KCD BUDAPEST · 30 OCTOBER 2026

Every Workload Deserves an Identity Instead of an API Key

Attest, exchange, deliver: replacing static keys with workload identity

Zsolt Varga · Nándor Krácser

riptides

WHO WE ARE

Your speakers today



Zsolt Varga

CTO, Riptides

wayne@riptides.io



Nándor Krácser

CISO, Riptides

nandor@riptides.io

riptides

WHY THIS MATTERS

Service credentials are where breaches start

39%

of breaches involve credential abuse at some stage, more than any other vector

Verizon DBIR 2026

28.6M

new secrets leaked on public GitHub in 2025, up 34% in a year

GitGuardian, Secrets Sprawl 2026

64%

of secrets found valid in 2022 still worked in January 2026

GitGuardian, Secrets Sprawl 2026

80:1

machine identities for every human in the average organization

CyberArk Identity Security Landscape 2025

WHY THIS MATTERS · 2026

Same pattern, every time: someone else got the credential

MARCH

Cisco via Trivy

A poisoned Trivy release stole CI secrets. AWS keys used, 300+ repos cloned.

AUGUST

npm worm

Stole npm tokens and cloud keys, then used them to infect 400+ more packages.

17 OCTOBER

OpenCode

Stored OpenAI, Anthropic and Google keys taken. Users saw charges within hours.

21 OCTOBER

Railway

Customer environment variables decrypted. Every user told to rotate every secret.

Every one was a bearer credential, valid for whoever held it. Validation on the server didn't help.

Just bájtkolbász on the Wire

How service-to-service authentication evolved, and why so little has changed

A SHORT HISTORY

For forty years, service-to-service auth has been the same trick: send some bytes, and let the other side check them.

The only real question is whether those bytes are the secret, or proof of one.

A SHORT HISTORY

Five generations of the same idea

1980s

Network trust

`source IP`

From the right address, you're trusted.

1990s

Service password

`svc:hunter2`

A technical user in a connection string.

2000s

API key

`sk_live_51H…`

A machine-made password, in config for years.

2012

Client credentials

`client_secret`

A long-lived secret traded for a one-hour token.

2018+

Workload JWT

`eyJhbGciOi…`

Issued to the workload, expires fast. Still bearer.

riptides

A SHORT HISTORY

A JWT is a password with a signature

	API key or client secret	Workload JWT
Sent	On every call	On every call
Usable by	Whoever holds it	Whoever holds it
Checked by	Comparing a stored copy	**The issuer's public key ↑**
Valid for	Until rotated	**Minutes to hours ↑**
If stolen	Impersonation	Impersonation, until expiry

Two rows got better. The one that matters didn't: holding the bytes is the identity.

BEARER TOKENS · THE ISSUES

What a bearer token can’t protect against

Replay from anywhere

A copied token works the same from an attacker’s laptop as from your pod.

Copies everywhere

Logs, env vars, app memory, crash dumps, proxies. Every hop is a leak path.

Expiry is the only defence

Short lifetimes shrink the window. They don’t close it.

No way to tell

The server can’t distinguish a copy from the original.

THE PATCHWORK

Two kinds of patchwork

DELIVERED · THE WORKLOAD IS PASSIVE

Someone else puts a secret within reach

sync operators, injectors, CSI drivers, CI/CD

Trusts: the deliverer's own credential and a static mapping.

Leaves: a ready-to-use secret at rest, and a deliverer that can read everything.

EXCHANGED · THE WORKLOAD PARTICIPATES

The workload trades what it has for a token

Vault login, client credentials, cloud STS

Trusts: whatever the workload presents first.

Leaves: secret zero. Shorter tokens downstream don't fix a weak first credential.

Around both: encrypted Git, rotation, scanning. Hygiene, not identity.

PATCHWORK 1 · DELIVERED

The delivered secret's journey



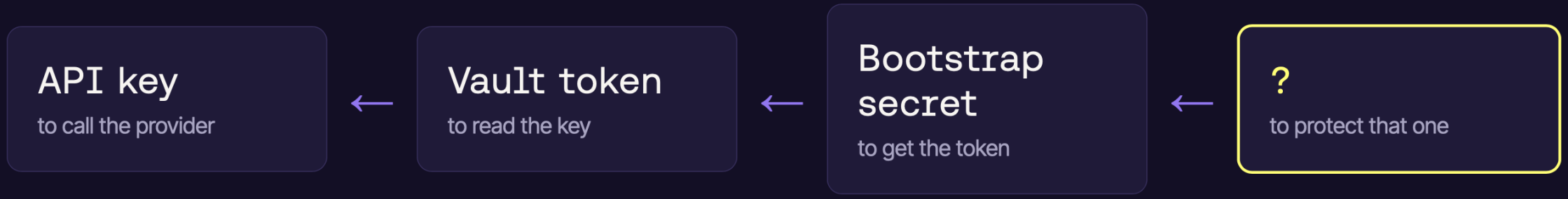
We moved the secret. We didn't remove it.

riptides

PATCHWORK 2 · EXCHANGED

The secret-zero problem

A chicken-and-egg problem: to get a credential, a workload needs a credential. Where does the first one come from?



