GitHub Actions: running them securely

GitHub Actions[1] are a powerful way of creating a pipeline to act on events in GitHub. By creating a workflow file you run actions on code updates to build your application, automate triaging tasks from issues, and loads of other helpful uses.

Author Rob Bos

Tyranny of the default

Every demo on GitHub Actions shows how easy it is to get started: add a text file with some actions in it and you are good to go. Unfortunately, this is highly insecure! To understand why, you need to know what the attack vectors of your workflow are and how you can guard yourself against them.

Let's start with an introduction to GitHub Actions first.

By storing the dotnetcore.yml file in the right location, you have added a new workflow that can be triggered on events. There are a lot of events available, from the push event in this example(1), to comments on an issue and closing of a Pull Request.

main ▾ dotnetcore-webapp / .github / workflows / dotnetcore.yml

```
name: .NET Core

on: [push]                                          1

jobs:                                               2
  build-and-deploy:
    environment: Production

    runs-on: ubuntu-latest

    steps:                                          3
    - uses: actions/checkout@v1
    - name: Setup .NET Core                         4
      uses: actions/setup-dotnet@v1
      with:
        dotnet-version: 3.0.100

    # dotnet build                                  5
    - name: Build with dotnet
      run: |
        dotnet build --configuration Release ./dotnet-core-webapp/dotnetcore-webapp.csproj
```

Make your own Octocat: https://myoctocat.com/

[1] https://github.com/features/actions

In the jobs[2] section you can create one or more jobs that will run on a specific runner that executes the steps[3] in the sequential order within the file. In this example the repository is checked out(3) first, then a version of the .NET Core tooling is installed[4] and in the last step the .NET Core project is built using the tools[5].

Know your GitHub Actions

When using GitHub Actions it is important to understand what the actions you use are doing. You can use any action by leveraging the setup from GitHub: the action identifier is the organization or username that is hosting the action, and the name of the repository it is in.

In this example you can find both actions in the 'docker' organization in their own repositories. Adding the action path to https://github.com/ straight to the action repo.

```
-name: Login to DockerHub
 uses: docker/login-action
 with:
   username: ${{ secrets.DOCKERHUB_
   USERNAME }}
   password: ${{ secrets.DOCKERHUB_
   TOKEN }}

- name: Build and push
  uses: docker/build-push-action
    with:
      push: true
        tags: user/app:latest
```

Having a valid action.yml in the repository makes it useable for every workflow. Using the action like this will ensure that the workflows will always download the latest available version of the repository and execute the code that is in it. This is also the greatest downside of actions: the default is already insecure! Anyone can create an action like this and there is no process that will check the action you are using for quality or security issues. Even limiting the actions people can use in your organization, to only the actions listed on the marketplace is insecure: there is no process that checks whether your action is doing malicious things.

The source of every action is public, which also means that you can look at the action repository and verify what it is doing when it runs. You can check whether it is sending your environment variables over to their own API for example, or logging your OS information together with your IP-address.

What are the risks?

It is wonderful to able to use actions that someone else already spent time and effort to create, potentially saving you a lot of time. However, this also adds some risk to your repository, the application you are creating and the setup around it. To get some understanding of the risk we need to look at the results of an attack on your workflows.

A malicious actor can wreak havoc on your application or its environment in three different ways:

1. data theft
2. data integrity breaches
3. availability

Data theft

By working their way into your workflows, people could get access to the code in your repository, but potentially also to the environment your workflow is running in. That environment could be set up to have API keys available for accessing services you need to build or deploy your application, or have certificates installed for code signing. It could even have access to an account on your cloud platform that has administrative rights and could get access to data or delete infrastructure there. Limiting the access for the runner that executes your workflow to the bare minimum is key in preventing against data theft.

When you run your workflow on hosted runners[2], it is GitHub's responsibility to keep them up to date with the latest OS and tool updates. To make sure the attack surface on them is as small as possible, they will create a completely new environment for each run and clean up the environment after it is no longer used.

