GRC ENGINEERING

A GRC Engineering Pipeline, Built in Public

From "it works" to "a stranger can verify it" — six weeks, one system, every claim runnable

By David Kramer, RHCA, CISSP · SevenBelow LLC Published July 2026 Version 1.0

CASE STUDY

The one-sentence version

I built a pipeline that takes an AWS resource from *"it works"* to *"a stranger can verify it satisfies NIST 800-53 without talking to me"* — and every claim in this article is checkable by running a command.

```
git clone https://github.com/sevenbelowllc/grc-engineering-club
cd grc-engineering-club && ./verify-pipeline.sh
```

Latest run: **14 checks, 14 passed, 0 skipped** (transcript). Run it yourself and the script verifies whatever your toolchain allows and names every check it cannot run. The ceiling for anyone but me is **12 passed, 1 skipped**: the two vault checks read a private Object Lock bucket, and an unreadable vault is reported as one `skipped` check — the verdict says `INCOMPLETE`, because a check that did not run is not a check that passed. The Debian transcript shows exactly that run.

Source: github.com/sevenbelowllc/grc-engineering-club

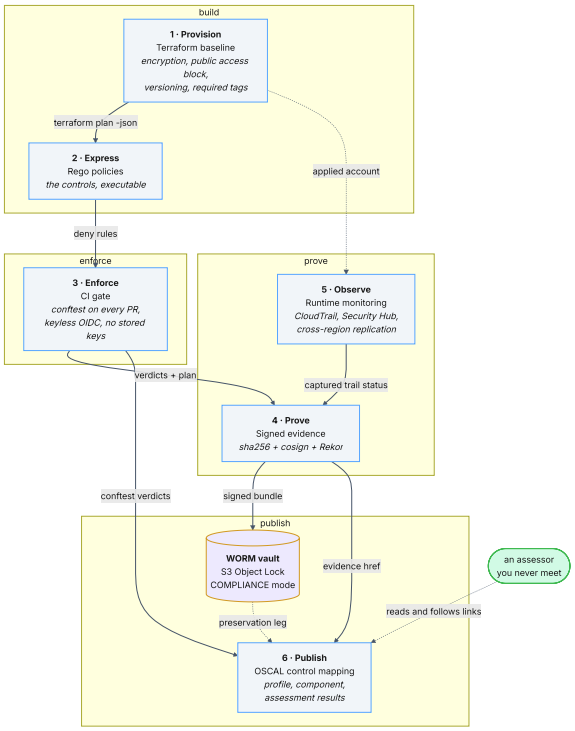
Why this exists

Most compliance evidence is a screenshot in a shared drive. It proves nothing in particular: not that the setting was real, not that it stayed real, not that nobody edited the PNG. The whole apparatus rests on an auditor's willingness to believe the person who produced it, which is why audit season feels like a trust exercise rather than an engineering one.

The alternative is not "better screenshots." It is treating control state as an artifact — something built by a pipeline, signed by a machine identity, recorded in a public log, stored where it cannot be deleted, and published in a format an assessor's tooling can read. That is a supply-chain problem, and the supply-chain tooling already exists.

This is a working, end-to-end demonstration of that idea, built over six weeks in a public repository.

The pipeline



Stage	What it does	The thing that makes it real
1 · Provision	Terraform baseline satisfying SC-28, AC-3, CM-6	Provider `default_tags` makes the tagging control impossible to forget on a new resource
2 · Express	Rego policies that read a Terraform plan	Rules match **by reference, not by value** — the bucket's name doesn't exist yet
3 · Enforce	GitHub Actions gate, required status check	A violation never reaches the account, because the merge is blocked
4 · Prove	SHA-256 + keyless cosign + Rekor + S3 Object Lock	Four separate properties, four separate mechanisms
5 · Observe	CloudTrail, Security Hub, cross-region replication	The *running* account, which is a different claim from the plan's intent

Stage	What it does	The thing that makes it real
6 · Publish	OSCAL profile, component, assessment plan and results	An assessor follows links instead of scheduling a call

The whole thing runs on keyless GitHub OIDC. There are no stored AWS credentials anywhere in the repository or its secrets.