Each exchange moves the problem one hop. It doesn't end the chain.

riptides

THE KUBERNETES ANSWER

But what about projected service-account tokens?

What it solves

- No pre-shared secret
- Issued per pod by the API server
- Audience-bound and short-lived
- Rotated by the kubelet, dies with the pod

What it doesn't

- A bearer JWT in a file in the container
- Identity is the service account, not the code
- Whoever can create the pod gets the token
- Only inside Kubernetes

It ends secret zero, then hands the app a bearer token again.

PROOF OF POSSESSION

Proof of possession keeps being reinvented

1988

Kerberos

tickets

Proves it holds the ticket's session key.

1990s

Client certs

mTLS

Proves it holds the private key in the handshake.

2006

HMAC signing

AWS SigV4

Signs each request. The secret is never sent.

2007

OAuth 1.0

oauth_signature

Signed every request. Dropped in OAuth 2.0.

2017+

SPIFFE

X.509-SVID

Short-lived, rotated keys per workload.

2020+

Bound tokens

RFC 8705, DPoP

A token that only works with the client's key.

Each time, convenience pulled us back to bearer tokens.

riptides

PROOF OF POSSESSION · THE ISSUES

Even with a key, the server can’t see the client

The server can check

- The signature is valid
- The issuer is trusted
- The certificate hasn’t expired
- The request is bound to the key

The server can’t check

- Where the key was stored
- Who else has read it
- Whether the caller is the real workload
- Whether the client was compromised

A leaky client still produces valid signatures.

PROOF OF POSSESSION · THE ISSUES

Proof of possession still needs a first key

01	02	03	04	05	06
Generate	**Prove**	**Issue**	**Store**	**Renew**	**Revoke**
On the workload, or handed to it?	**Why should the CA sign this request?**	Which identity goes into the certificate?	Who else can read the private key?	Signed with the current key, so trust is inherited.	Who notices that it leaked?

Every renewal inherits the trust of the first issuance. Step 02 decides everything.

riptides

THE PATCHWORK, AGAIN

Same patches, now for keys

DELIVERED · THE WORKLOAD IS PASSIVE

A tool writes the key and certificate for it

certificate Secrets, mounted key files

Trusts: the issuing tool's credential and a static mapping.

Leaves: a private key at rest in the pod, readable like any other secret.

EXCHANGED · THE WORKLOAD PARTICIPATES

The workload trades a token for a certificate

PKI login, ACME, join tokens

Trusts: the token it presents first.

Leaves: secret zero, now guarding a private key.

Key in a KMS or HSM? The app calls it with yet another credential.

THE WAY OUT

Move the key below the application

IN THE APP

File or env var

Readable by anyone with the pod spec, the image, a shell or a crash dump.

IN THE APP

App memory

Any code in the process, dependencies included, can read it or use it.

BESIDE THE APP

Sidecar proxy

Outside the app process. Adds the proof on the wire; the app never sees the key.

BELOW THE APP

Kernel or platform layer

Below every application. Backed by a TPM, the key can't even be exported.

The further below the app, the less of the app's code can reach the key.

THE WAY OUT

Proof of possession, owned by the platform

01

Attest

SA token, node, TPM

The platform observes what is running. No secret needed.

02

Generate

kernel, TPM, agent

The key is created below the app and never exported.

03

Issue

platform CA

A short-lived identity, bound to the attested facts.

04

Prove

proxy, kernel

The proof is added on the wire, outside the app.

05

Renew

the platform

Re-attested each time, not just re-signed.

The application never holds a credential, so nothing in it can leak one.

riptides

THE LESSON

Verifying credentials on the server is solved. Fixing the client means taking the credential out of it: proof of possession, attested and held by the platform.

riptides

Meanwhile, in your cluster

Most workloads still hold a long-lived API key. Here's why, and how to get rid of it.

THE DEFAULT

Twelve pods, one key

```yaml
kind: Deployment
spec:
 replicas: 12
 …
    containers:
    - name: support-bot
      env:
      - name: ANTHROPIC_API_KEY
        valueFrom:
          secretKeyRef:
            name: anthropic
            key: api-key
```

AUGUST 2025 · THE NX “S1NGULARITY” ATTACK

Malware went looking for keys, and found them

2,349 credentials stolen

1,079 developer machines and CI runners

1,000+ still valid when researchers checked

Nothing was cracked. The payload read env vars and files already on disk.

