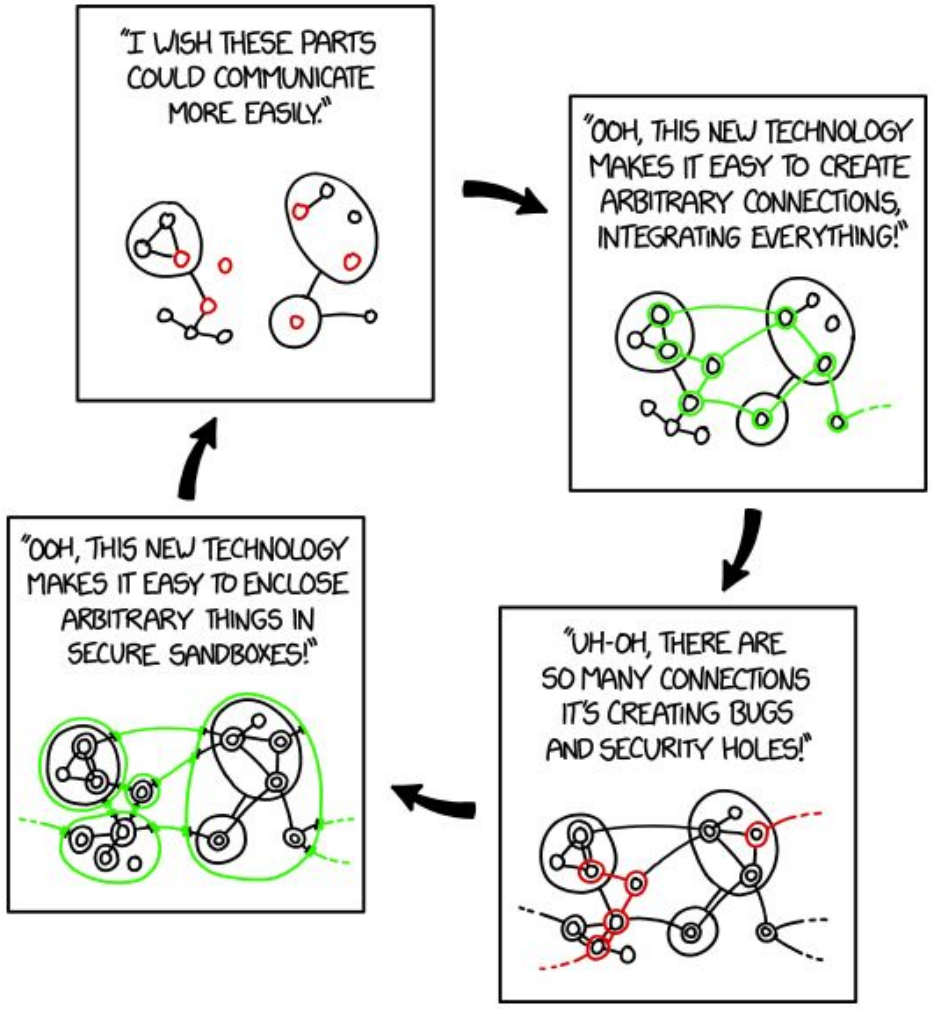
SIG Auth Deep Dive

Margo Crawford, VMWare
Mike Danese, Google

Agenda

SIG Auth Deep Dive

What do we do?



SIG Auth Deep Dive

What do we do?

SIG-Auth is responsible for features in Kubernetes that control and **protect access** to the API and other core components. This includes **authentication** and **authorization**, but also encompasses features like **auditing** and some **security policy**.

https://github.com/kubernetes/community/blob/master/sig-auth/charter.md

SIG Auth Deep Dive

Subprojects

- Audit Logging
- Authenticators
- Authorizers
- Certificates
- Encryption At Rest
- Node Identity and Isolation
- Policy Management
- Service Accounts
- Multi Tenancy
- Hierarchical Namespace Controller
- Secrets Store CSI Driver

And coming soon…

- kube-rbac-proxy

SIG Auth Deep Dive

Highlight Reel: Year(-ish) In Review

- Pod Security Admission is GA in 1.25!
- Certificates API is GA!
- Exec Auth Kubectl Credential Provider is GA!
- TokenRequest, et al. is GA!
- Secret Store CSI is GA!

SIG Auth Deep Dive

Highlight Reel: Trust Anchor Sets

kubernetes/enhancements#3258

- Non-namespaced ConfigMaps for cert bundles
  - Solves scalability issues in clusters with large # ns's
- Referenceable by Dynamic Webhook and API config
- Useful for signer anchor distribution (e.g. svc mesh)

SIG Auth Deep Dive

Highlight Reel: KMS ~~Observability~~ v2

kubernetes/enhancements#3302

- Working group focusing on bring KMS to GA
- Initial proposal on observability improvements
- Broadened to key rotation, scalability, health checking

