**Layers are immutable and already published. Deleting the tag does not delete the layer**	`docker history` · per-layer filesystem diff · registry retention policy
CI variables & repo secrets	Set once, often years ago	**No owner, no expiry, no review. Frequently duplicated across projects**	Provider API listing per org, project and environment
CI job logs	Echoed, or printed by a tool that did not mask it	**Logs outlive the job and the pipeline**	Search job logs for the value **and** for its masked form
Developer machines	`.env`, `~/.aws/credentials`, shell history	**Outside every scanner you own, and outside your change window**	**You cannot. This tier is the argument for rotation**
Tickets, chat, wiki	Pasted during an incident or onboarding	**Full-text searchable by anyone with a seat, indefinitely**	Search the way an attacker would. The Salesloft actors did exactly this
Backups & snapshots	Whatever was on the volume at the time	**A restore re-introduces the pre-rotation value without anyone noticing**	Inventory restore paths, not just backups. Ask what a restore would bring back
Monitoring & APM	Captured in request payloads or env dumps	**Retention is usually longer than your log retention**	Check payload and environment capture settings per agent

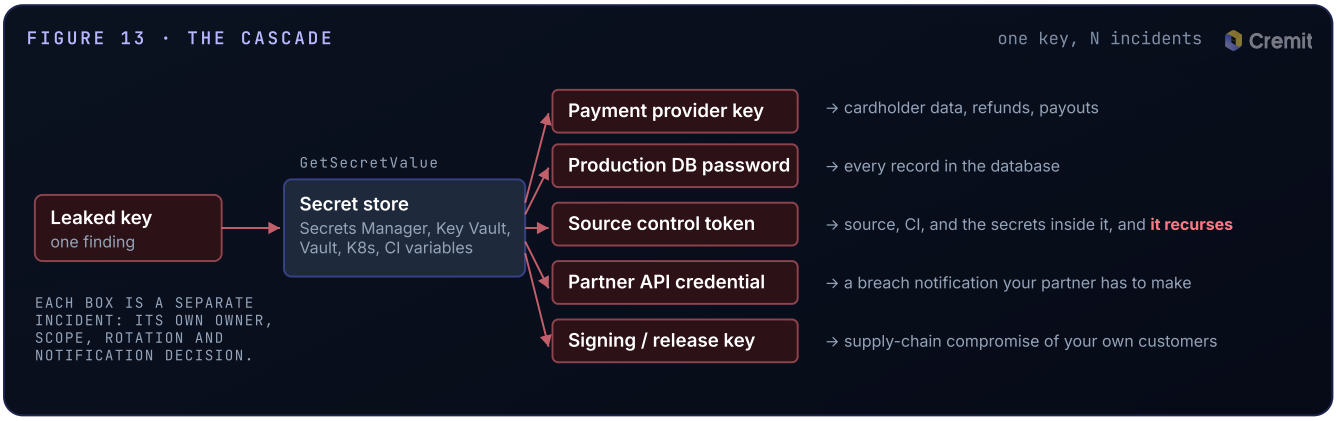
⚠ THE SPRAWL RULE

A credential's blast radius includes **every copy of it**, and the copy you did not know about is the one that ends the incident badly. Cloudflare missed four out of thousands. The Internet Archive missed one Zendesk token and was breached again inside a month. **Neither missed a copy they had listed; both missed a copy they had never enumerated.** The time to build the list of resting places is before you need it: during an incident you will only ever find the ones you already knew about.

SECRET STORES · 1 OF 2

One credential that reaches a vault is not one incident.

The moment the compromised identity can call `GetSecretValue`, every secret it could have read becomes its own incident with its own blast radius, its own owner, and its own rotation. **The correct response is to fork, not to widen.** And the first question, whether the attacker read them, is answerable on some platforms and structurally unanswerable on others.



Default logging of a secret read

The most important table in this report. Which row your platform sits in decides whether the question has an answer at all, and it was settled long before the incident.

STORE	READ OPERATION	LOGGED BY DEFAULT	WHERE IT LANDS, AND WHAT TO DO ABOUT IT
AWS Secrets Manager	`GetSecretValue`	✓ Yes	CloudTrail management event. 90 days in Event history; longer with a trail. **The good case.**
AWS SSM Parameter Store	`GetParameter(s)`	✓ Yes	Also a management event. Pair with `kms:Decrypt` for `SecureString`.
GCP Secret Manager	`AccessSecretVersion`	× No	`DATA_READ`. Data Access logging org-wide, or reads leave no trace.
Azure Key Vault	`SecretGet`	× No	Needs a diagnostic setting sending `AuditEvent` to Log Analytics. Off until you create one.
HashiCorp Vault	`kv read`	× No	Requires an enabled audit device. Values are HMAC'd, so you see which secret, not the secret.
Kubernetes Secrets	`get secret`	× No	Needs an API server audit policy at `Metadata` level or above. Also check who can read the etcd backups.
CI/CD variables	injection at build	× No	No per-read record exists. Scope is **every job that ran** in the window, which is what made Codecov and CircleCI so expensive.

