GitHub Security Fundamentals

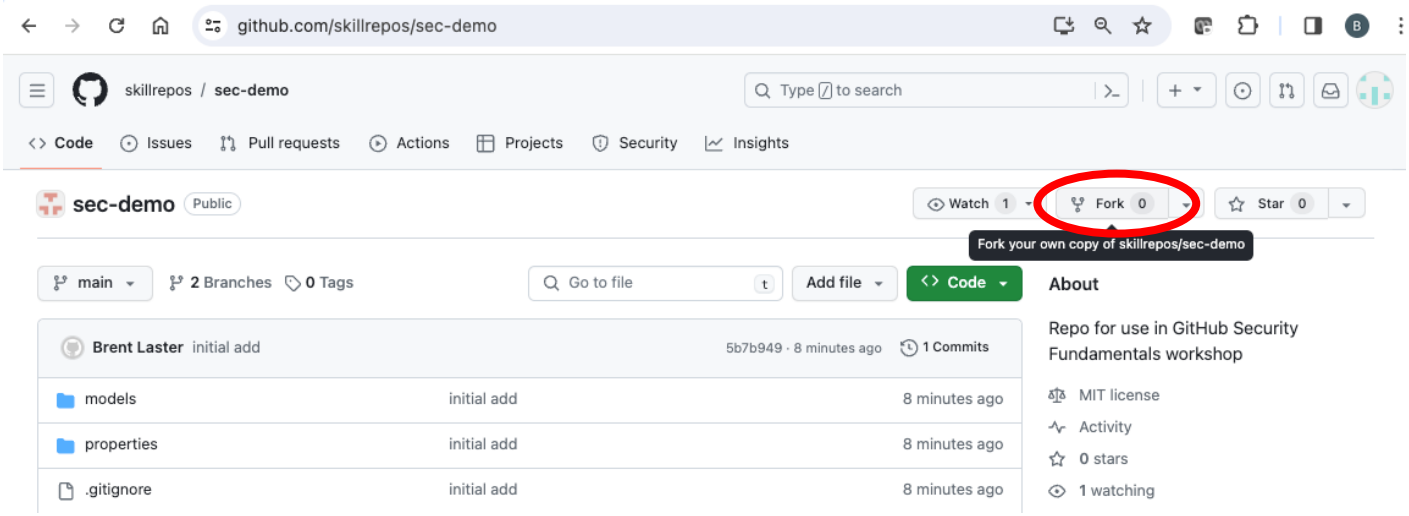
Best practices, tools, and strategies for creating and collaborating securely
Revision 1.4 – 02/10/24
Tech Skills Transformations LLC / Brent Laster

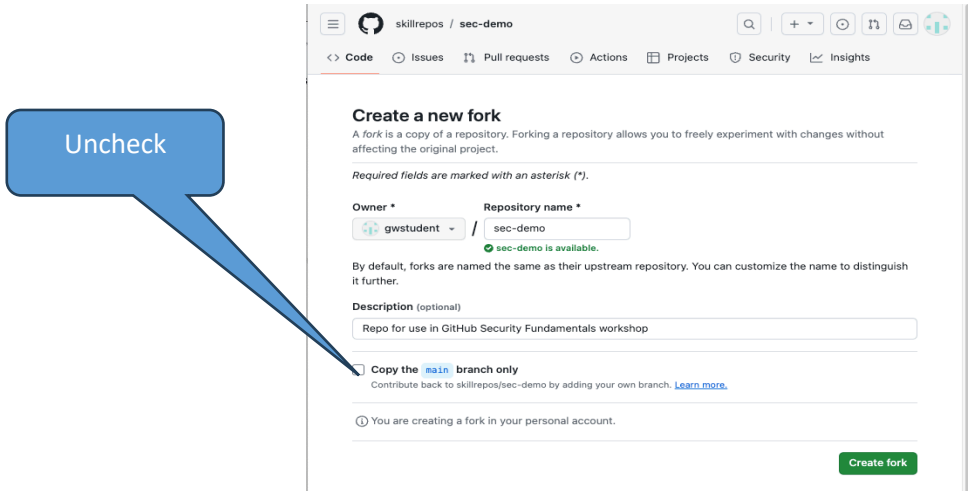
Lab 1 – Getting Started

Purpose: *In this lab, we'll get a quick start learning about some general security settings and authentication with Personal Access Tokens*

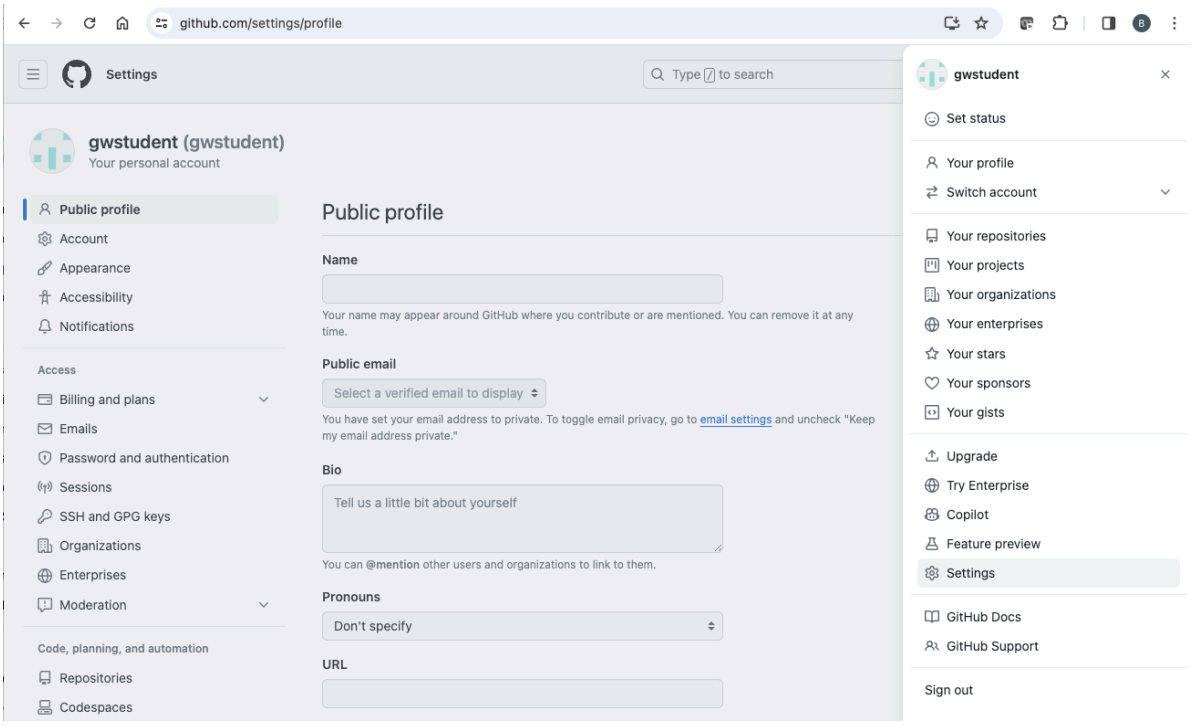
1. Log in to GitHub with your primary GitHub account.

2. Go to https://github.com/skillrepos/sec-demo and fork that project into your own GitHub space. Do this by clicking on the ***Fork*** button. On the next screen, make sure to uncheck the ***Copy the main branch only*** checkbox and then click the ***Create Fork*** button.

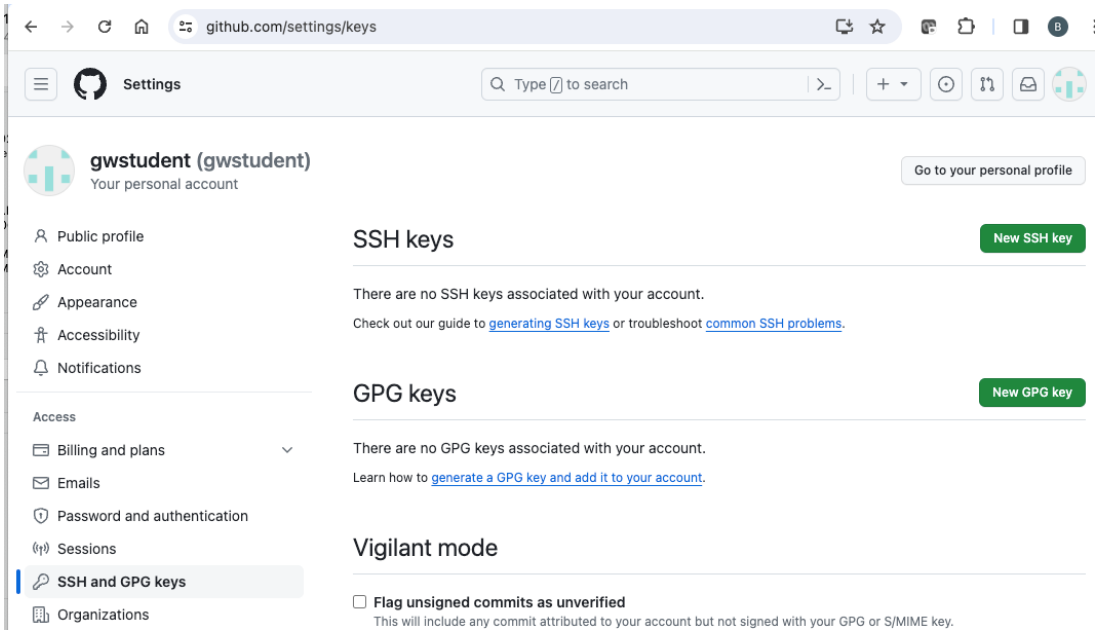




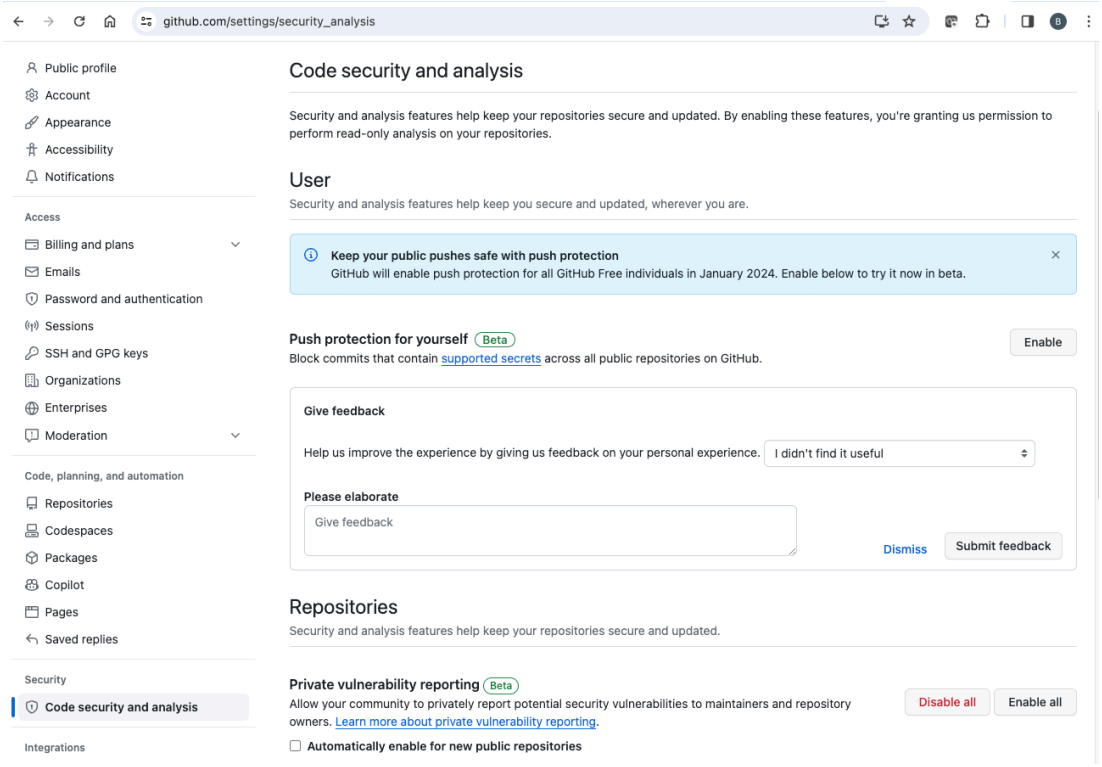
3. Now you'll be on your fork of the repo. Let's take a look at a couple of developer settings for security. Click on the *icon for your userid* in the upper right of the screen and then click on ***Settings*** near the bottom of the list that pops up. (Alternatively, you can go to *github.com/settings/profile directly*.)



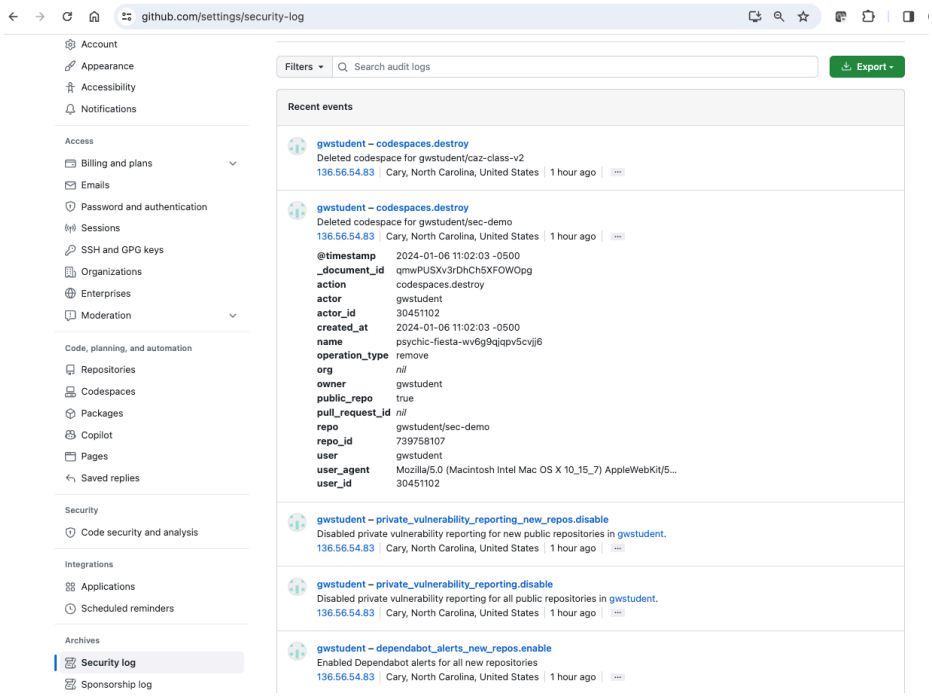
4. In the ***Settings*** screen for your userid, let's look at a few items related to security at the user level. First, select ***the SSH and GPG keys*** entry on the left. This is where you can add keys for accessing the repositories via SSH and GPG keys for signing commits/tags.



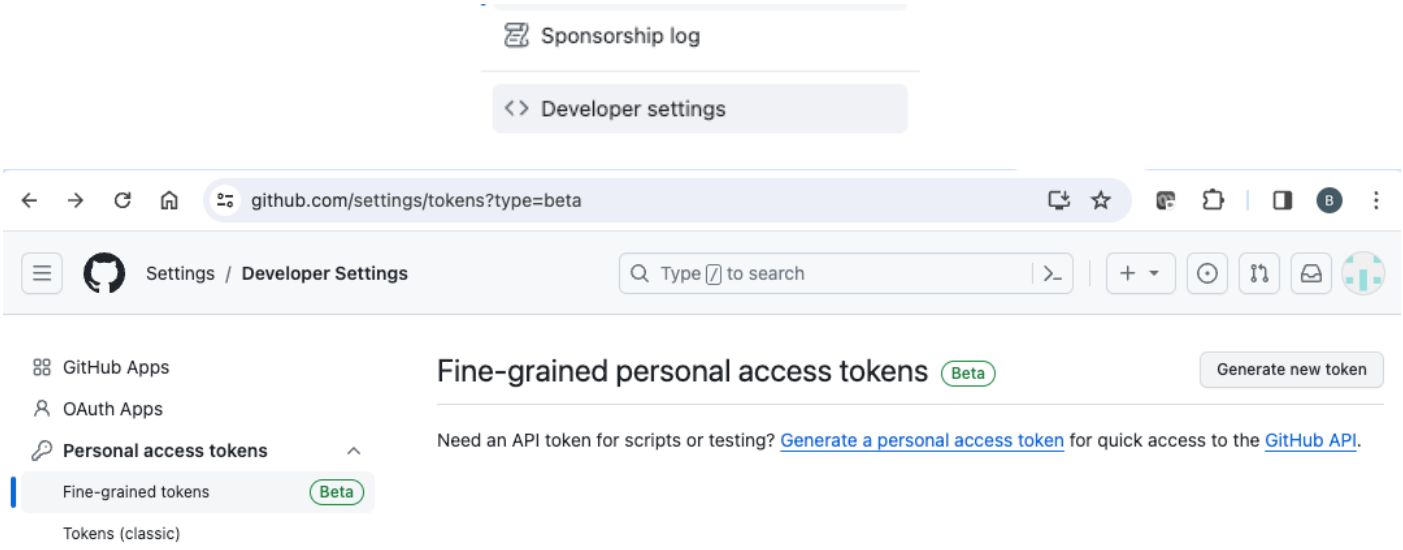
5. Next, select the ***Code security and analysis*** entry on the left. You don't have to change anything on this screen, just scroll around and note the different categories for ***User*** and ***Repositories***.