If you run the workflows on private runners[3], taking all these security measures is up to you. Keep in mind that you are taking that responsibility when you install a private runner. You need to secure the OS, limit access the account the workflow is running under to only the things it needs access to (so do not assign network admin permissions to it!). You also need to keep the tools on that machine up to date with all the security patches.

Data integrity breaches

If a malicious actor has a way to get into your workflow or execution environment, they can also inject malicious code into your application. Most workflows create an artifact to deploy into an environment and store the artifacts in the pipeline environment. A possibility is that the attacker injects something into the artifact and the deployment will then deploy the malicious code for you! The recent Solorigate[4] attack is a prime example of this type of attack. Adding one malicious assembly before the artifact was uploaded (and avoiding a lot of different detection methods) was the central point the attack was executing.

Other examples of data integrity breaches are poisoning your dependency cache: there are a lot of blogposts[5] available explaining that you need to verify the dependencies you are using with, for example SHA512 hashes of the commit[6] to make sure you are not unknowingly pulling in a newer version of the dependency when you build your application.

Something similar happens with typo squatting attacks[7]: can you spot the difference between using 'npm install crossenv' and 'npm install cross-env'? An easy mistake to make, but if the first one is a malicious copy of the package

[2] https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners
[3] https://docs.github.com/en/actions/hosting-your-own-runners
[4] http://xpir.it/Solorigate
[5] https://xpirit.com/99-of-code-isnt-yours/
[6] https://w3c.github.io/webappsec-subresource-integrity/
[7] https://snyk.io/blog/typosquatting-attacks/

you need, with some bonus code that executes at runtime, you might be compromised before you know it! These attacks are now getting even more sophisticated by finding out the names of internal packages you use and host a malicious version on the public repository site. Most package tools have a default to check the public hosted endpoints first. If the package is not found there, it will try the same on internal endpoints. Take a good look at those configurations you are using.

Availability

An attack vector that seems less likely is injecting something into your workflow that will cause the workflow to stop running. These days, most DevOps teams are very dependent on their pipelines to push code to production, and they have a hard time getting updates out if their pipelines are not working anymore. To limit what engineers have access to, everything is locked down and only a service account has access to production. What if your application is down, or worse: vulnerable to an attack? What if someone can trigger your workflow to be unable to execute, right at that moment? Does your DevOps team have a 'break glass' option[8] to fix the vulnerability without their pipelines?

Attack vectors

By pulling in the action from the internet you are executing its code in your environment: this can be a hosted runner on GitHub's infrastructure, or your own runner in your own cloud environment.

The code in the action can do multiple things: it can send out your data, code or environment setup (SSH Keys, locally stored certificates, etc.) to an endpoint of their own and exfiltrate data that way. They can also try to get access to your environment or your GitHub setup: either the code in the repository itself or even try to get administrative access to the complete repository. They could pull in extra dependencies in your code, add other actions to your workflow, or even misuse your action runs with Bitcoin miners for their own gain.

There are multiple ways to try and get in. Every now and again GitHub has 'Capture The Flag' (CTF) events where they invite the community to try out a repository and gain access. From those events they learn a lot about their setup and ways to break the security around the repository. A basic example of an attack vector is the use of sending in a Pull Request that alters the workflow files itself by adding in a malicious action. More sophisticated attacks examples are adding JavaScript in the issue comment that is being picked up by the workflow and not handled securely: the JavaScript is then executed by logging it to the output for example (helpful to see them in the logs) which in turn enables the attacker to break out of the action environment itself and run a process on the runner environment. With that setup someone can create a new Pull Request for the repository that added the next step of the attack by writing code back into the repository. From the CTF events we learn the new ways to get access, and GitHub can try to prevent those types of attack.