Salesloft Drift

2025 Cremit

SECOND-ORDER REACH AS THE OBJECTIVE

Stolen OAuth tokens for a chat integration gave access to the Salesforce tenants of **700+ organisations** over ten days. The actors did not stop at the CRM data. They queried Cases, the support tickets, and searched the text specifically for **AWS access keys, Snowflake tokens, VPN credentials and passwords**, then used what they found to move into those environments.

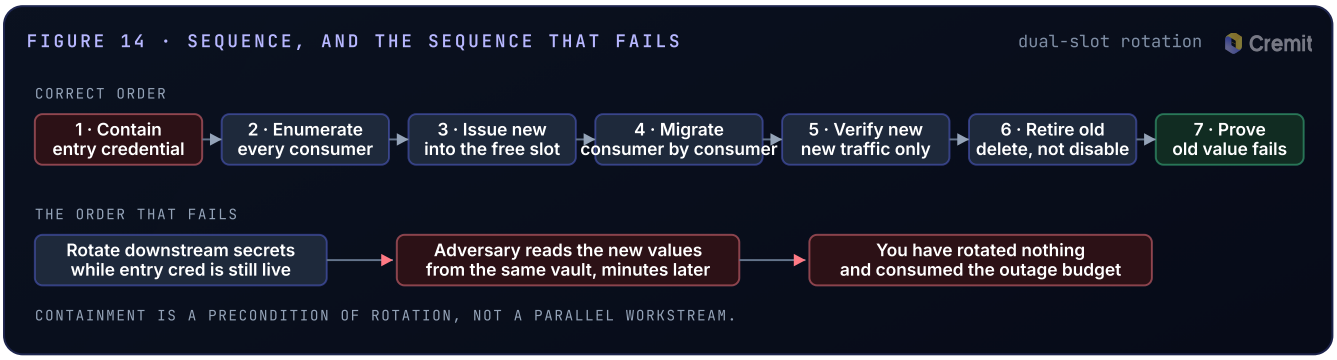
TAKEAWAY

Your ticketing system, your wiki and your chat history are secret stores. They have no `GetSecretValue` to audit, and the attackers already know to look there.

SECRET STORES · 2 OF 2

Rotation has an order, and an ending you have to prove.

Two failures account for most botched rotations: rotating downstream secrets while the adversary still holds the entry credential and reads the new values; and declaring rotation complete without a ledger.



Rotation patterns and their consequences

PATTERN	WHERE YOU MEET IT	CONSEQUENCE
Dual slot	AWS IAM (2 keys), Azure Storage (key1/key2), GCP SA keys	Zero downtime if you migrate before deleting
Overlapping validity	Most SaaS: multiple live tokens allowed	Same as dual slot; just track both
Single value	APIs that allow one active key; some webhooks	Plan a cutover window and announce it
Not individually revocable	Ad-hoc Azure SAS; anything signed by a shared key	Rotating the signer breaks every sibling. Build the consumer list first

CircleCI

2023 Cremit

WHEN "ROTATE EVERYTHING" IS THE ONLY CORRECT ANSWER

Malware on an engineer's laptop stole an SSO session for an identity that could mint production tokens. Customer variables, tokens and keys were exfiltrated: encrypted at rest, but the keys came from a running process. **Every customer was told to rotate every secret in the platform** and to audit two weeks of their own logs.

TAKEAWAY

When the store itself is breached, per-secret triage is not available. The organisations that recovered fastest already had an inventory of which secrets lived there and who consumed them.

The rotation ledger

One row per secret in scope. A rotation is complete when every row reaches the last column, not when the ticket is closed. **"Believed unused" is not a value this table accepts.**

SECRET	STORE	CONSUMERS FOUND	OWNER	NEW ISSUED	CONSUMERS MIGRATED	OLD DELETED	OLD VALUE PROVEN DEAD
`prod/db/app`	Secrets Manager	08	platform	✓	8 / 8	✓	401 confirmed
`stripe/live`	Secrets Manager	02	payments	✓	2 / 2	✓	401 confirmed
`partner/sftp`	Vault kv/	03	data	✓	2 / 3	–	× open
`legacy/etl`	CI variable	unknown	× unassigned	–	–	–	× open

The last two rows are the incident and what you report. The first two are paperwork.