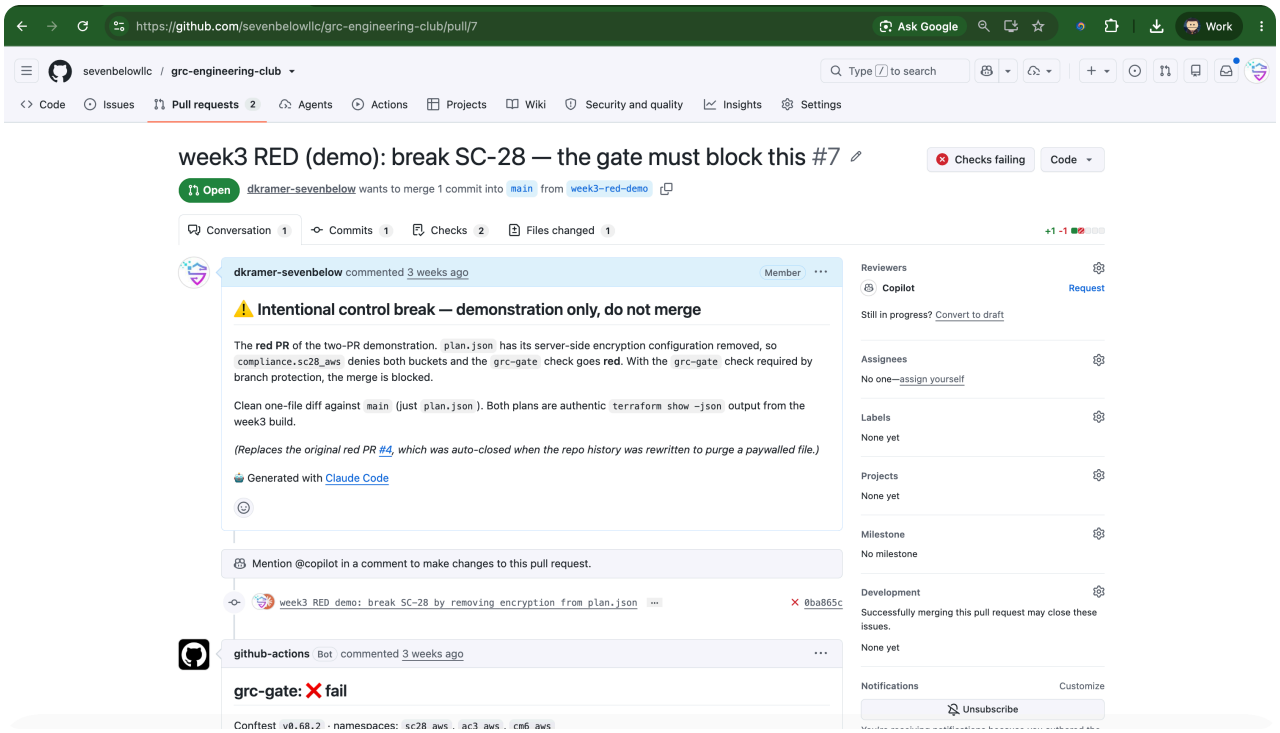
Proof

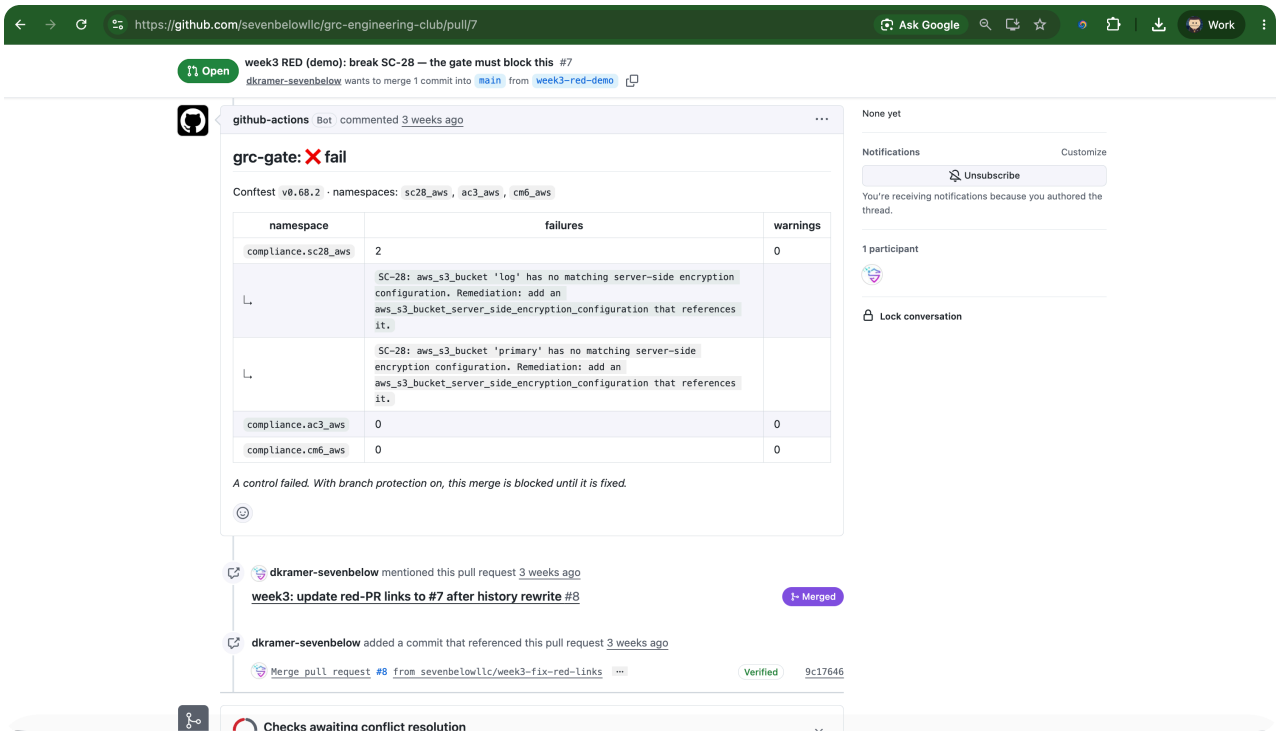
Evidence over adjectives. Every row links to something that ran.