[8] https://docs.microsoft.com/en-us/azure/active-directory/roles/security-emergency-access?WT.mc_id=AZ-MVP-5003719

Securing the actions you run

There are several measures you can take to secure your actions. Just using the latest version of the action is not a good idea: new code could have nasty side-effects like introducing new vulnerabilities, as we have seen in the previous paragraphs. The action repository might even be taken over by a new maintainer with ill intent and still compromise your setup. That is why running the action (as displayed in every demo!) like this example is a bad idea:

```
- name: Login to DockerHub
  uses: docker/login-action
  with:
    username: ${{ secrets.DOCKERHUB_USERNAME }}
    password: ${{ secrets.DOCKERHUB_TOKEN }}

- name: Build and push
  uses: docker/build-push-action
  with:
    push: true
    tags: user/app:latest
```

Option 1: Version tags

You can add the version number of the action to the end of the configuration, but there is no way to verify if it is still the same code: the tag can be reused with new code changes in it, so adding this does not add real security to it.

```
uses docker/login-action@v1
```

Option 2: At least start here

Start by verifying the actions you are running by looking into the action's repository. Have a sanity check on the code in the repository and use the commit SHA from GitHub to add that at the end of your action configuration:

```
name: Login to DockerHub
 uses: docker/login-action@
 e2302b10ccc2c798f917336fe81ce41ea8dea0fd
 with:
   username: ${{ secrets.DOCKERHUB_USERNAME }}
   password: ${{ secrets.DOCKERHUB_TOKEN }}

- name: Build and push
  uses: docker/build-push-action@
  0ec1157bb54f3e4676c823ef3497b53135ed39de
    with:
      push: true
       tags: user/app:latest
```

The commit SHA is immutable: if the code in the repository changes, the SHA will be different. This is the only secure way to know for sure that the code you are executing is the code you have checked yourself and that you have approved the risks that come from using it.

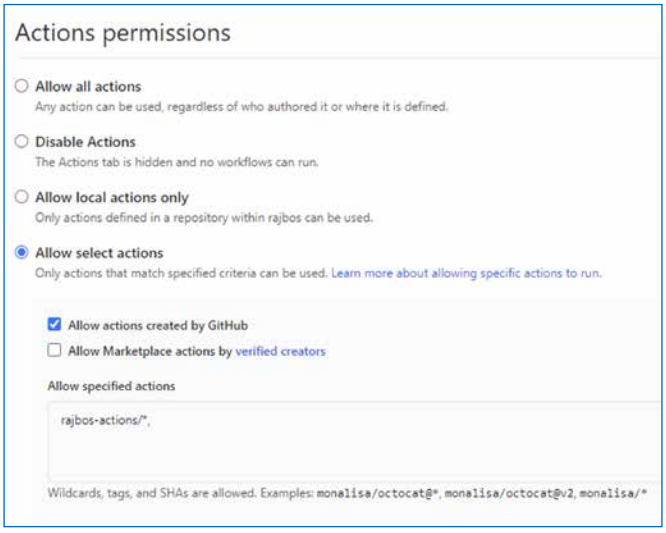
Staying up to date

Now that we are using the actions as securely as we can (by checking what it is **actually** doing and making sure no unseen changes can be added), the next question needs to be answered: how do we still get updates?

Since there is no update feed on the marketplace, or a blog that can be followed, I created a Twitter bot[9] that will regularly check for new or updated actions and will tweet them out. Checking the used action versions in your workflow files and updating them automatically can be done by using Dependabot[10]: it will scan your workflow files on a schedule and create a Pull Request for each updated action. This will give you a chance to manually verify the incoming changes and then accept the pull request.

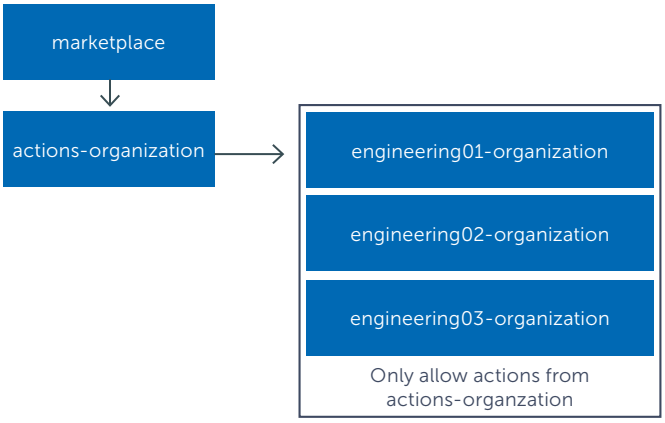
Option 3: Forking the action repository

The ultimate security setup I have found is forking the action repository to a specific organization for it. This way of working was suggested previously in documentation, but has not gained momentum.