FILLING THE LAST COLUMN

```
# From a recorded host, with the old value, once. The denial
# it leaves in the log IS the evidence.
aws sts get-caller-identity    # → InvalidClientTokenId
curl -so /dev/null -w '%{http_code}' \
  -H "Authorization: Bearer $OLD" "$URL"
#  401  ← this line and its UTC time go in the ledger
```

⚠ ONE DENIAL IS NOT ALWAYS ENOUGH

The old value can be refused while the sessions it minted still work. **The last column closes only when both are dead.**

SaaS KEYS

Four questions decide what a provider will tell you.

SaaS credentials have no CloudTrail. What you can establish is entirely determined by what the vendor chose to expose, and that varies from **excellent** to **nothing at all**. These four questions, run over the top twenty integrations, give the realistic investigation ceiling.

Q1

Can I read the credential's scope without using it?

If not, you cannot bound the blast radius except by assuming the maximum the integration ever needed.

Q2

Is there a per-request log, with source IP?

Aggregate usage counters tell you that something happened. Only per-request logs tell you what and from where.

Q3

Can I revoke this token alone?

Or does revoking it take down every sibling credential signed by the same key?

Q4

How long until revocation is real?

Cached tokens, edge propagation and refresh windows all extend how long the adversary keeps working.

CREDENTIAL	Q1: SCOPE READABLE?	Q2: PER-REQUEST LOG?	Q3/Q4: REVOCATION
GitHub classic PAT `ghp_…`	Yes. The `X-OAuth-Scopes` response header lists them	Org audit log; source IP only if IP logging is enabled on the plan	Delete the token. Immediate, and independent of other tokens.
GitHub fine-grained PAT `github_pat_…`	No. There is no introspection endpoint; `X-Accepted-GitHub-Permissions` tells you what an endpoint needs, not what you hold	Same as above	Delete the token
GitHub App installation `ghs_…`	Installation permissions are enumerable	Org audit log	Short-lived (about an hour); also revocable directly
Slack bot / user token `xoxb-` · `xoxp-`	`auth.test` for identity; granted scopes are returned by the Web API	Audit Logs API is Enterprise Grid only. On other plans, no usable trail exists.	Revoke the token or rotate the app credentials
Stripe `sk_live_` · `rk_live_`	Restricted keys declare their scope	Full request log with source IP in the dashboard. The only row here that answers per request.	Roll the key, with an optional grace period for migration
Model provider API key `sk-` · `sk-ant-`	Project or workspace scoping at best	Usage and spend, aggregated. An anomalous bill is often the only signal, which is what makes LLMjacking profitable.	Delete the key; immediate
Package registry token `npm_` · `pypi-`	Granular tokens declare scope; legacy ones do not	Publish events are visible; read use generally is not	Revoke, but treat any publish in the window as a supply-chain incident

xAI

2025 · Cremit

DETECTION WITHOUT AN OWNER

A staff member committed an `x.ai` API key to a public repository. A scanning vendor notified the developer on **2 March**. On **30 April** the key was still valid, and was only revoked after the vendor escalated directly to the security team. The exposure reached roughly **60 models**, including unreleased and private ones built on internal company data.

TAKEAWAY

The detection worked perfectly. Everything after it did not. Route findings to a team with a revocation SLA, never to the individual who committed them.

Shai-Hulud

2025 · Cremit

YOUR SCANNER, THEIR TOOLING

A self-replicating npm worm compromised maintainer tokens and ran `TruffleHog`, an ordinary open-source secret scanner, on build hosts to harvest AWS, GCP and Azure credentials, then propagated to further packages. The first wave hit 500+ package versions; the second, **796 packages with 20M+ weekly downloads**.

TAKEAWAY

The asymmetry comes from cadence. They scan continuously and you scan on a schedule. Closing that gap makes most of this class of incident a non-event.

EVERYTHING WORTH COLLECTING, ORDERED BY HOW FAST IT DISAPPEARS

Evidence decays at different speeds. That is the order to collect it in.

The evidence that answers the hardest questions usually has the shortest life, and some of it is destroyed by your own containment actions. Anything in Tier 1 you do not have within the hour, you will not have at all.