SIG Auth Deep Dive

Highlight Reel: SvcAcct. Token Secret Deprecation

kubernetes/enhancements#2799

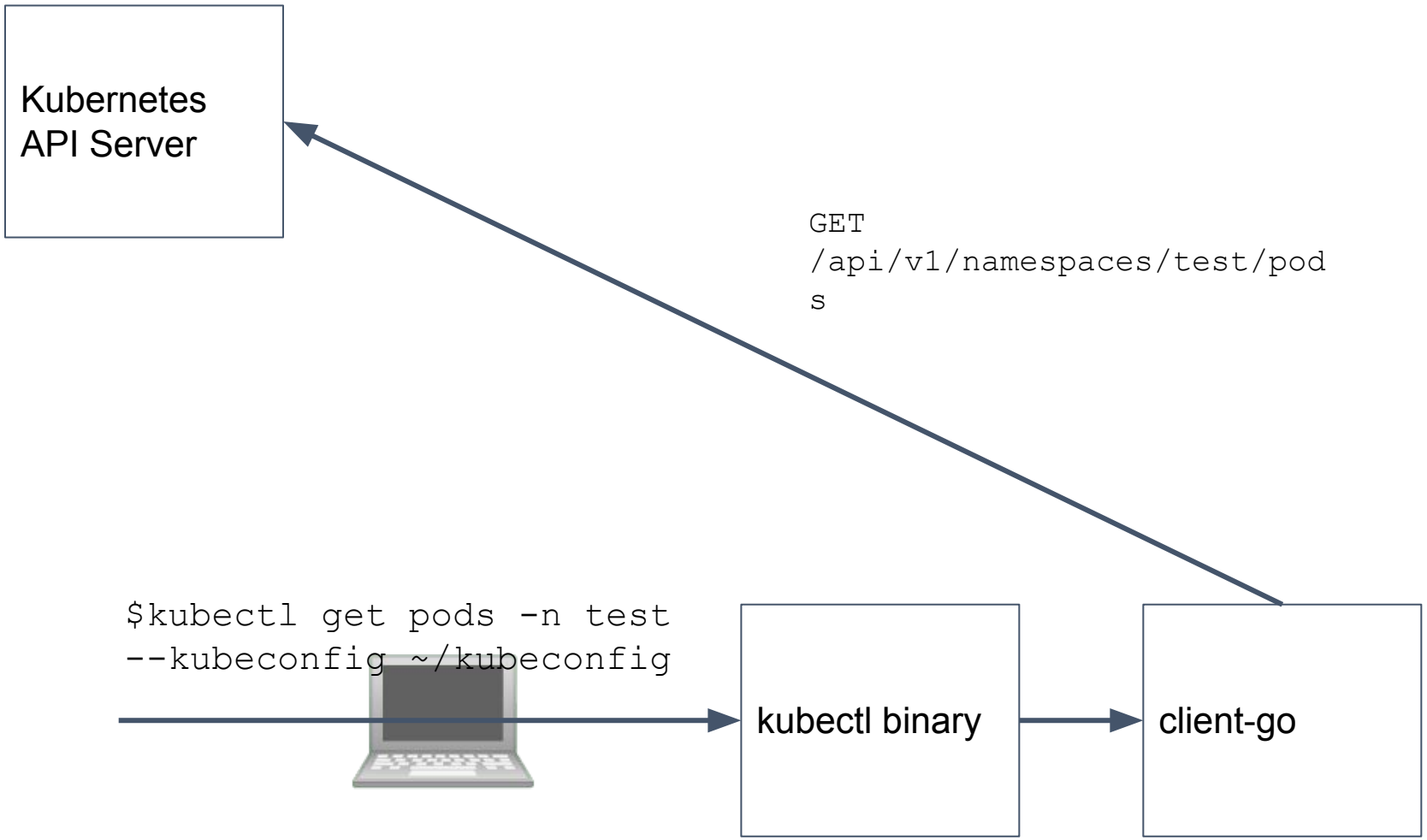
- Continuation of bound service account token work
- Goal is to not auto-create (and remove) unused token secrets
- Accessing token secrets bumps timestamp

Sample Implementation Walkthrough: client-go Credential Plugin

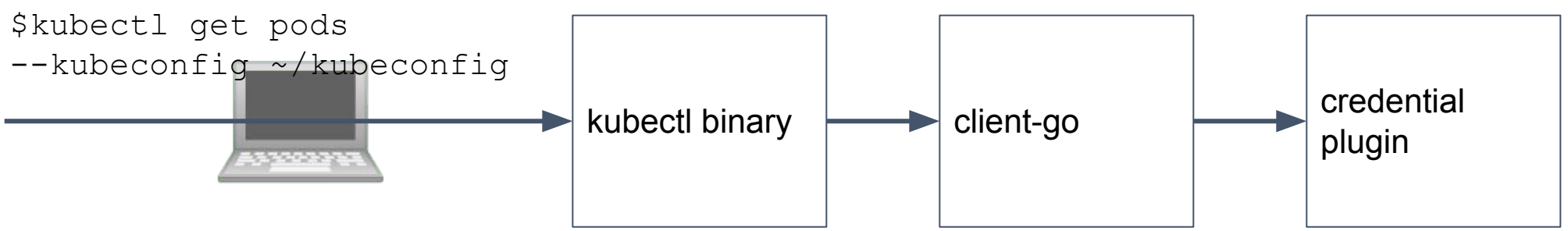
What is it?

- k8s.io/client-go executes an external command to receive user credentials.
- The plugin implements protocol specific logic (LDAP, OAuth2, etc.), then returns opaque credentials to use.
- Usually requires webhook token authenticator to interpret the credential

https://github.com/kubernetes/enhancements/tree/master/keps/sig-auth/541-external-credential-providers
https://github.com/vmware-tanzu/pinniped


Kubernetes
API Server
GET
/api/v1/namespaces/test/pod
s
$kubectl get pods -n test
--kubeconfig ~/kubeconfig
kubectl binary
client-go