Source: GitGuardian, “The Nx s1ngularity attack: inside the credential leak”

riptides

THREE GAPS

So why is the key still in the Secret?

01

Attest

Something has to prove what the workload is before anyone gives it a credential.

02

Exchange

Every provider has its own protocol, audience, claims and expiry rules.

03

Deliver

The short-lived credential still has to reach the request without leaking.

01 · ATTEST

How SPIRE gives a pod its identity

1 **The pod calls the Workload API**

A Unix socket mounted into the pod. No token, no password.

2 **The agent identifies the caller**

Process ID from the socket, then the kubelet: namespace, service account, labels, image.

3 **The server matches a registration entry**

Those selectors map to a SPIFFE ID: spiffe://acme.org/ns/payments/sa/checkout

4 **The pod gets its SVIDs**

An X.509 certificate for mTLS and a JWT for providers. Short-lived, rotated automatically.

The workload never presents a secret to get its identity. That closes the secret-zero gap.

02 · EXCHANGE

Identity in, short-lived credential out

JWT-SVID

```
{
  "sub": "spiffe://acme.org/…/checkout",
  "aud": "sts.amazonaws.com",
  "exp": in 5 minutes
}
```

→

Provider token endpoint

Checks the signature against your keys, then aud, sub and the trust policy.

→

Short-lived credential

An AWS session, a GCP access token, a Vault secret or an API token.

The workload proves who it is. The provider's own IAM decides what it can do.

02 · EXCHANGE

Same idea, seven dialects

ONE CALL

AWS → Access key + session token
STS AssumeRoleWithWebIdentity

Azure → OAuth2 access token
Entra JWT client assertion

Oracle Cloud → Session token + your key
Token exchange for a UPST

OpenAI → API access token
RFC 8693 token exchange

Anthropic → API access token
RFC 7523 JWT bearer grant

TWO CALLS

Google Cloud → OAuth2 access token
STS exchange + SA impersonation

Vault / OpenBao → Any secret Vault holds
JWT login, then read

An exchange first, then a second call for the real credential.

Each difference is small. Seven of them is a project.

02 · EXCHANGE

tokenex: one Go interface for all of them

An open source (MIT) Go library. Give it an identity token source, get back a channel of credentials that stays fresh.

```
go get go.riptides.io/tokenex
```

In production: 13 credential source types in the Riptides platform run on it.

IDENTITY TOKENS FROM

- SPIFFE Workload API
- GitHub Actions
- Static / your own

CREDENTIALS FOR

- AWS
- Google Cloud
- Azure
- Oracle Cloud
- Vault / OpenBao
- CyberArk Conjur
- RFC 8693 · OpenAI
- RFC 7523 · Anthropic
- OAuth2 client credentials
- OAuth2 auth code
- GitHub App
- Kubernetes Secret

02 · EXCHANGE

SPIRE to AWS in a dozen lines

```
idp, _ := spiffe.NewIdentityTokenProvider(
    spiffe.WithAudience("sts.amazonaws.com"))

awsp, _ := aws.NewCredentialsProvider(ctx, log, &awssdk.Config{Region: region})

creds, _ := awsp.GetCredentials(ctx, idp,
    aws.WithRoleArn("arn:aws:iam::123456789012:role/checkout"))

for res := range creds {
    if res.Err != nil { return res.Err } // refresh failed: retry
    c := res.Credential.(*credential.AWSCreds)
    use(c) // arrives again before every expiry
}
```

02 · EXCHANGE

Refreshing is the hard part



When

With 20% of the lifetime left, kept between 1 and 5 minutes before expiry.

Jitter

Up to 10% extra, so a fleet of replicas doesn't hit the token endpoint at the same moment.

Events

Each new credential is an Update. A failure is an Err that closes the channel; the caller owns the retry.

03 · DELIVER

Delivery options and their trade-offs

Secret / env

App changes
None

App can read it
Yes

Rotation
Restart or reload

Protocols
Any

Moving parts
Sync controller

Mounted file

App changes
Reload on change

App can read it
Yes

Rotation
Re-read the file

Protocols
Any

Moving parts
CSI driver or helper

Sidecar agent

App changes
None

App can read it
Yes

Rotation
Agent renews

Protocols
Any

Moving parts
An agent per pod

Egress injection

App changes
None

App can read it
No

Rotation
Transparent

Protocols
HTTP(S), per auth scheme

Moving parts
Proxy + TLS interception

riptides

03 · DELIVER

Injection isn't free

TLS interception

To change an HTTPS request, the proxy terminates TLS. The app has to trust its CA.