TIER	ARTEFACT	ANSWERS	HOW TO PULL IT, AND WHY IT DECAYS
1	**IAM key last-used** AWS	Latest confirmed use of the key	`iam get-access-key-last-used` **overwritten the instant you test the key yourself**
1	**The exposure artefact**	Where it leaked, and since when	Clone the repo, save the raw page, screenshot with URL and UTC clock visible. **The owner may delete or privatise it as soon as they are told**
1	**Dangling git objects**	Whether it was ever public in a deleted commit	`git fsck --unreachable --dangling` **a repo delete or GC takes this with it**
1	**Entra sign-in logs** AZURE	Whether the service principal signed in, from where	`AADServicePrincipalSignInLogs` **7 days on Free, and upgrading does not backfill**
1	**CI job logs**	Echoed secrets, and which jobs consumed them	Export the job log archive for the window. **Retention is often 30 days or fewer and rolls continuously**
1	**SaaS usage & billing view**	Anomalous volume, often the only signal	Screenshot the per-day usage graph before rotation resets the context
1	**Data Access logs** GCP	Whether secrets were read	`gcloud logging read` **30-day _Default retention, and only if it was already enabled**
2	**Entitlement snapshot** AWS GCP AZURE	The blast radius as it was	`get-account-authorization-details · analyze-iam-policy · role assignment list` **your own remediation changes it**
2	**CloudTrail slice** AWS	Every call made by the key	`cloudtrail lookup-events --lookup-attributes AttributeKey=AccessKeyId` **90 days, per Region: loop all Regions**
2	**GuardDuty / SCC / Defender**	Behavioural corroboration	`guardduty list-findings` **roughly 90 days**
2	**Key Vault audit** AZURE	Whether secrets were read	`AzureDiagnostics … MICROSOFT.KEYVAULT` **exists only if a diagnostic setting was configured**
2	**Storage / SAS usage** AZURE	Which SAS token, what it touched, how much left	`StorageBlobLogs … AuthenticationType == "SAS"` **correlate by AuthenticationHash**
2	**Credential inventory**	Attacker-created persistence	`iam get-credential-report · sa keys list · az ad app credential list` **compare against your own change record**
2	**SaaS audit log**	Clones, exports, permission changes	Org audit log API. **Plan-gated on several platforms; Slack requires Enterprise Grid**
3	**CloudTrail S3 trail** AWS	Long-window timeline	Query with Athena or ingest into CloudTrail Lake. **Durable; collect at leisure**
3	**Admin Activity** GCP	Persistence and escalation	`_Required bucket` **400 days, always on, free**
3	**Azure Activity Log** AZURE	Control-plane changes	`az monitor activity-log list` **90 days**
3	**Registry & image history**	Whether the secret shipped in an artefact	`docker history · per-layer diff, plus registry pull logs`

⚠ COLLECT THE ENTITLEMENT SNAPSHOT BEFORE YOU REMEDIATE

Every other row describes something an outside force decays. This one you destroy yourself: the moment you detach a policy, disable a principal or delete a binding, the record of what the credential could have reached is gone, and that is precisely the question regulators, customers and your own executives will ask. **Snapshot the authorisation state to a file first. It takes one command and ninety seconds.**

COPY, FILL IN THE VARIABLES, RUN

Its output lands in one case folder, sealed with one hash.

The order matters: the first two commands capture what the next one would destroy. Run it from a recorded host with your own credentials, never the compromised ones, and keep the session transcript.

1 · SETUP, THEN AWS IDENTITY AND ENTITLEMENT

```
#!/usr/bin/env bash. Your own credentials, never the leaked ones.
set -euo pipefail
CASE="case-$(date -u +%Y%m%dT%H%M%SZ)"
FROM="2026-06-01T00:00:00Z"          # start of the exposure window
KEY="AKIA..."                         # the leaked identifier
mkdir -p "$CASE"/{aws,gcp,azure,git,saas,notes}
exec > >(tee -a "$CASE/notes/collection.log") 2>&1
echo "op=$(whoami) host=$(hostname) egress=$(curl -s ifconfig.me)" | tee "$CASE/notes/chain-of-custody.txt"
# 1. FIRST. This is destroyed the moment anyone authenticates with the key.
aws iam get-access-key-last-used --access-key-id "$KEY" > "$CASE/aws/last-used.json"
aws sts get-access-key-info --access-key-id "$KEY" > "$CASE/aws/owner.json"
# 2. Entitlement snapshot, BEFORE remediating.
aws iam get-account-authorization-details --filter User Role Group LocalManagedPolicy > "$CASE/aws/authz.json"
aws iam generate-credential-report >/dev/null
aws iam get-credential-report --query Content --output text | base64 -d > "$CASE/aws/cred-report.csv"
```