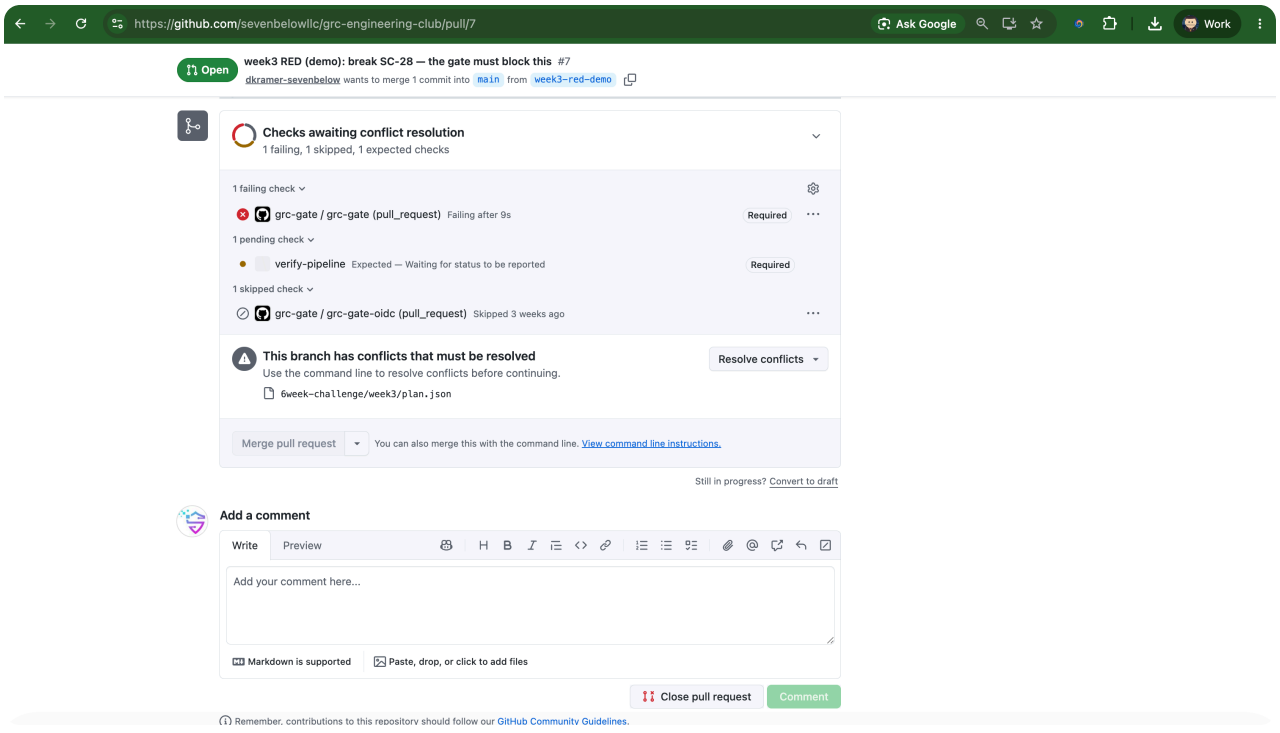
Claim	Proof
The policies are unit-tested	`opa test policies/ -v` → **PASS: 6/6**
A compliant change merges	**PR #3** — green run, all three namespaces pass
A non-compliant change **cannot** merge	**PR #7** — SC-28 denies both buckets, exit 1, merge blocked
Evidence survives the failure	The red run still signed and uploaded its bundle — `if: always()` in `grc-gate.yml`
The evidence is intact, authentic and timely	`verify-evidence.sh` → **CHAIN INTACT**, signed by the workflow in PR #9
Tampering is detected	The same script on a modified bundle → **FAIL**
The evidence cannot be deleted	Hard delete with **admin** credentials → *"Access Denied because object protected by object lock"*