Kubernetes
API Server



Kubernetes
API Server

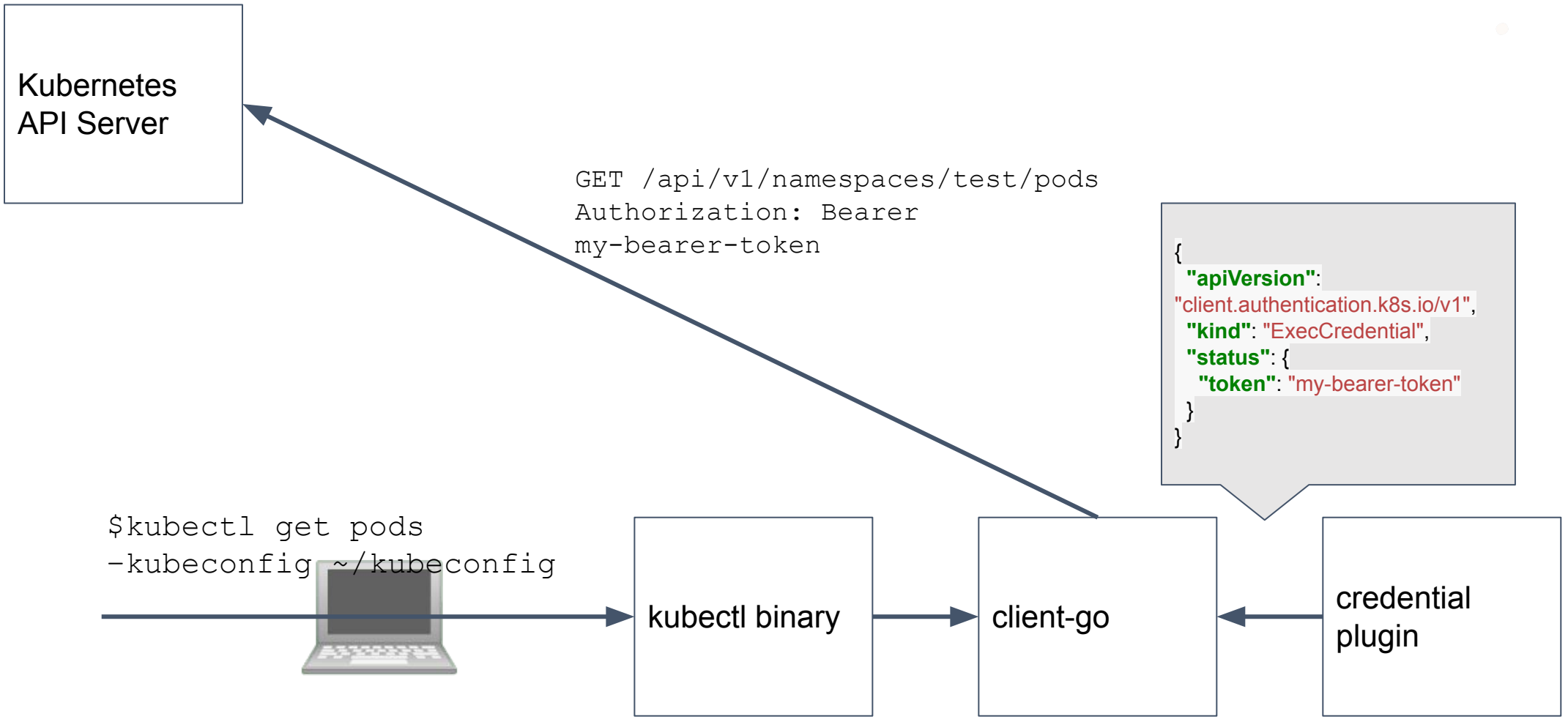

```
{
  "apiVersion":
"client.authentication.k8s.io/v1",
  "kind": "ExecCredential",
  "status": {
    "token": "my-bearer-token"
  }
}
```

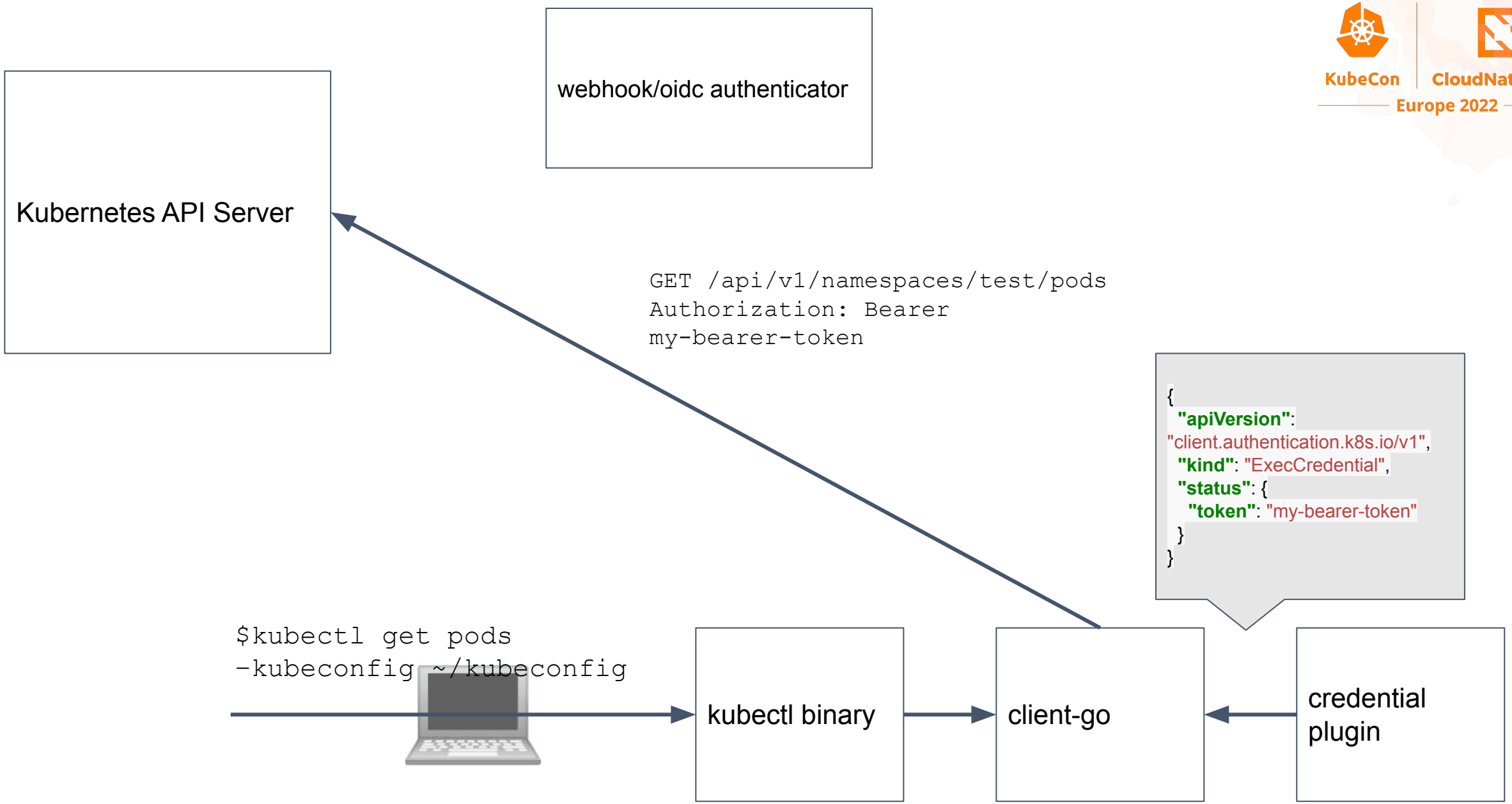

```
$kubectl get pods
--kubeconfig ~/kubeconfig
```