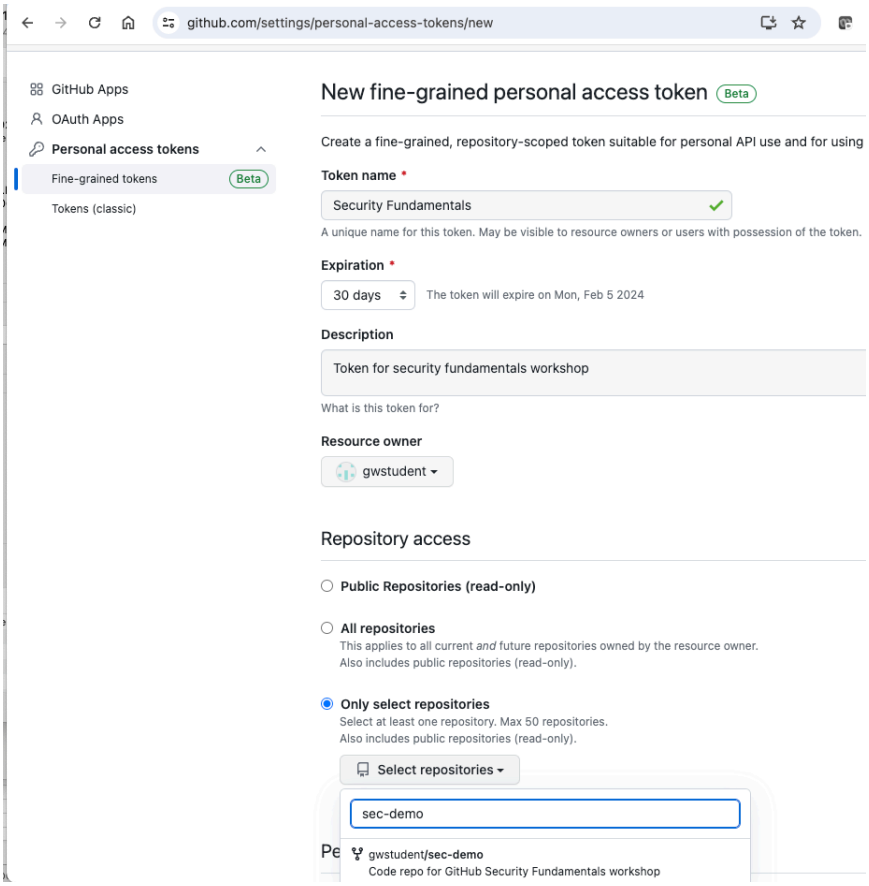
6. Now, in the left menu, near the bottom, click on ***Security log.*** You can scroll back through your history, expand entries or search.



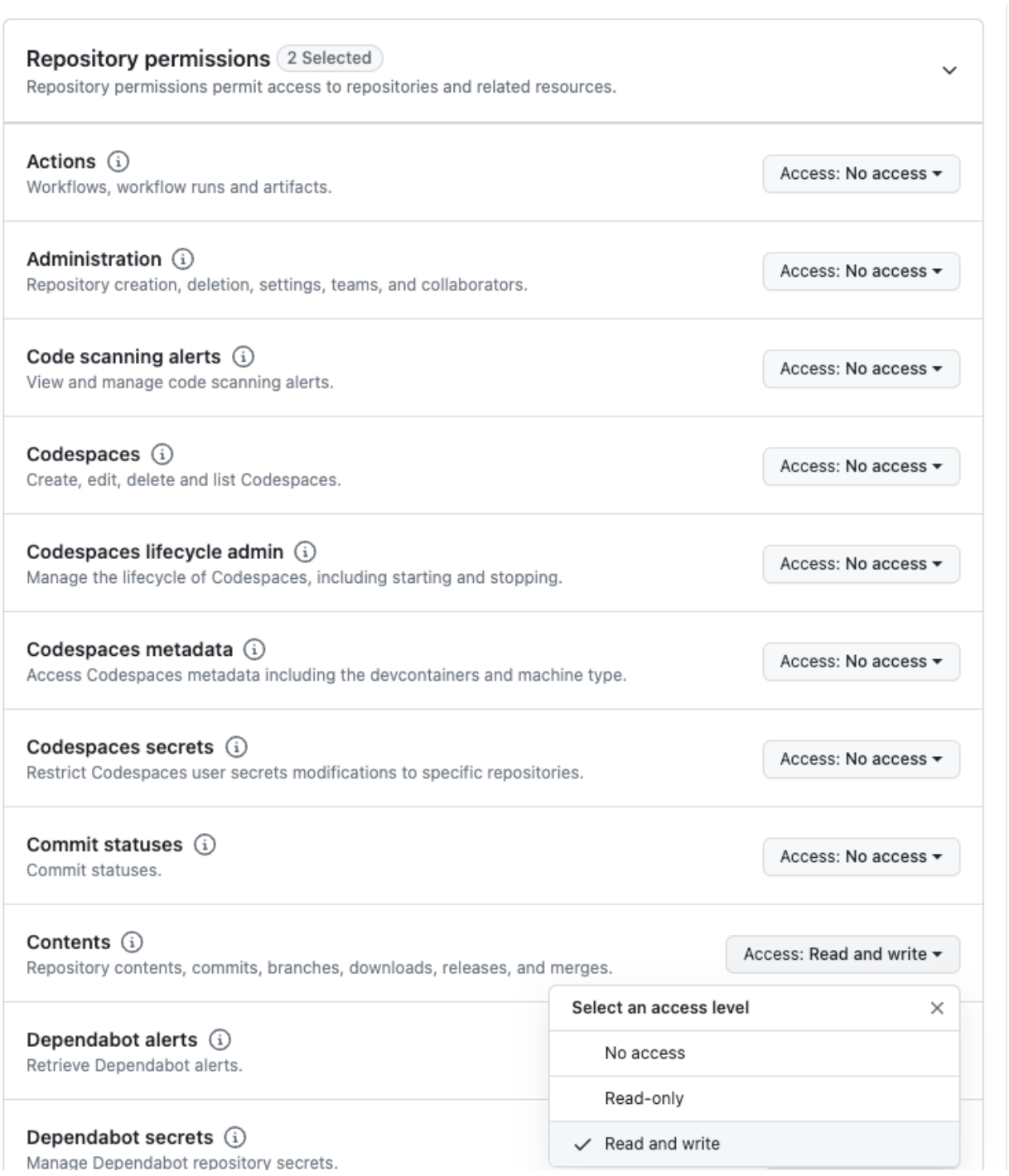
7. Let's now create a Personal Access Token (PAT). Click on ***Developer settings,*** *then click on* ***Personal access tokens,*** then ***Fine-grained tokens***. Click on the ***Generate new token*** button.



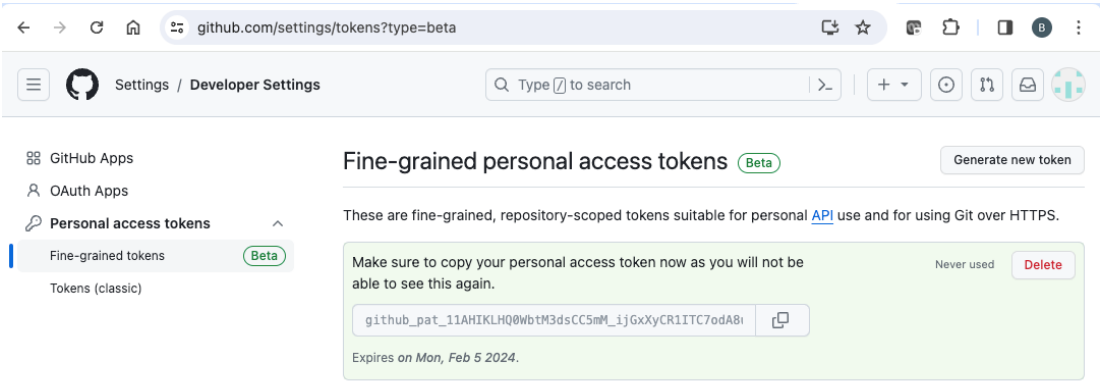
8. Enter text for the ***Token Name*** field and a ***Description***. You can leave the ***Expiration*** timeframe and ***Resource owner*** as-is. Under ***Repository access***, click on ***Only select repositories***, search for your repository "*sec-demo"* and select it.



9. Under the ***Repository permissions*** section, find the ***Contents*** row and change the selection to ***Access: Read and write***.



10. Click on the green ***Generate token*** at the bottom. A new token will be generated. Copy the token and store it somewhere for use in the next steps.



11. Go to a terminal, clone your repo down using the https protocol. (Substitute your actual GitHub userid for <github-userid>.) Then cd into the directory, make a simple change or create a new file. Add and commit the change, and then push it.

$ git clone https://github.com/<github-userid>/sec-demo

$ cd sec-demo

$ git checkout dev

$ echo "Repo for security demos :lock:" > README.md

$ git add README.md

$ git commit -m "Add README file"

$ git push -u origin dev

12. When you do the push, you'll be prompted for username (your GitHub username) and then a sign-in/Private Access Token or password. Wherever it asks for a token or a password, you can just copy and paste in the token you generated in GitHub in the previous steps. An example dialog that may come up is shown below.

If instead, you are on the command line and prompted for a password, just paste the token in at the prompt. Note that it will not show up on the line, but you can just hit enter afterwards.

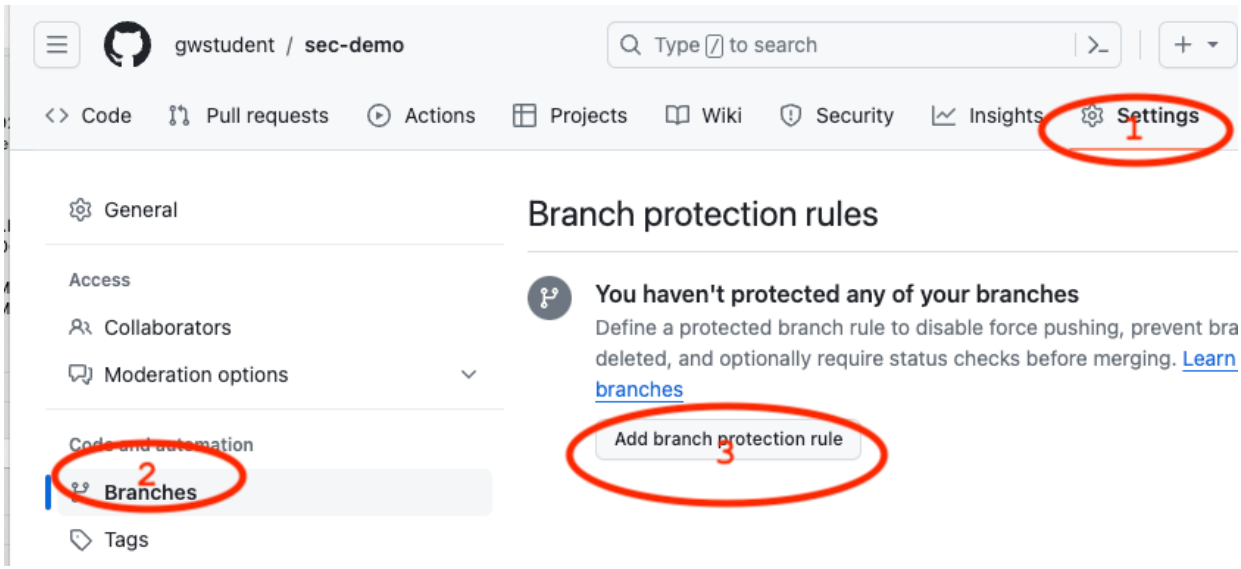
```
developer@Bs-MacBook-Pro sec-demo % git push -u origin dev
Username for 'https://github.com': gwstudent
Password for 'https://gwstudent@github.com':
```

END OF LAB

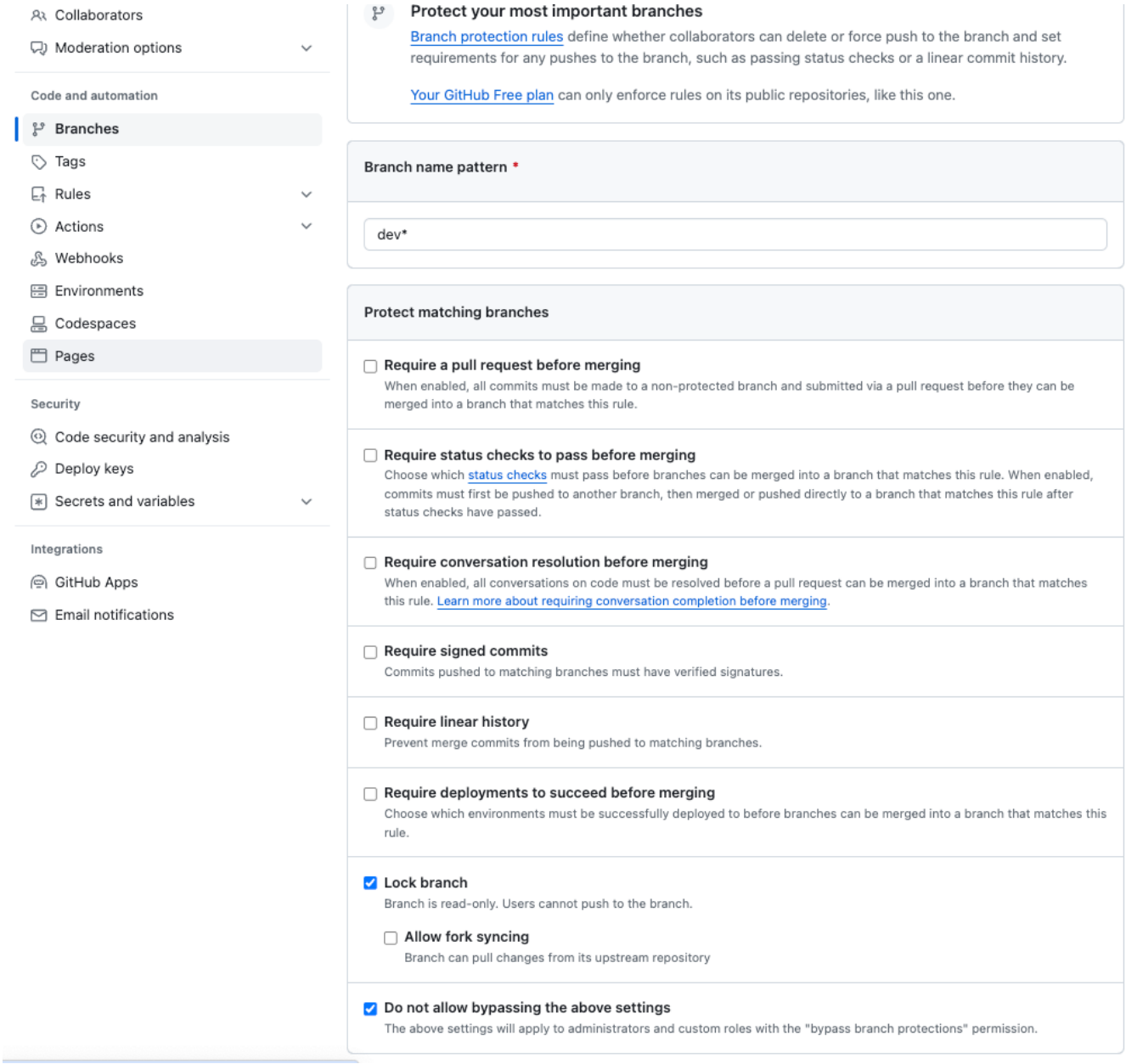
Lab 2 – Branch protection

Purpose: *In this lab, we'll see how to set up some branch protection rules and use them*

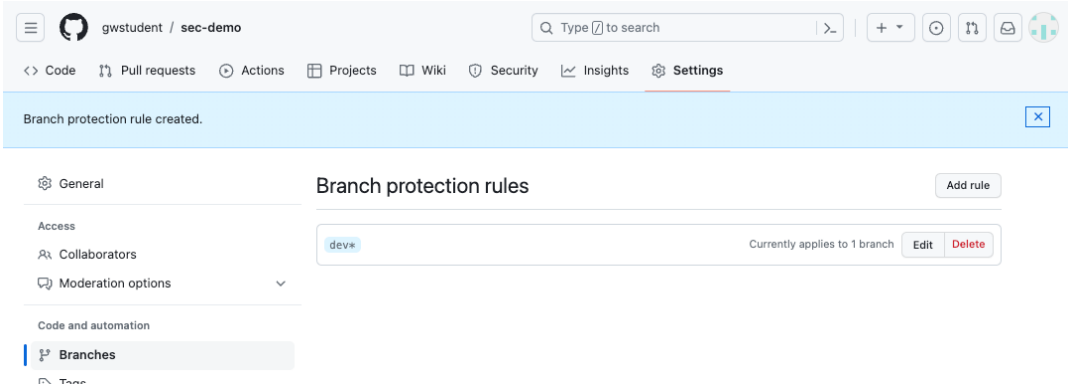
1. Our current repo does not have any protection rules setup. Let's set some up. We have two options, standalone branch protection rules or rulesets. First, we'll look at the standalone option. Go to the ***Settings*** tab for the repository, then select ***Branches*** on the left. In the new screen, click on the ***Add branch protection*** rule button.



2. At the end of the last lab, we pushed to branch "dev" in our repository. Let's lock down all dev* branches. In the ***Branch name pattern*** field, enter the pattern **dev*.** Under the ***Protect matching branches*** section, check the box in the row for ***Lock branch***. Also, check the box in the row for ***Do not allow bypassing the above settings*** - since you're the owner of the repo.