2 · THE TRAIL, EVERY REGION · THEN GCP

```
aws cloudtrail describe-trails > "$CASE/aws/trails.json"
for r in $(aws ec2 describe-regions --query 'Regions[].RegionName' --output text); do
  aws cloudtrail lookup-events --region "$r" --start-time "$FROM" \
    --lookup-attributes AttributeKey=AccessKeyId,AttributeValue="$KEY" > "$CASE/aws/ct-$r.json" || true
done
P="prod-analytics-01"; ORG="<org-id>"; SA="etl@$P.iam.gserviceaccount.com"; KEYID="a1b2c3..."
gcloud asset analyze-iam-policy --organization=$ORG --identity=serviceAccount:$SA --format=json > "$CASE/gcp/iam.json"
gcloud iam service-accounts keys list --iam-account="$SA" --managed-by=user --format=json > "$CASE/gcp/sa-keys.json"
# Every call signed by THIS key.
Q="protoPayload.authenticationInfo.serviceAccountKeyName:\"$KEYID\""
gcloud logging read "$Q AND timestamp≥\"$FROM\"" --project="$P" --format=json > "$CASE/gcp/key-usage.json"
```

3 · AZURE AND GIT, THEN SEAL THE FOLDER

```
APPID="0000...0000"; WS="<workspace-id>"
az ad app credential list --id "$APPID" -o json > "$CASE/azure/app-credentials.json"
az role assignment list --assignee "$APPID" --all --include-inherited \
  --include-groups -o json > "$CASE/azure/rbac.json"
G="https://graph.microsoft.com/v1.0"; SP="servicePrincipals(appId='$APPID')"
az rest --method get --url "$G/$SP/appRoleAssignments" > "$CASE/azure/graph-roles.json"
# Sign-ins: run the KQL from the Azure evidence chapter and export the result here.
# Git: all history by content, plus unreachable objects.
git log --all --format='%H %aI %cI %ae' -S"$KEY" > "$CASE/git/hits.txt"
git fsck --unreachable --dangling --no-reflogs > "$CASE/git/dangling.txt"
# Seal it: hash every file, then the archive.
find "$CASE" -type f -exec shasum -a 256 {} \; | sort -k2 > "$CASE/MANIFEST.sha256"
tar -czf "$CASE.tar.gz" "$CASE" && shasum -a 256 "$CASE.tar.gz" | tee "$CASE.tar.gz.sha256"
```

ALSO RECORD, BY HAND

- Who discovered it, how, and the exact UTC time
- Every probe you ran: command, time, source IP, user agent
- Which containment option you chose where more than one existed, and who authorised it
- Any source you could not collect, and why. A documented gap is a finding; an undocumented one is a liability

ⓘ ADAPT BEFORE YOU RUN IT

A template, not a product. Region loops cost API calls and `lookup-events` is paginated and rate-limited, which is a poor thing to discover mid-incident. The script pulls the provider rows; the KQL exports, GuardDuty, CI and SaaS logs, the Azure activity log, registry history and screenshots are yours.

THE EXACT STRINGS, NOT THE GENERAL IDEA

Every answer here is a field in someone's log.

Knowing CloudTrail records the call is not the same as knowing which field in the JSON holds the attacker's account number. These are the paths that carry the answer.

AWS CloudTrail

FIELD PATH	WHAT IT SETTLES
`userIdentity.accessKeyId`	The `AKIA/ASIA` that signed the call
`…sessionContext.attributes .creationDate`	Session start; the value `aws:TokenIssueTime` compares against
`userIdentity.invokedBy`	Present when an AWS service made the call. **Absent on attacker traffic**
`requestParameters.roleArn responseElements. assumedRoleUser.arn`	On `AssumeRole`: the next hop and the session it produced
`requestParameters.secretId`	On `GetSecretValue`: **which secret** was read
`requestParameters. withDecryption`	On `GetParameter`: whether the plaintext was returned
`…createVolumePermission .add.items[].userId · .group`	**On `ModifySnapshotAttribute`: the AWS account ID they shared to, or `all` for public**
`SharedSnapshotCopyInitiated SharedSnapshotVolumeCreated`	**Logged in your account when they actually take the copy. Proof the exfiltration completed**
`errorCode`	Denials; the enumeration fingerprint

GCP Cloud Audit Logs

FIELD PATH (UNDER PROTOPAYLOAD)	WHAT IT SETTLES
`authenticationInfo .serviceAccountKeyName`	**Which key signed it.** `//iam.googleapis.com/projects/P/serviceAccounts/SA/keys/KEY_ID`
`….serviceAccountDelegationInfo[] ….principalEmail`	**The impersonation chain in order, and the identity the API saw**
`authorizationInfo[] .permission · .granted`	Per-permission allow or deny, not just the call result
`status.code`	Non-zero is a denial
`methodName`	e.g. `…SecretManagerService.AccessSecretVersion`
`requestMetadata.callerIp . callerSuppliedUserAgent`	Source. `gcloud` and client libraries differ