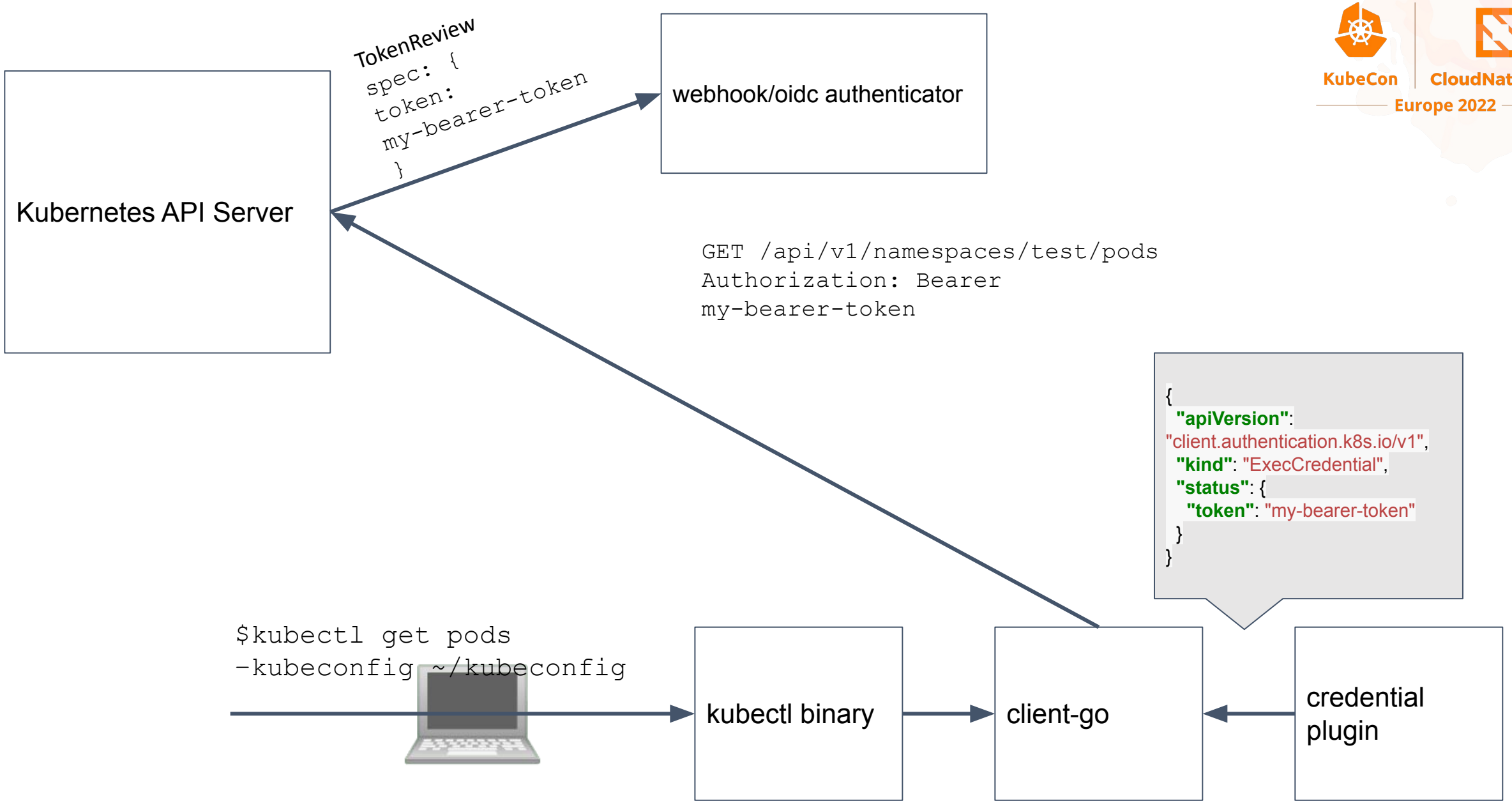
kubectl binary

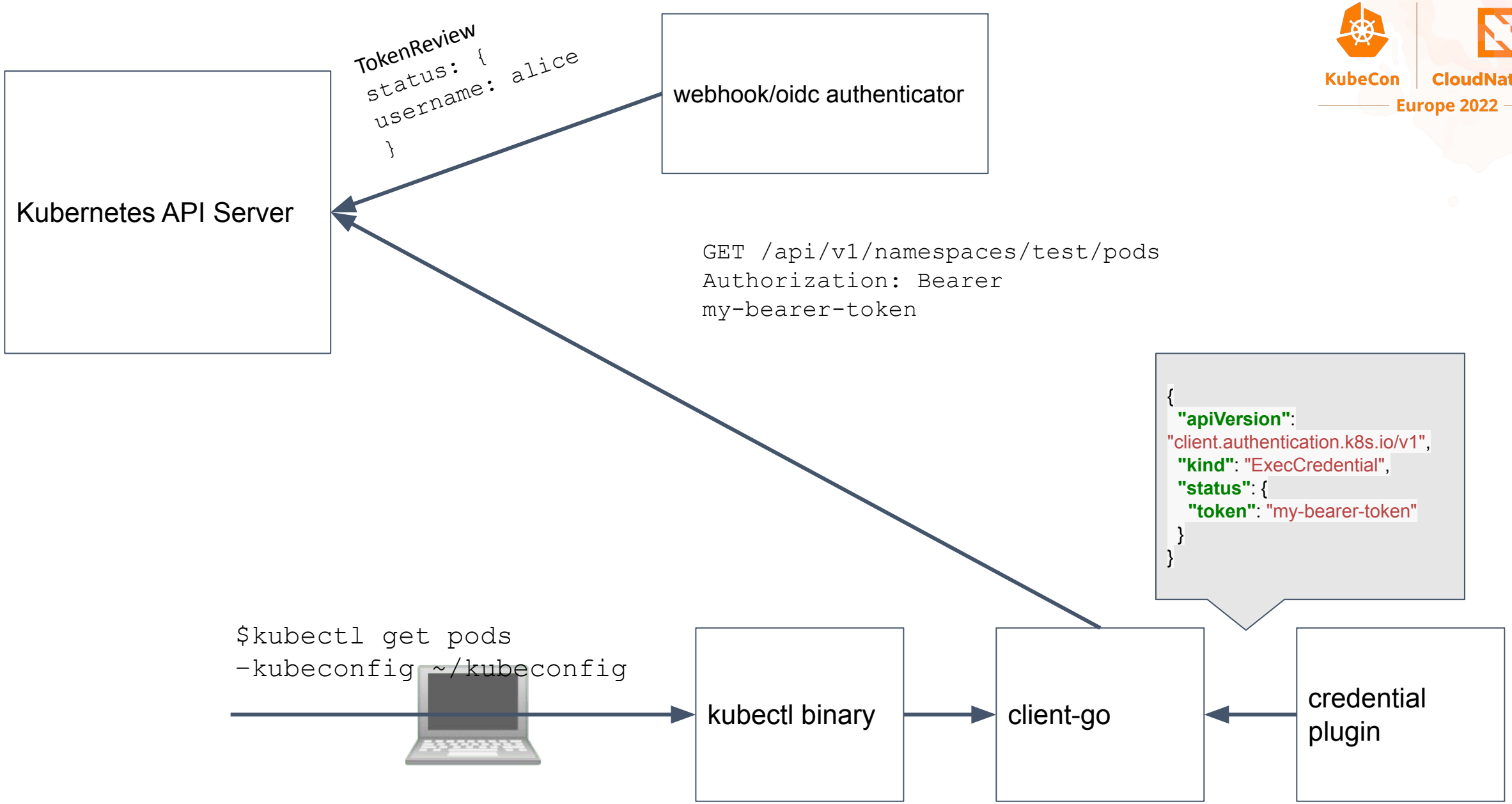
client-go

credential
plugin


Kubernetes
API Server
GET /api/v1/namespaces/test/pods
Authorization: Bearer
my-bearer-token
{
"apiVersion":
"client.authentication.k8s.io/v1",
"kind": "ExecCredential",
"status": {
"token": "my-bearer-token"
}
}
$kubectl get pods
-kubeconfig ~/kubeconfig
kubectl binary
client-go
credential
plugin


webhook/oidc authenticator
Kubernetes API Server
GET /api/v1/namespaces/test/pods
Authorization: Bearer
my-bearer-token
{
"apiVersion":
"client.authentication.k8s.io/v1",
"kind": "ExecCredential",
"status": {
"token": "my-bearer-token"
}
}
$kubectl get pods
-kubeconfig ~/kubeconfig
kubectl binary
client-go
credential
plugin


TokenReview
spec: {
token:
my-bearer-token
}
webhook/oidc authenticator
Kubernetes API Server
GET /api/v1/namespaces/test/pods
Authorization: Bearer
my-bearer-token
{
"apiVersion":
"client.authentication.k8s.io/v1",
"kind": "ExecCredential",
"status": {
"token": "my-bearer-token"
}
}
$kubectl get pods
-kubeconfig ~/kubeconfig
kubectl binary
client-go
credential
plugin


TokenReview
status: {
username: alice
}
webhook/oidc authenticator
Kubernetes API Server
GET /api/v1/namespaces/test/pods
Authorization: Bearer
my-bearer-token
{
"apiVersion":
"client.authentication.k8s.io/v1",
"kind": "ExecCredential",
"status": {
"token": "my-bearer-token"
}
}
$kubectl get pods
-kubeconfig ~/kubeconfig
kubectl binary
client-go
credential
plugin

```yaml
apiVersion: v1
kind: Config
users:
- name: my-user
  user:
    exec:
      # Command to execute. Required.
      command: "example-client-go-exec-plugin"

      # API version to use when decoding the
ExecCredentials resource. Required.
      apiVersion: "client.authentication.k8s.io/v1"

      # Arguments to pass when executing the plugin.
Optional.
      args:
      - "arg1"
      - "arg2"

      interactiveMode: IfAvailable
      installHint: Don’t curl pipe to bash!
...
```