Claim	Proof
The OSCAL is schema-valid	`trestle validate -a` → **5 documents VALID**
An assessor can traverse it	`./traverse.sh` → **4/4 controls** walked from profile to verified evidence
The converter fails when it should	Three guard cases, exit 3, nothing written
The verifier itself is gated	`verify-pipeline` is a **required status check** on `main` — run 30605526989 passed the fourteen-check verifier with zero skips, against live AWS

The row that matters most, seen rather than linked — PR #7, the intentional SC-28 break, as a reviewer sees it:




https://github.com/sevenbelowllc/grc-engineering-club/pull/7
Ask Google
Work
Open
week3 RED (demo): break SC-28 — the gate must block this #7
dkramer-sevenbelow wants to merge 1 commit into main from week3-red-demo
github-actions Bot commented 3 weeks ago
grc-gate: ❌ fail
Conftest v0.68.2 · namespaces: sc28_aws, ac3_aws, cm6_aws
namespace
failures
warnings
compliance.sc28_aws
2
0
SC-28: aws_s3_bucket 'log' has no matching server-side encryption configuration. Remediation: add an aws_s3_bucket_server_side_encryption_configuration that references it.
SC-28: aws_s3_bucket 'primary' has no matching server-side encryption configuration. Remediation: add an aws_s3_bucket_server_side_encryption_configuration that references it.
compliance.ac3_aws
0
0
compliance.cm6_aws
0
0
A control failed. With branch protection on, this merge is blocked until it is fixed.
dkramer-sevenbelow mentioned this pull request 3 weeks ago
week3: update red-PR links to #7 after history rewrite #8
Merged
dkramer-sevenbelow added a commit that referenced this pull request 3 weeks ago
Merge pull request #8 from sevenbelowllc/week3-fix-red-links
Verified
9c17646
Checks awaiting conflict resolution
None yet
Notifications
Customize
Unsubscribe
You're receiving notifications because you authored the thread.
1 participant
Lock conversation


https://github.com/sevenbelowllc/grc-engineering-club/pull/7
Ask Google
Work
Open
week3 RED (demo): break SC-28 — the gate must block this #7
dkramer-sevenbelow wants to merge 1 commit into main from week3-red-demo
Checks awaiting conflict resolution
1 failing, 1 skipped, 1 expected checks
1 failing check
grc-gate / grc-gate (pull_request) Failing after 9s
Required
1 pending check
verify-pipeline Expected — Waiting for status to be reported
Required
1 skipped check
grc-gate / grc-gate-oidc (pull_request) Skipped 3 weeks ago
This branch has conflicts that must be resolved
Use the command line to resolve conflicts before continuing.
Resolve conflicts
6week-challenge/week3/plan.json
Merge pull request
You can also merge this with the command line. View command line instructions.
Still in progress? Convert to draft
Add a comment
Write
Preview
Add your comment here...
Markdown is supported
Paste, drop, or click to add files
Close pull request
Comment
Remember, contributions to this repository should follow our GitHub Community Guidelines.

Three decisions worth stealing

1. Match by reference, not by value

The obvious way to write *"every bucket must be encrypted"* is to compare the encryption resource's `bucket` field to the bucket's name. It does not work.

At plan time the bucket name is `grc-challenge-dev-data-${random_id.suffix.hex}` — a value that does not exist yet. A name-matching rule cannot evaluate the plan at all, so it silently passes, which is the worst available outcome: a green gate that has checked nothing.