Two commands people miss

VAULT · FIND A SECRET YOU CANNOT SEE

```
# Audit devices HMAC every value, so you cannot grep the
# plaintext. Ask Vault for the HMAC of the leaked value and
# grep for that instead.
vault write sys/audit-hash/<device> \
  input="<leaked value>"
#  hash: hmac-sha256:9f2c...  →  grep the audit log for it
```

KUBERNETES · BLAST RADIUS IN ONE LINE

```
kubectl auth can-i --list \
  --as=system:serviceaccount:<ns>:<sa>
```

AZURE Decoding a SAS query string

Read left to right: the token states its own scope, its own expiry and whether you can revoke it without rotating the account key.

PARAM	MEANING	WHAT TO CONCLUDE
`ss`	Signed services	Account SAS only. `b q t f` = blob, queue, table, file
`srt`	Signed resource types	**Service, Container, Object. `sco` is account-wide, which is what made the Microsoft AI token so damaging. Service SAS uses `sr` instead: blob, container, directory**
`sp`	Permissions	**`r w d l a c u p x y i t f`. `rwdlacupx` is everything, including delete**
`st · se`	Start and expiry	**On an ad-hoc SAS this is the only limit that exists. Check the year before you assume it lapsed**
`sip · spr`	IP range and protocol	No `sip` means usable from anywhere. Present narrows the investigation considerably
`si`	Stored access policy id	**Present means you can revoke by deleting the policy, with no key rotation**
`sk*`	User delegation key fields (`skoid sktid skt ske sks skv`)	**Present means `az storage account revoke-delegation-keys` works and account keys stay untouched**
`sig`	The HMAC itself	Correlate usage by `AuthenticationHash` in `StorageBlobLogs`

No `si` and no `skoid`: the only revocation is rotating the account key, and everything signed by it dies too.

HOW YOU KNOW IT IS ACTUALLY OVER

“We rotated it” is a claim. These six gates are evidence.

Most repeat incidents are not new intrusions; they are the first incident, closed early. Every gate below must produce an artefact you could hand to an auditor. A gate that cannot be evidenced has not passed; it has been assumed.

FIGURE 15 · SIX GATES, ALL MUST PASS

closure Cremit

G1	G2	G3	G4	G5	G6
Old value dead a logged denial, with a timestamp	**Ledger complete** no “unknown” rows, no unassigned owner	**No persistence** credential inventory diffed vs change record	**Surfaces mapped** including the ones you cannot remove	**Gaps declared** every unanswerable question, in writing	**Radius re-measured** escalation during the incident changed it

Any gate failing sends you back to step 04 of the triage tree. There is no partial close.

Cloudflare

2023 Cremit

GATE 2, FAILED ON FOUR ROWS

Thousands of leaked credentials were rotated after the Okta compromise. Four were not: one service token and three service accounts, skipped because **“mistakenly it was believed they were unused.”**

Compromise at the identity provider on **18 Oct**. The threat actor began probing with the four survivors on **14 Nov**, and was detected on **23 Nov**, by which point access had reached Atlassian, Jira and source control.

TAKEAWAY

Twenty-seven days between an incomplete close and a second intrusion. “Believed unused” is a hypothesis; a ledger wants an observation.

Internet Archive

2024 Cremit

THE SAME GATE, TWICE IN A MONTH

A GitLab config file holding an auth token had been reachable since at least December 2022, leading to a breach of **31 million** accounts. Weeks later they were breached again, through **a Zendesk token exposed in that first incident and never rotated**, exposing 800,000+ support tickets.

TAKEAWAY

The second incident was not bad luck; it was the first incident's Gate 2 being closed late.

The five numbers to report

METRIC	DEFINITION	WHY THIS ONE
Exposure window	`T1 → T4`	The number bounding what must be assumed compromised
Time to detect	`T1 → T3`	The interval you can actually improve. The rest is process
Time to contain	`T3 → T4`	Tests the runbook, not the tools
Adversary dwell	`T2 → T4`	A range when logs cannot pin T2. Never zero
Rotation completeness	`rotated / in scope`	**Below 100% is an open incident, whatever the ticket says**

⚠ LANGUAGE DISCIPLINE IN THE WRITE-UP

“No evidence of misuse” and **“evidence of no misuse”** are different findings, and only one is usually available to you. If Data Access logging was off, if sign-in retention had rolled, if storage logging was never on, write that sentence explicitly, in the executive summary, not in an appendix. **Every non-engineer will otherwise read the first phrase as the second.**

The executive summary, in the order it should be read

Six sentences. Whoever reads only this should be able to decide with it, and should not be able to mistake an unanswered question for a resolved one.