Sample Plugin

- Prompts user to log in to an OIDC identity provider using the password flow
- Takes OIDC issuer and client id as inputs
- Takes the ID token from the OIDC provider and wraps it in an ExecCredential

Entrypoint

- Cobra setup

Entrypoint


```go
func init() {
    // initialize the command using cobra
    var (
        cmd = &cobra.Command{
            Args:         cobra.NoArgs,
            Use:          "login --issuer ISSUER --client-id CLIENTID",
            Short:        "Login using an OpenID Connect provider with PKCE flow",
            SilenceUsage: true,
        }
        flags oidcLoginFlags
    )

    // two flags, both are required
    cmd.Flags().StringVar(&flags.issuer, name: "issuer", value: "", usage: "OpenID Connect issuer URL")
    cmd.Flags().StringVar(&flags.clientID, name: "client-id", value: "", usage: "OpenID Connect client ID")
    if err := cmd.MarkFlagRequired( name: "issuer"); err != nil : err *
    if err := cmd.MarkFlagRequired( name: "client-id"); err != nil : err *

    cmd.RunE = func(cmd *cobra.Command, args []string) error { return runOIDCLogin(cmd, flags) }
    rootCmd.AddCommand(cmd)
}
```

KUBERNETES_EXEC_INFO


```yaml
kind: ExecCredential
apiVersion: client.authentication.k8s.io/v1
spec:
  interactive: false
  cluster:
    server: https://1.2.3.9:8443
    tls-server-name: foo
    insecure-skip-tls-verify: false
    certificate-authority-data: ABCD...
    proxy-url: http://127.0.0.1:7434
    config:
      arbitrary: extra
      data: true

```

Caching

- We need to cache credentials so we don’t have to fetch them every time a user runs a kubectl command
- Store the token in a yaml file on your home directory
- Fetch credentials at a key hashed from the arguments to the command

Caching: get from cache


```
// Check the cache for a previous session issued with the same parameters.
sort.Strings(h.scopes)
cacheKey := SessionCacheKey{
    Issuer:      h.issuer,
    ClientID:    h.clientID,
    Scopes:      h.scopes,
    RedirectURI: (&url.URL{Scheme: "http", Host: h.listenAddr, Path: h.callbackPath}).String(),
}

// If the ID token is still valid for a bit, return it immediately and skip the rest of the flow.
cached := h.cache.GetToken(cacheKey)
if cached != nil && cached.IDToken != nil && time.Until(cached.IDToken.Expiry.Time) > minIDTokenValidity {
    h.logger.V(debugLogLevel).Info( msg: "Found unexpired cached token.")
    return cached, nil
}
```

Caching: insert into cache


```
// If we got tokens, put them in the cache.
if err == nil {
    h.cache.PutToken(cacheKey, token)
}
```

Reading username and password from stdin

Packaging the token into an ExecCredential

```go
func tokenCredential(token *oidctypes.Token) *clientauthv1beta1.ExecCredential {
    cred := clientauthv1beta1.ExecCredential{
        TypeMeta: metav1.TypeMeta{
            Kind:       "ExecCredential",
            APIVersion: "client.authentication.k8s.io/v1beta1",
        },
        Status: &clientauthv1beta1.ExecCredentialStatus{
            Token: token.IDToken.Token,
        },
    }
    if !token.IDToken.Expiry.IsZero() {
        cred.Status.ExpirationTimestamp = &token.IDToken.Expiry
    }
    return &cred
}
```

SIG Auth Deep Dive

Highlight Reel: Client Exec Proxy Auth

kubernetes/enhancements#2693

- Allows for richer authentication protocol
- Counterpart of authenticating proxies
- AWSv4 in the works, lays groundwork for Kerberos, ALTS, etc…

SIG-Auth Deep Dive

Where can you find us?