The plan does contain something stable. Under `configuration`, every resource records the *symbolic addresses* it references:

```
# The encryption resource records "aws_s3_bucket.primary.id" — an address, not a
# name. That exists at plan time. The name does not.
encryption_references(bucket_addr) if {
    some r in config_resources
    r.type == "aws_s3_bucket_server_side_encryption_configuration"
    some ref in r.expressions.bucket.references
    references_bucket(ref, bucket_addr)
}

deny contains msg if {
    some bucket in config_resources
    bucket.type == "aws_s3_bucket"
    not encryption_references(sprintf("aws_s3_bucket.%s", [bucket.name]))
    msg := sprintf("SC-28: aws_s3_bucket '%s' has no matching server-side encryption confi
}
```

Matching on the address works before apply, works at any module depth, and works regardless of what the resource ends up being called.

The general lesson: **policy-as-code that runs before apply has to reason about intent, not state.** Most examples on the internet quietly assume otherwise, and quietly pass.

2. The control mapping lives in the OSCAL, not in the converter

Week six's centrepiece is a converter that turns

```
{ "namespace": "compliance.sc28_aws", "successes": 1 }
```

into

```
{ "target-id": "sc-28_obj", "status": { "state": "satisfied" } }
```

— the shape an assessor's tooling consumes. Somebody has to carry the verdict across that gap. If that somebody is a person, they do it again every time the gate runs, and the moment they stop, the mapping goes stale without anyone noticing.

The tempting design is a dictionary in the converter mapping package names to control IDs. I deliberately didn't write one. That mapping lives in the OSCAL component definition, as a `policy-package` prop on each `implemented-requirement`, because that is the document whose *job* is to say how a control is implemented. The converter reads it back out at conversion time:

```
# The entire mapping logic.
packages = [p["value"] for p in req.get("props", [])
            if p.get("name") == "policy-package"]
```

Consequences: add a control to the component definition and its verdicts start converting, with no change to the converter. Rename a Rego package without updating the component and the build fails rather than silently dropping a control. And the converter cannot claim a control the component does not implement, because it has no independent source for the list.

A hardcoded table would have been a third place the truth lives, and the third place is always the one nobody updates.

3. An absent verdict is not a passing verdict

The guard I am proudest of runs in the direction nobody checks. Before writing anything, the converter verifies that every policy package the component *claims* gates a control actually appears in the policy engine's output:

```
mapping error: the component definition claims these policy packages gate a
control, but conftest reported no result for them:
  compliance.sc28_aws  (control sc-28)
An absent verdict is not a passing verdict. Either the gate did not run the
policy, or the component definition is claiming a rule that no longer exists.
```

Here is the failure it prevents. A policy file gets renamed, moved, or picks up a syntax error that stops it loading. Conftest returns one fewer entry and **exit code 0**. The gate is green. The generated assessment-results document is schema-valid. It simply no longer mentions SC-28 — and nothing, anywhere, turns red.

That is how automated compliance rots: not with a failure, but with a silence. Checking for the silence costs eight lines of Python.

The same principle earned its keep a second time, in an unrelated subsystem. `verify-pipeline.sh` exits `0` when checks are *skipped* — right for a laptop with a partial toolchain, wrong for CI, where a skip means an install step broke. A CI job that trusted the exit code alone would go green having run six checks instead of fourteen. So the job parses the summary line and fails the build on a non-zero skip count. Two guards, two subsystems, one failure mode: a green signal that quietly covers less than it appears to.

The part I would not automate away

Three of the four controls are enforced at plan time. **AU-3 is not, and that is deliberate.**

A Terraform plan can show that a CloudTrail trail *will be created* with the right arguments. It cannot show that the trail is *delivering records*, because delivery is a runtime property of a system that does not exist yet at plan time. Writing a plan-time rule for AU-3 would have produced a fourth green check that attests to nothing — and a fourth green check is worth less than an honest gap, because it spends credibility rather than earning it.

So AU-3's evidence is captured `get-trail-status` output from the running account, showing `IsLogging: true` with recent log and digest delivery. That is a **narrower** claim than the plan-time controls make — true of one moment, not of every future pull request — and a **stronger** one, because it is about the system rather than about the intent.

The OSCAL says this in the model rather than in a footnote: the assessment plan carries two `control-selections`, one per method, and the difference between them *is* the assurance boundary of the pipeline.

What this does not prove

An assessor's first question about any automated control is *"what does it miss?"* The answer is the same whoever produces it. The only variable is whether it came from the builder or from the auditor.

So the repository ships an **assurance boundary document** listing nine limits, each with what would close it. The four that matter most:

Plan-time is not runtime. The gate proves no *merged change* intends to violate a control. It does not prove the control is in force in the account right now.