3. Click the ***Create*** button at the bottom of the page to save the rule and apply it.



4. Let's see how the branch rule works in practice. Go back to your terminal and make another change, commit it, and then push it. (You'll need to have your fine-grained token from the previous lab.) An example change is shown below, but you can make any change you want.

$ echo "For educational purposes only" >> README.md

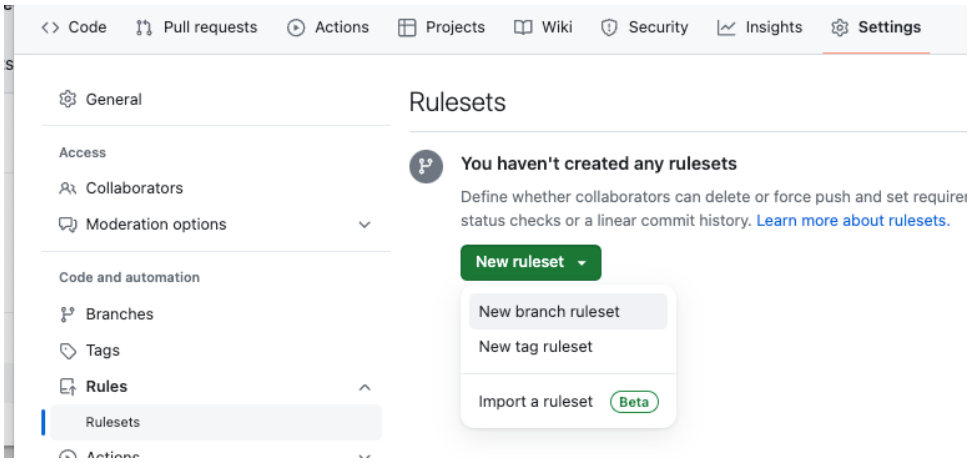
$ git commit -am "Update README"

$ git push origin dev

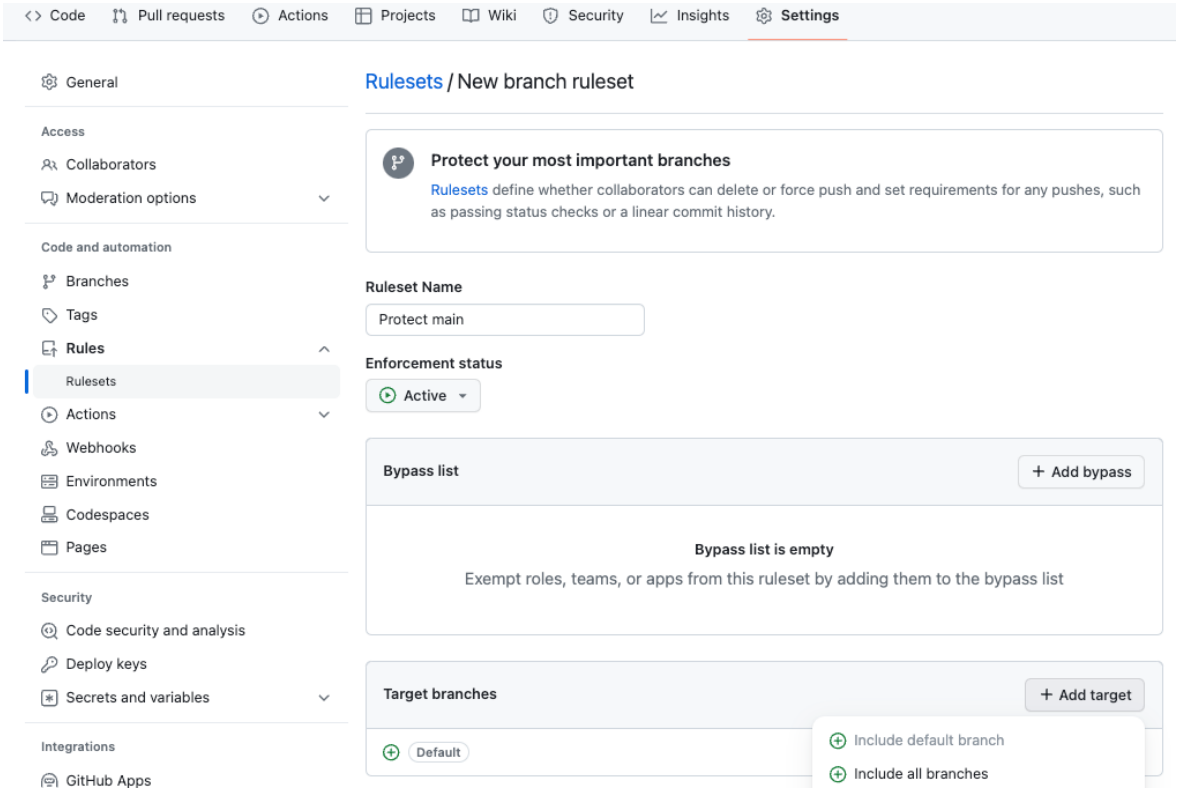
5. At this point, you'll see an error message for your attempted push since the dev branch is locked down.

```
developer@Bs-MacBook-Pro sec-demo % git push origin dev
Username for 'https://github.com': gwstudent
Password for 'https://gwstudent@github.com':
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 321 bytes | 321.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote: error: GH006: Protected branch update failed for refs/heads/dev.
remote: error: Cannot change this locked branch
To https://github.com/gwstudent/sec-demo
 ! [remote rejected] dev -> dev (protected branch hook declined)
error: failed to push some refs to 'https://github.com/gwstudent/sec-demo'
```

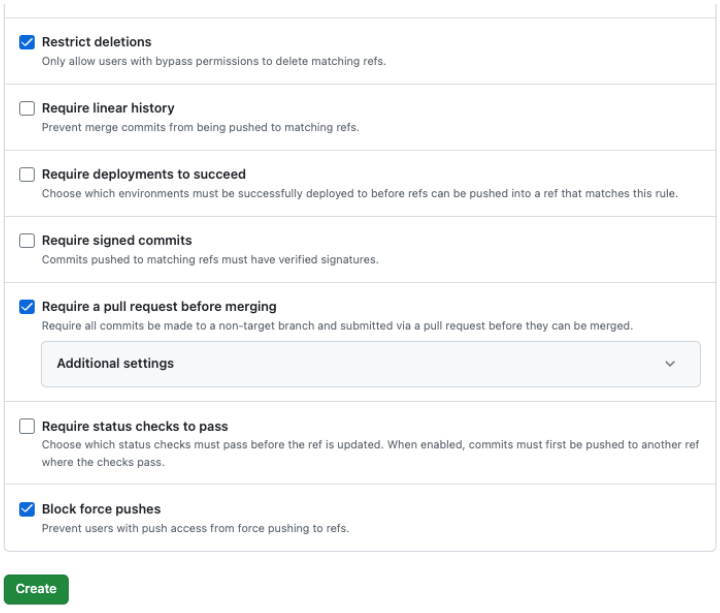
6. The other option for protecting branches is rulesets. Let's set up a ruleset next. Still in the repository's ***Settings*** tab, select ***Rules*** on the left, then ***Rulesets***. Then click on the ***New ruleset*** button and ***New branch ruleset***.



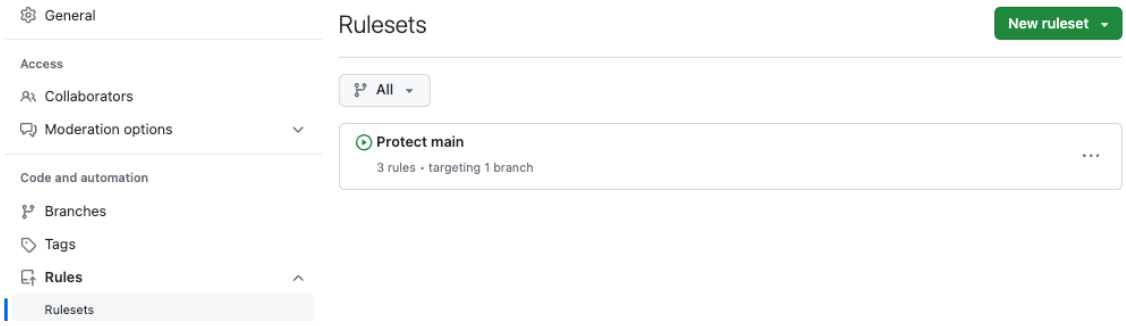
7. Enter a ***Ruleset Name*** (such as "Protect main"), set the ***Enforcement*** status to **Active**, and for ***Target branches***, you can just click ***+Add target*** and select "*Include default branch*".



8. For the actual ***Branch protections***, you can just leave the ones checked that are already selected, and check the box for the one for ***Require a pull request before merging***. You don't need to check any of the other boxes for *Additional settings* right now. Then click the ***Create*** button.



Afterwards, if you click on ***Rules*** or *Rulesets* again, you should see the item you just created.



9. Now, let's see this one in practice. Switch back to your terminal and attempt to push the change out to your GitHub repo. Once again, you'll need your fine-grained token.

$ git checkout main
$ echo "Copyright 2024" >> LICENSE
$ git commit -am "update license"
$ git push origin main

10. You should see a message like the following indicating that you can push the changed code due to the repository rule violation.

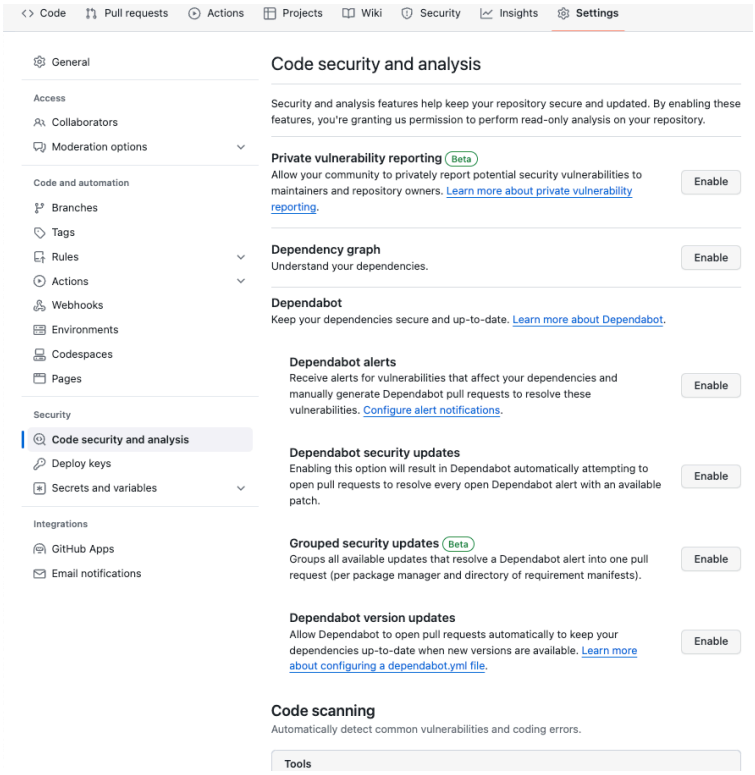
```
developer@Bs-MacBook-Pro sec-demo % git push origin main
Username for 'https://github.com': gwstudent
Password for 'https://gwstudent@github.com':
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes | 323.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote: error: GH013: Repository rule violations found for refs/heads/main.
remote: Review all repository rules at http://github.com/gwstudent/sec-demo/rule
s?ref=refs%2Fheads%2Fmain
remote:
remote: - Changes must be made through a pull request.
remote:
To https://github.com/gwstudent/sec-demo
 ! [remote rejected] main -> main (push declined due to repository rule violatio
ns)
error: failed to push some refs to 'https://github.com/gwstudent/sec-demo'
```

END OF LAB

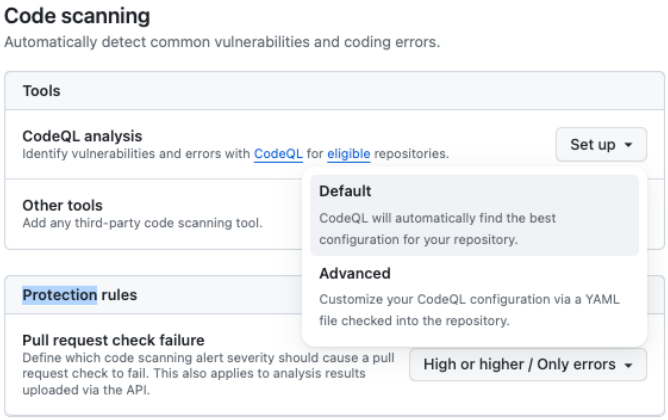
Lab 3 – Code Scanning

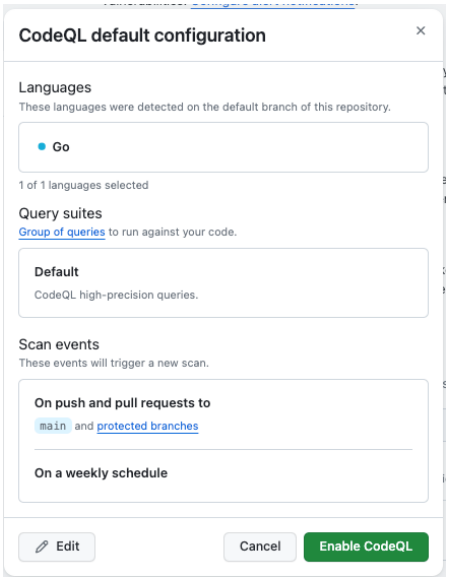
Purpose: *In this lab, we'll see how to enable code scanning on our repository*

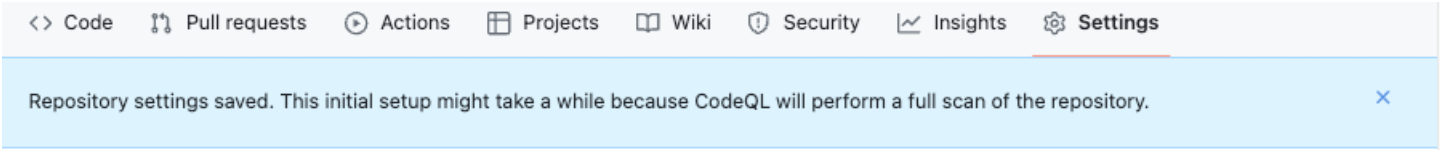
1. An important part of keeping your repository secure is code scanning. Let's enable the default setup for code scanning in our repo via CodeQL and GitHub Actions. Go to your repo's ***Settings*** tab and under "Security" on the left, select ***Code security and analysis***. Near the bottom, you'll see the "Code Scanning" section.



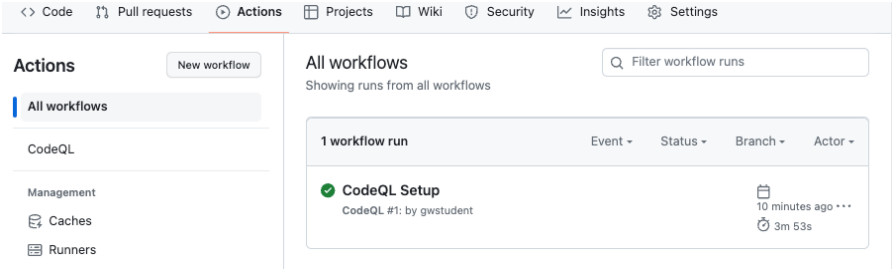
2. In the ***Tools / CodeQL*** analysis section, click on the ***Set up*** button and select the ***Default*** configuration. On the dialog that comes up with the default options, click the ***Enable CodeQL*** button. After that, you'll see a blue bar saying that a scan of the entire repo will happen.

