Quarkus - Security Tips and Tricks

HttpAuthenticationMechanism Customization

One can customize `HttpAuthenticationMechanism` by registering a CDI implementation bean. In the example below the custom authenticator delegates to `JWTAuthMechanism` provided by `quarkus-smallrye-jwt`:

```java
@Alternative
@Priority(1)
@ApplicationScoped
public class CustomAwareJWTAuthMechanism implements
HttpAuthenticationMechanism {

    private static final Logger LOG =
LoggerFactory.getLogger(CustomAwareJWTAuthMechanism.class);

    @Inject
    JWTAuthMechanism delegate;

    @Override
    public Uni<SecurityIdentity> authenticate(RoutingContext
context, IdentityProviderManager identityProviderManager) {
        // do some custom action and delegate
            return delegate.authenticate(context,
identityProviderManager);
    }

    @Override
    public Uni<ChallengeData> getChallenge(RoutingContext context)
{
        return delegate.getChallenge(context);
    }

    @Override
    public Set<Class<? extends AuthenticationRequest>>
getCredentialTypes() {
        return delegate.getCredentialTypes();
    }

    @Override
    public HttpCredentialTransport getCredentialTransport() {
        return delegate.getCredentialTransport();
    }

}
```

Security Identity Customization

Internally, the identity providers create and update an instance of the `io.quarkus.security.identity.SecurityIdentity` class which holds the principal, roles, credentials which were used to authenticate the client (user) and other security attributes. An easy option to customize `SecurityIdentity` is to register a custom `SecurityIdentityAugmentor`. For example, the augmentor below adds an addition role:

```
import io.quarkus.security.identity.AuthenticationRequestContext;
import io.quarkus.security.identity.SecurityIdentity;
import io.quarkus.security.identity.SecurityIdentityAugmentor;
import io.quarkus.security.runtime.QuarkusSecurityIdentity;
import io.smallrye.mutiny.Uni;

import javax.enterprise.context.ApplicationScoped;
import java.util.function.Supplier;

@ApplicationScoped
public class RolesAugmentor implements SecurityIdentityAugmentor {

    @Override
    public int priority() {
        return 0;
    }

    @Override
    public Uni<SecurityIdentity> augment(SecurityIdentity identity,
AuthenticationRequestContext context) {
        return Uni.createFrom().item(build(identity));

        // Do 'return context.runBlocking(build(identity));'
        // if a blocking call is required to customize the identity
    }

    private Supplier<SecurityIdentity> build(SecurityIdentity
identity) {
        if(identity.isAnonymous()) {
            return () -> identity;
        } else {
            // create a new builder and copy principal, attributes,
credentials and roles from the original identity
            QuarkusSecurityIdentity.Builder builder =
QuarkusSecurityIdentity.builder(identity);

            // add custom role source here
            builder.addRole("dummy");
            return builder::build;
        }
    }
}
```

Here is another example showing how to use the client certificate available in the current Mutual TLS request to add more roles:

```
import java.security.cert.X509Certificate;
```

```java
import io.quarkus.security.credential.CertificateCredential;
import io.quarkus.security.identity.AuthenticationRequestContext;
import io.quarkus.security.identity.SecurityIdentity;
import io.quarkus.security.identity.SecurityIdentityAugmentor;
import io.quarkus.security.runtime.QuarkusSecurityIdentity;
import io.smallrye.mutiny.Uni;

import javax.enterprise.context.ApplicationScoped;
import java.util.function.Supplier;
import java.util.Set;

@ApplicationScoped
public class RolesAugmentor implements SecurityIdentityAugmentor {

    @Override
    public int priority() {
        return 0;
    }

    @Override
    public Uni<SecurityIdentity> augment(SecurityIdentity identity,
AuthenticationRequestContext context) {
        return Uni.createFrom().item(build(identity));
    }

    private Supplier<SecurityIdentity> build(SecurityIdentity
identity) {
        // create a new builder and copy principal, attributes,
credentials and roles from the original identity
        QuarkusSecurityIdentity.Builder builder =
QuarkusSecurityIdentity.builder(identity);

        CertificateCredential certificate =
identity.getCredential(CertificateCredential.class);
        if (certificate != null) {

builder.addRoles(extractRoles(certificate.getCertificate()));
        }
        return builder::build;
    }

    private Set<String> extractRoles(X509Certificate certificate) {
        String name =
certificate.getSubjectX500Principal().getName();

        switch (name) {
            case "CN=client":
                return Collections.singleton("user");
            case "CN=guest-client":
                return Collections.singleton("guest");
```

```
            default:
                return Collections.emptySet();
        }
    }
}
```

Custom JAX-RS SecurityContext

If you use JAX-RS `ContainerRequestFilter` to set a custom JAX-RS `SecurityContext` then make sure `ContainerRequestFilter` runs in the JAX-RS pre-match phase by adding a `@PreMatching` annotation to it for this custom security context to be linked with Quarkus `SecurityIdentity`, for example:

```java
import java.security.Principal;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.PreMatching;
import javax.ws.rs.core.SecurityContext;
import javax.ws.rs.ext.Provider;

@Provider
@PreMatching
public class SecurityOverrideFilter implements
ContainerRequestFilter {
    @Override
    public void filter(ContainerRequestContext requestContext)
throws IOException {
        String user = requestContext.getHeaders().getFirst("User");
        String role = requestContext.getHeaders().getFirst("Role");
        if (user != null && role != null) {
            requestContext.setSecurityContext(new SecurityContext()
{

                @Override
                public Principal getUserPrincipal() {
                    return new Principal() {
                        @Override
                        public String getName() {
                            return user;
                        }
                    };
                }

                @Override
                public boolean isUserInRole(String r) {
                    return role.equals(r);
                }
```

```
                @Override
                public boolean isSecure() {
                    return false;
                }

                @Override
                public String getAuthenticationScheme() {
                    return "basic";
                }
            });
        }

    }
}
```

Disabling Authorization

If you have a good reason to disable the authorization (for example, when testing) then you can register a custom `AuthorizationController`:

```
@Alternative
@Priority(Interceptor.Priority.LIBRARY_AFTER)
@ApplicationScoped
public class DisabledAuthController extends AuthorizationController
{
    @ConfigProperty(name = "disable.authorization")
    boolean disableAuthorization;

    @Override
    public boolean isAuthorizationEnabled() {
        return disableAuthorization;
    }
}
```

Registering Security Providers

When running in native mode, the default behavior for GraalVM native executable generation is to only include the main "SUN" provider unless you have enabled SSL, in which case all security providers are registered. If you are not using SSL, then you can selectively register security providers by name using the `quarkus.security.security-providers` property. The following example illustrates configuration to register the "SunRsaSign" and "SunJCE" security providers:

Example Security Providers Configuration

```
quarkus.security.security-providers=SunRsaSign,SunJCE
```

Reactive Security

If you are going to use security in a reactive environment, you will likely need SmallRye Context Propagation:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-context-propagation</artifactId>
</dependency>
```

This will allow you to propagate the identity throughout the reactive callbacks. You also need to make sure you are using an executor that is capable of propagating the identity (e.g. no `CompletableFuture.supplyAsync`), to make sure that Quarkus can propagate it. For more information see the Context Propagation Guide.

References

- Quarkus Security