1. **What was exposed.** Credential class, the identity, and the window in dates. Not “an API key”.
2. **What it could reach.** The blast radius in terms of data and systems, not IAM policy names.
3. **What we can and cannot say about use.** Confirmed activity, and the questions your logs could not answer.
4. **What we did.** Containment and rotation, and who authorised any option declined for availability.
5. **Notification position.** Regimes assessed, the conclusion for each, and the clock you are working to.
6. **What changes now.** The one logging or ownership gap exposed, with an owner and a date. One beats twelve.

EVERY NON-OBVIOUS CLAIM

Primary sources, claim by claim

Checked **25 August 2026**. Provider behaviour is the part most likely to have moved since, so re-read their documentation before relying on a containment step. A claim that is not here should not have been in the report.

AWS

- Quarantine deny list, and its stated purpose · AWS managed policy reference
- Leaving the policy attached until remediation completes · AWS re:Post
- IAM unique ID prefixes · IAM identifiers reference
- Session revocation, `aws:TokenIssueTime`, propagation, service-linked roles · Revoking IAM role temporary credentials
- Event history: 90 days, management events, per Region · Working with CloudTrail event history
- `GetSecretValue` recorded by default · Logging Secrets Manager events
- `GetParameter` is a read-only management event · Logging management events
- `get-parameters-by-path` flags · AWS CLI reference
- Snapshot sharing as exfiltration; victim-side events · Datadog Security Labs, Stratus Red Team, Elastic

GCP, Azure, and the rest

- SA access tokens cannot be revoked; disable the account for 60 minutes · Google Cloud, mitigating compromised gcloud OAuth tokens
- `AccessSecretVersion` is `DATA_READ`, off by default · Secret Manager audit logging
- Retention: 400 days fixed, 30 days configurable · Cloud Logging quotas
- Entra sign-in retention, and that upgrading is not retroactive · Microsoft Entra data retention
- Key Vault logs nothing without a diagnostic setting · Enable Key Vault logging
- A SAS cannot be revoked individually; creation is untracked · MSRC advisory, Wiz research
- SAS parameters and what each implies · Create a service SAS
- Fine-grained PATs have no introspection · GitHub REST docs, community discussion 156115
- GitHub token formats end in a CRC32 checksum · GitHub Engineering
- Force-pushed commits stay fetchable · Truffle Security, Neodyme
- `sys/audit-hash` returns the HMAC as logged · HashiCorp Vault
- `kubectl auth can-i --list --as=` · Kubernetes authorization

Incidents

- **Uber** 2016 · Breaches.cloud, NBC News
- **Codecov** 2021 · Codecov post-mortem, GitGuardian
- **Heroku / Travis** 2022 · GitHub security blog
- **Toyota** 2022 · BleepingComputer, The Register
- **CircleCI** 2023 · CircleCI incident report, TechCrunch
- **Microsoft AI** 2023 · Wiz research, MSRC
- **EleKtra-Leak** 2023 · Unit 42
- **Cloudflare** 2023 · Cloudflare incident report
- **Mercedes-Benz** 2024 · RedHunt Labs, BleepingComputer
- **Sisense** 2024 · Krebs on Security, TechCrunch
- **Internet Archive** 2024 · BleepingComputer, Security Affairs
- **LLMjacking** 2024 · Sysdig
- **xAI** 2025 · Krebs on Security
- **Salesloft Drift** 2025 · AppOmni, Anomali
- **Shai-Hulud** 2025 · Unit 42, Datadog Security Labs

Our own analysis, not a vendor statement

- **The discovery table and curve.** Standard Poisson first arrival, computed from the formula printed on the page. The four rates are chosen, not measured, and the page says so
- **Deny list 28 → 89 → 99, nothing removed, and beyond S3 reads no data-access action denied in any version.** Our diff of the three published policies. AWS edits these in place, so re-run it
- **The actions quarantine leaves open.** Read out of the V3 JSON. Not an AWS statement
- **The account ID inside an access key ID.** Method from Aidan Steele, Tal Be'ery and Truffle Security; run before publishing, so the worked example is real output
- **The seven-hop chain and the sprawl tiering.** An observed pattern, said as such on the page. The primitives are documented individually; the sequence is first-party

Timing and the regulatory clock

- First abuse of a planted key at ~1 minute · Comparitech honeypot
- Quarantine at 2 min, actor 4 min after that, 400+ calls in 7 · Unit 42
- GDPR Art. 33 · Regulation (EU) 2016/679 · NIS2 Art. 23 · Directive (EU) 2022/2555 · SEC Item 1.05 · 2023 disclosure rules

ⓘ CLAIMS THIS REPORT DELIBERATELY DOES NOT MAKE

That anyone named was negligent. Each case illustrates a failure mode common across the industry. **That the "X% of repositories contain a secret" kind of market statistic holds**: vendor telemetry is not reproducible and dates badly. **That any product prevents these incidents. That LLMjacking has a dollar figure** beyond "tens of thousands per day": the larger numbers are projections, not invoices.