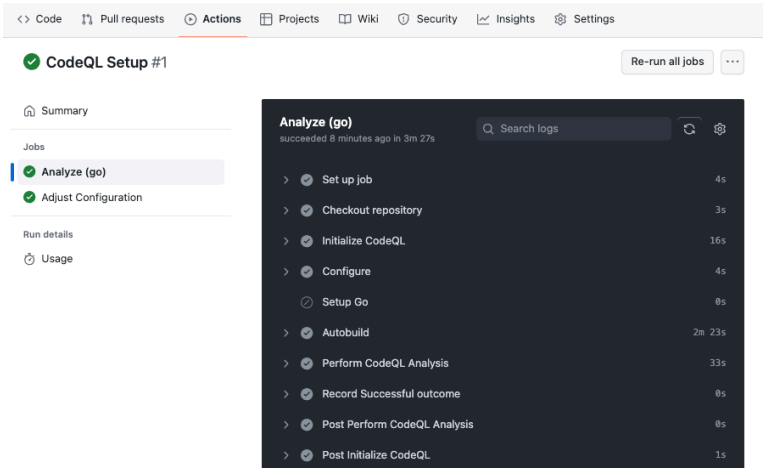




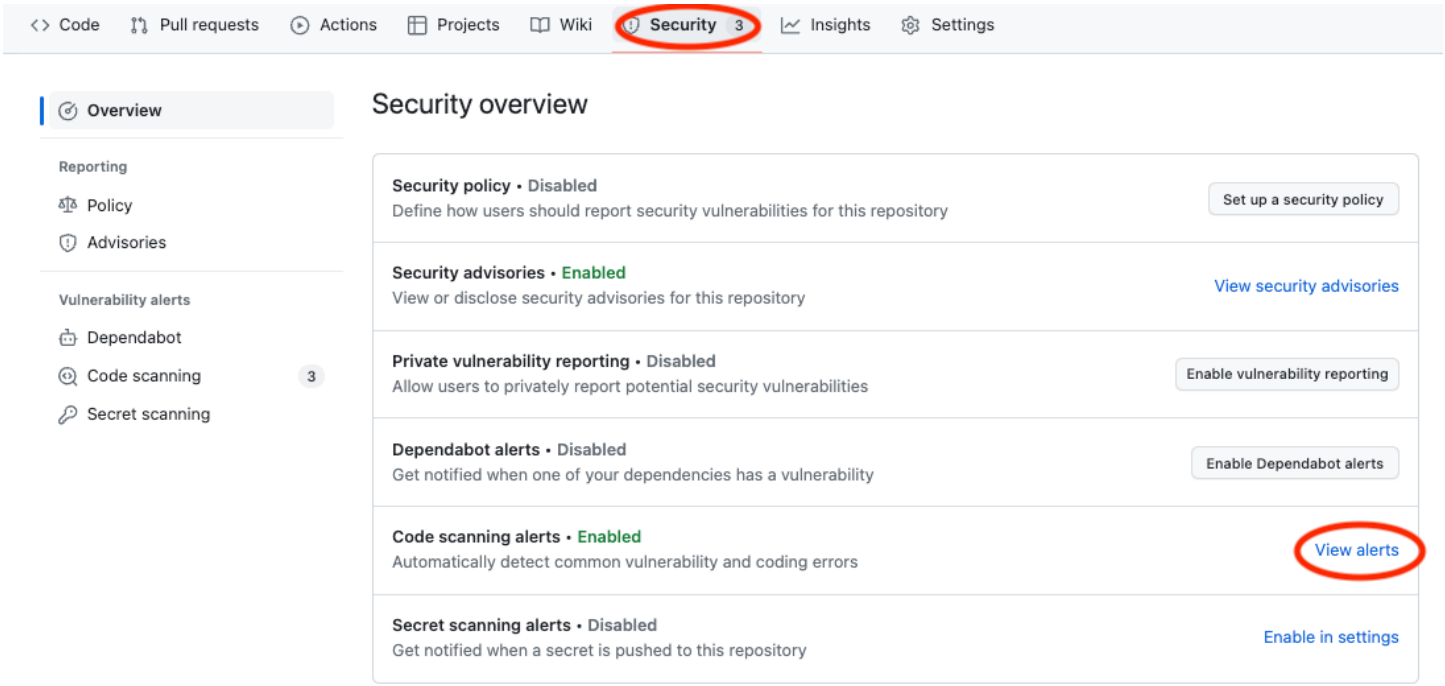


3. The setup will take several minutes to run. After it starts, you can click on the ***Actions*** tab in your repository, and you'll see that you have a new Actions workflow named "CodeQL" that is running against your repo. After it completes, if you want, you can click on the ***CodeQL Setup*** item under ***All workflows*** on the right and drill in to the jobs that ran.

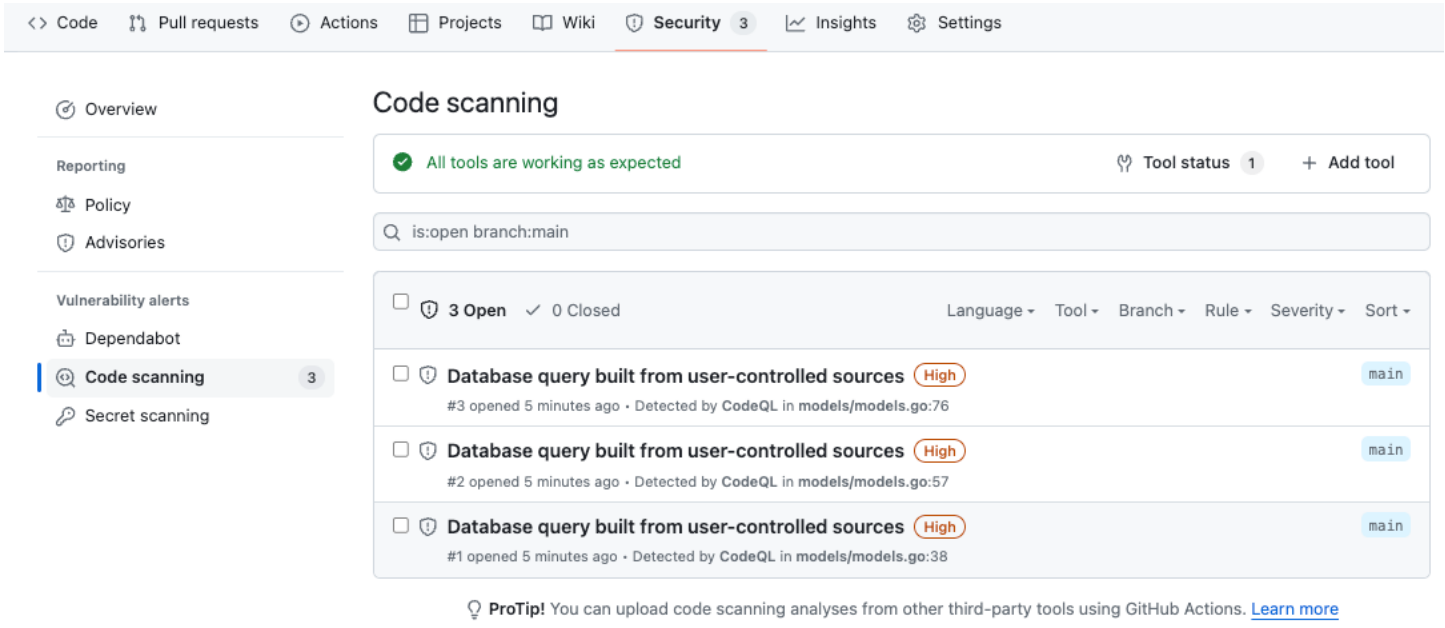




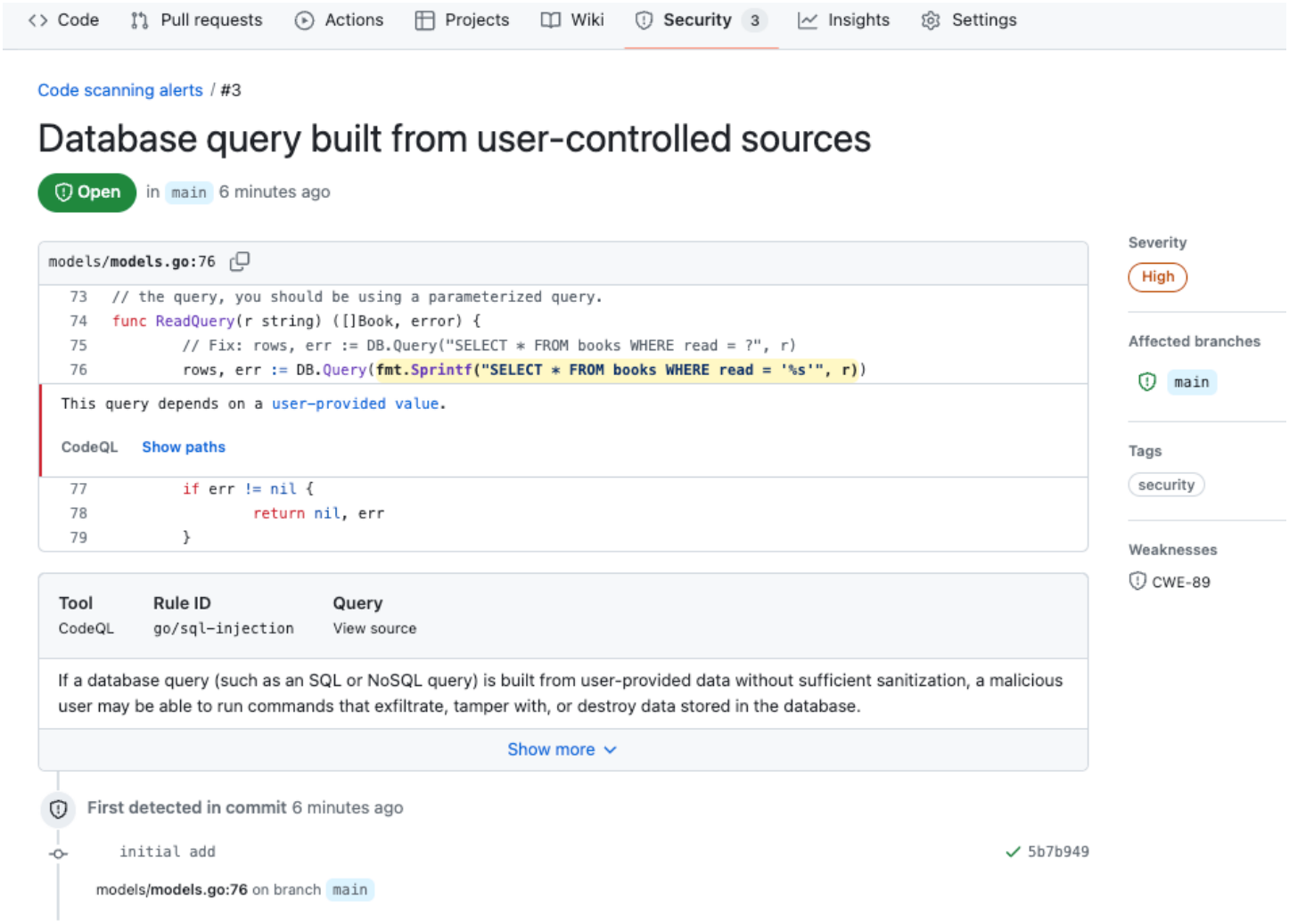
4. Now, let's see if any vulnerabilities were found. Click on the ***Security*** tab of the repo, find the row under ***Security Overview*** for ***Code scanning alerts*** and select ***View alerts***. The ***Security*** tab at the top should show a 3 next to it indicating that vulnerabilities were found.



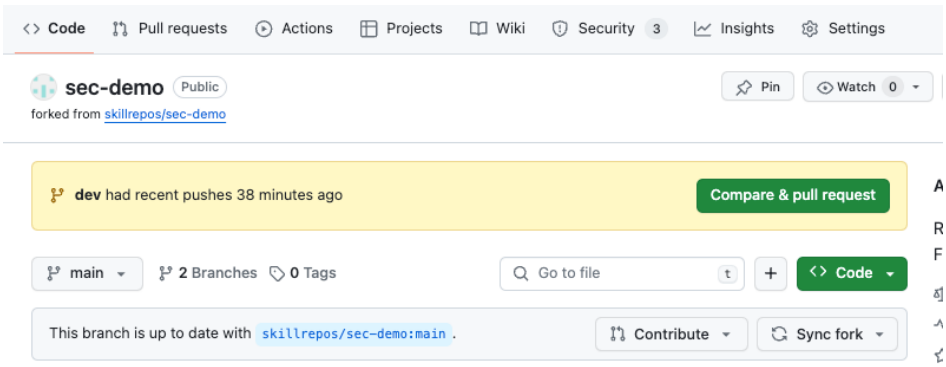
5. On the next screen, you should see a set of alerts found in the current code.



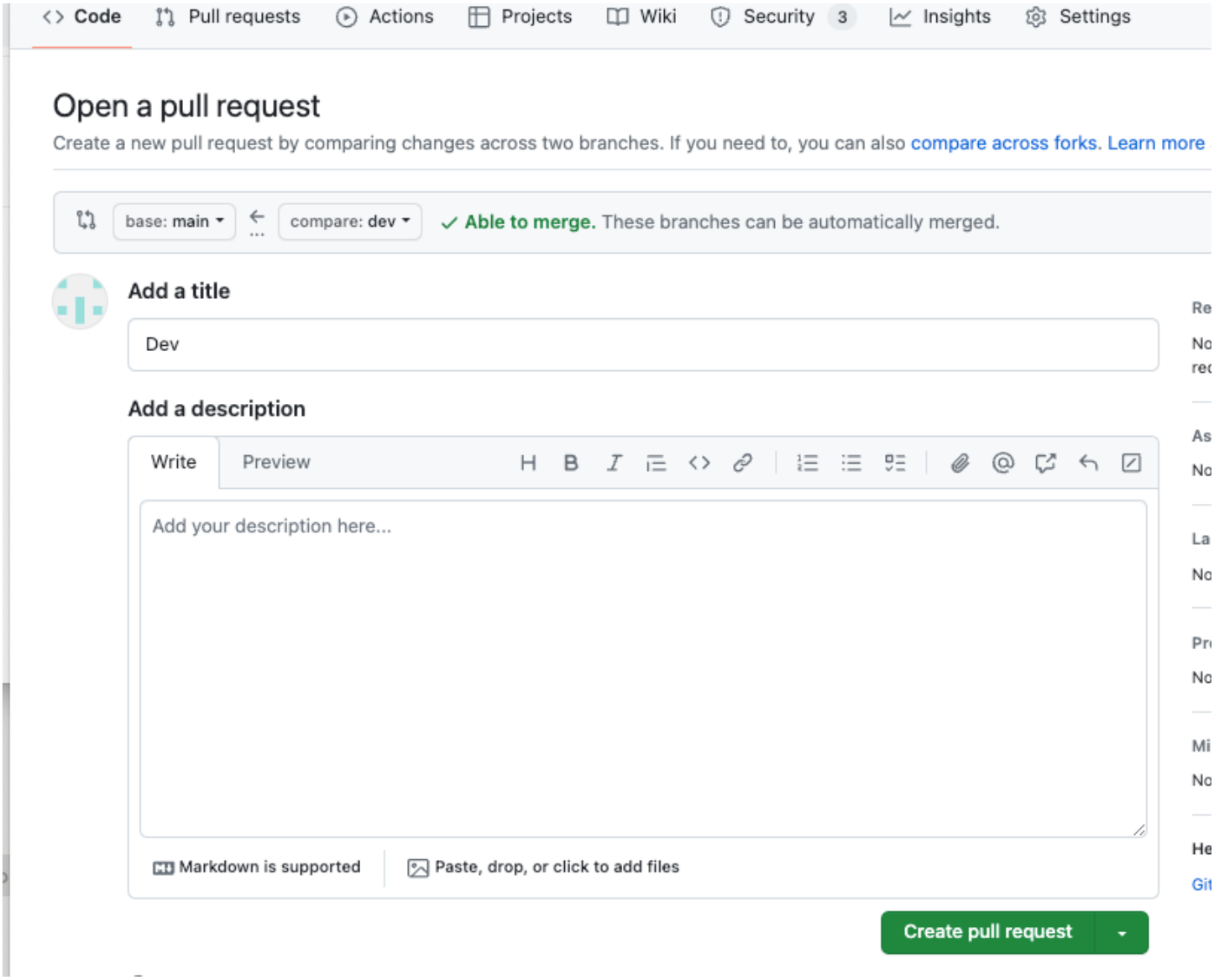
6. Click on the top one and you'll see more detail about the issue. This includes a link to more info about the vulnerability on the right under "Weaknesses".



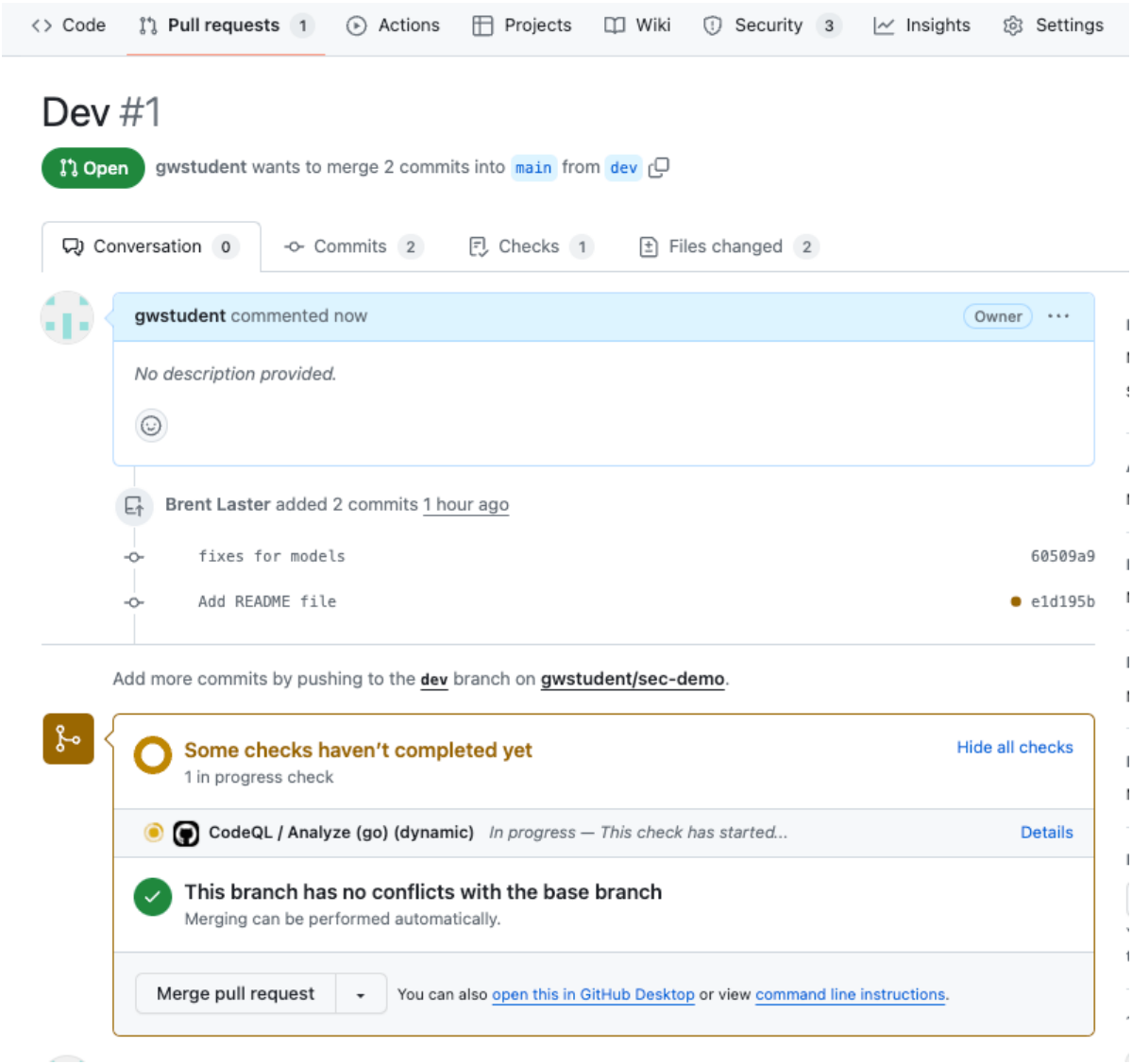
7. The *dev* branch already has fixes in it for these issues. Go back to the ***Code*** tab in the repo. Near the top, there should be a yellow bar noting that there were recent pushes to *dev* and showing a button to ***Compare & pull request***. (If you don't see this, you can initiate a new pull request via the ***Pull requests*** menu at the top.) Click on that button.