Protocol awareness

A bearer token is one header. AWS SigV4 signs the whole request. Every scheme is code.

Coverage

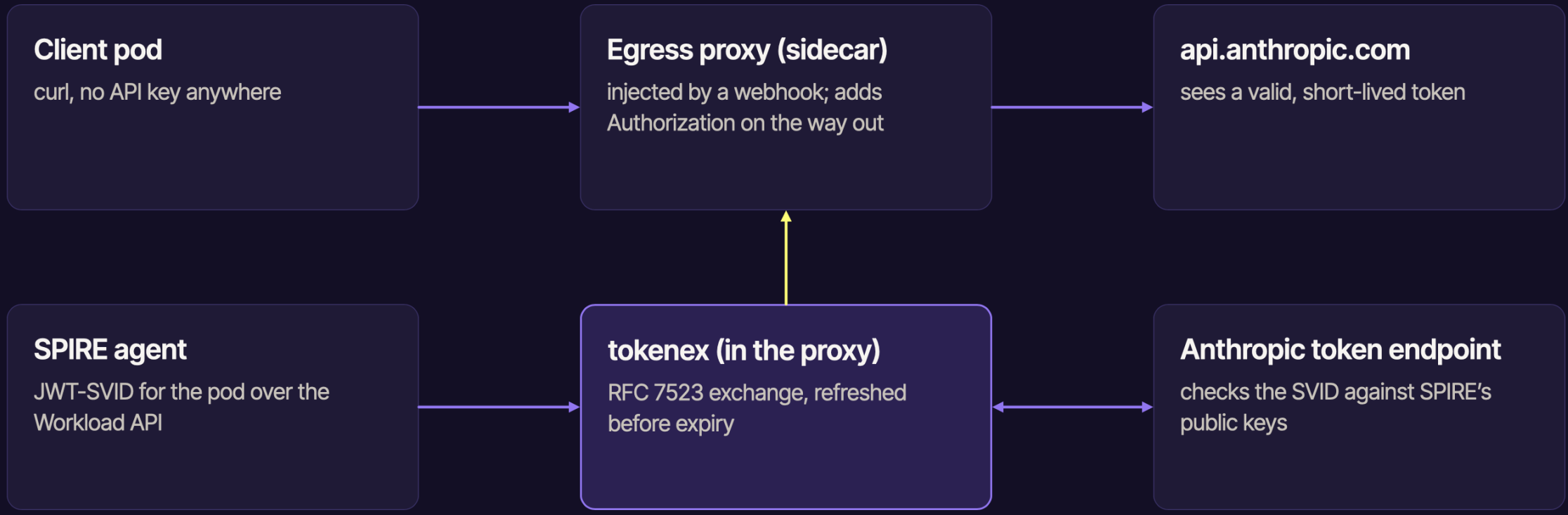
HTTP/1.1 is straightforward. HTTP/2, gRPC and non-HTTP protocols take more work.

Debugging

The app never sees the credential, so a 401 has to be debugged from the proxy side.

DEMO · ALL OPEN SOURCE

No key in the pod, and it still works



Code: github.com/riptideslabs/kcd-budapest-demo

riptides

DEMO · ALL OPEN SOURCE

What you'll see

1. The pod has no key: env, files and Secrets are clean
2. The API call fails with 401
3. Turn on the egress proxy for the pod
4. The same command succeeds
5. A fake exfiltration script finds nothing to steal

TAKEAWAYS

Start on Monday

01

Attest

Give one workload a SPIFFE ID with SPIRE, or start with projected service account tokens.

02

Exchange

Turn on federation for one provider. Let tokenex do the exchange and the refresh.

03

Deliver

Delete one API key from one Secret. Then the next one.

riptides

THE NEXT STEP

What if the platform did all of it?

YOUR WORKLOAD

Plain request

no credential

→

SECURITY LAYER IN YOUR INFRASTRUCTURE

01	**Attest**	Identity from what the platform observes
02	**Authorize**	Policy decides each access at runtime
03	**Inject**	Credential added on the wire, never in memory
04	**Contain**	Egress enforced, not opt-in
05	**Attribute**	Every connection tied to its workload

ONE LAYER · KUBERNETES, BARE METAL, ENDPOINT

→

THE API

Verified request

short-lived, bound, attributed

No SDK. No code changes. Nothing for the app to hold. That's the layer we provide.

riptides

THANK YOU

riptides

Questions?

TOKENEX

github.com/riptideslabs/tokenex

RIPTIDES

riptides.io
blog.riptides.io

DEMO

github.com/riptideslabs/kcd-budapest-demo

GITHUB & LINKEDIN

Zsolt Varga · @waynz0r
Nándor Krácser · @bonifaido