Slack channel: #sig-auth

Home page: https://github.com/kubernetes/community/tree/master/sig-auth

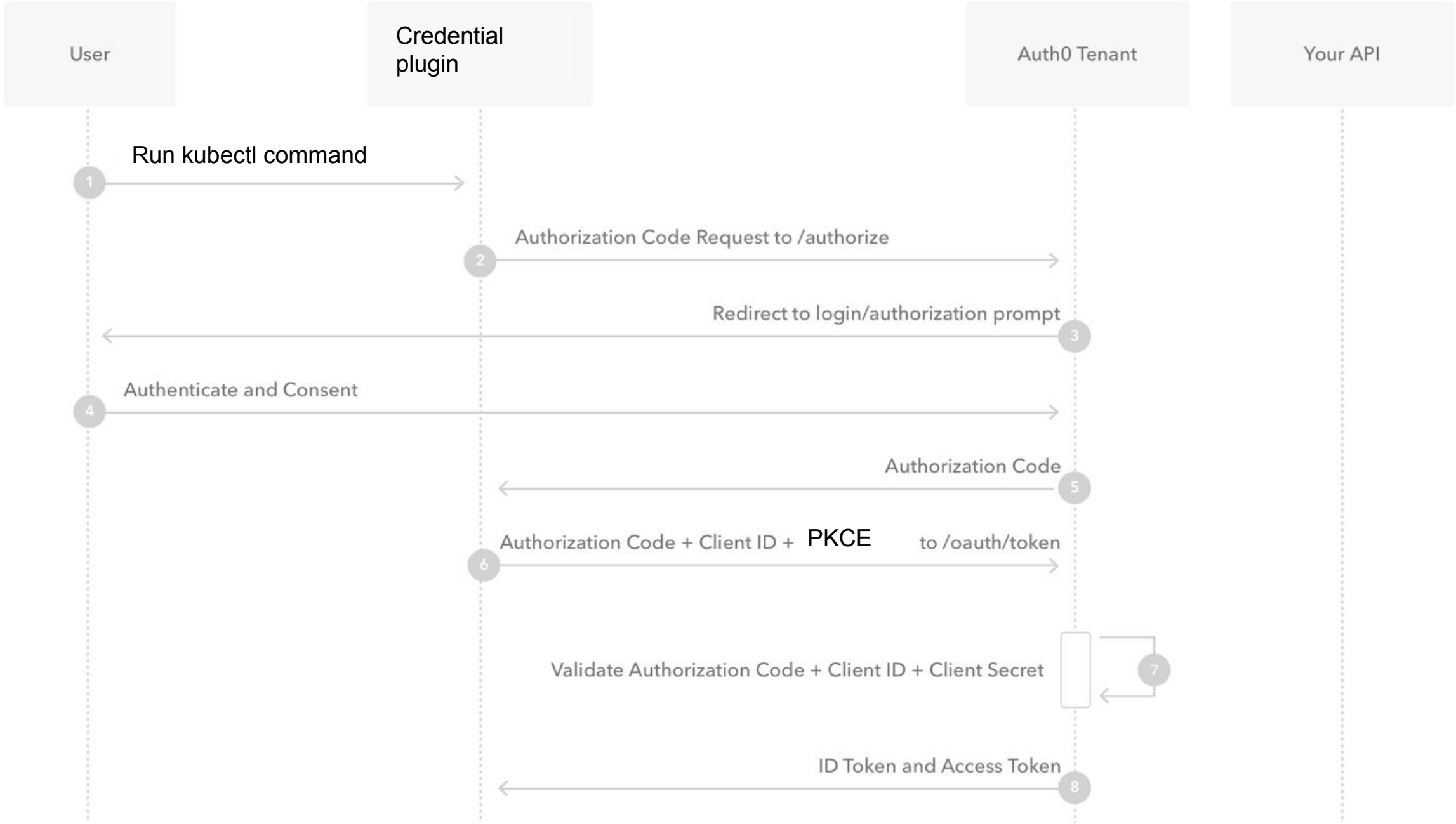
Mailing list: https://groups.google.com/forum/#!forum/kubernetes-sig-auth

Bi-weekly meetings Wednesday at 11PT (agenda/recordings links on home page)

Q & A

OIDC Flow

- Plugin starts a localhost listener
- Plugin opens user’s browser to their OIDC provider, set it to redirect to the localhost listener
- User logs on
- Redirect is sent with authcode as param
- Plugin takes authcode and uses it to ask the OIDC provider for tokens