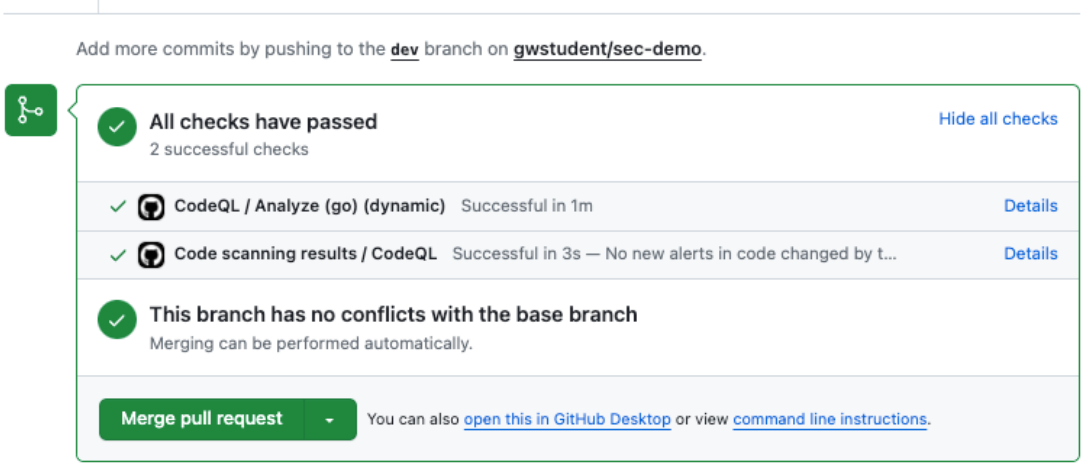
8. On the next screen, in the dropdowns on the gray bar, make sure to select the *main* branch of your current repo as the **base** on the left and the *dev* branch of your current repo as the **compare** selection. Enter a title and description if you want and then click on the green ***Create pull request*** button.



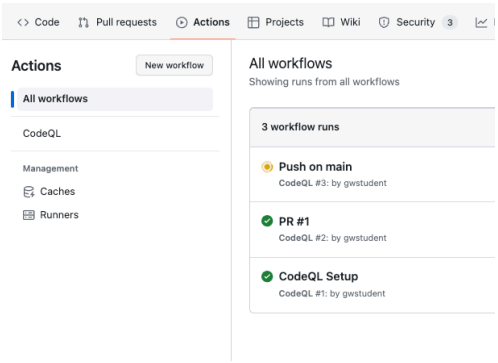
9. On the next screen, notice that the CodeQL analysis is automatically added as a check for being able to merge the changes. While you're waiting for it to complete, you can look at at the changes in the ***Commits*** tab or ***Files changed*** tab.



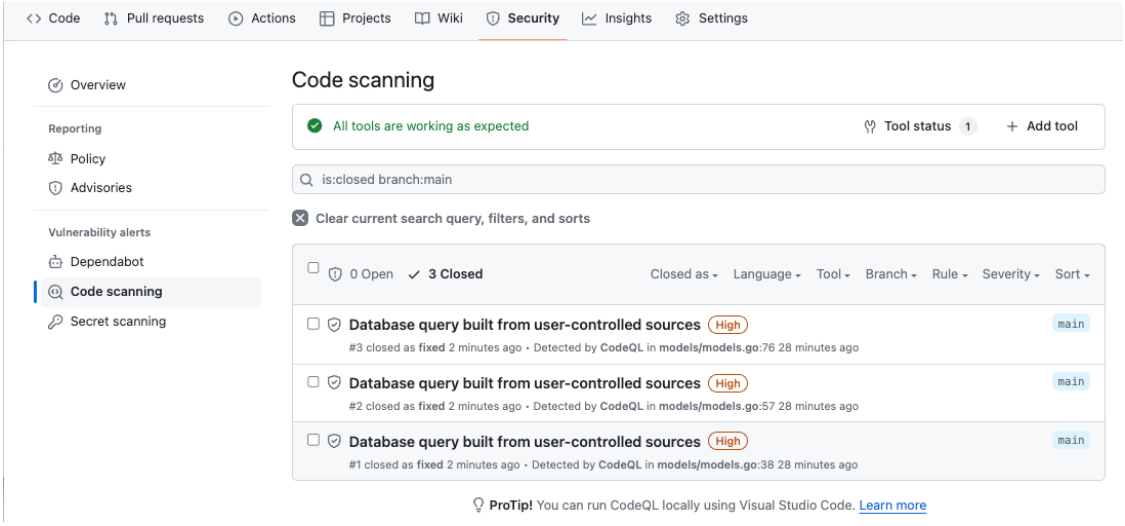
10. Once the checks have completed, the screen will show *All checks have passed*. You can then go ahead, and ***Merge pull request*** and ***Confirm merge***.

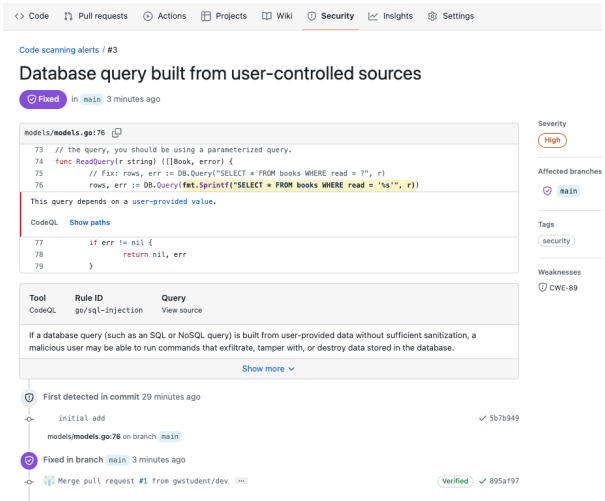


11. When the push is made to main as a result of the merge, this will kick off another run of the *CodeQL* workflow. You'll be able to see that in the Actions menu.



12. After this completes, if you go back to the ***Security*** tab, you'll see that the alerts have been closed as fixed. (Click on the ***3 Closed*** link and individual items to see details.)



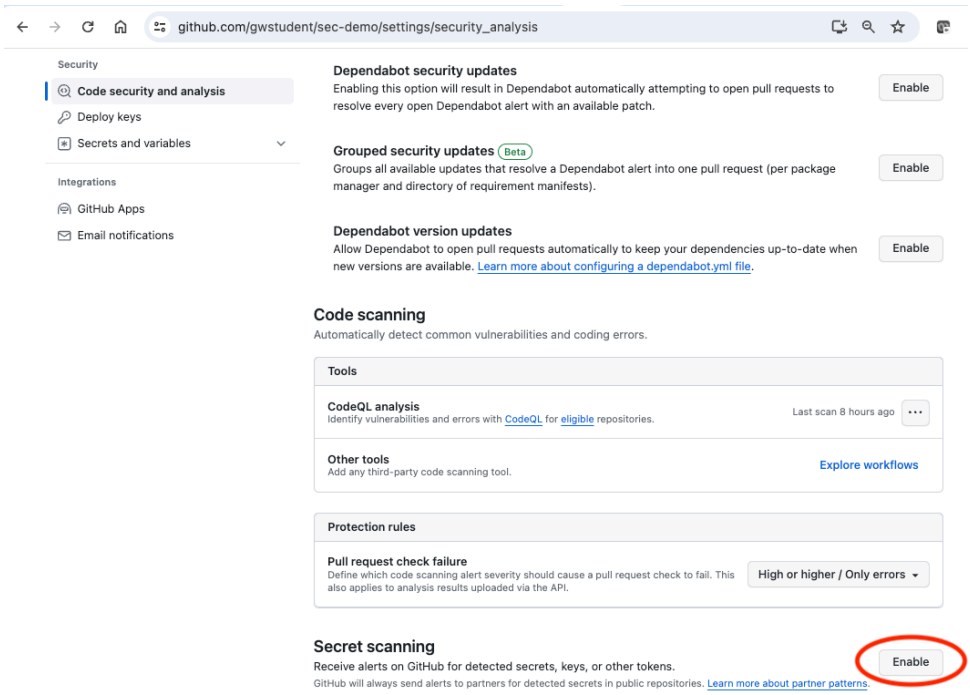


END OF LAB

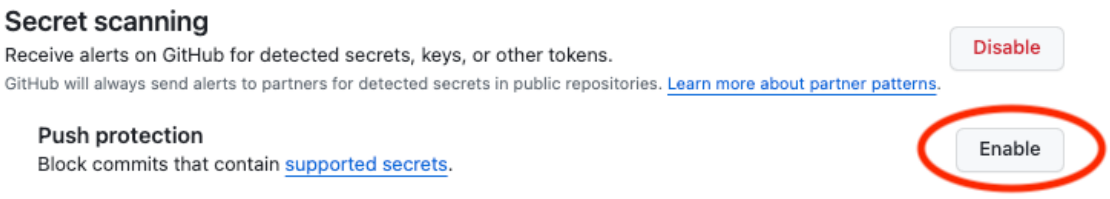
Lab 4 – Secret Scanning

Purpose: *In this lab, we'll see how to enable secret scanning on our repository*

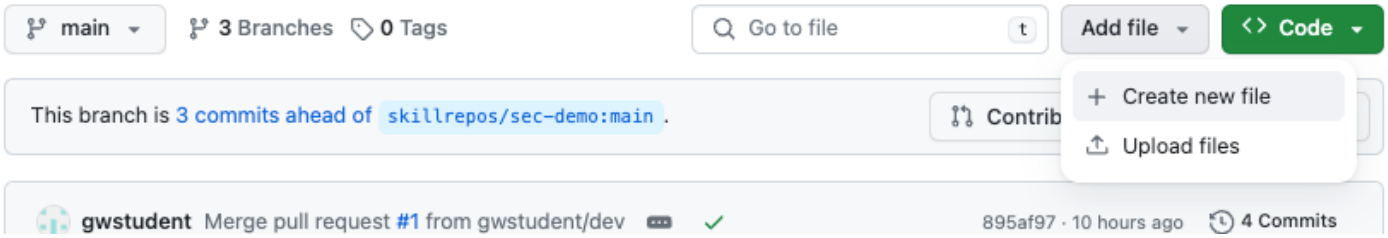
1. Just as we used code scanning to find vulnerabilities in the code in our repositories, we can also have GitHub scan for secrets that may be exposed in our repository files. To enable that functionality, go to the repository's ***Settings***, then back to ***Code security and analysis*** on the left and then scroll down to the bottom of the page. In the ***Secret scanning*** section, click the ***Enable*** button.



2. Also select the button to *Enable* the **Push protection** option to *Block commits that contain supported secrets*.

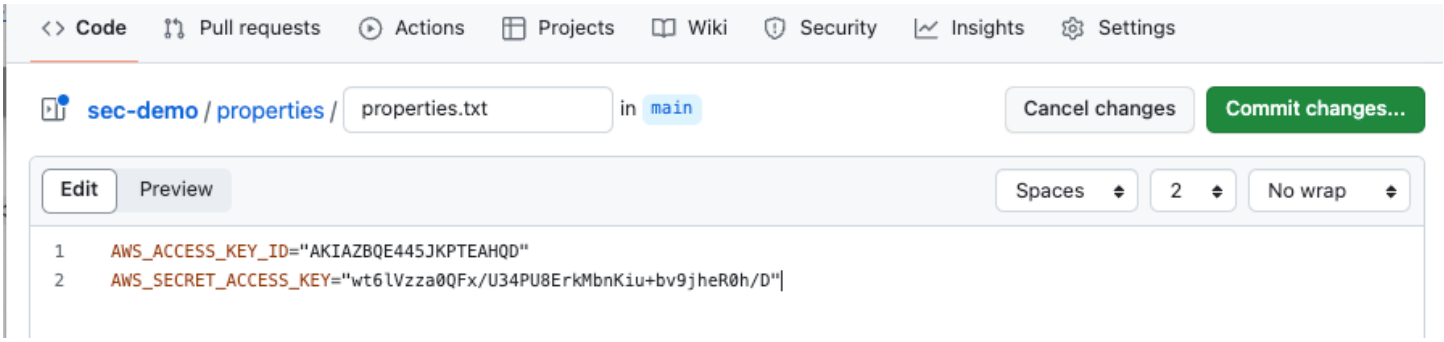


3. Let's add a file with secrets to see how the blocking works. In the ***Code*** tab, click on ***Add File*** and then ***+ Create new file***.

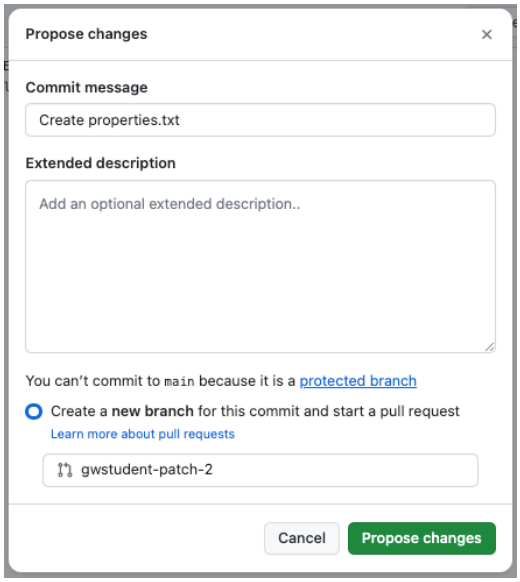


4. In the new file screen, name the file properties/properties.txt (add that path in the text entry box at the top after "sec-demo") and copy and paste the following content into the contents area under the "Edit" button.

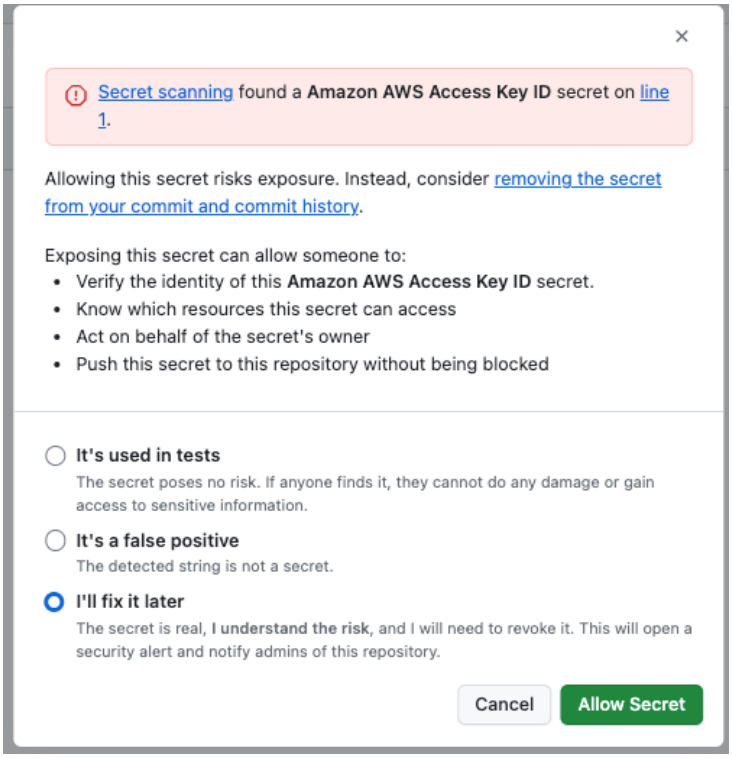
AWS_ACCESS_KEY_ID="AKIAZBQE445JKPTEAHQD"
AWS_SECRET_ACCESS_KEY="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"



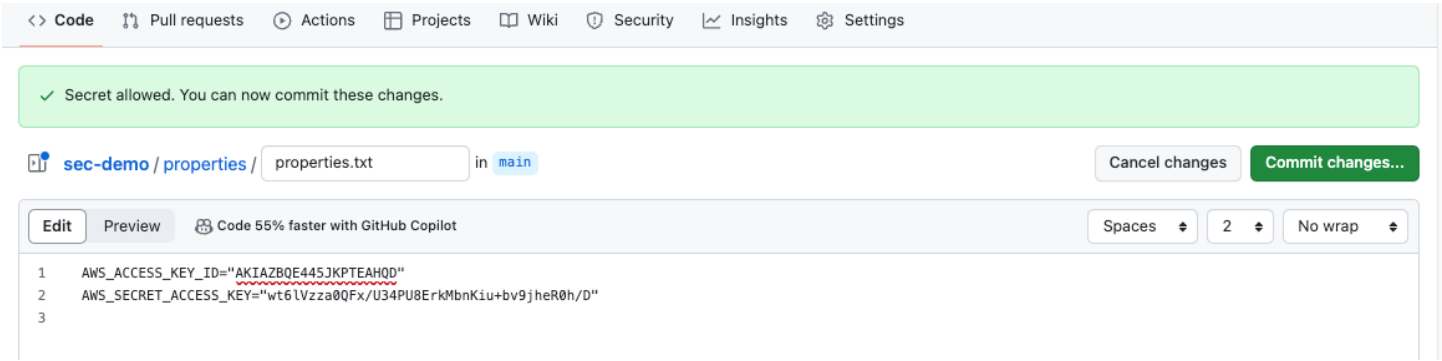
5. Click on the ***Commit changes…*** button. Since we have the current branches protected, go ahead and accept the option to create a new branch and click the ***Propose changes*** button.