Drift is mostly invisible. A change made in the console is recorded by CloudTrail — forensically recoverable, which is not the same as the control holding.

The chain proves custody, not correctness. If `terraform plan` had produced a wrong answer, the pipeline would have signed the wrong answer faithfully, and

every verification would still pass.

Thirty days of Object Lock is a demo value. COMPLIANCE mode is real — a hard delete with admin credentials is refused — but SEC 17a-4 retention is measured in years, and requires a designated third party and an audited process besides.

What I would build next

This pipeline proves controls for **one account that already exists**. The interesting problem is the one before it: standing up a client's foundation so these controls are true from the first `apply`.

Concretely — an interactively-bootstrapped AWS Organizations landing zone. Admin, NonProd and Prod accounts; OUs and SCPs; an org-level CloudTrail, a Config aggregator, Security Hub and GuardDuty enabled at the organization; Identity Center; and Terraform state bootstrapped before any of it — all driven by a single idempotent shell script an engineer runs once per client.

I scoped it during this build and **deliberately did not start it**, for reasons worth stating because the reasoning is the transferable part:

It would orphan the evidence. Everything captured across the six weeks — the signed bundle, the plan, twelve Security Hub findings, a 53-object replica listing — was captured against a single-account topology. Rebuilding as multi-account invalidates all of it to re-prove controls that are already proven.

The iteration loop is irreversible. Closing an AWS member account triggers a 90-day suspended state; root email addresses must be globally unique across all of AWS and cannot be reused on a retry; an organization with member accounts cannot be deleted. Every test run burns addresses and leaves 90-day residue.

It is 40–80 hours to a client-ready standard. There were five days.

Knowing which work *not* to start is most of the job.

The narrower next steps, in order: run the same Rego rules against live state on a schedule so plan-time and runtime verdicts can be compared; generate AWS Config rules from the same Rego source so the two checks cannot disagree by accident; and capture runtime evidence from a scheduled workflow rather than by hand, so every signature in the chain is a workflow signature rather than a person's.

What actually clicked

Compliance evidence is a supply-chain problem, and the tooling already exists. Cosign, Rekor and S3 Object Lock were built for software artifacts, not audits. But the four properties they provide — integrity, authenticity, timeliness, preservation — are precisely the four an auditor is trying to establish about a screenshot, and are precisely the four a screenshot cannot provide.

The realisation that reframed everything after it: **those four are separate properties requiring separate mechanisms.**

Property	Mechanism	Proves	Says nothing about
Integrity	SHA-256 sidecar	The bytes did not change	Who produced them
Authenticity	cosign keyless signature	Which identity signed	When
Timeliness	Rekor transparency log	The signature existed at a time	Whether it can be deleted
Preservation	S3 Object Lock, COMPLIANCE	It cannot be deleted	Whether the content is true

Compliance programmes routinely collapse all four into *"we have evidence"*, and then discover during an audit which one they were actually missing.

And the second thing: a blocked merge beats a caught mistake. Detection is the industry's default posture, and it is a posture of permanent lateness — every finding is a description of something that already happened. Moving the same rule from "monitor" to "required status check" changes what it *is*. The violation stops being an incident and becomes a build error: nobody escalates it, nobody writes a ticket, nobody explains it to an auditor. It is just a red X and a one-line fix.

The argument applies to the verification as much as to the controls. The fourteen-check verdict this article opens with used to be something I remembered to run; it now runs on every pull request and is itself a required status check alongside the gate. The check that proves the pipeline works is no longer allowed to be the one nobody ran.

Try it

```
git clone https://github.com/sevenbelowllc/grc-engineering-club
cd grc-engineering-club

./verify-pipeline.sh    # every eligibility check, one verdict
./traverse.sh           # profile → component → evidence → CHAIN INTACT
```

No AWS account required — the two vault checks report themselves skipped and the verdict says `INCOMPLETE`, because a check that did not run is not a check that passed. No cooperation from me required. That was the point.

What I learned

The biggest thing I learned was working with OSCAL and Rego and putting them into action. I really love the policy-as-code methodology: OSCAL's Lego-like models (*Catalog*, *Profile*, *Component Definition*, *System Security Plan*, *Assessment Plan*, *Assessment Results*, *POA&M*) snap it all together, and Rego evaluates the reality. This challenge really opened my eyes to Policy as Code and I can't wait to start implementing it everywhere I can!

Built for the GRC Engineering Club six-week challenge, July 2026. The repository is public and the pipeline is live.