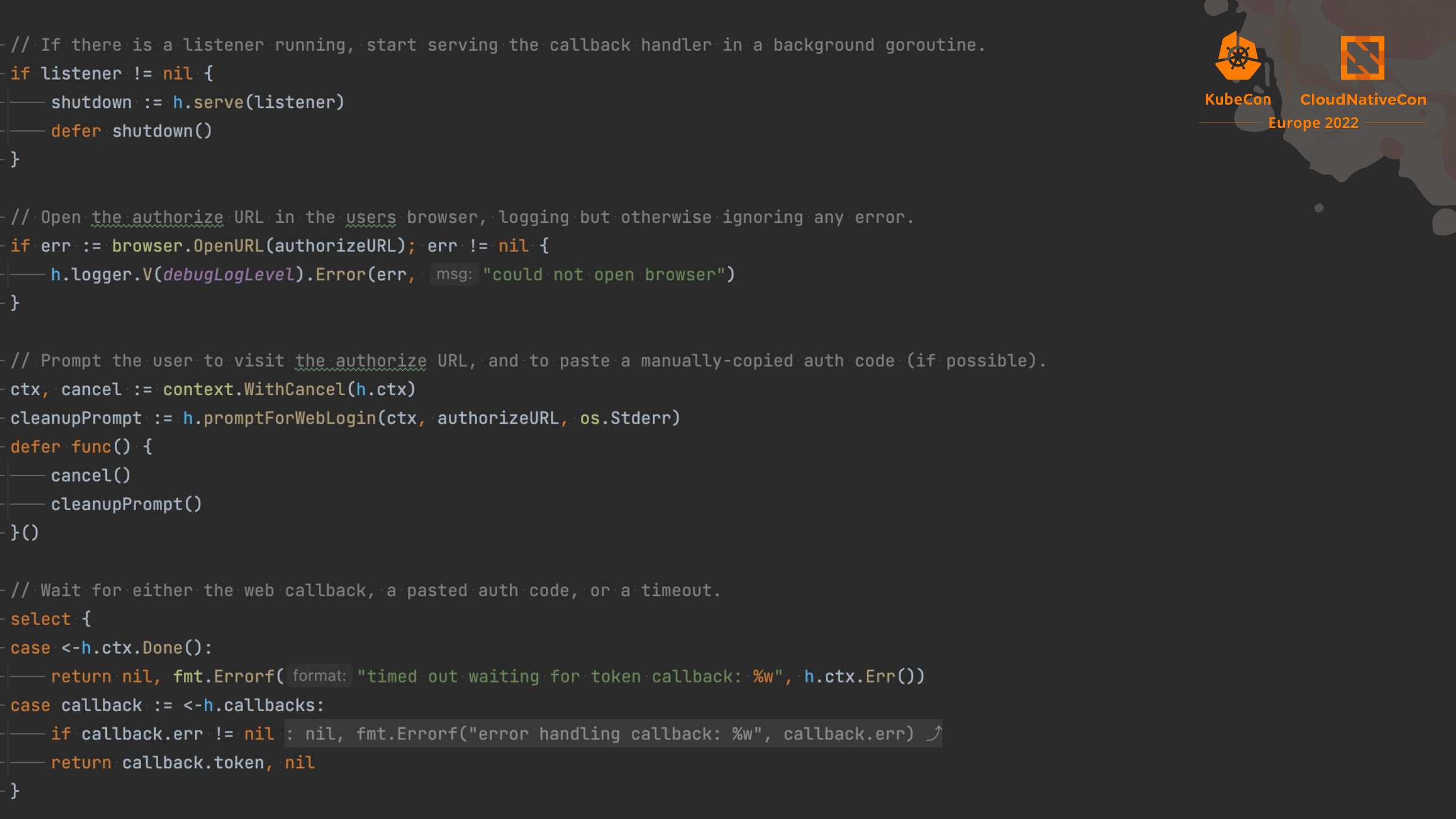

User
Credential plugin
Auth0 Tenant
Your API
Run kubectl command
1
Authorization Code Request to /authorize
2
Redirect to login/authorization prompt
3
Authenticate and Consent
4
Authorization Code
5
Authorization Code + Client ID + PKCE to /oauth/token
6
Validate Authorization Code + Client ID + Client Secret
7
ID Token and Access Token
8

OIDC Flow: Open localhost listener

```
// Attempt to open a local TCP listener, logging but otherwise ignoring any error.
listener, _ := net.Listen( network: "tcp", h.listenAddr)

// If the listener failed to start and stdin is not a TTY, then we have no hope of succeeding,
// since we won't be able to receive the web callback and we can't prompt for the manual auth code.
if listener == nil && !term.IsTerminal(stdin()) {
    return nil, fmt.Errorf( format: "login failed: must have either a localhost listener or stdin must be a TTY")
}

// Update the OAuth2 redirect_uri to match the actual listener address (if there is one), or just use
// a fake ":0" port if there is no listener running.
redirectURI := url.URL{Scheme: "http", Path: h.callbackPath}
if listener == nil {
    redirectURI.Host = "127.0.0.1:0"
} else {
    redirectURI.Host = listener.Addr().String()
}
h.oauth2Config.RedirectURL = redirectURI.String()
```




```
// If there is a listener running, start serving the callback handler in a background goroutine.
if listener != nil {
    shutdown := h.serve(listener)
    defer shutdown()
}

// Open the authorize URL in the users browser, logging but otherwise ignoring any error.
if err := browser.OpenURL(authorizeURL); err != nil {
    h.logger.V(debugLogLevel).Error(err, msg: "could not open browser")
}

// Prompt the user to visit the authorize URL, and to paste a manually-copied auth code (if possible).
ctx, cancel := context.WithCancel(h.ctx)
cleanupPrompt := h.promptForWebLogin(ctx, authorizeURL, os.Stderr)
defer func() {
    cancel()
    cleanupPrompt()
}()

// Wait for either the web callback, a pasted auth code, or a timeout.
select {
case <-h.ctx.Done():
    return nil, fmt.Errorf( format: "timed out waiting for token callback: %w", h.ctx.Err())
case callback := <-h.callbacks:
    if callback.err != nil : nil, fmt.Errorf("error handling callback: %w", callback.err)
    return callback.token, nil
}
```

OIDC Flow: Handling the callback

```go
func (h *handlerState) serve(listener net.Listener) func() {
	mux := http.NewServeMux()
	mux.Handle(h.callbackPath, httperr.HandlerFunc(h.handleAuthCodeCallback))
	srv := http.Server{
		Handler:     securityheader.Wrap(mux),
		BaseContext: func(_ net.Listener) context.Context { return h.ctx },
	}
	go func() { _ = srv.Serve(listener) }()
	return func() {
		// Gracefully shut down the server, allowing up to 100ms for
		// clients to receive any in-flight responses.
		shutdownCtx, cancel := context.WithTimeout(h.ctx, 100*time.Millisecond)
		_ = srv.Shutdown(shutdownCtx)
		cancel()
	}
}
```


```go
func (h *handlerState) handleAuthCodeCallback(w http.ResponseWriter, r *http.Request) (err error) {
    // If we return an error, also report it back over the channel to the main CLI thread.
    defer func() {
        if err != nil {
            h.callbacks <- callbackResult{err: err}
        }
    }()

    var params url.Values
    if h.useFormPost {...} else {
        // Return HTTP 405 for anything that's not a GET.
        if r.Method != http.MethodGet {...}

        // Pull response parameters from the URL query string.
        params = r.URL.Query()
    }

    // Validate OAuth2 state and fail if it's incorrect (to block CSRF).
    if err := h.state.Validate(params.Get( key: "state")); err != nil {...}

    // Check for error response parameters. See https://openid.net/specs/openid-connect-core-1_0.html#AuthError.
    if errorParam := params.Get( key: "error"); errorParam != "" {...}

    // Exchange the authorization code for access, ID, and refresh tokens and perform required
    // validations on the returned ID token.
    token, err := h.redeemAuthCode(r.Context(), params.Get( key: "code"))
    if err != nil : httperr.Wrap(http.StatusBadRequest, "could not complete code exchange", err) ↗

    h.callbacks <- callbackResult{token: token}
    _, _ = w.Write([]byte("you have been logged in and may now close this tab"))
    return  err: nil
}
```