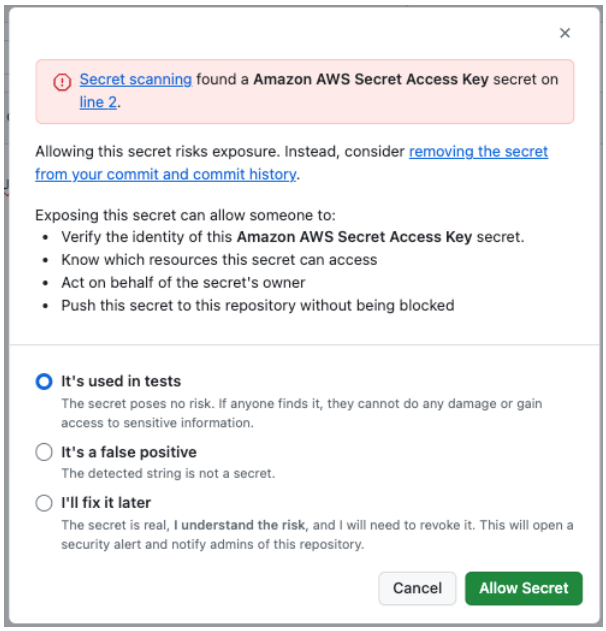
6. At this point, you'll see a warning dialog because we have secret scanning turned on in the repository and we are trying to push a file with a secret in it. For purposes of this exercise, just select the "*I'll fix it later*" option and then click on the ***Allow Secret*** button.



7. At this point, you'll see a screen acknowledging that the secret is allowed and you can commit. Click on the ***Commit changes…*** button.

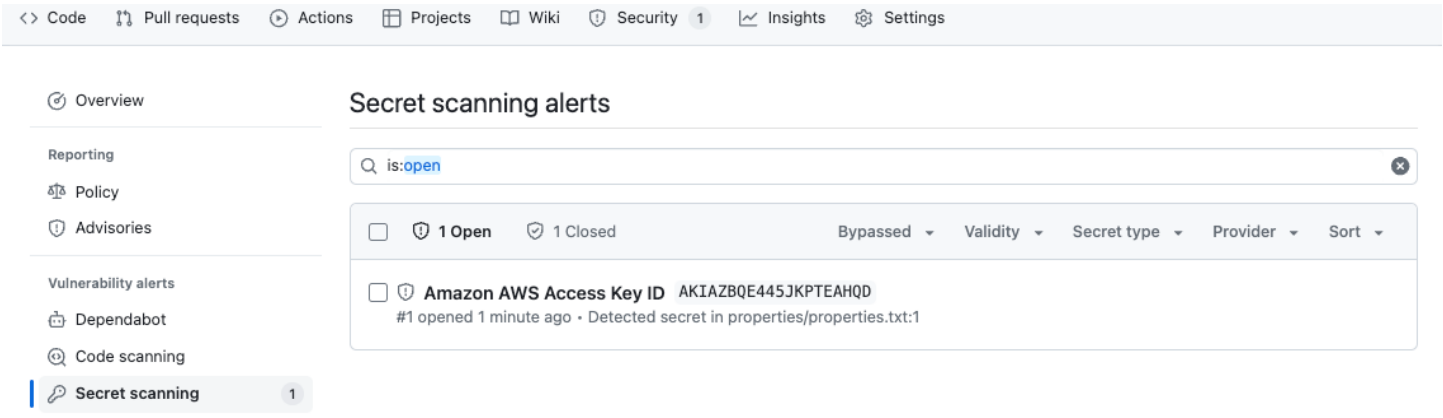


8. Click on the ***Propose changes*** button on the next dialog. Then you'll be at a similar dialog for the second secret as you had for the first one. This time, select "*It's used in tests*" and then click on the ***Allow Secret*** button.



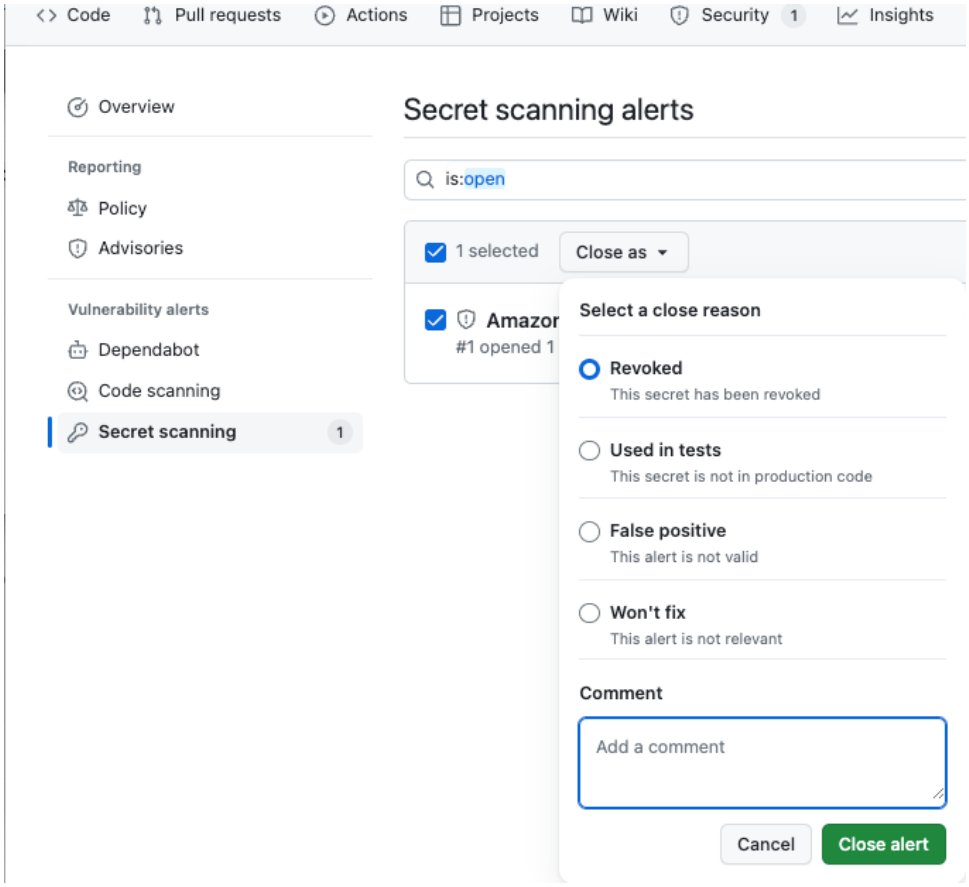
9. As before, you'll see the screen saying the secret is allowed. Click on the ***Commit changes…*** button and then the ***Propose changes*** button on the next dialog.

10. Now, if you click on the *Security* tab, the *Secret Scanning alerts* section on the page, and the *View detected secrets* options on the side, you'll see the one secret we said we would fix later.

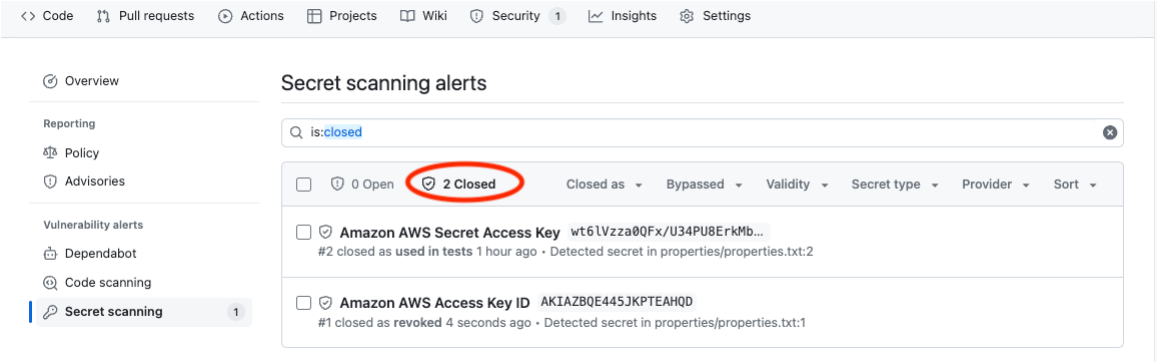


11. Assume that we have actually revoked this secret and now want to close out the alert. Check the box next to the secret to select it. A new button titled ***Close as*** will appear. Click on that and select

Revoked as the reason to close the alert. Enter a comment if desired and then click on the ***Close alert*** button.



12. After closing the alert, you can click on the 2 Closed link and see the alert that was automatically closed when you said it was used in tests and the one you recently closed as revoked.

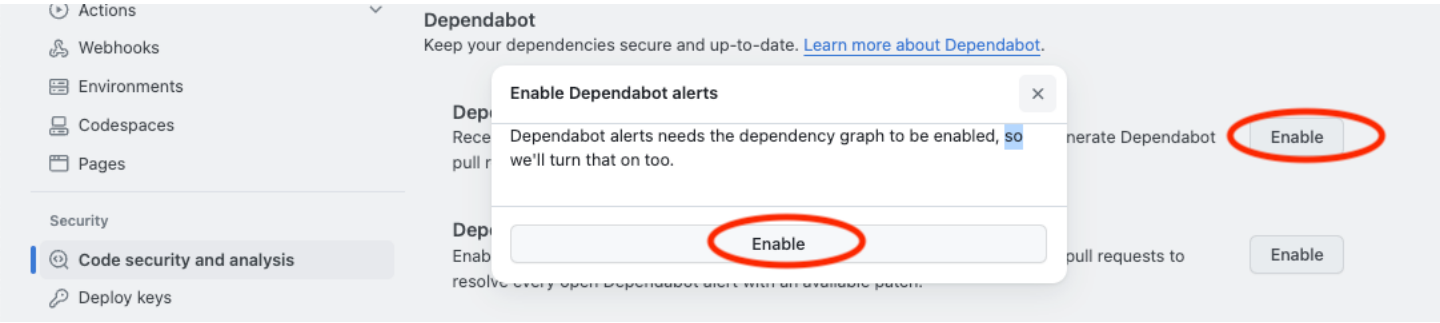


END OF LAB

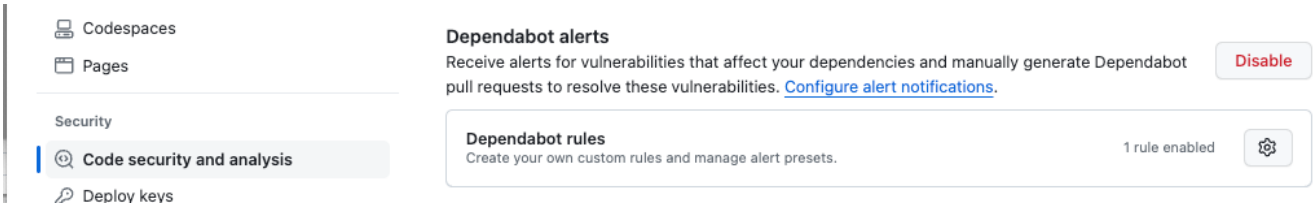
Lab 5 – Working with Dependabot

Purpose: *In this lab, we'll see how to use Dependabot to keep dependencies up to date.*

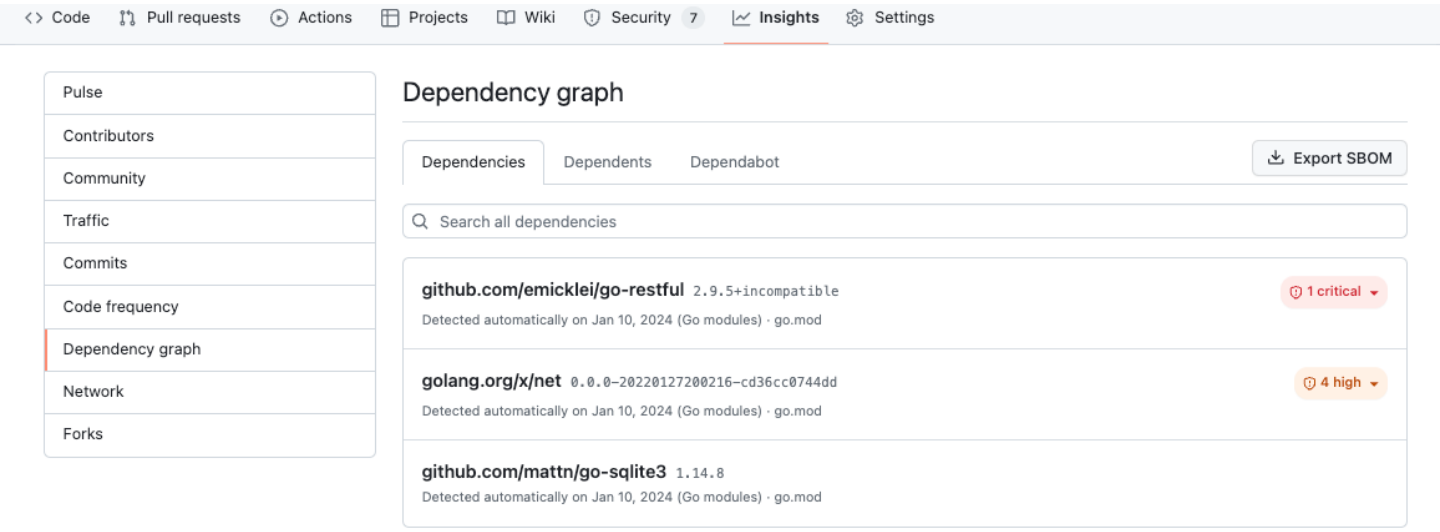
1. Let's enable Dependabot functionality in our repository. Go to the ***Settings*** tab for the repository, then select ***Code security and analysis*** on the left side again and in the *Dependabot* section of that page, click the button for *Dependabot alerts* to enable them. In the dialog that comes up about Enable Dependabot alerts, you can just click on the **Enable** button.



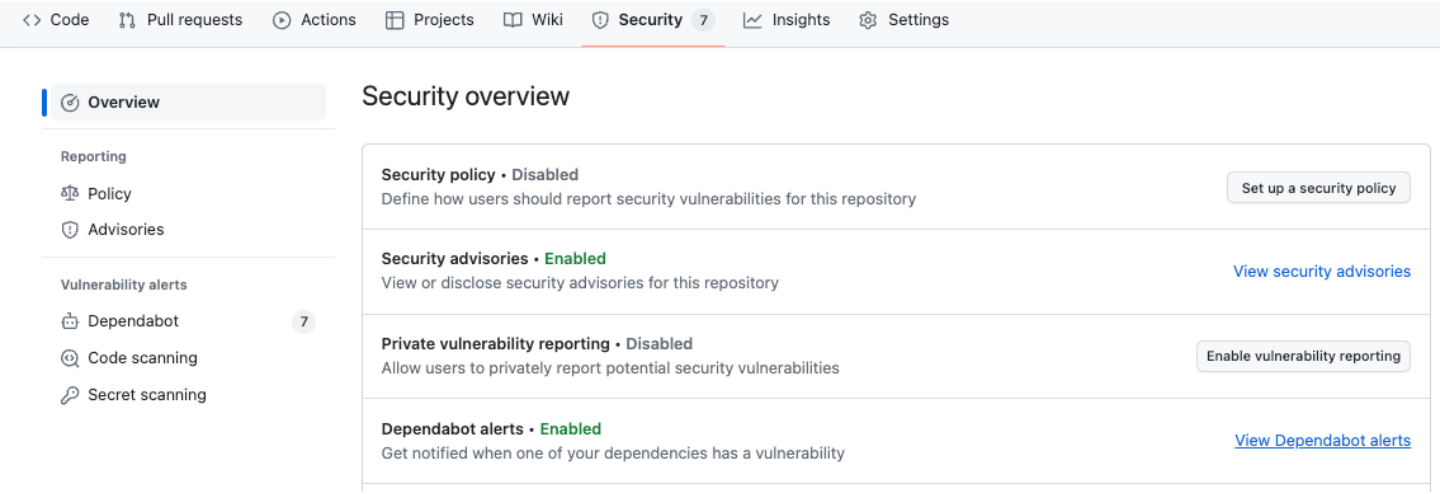
2. After this, you'll see that Dependabot has 1 rule enabled. You can click on the gear if you want to see more details about the rule.