Forking the repository gives you full control over the actions as well as their updates. It also provides a clear audit trail of the actions and secures you from actions being pulled by the maintainer. Additionally, you have a backup if the action gets deleted / renamed / moved to a different repository by the publisher. Remember the availability issues that can occur? This helps preventing that as well. You can now secure your other organizations (or separate repositories) to only allow actions being run from the forked repositories.

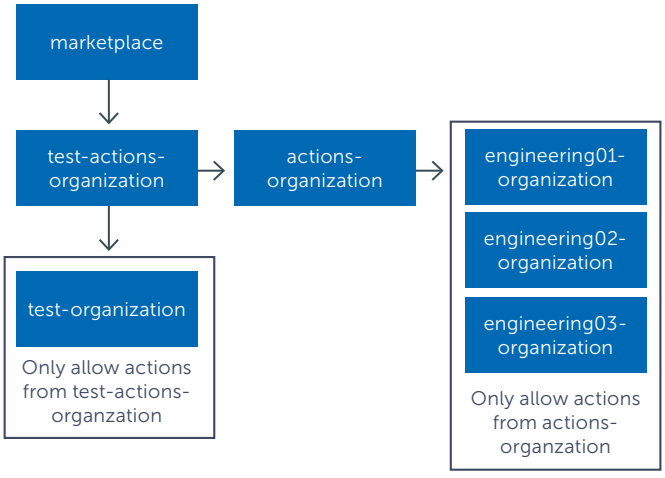
This is also an ideal strategy for enterprise organizations. You can create a specific actions-organization in which you fork all the actions you need. Then lock down the normal organization(s) everyone is using to only allow actions from the actions-organization. The setup would look like this:



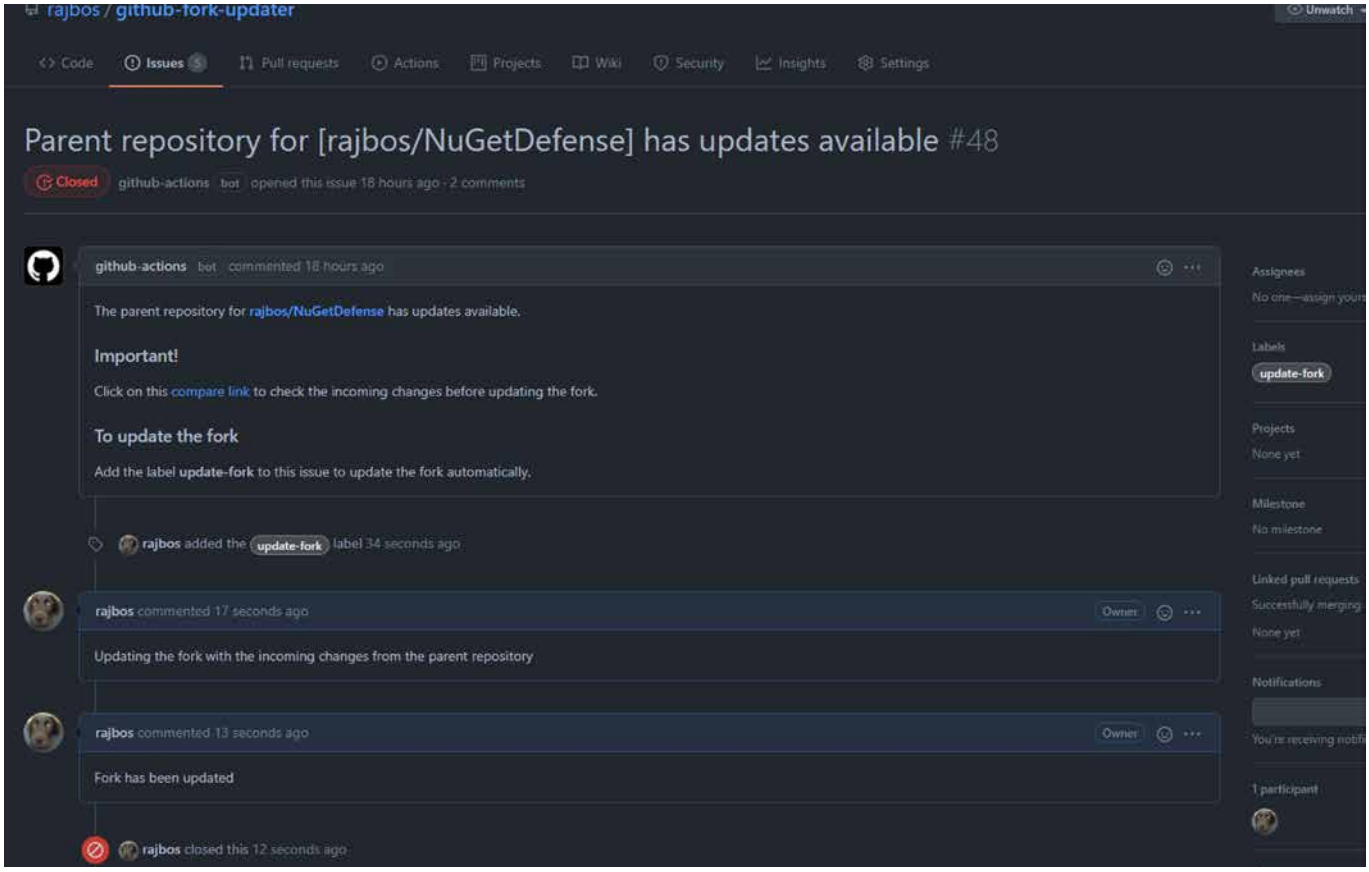
Enable your DevOps engineers!

Do not lock out you DevOps engineers: it is part of the DevOps way of working to let them take control over the tools they use. Add an organization in which people can pull in new actions to test with and validate their workflows, so they can still use new actions that you have not forked yet. They take ownership of the actions they want to use and fork the actions themselves!

That way they have full autonomy and will not be waiting for someone's approval before they can test new actions or updates.



[9] https://twitter.com/githubactions
[10] https://docs.github.com/en/github/administering-a-repository/keeping-your-actions-up-to-date-with-dependabot



Keeping your forks up to date

Now that you have secured your organization and made sure you are not blocking your DevOps engineers by empowering them to take control over the actions, you need a way to update your forks (all of them). To make this as easy and still secure as possible, I created the GitHub Fork Updater repository[11]: a specific repository that has everything in it you need. Fork it, add some configuration so that it can update all repositories in that organization, and you are good to go! The update works as follows:

1. On a schedule, check all repositories in the organization of the fork using a workflow.
2. If there are updates, create an issue in the fork-updater repository.
3. With the default GitHub notification setup, your engineers will get notified of new issues.
4. They can check the issue and do the security check on the incoming changes using a special link in the issue.
5. By adding a label on the issue, they will indicate that they have validated the incoming changes and that they want to pull them into the forked repository.
6. A workflow is triggered on the labeling of the issue and the fork will be updated.
7. The issue is closed.

Summary

Using GitHub Actions from the marketplace is not secure by default: there are no real checks on the code they are executing, and it is up to you to verify whether the actions are safe to use.

Empower your DevOps engineers to take ownership of the actions by forking the repositories and doing the due diligence on them to make sure they will not send out your data to some unknown third party. This can be done by setting up a secured configuration with additional organizations in your GitHub account and forking all the actions you want to use there. Keeping your forks up to date can be automated as much as you can by leveraging the GitHub Fork Updater to stay on top of changes. Always verify the incoming changes! </>

Rob Bos
Consultant

xpirit.com/rob

[11] https://github.com/rajbos/github-fork-updater