No customer of ours is named anywhere in this report, and no leaked key appears in it; both keys printed are public examples.

ON THE KOREAN EDITION

A Korean edition carries the same claims, figures and commands, and adds **Article 34 of Korea's PIPA** to the regulatory table.

FROM ONE INCIDENT TO A STANDING STATE

What the platform carries, and what stays yours

The incidents in this report are almost all **failures of sustaining, not of knowing**. Nothing in the previous thirty-two pages requires a vendor: an inventory, an owner, provider-accurate containment, and a ledger. **Cremit Platform** exists because those four are difficult to sustain by hand across every repository, bucket and workspace an engineering organisation touches.

PHASE	WHAT THE PLATFORM DOES	WHAT STAYS YOURS
02 · Scope exposure	Continuous scanning of GitHub and GitHub Packages, GitLab, Bitbucket, AWS S3, Confluence, Jira, Notion, Slack and Google Drive, across **commit history**, not just the current head. The Salesloft actors went looking in tickets and chat; those are connectors here.	Which organisations and repositories are in scope, and any surface outside those connectors
01/04 · Validity	**Validity verification.** A finding carries a verification status that is re-checked over time, so the question is "this credential authenticates" rather than "this matched a pattern". That distinction is what separates the xAI case from a closed ticket.	Confirming the account is yours, and the safe-probe discipline
03 · Blast radius	**Permission Analyzer** for AWS and GCP: connect a role or a service account key and read back the effective permission set, computed rather than assembled by hand.	Azure; resource-policy edge cases; and the judgement about what those permissions mean for your data
05 · Contain	Findings promote to **Incidents** with an assignee, status and full history, plus alarms and notifications: the owner and the clock that the xAI timeline never had.	The revocation itself. Cremit Platform does not hold credentials that can rotate your secrets
06 · Prove closure	Verification status flips when a credential dies, and the **inventory** tracks which secrets are live and which are publicly reachable: so the ledger is maintained rather than assembled during an incident.	Ledger rows for consumers no connector can see
Detection quality	Detection rules live in Git as the source of truth and sync to the platform, with false-positive benchmarking. A rule change is a reviewable commit.	Deciding what counts as a finding in your environment

ⓘ WHAT NO SCANNER FIXES

Law 5 and Law 6 are configuration decisions, and they have to be made before the incident. Whether GCP Data Access logging is on, whether Key Vault has a diagnostic setting, whether your CloudTrail captures data events, whether you can survive disabling a service account for sixty minutes: no tool retroactively creates the evidence, and none will take the outage decision for you. If you read one page of this document twice, make it the secret-store comparison.

THIS WEEK, WITHOUT BUYING ANYTHING

- Answer the secret-store comparison for your own platforms. Write the answers down.
- Name an owner and a revocation SLA for secret findings, not the committer.
- Run the collection script once, while nothing is on fire.
- Pull your ten oldest long-lived credentials and answer, for each, who uses it.

ON INTERNAL DISTRIBUTION

Cremit Platform finds, verifies and governs non-human credentials. This report is part of the Cremit Research incident response series.

cremit.io · ben@cremit.io

THE THREE INTERVALS

T1 → T3

Time to detect. Shortened only by continuous coverage of every surface a secret can land on. Toyota lost five years here; xAI lost sixty days after detection.

T3 → T4

Time to contain. Shortened by a named owner and provider-accurate runbooks: the 60-minute wait, the session deny, the SAS decision, all settled in advance.

T4 → T6

Time to actually close. Shortened by an inventory. Cloudflare and the Internet Archive both lost this one to four words: **believed to be unused**.

Cremit

CREMIT RESEARCH · INCIDENT RESPONSE SERIES

One key. One identity. Everything it reaches.

An exposed credential is not an incident about a string. It is an incident about everything that string could become, and about how much of that you can still prove.

PUBLISHED 4 September 2026
SOURCES Appendix C, page 32, lists them all
CONTACT ben@cremit.io