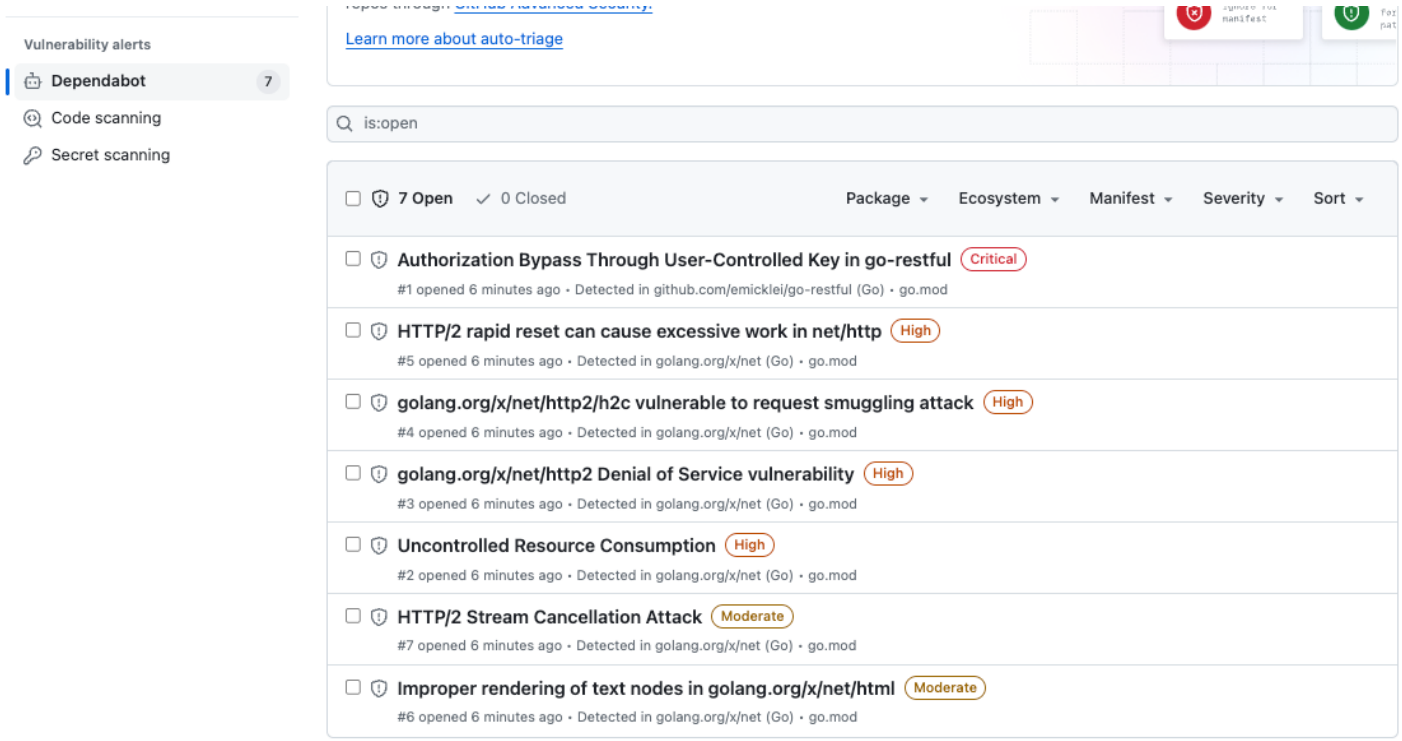
3. If you want to see the dependency graph info, you can click on the ***Insights*** tab at the top and then the ***Dependency graph*** item on the left.



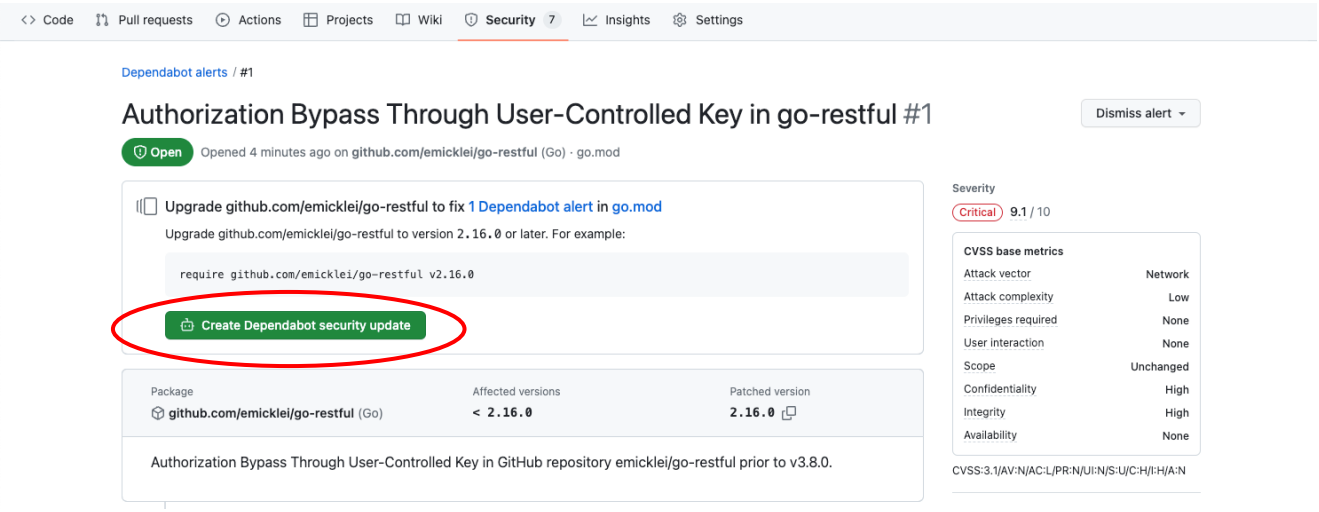
4. To see the issues Dependabot found, next click on the ***Security*** tab and then in the ***Dependabot alerts*** row, click on *View Dependabot alerts*.



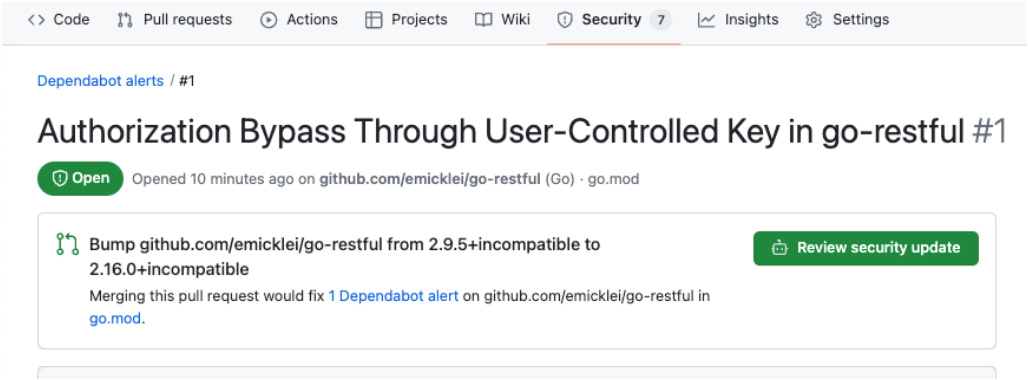
5. At this point, you'll see 7 dependabot alerts that show up, ranging from Critical to Moderate in severity.



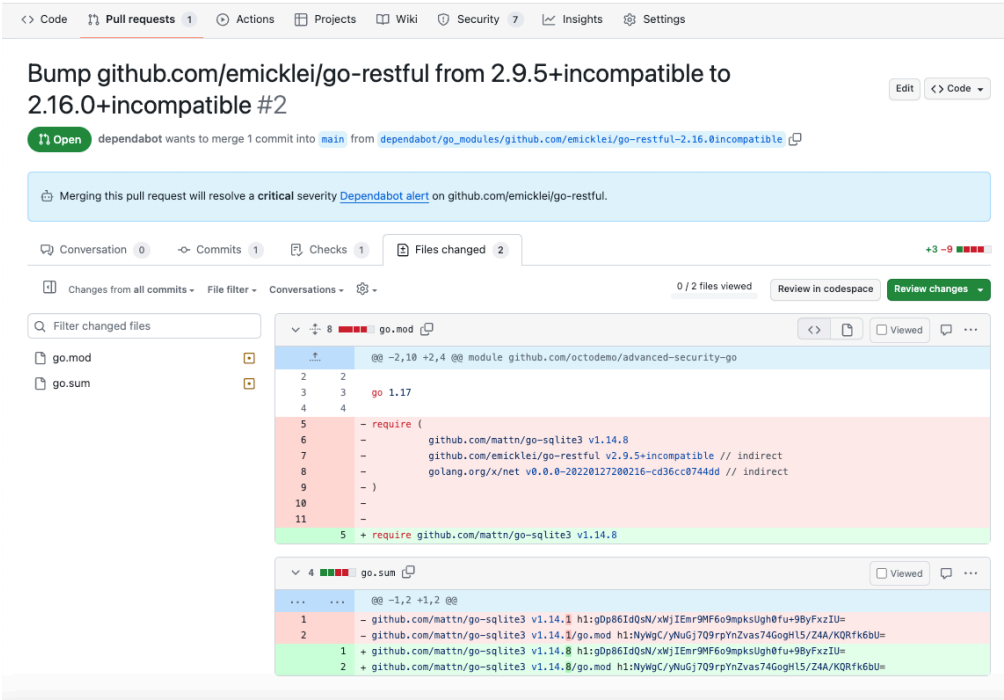
6. Click on the first alert. You can review the information shown and then click on the ***Create Dependabot security update*** button.



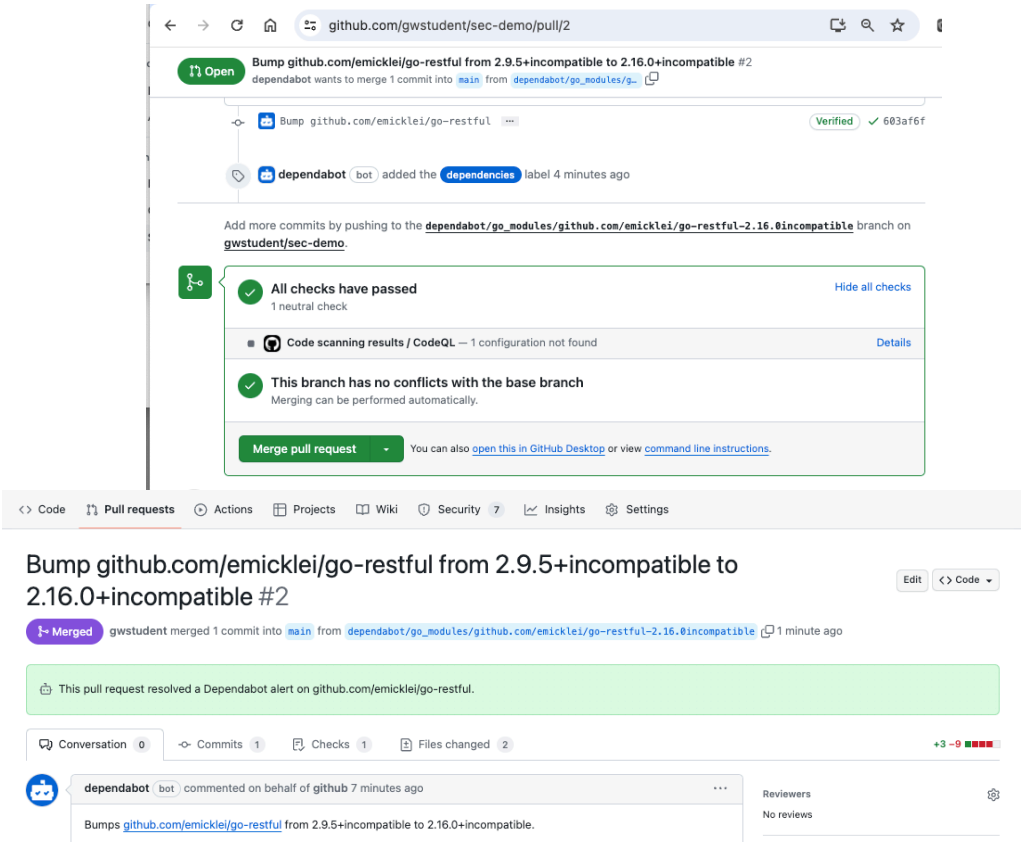
7. After the creation step runs, GitHub will generate a new security update. Click on the ***Review security update*** button to see details of the update.



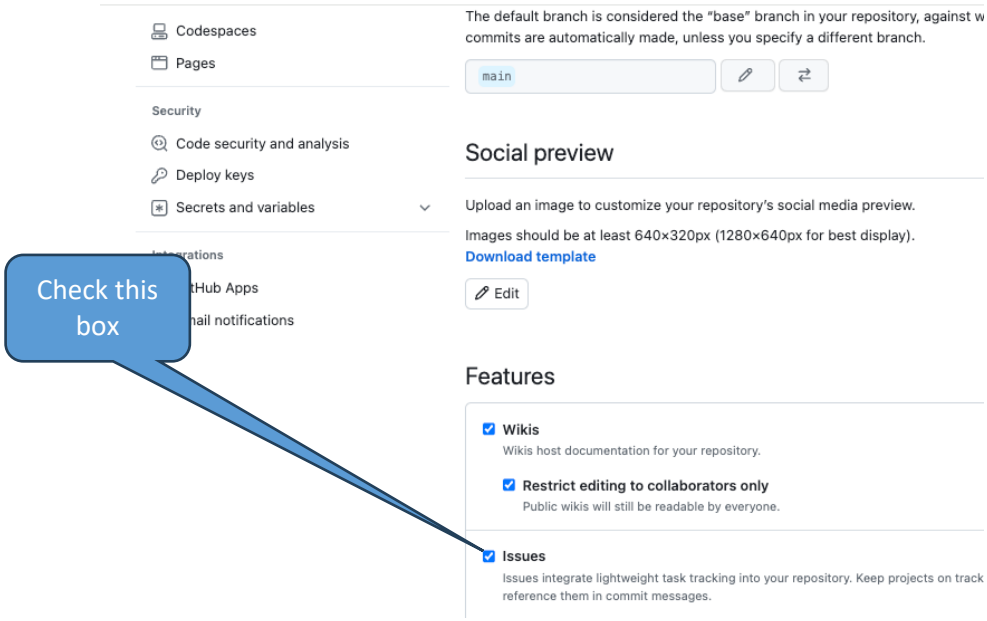
8. This will open up the security update which is actually a pull request. Click on the ***Commits*** tab and then the ***Files changed*** tab to see the changes. In this case, the two dependencies that were causing issues weren't needed, so Dependabot removed them. More typically, it would create a pull request to update the version.



9. When ready, change back to the **Conversation** tab of the pull request and click the **Merge pull request** button, followed by the **Confirm merge** button. After the merge is complete, you'll see a green bar telling you the pull request resolved the Dependabot alert.



10. In preparation for the next lab, enable the Issues functionality on your repo by going to the ***Settings*** tab, scrolling down to ***Features***, and then checking the box in the ***Issues*** section.

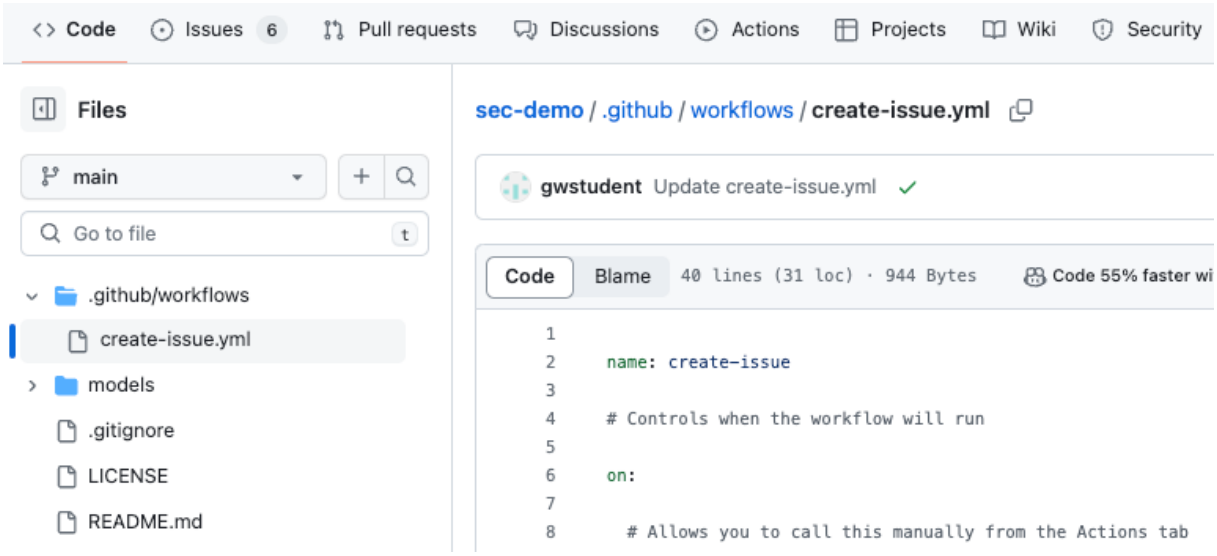


END OF LAB

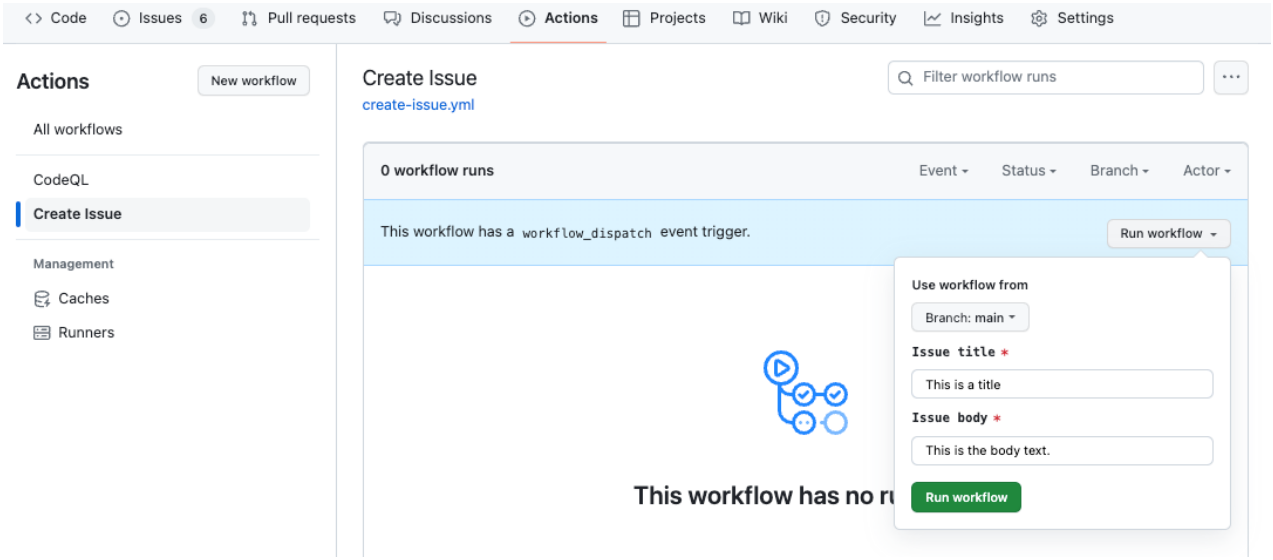
Lab 6 – Securing GitHub Actions

Purpose: *In this lab, we'll see how to use do some preventative security changes for GitHub Actions.*

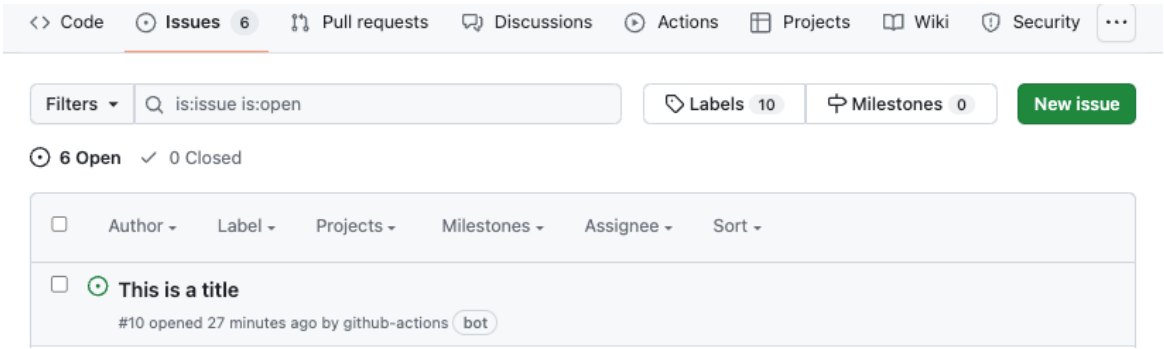
1. When the dev branch was merged with the pull request earlier, it added a GitHub Actions workflow that can be used to automatically create a GitHub issue. In the *Code* tab, find the file *.github/workflows/create-issue.yml* and click on it and take a look at the contents.



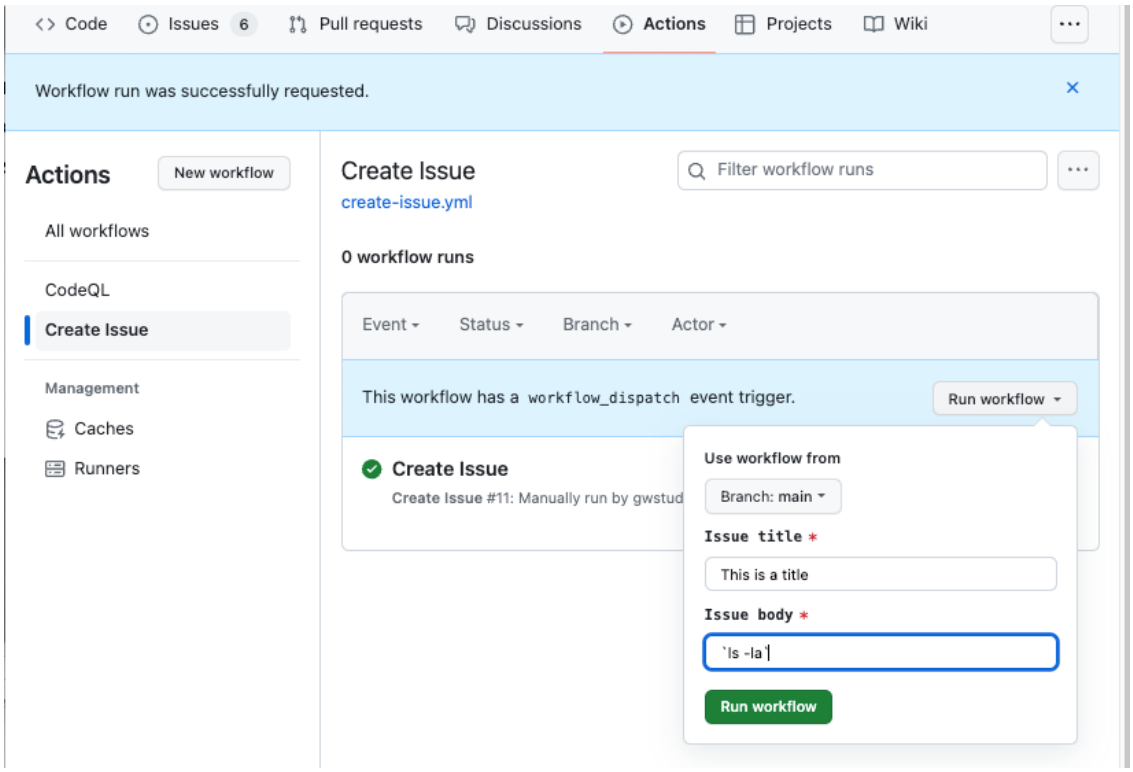
2. We can run this workflow manually to create an issue. Click on the ***Actions*** tab and then under the *All workflows* section on the left, select the ***Create issue*** item. Then click on the ***Run workflow*** button and in the dialog that comes up, select ***Branch: main***, and then you can put in any text for ***Issue title*** and ***Issue body***. Finally, click on the green button to **Run workflow**.



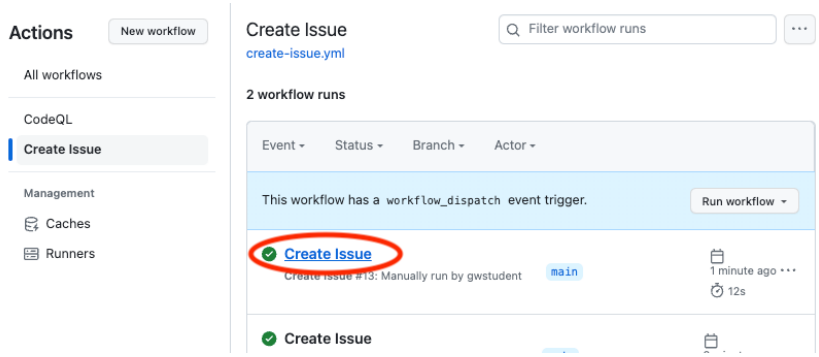
3. After the run of the workflow completes, you should have a new *GitHub Issue* visible under the ***Issues*** tab.



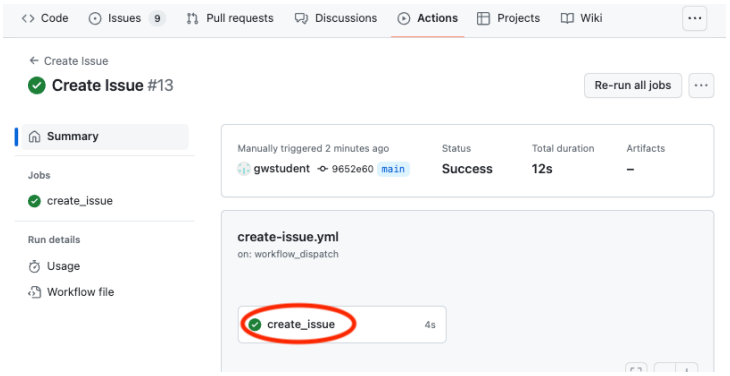
4. The code in this workflow is vulnerable to script injection because of the way the variables are evaluated in the code. To see this, in the *Actions* tab, run the workflow again, put what you want in the *Issue title* field, but in the *Issue body* field, enter the following as the arguments (NOTE: That is two backquotes around ls -la) `` `ls -la` ``



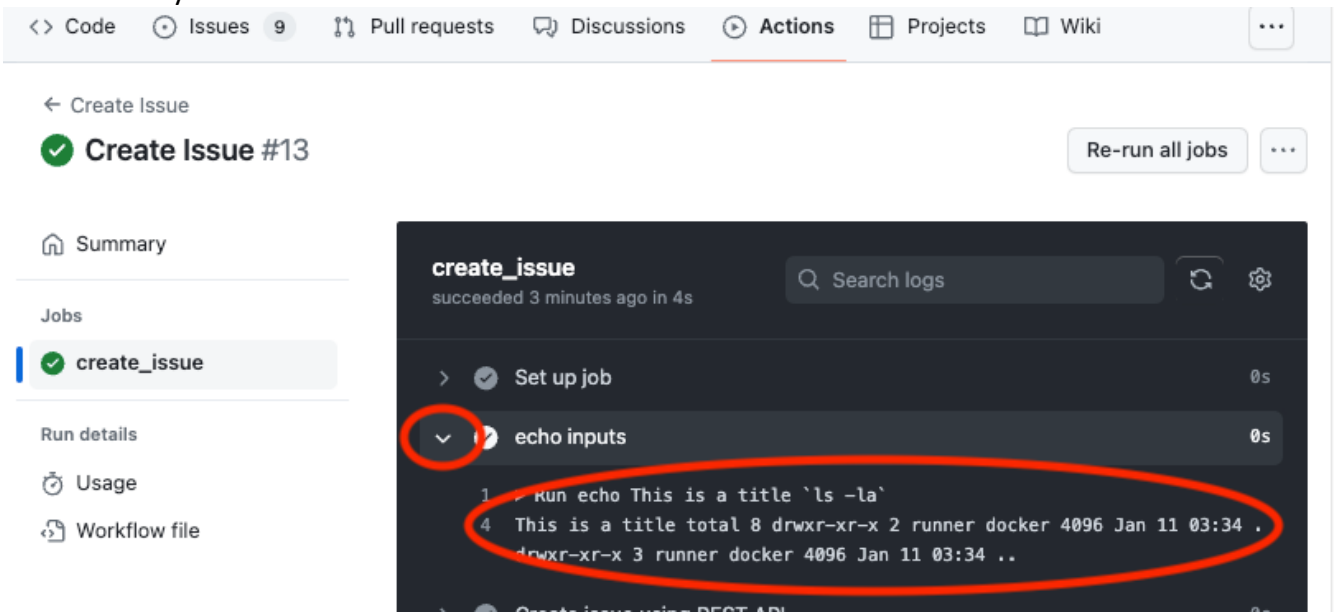
5. After the workflow runs, you'll have another issue created, but the more interesting part is in the execution of the action. Click on the row for the most recent run.



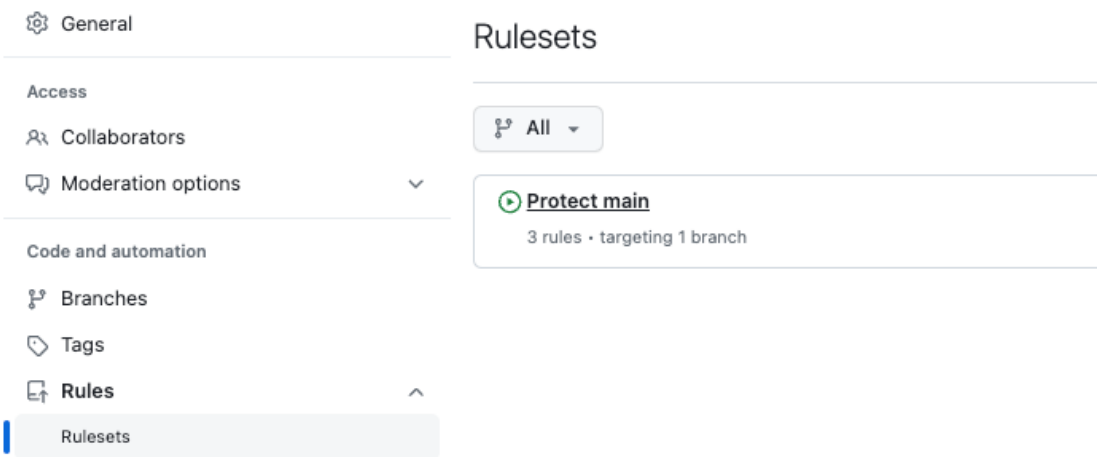
Then click on the job name in the next screen.



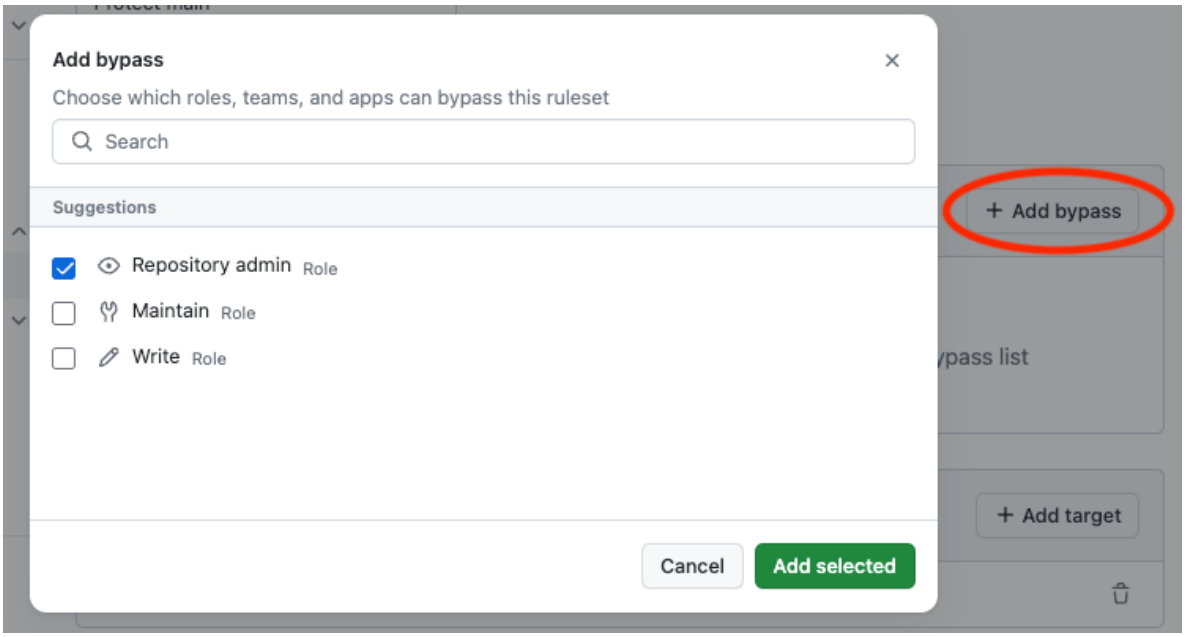
Then click to expand the *echo inputs* step in the log. Notice that the text we put in as the body of the issue was actually executed and run as a command.



6. Now, let's change the code to not directly interpret the values passed in, but to use environment variables instead. To make things simpler, let's add a bypass to the ruleset to allow us to commit directly to main for this activity. Go to the repository's ***Settings***, then select ***Rules*** / ***Rulesets*** on the left, and click on the ruleset to the right that you previously setup.

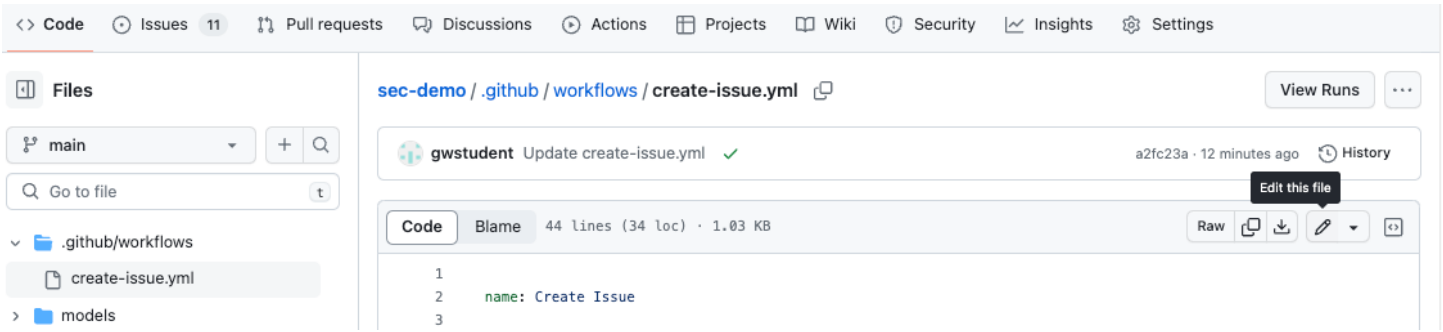


Next, in the page for the ruleset, in the ***Bypass list*** section, click on ***+ Add bypass.*** In the ***Add bypass*** dialog that comes up, select the ***Repository admin*** role, and check that box. Then click on the ***Add selected*** button.



Be sure to click on ***Save changes*** at the bottom of the screen to persist the bypass. After that, you should see a quick pop-up acknowledging the change.

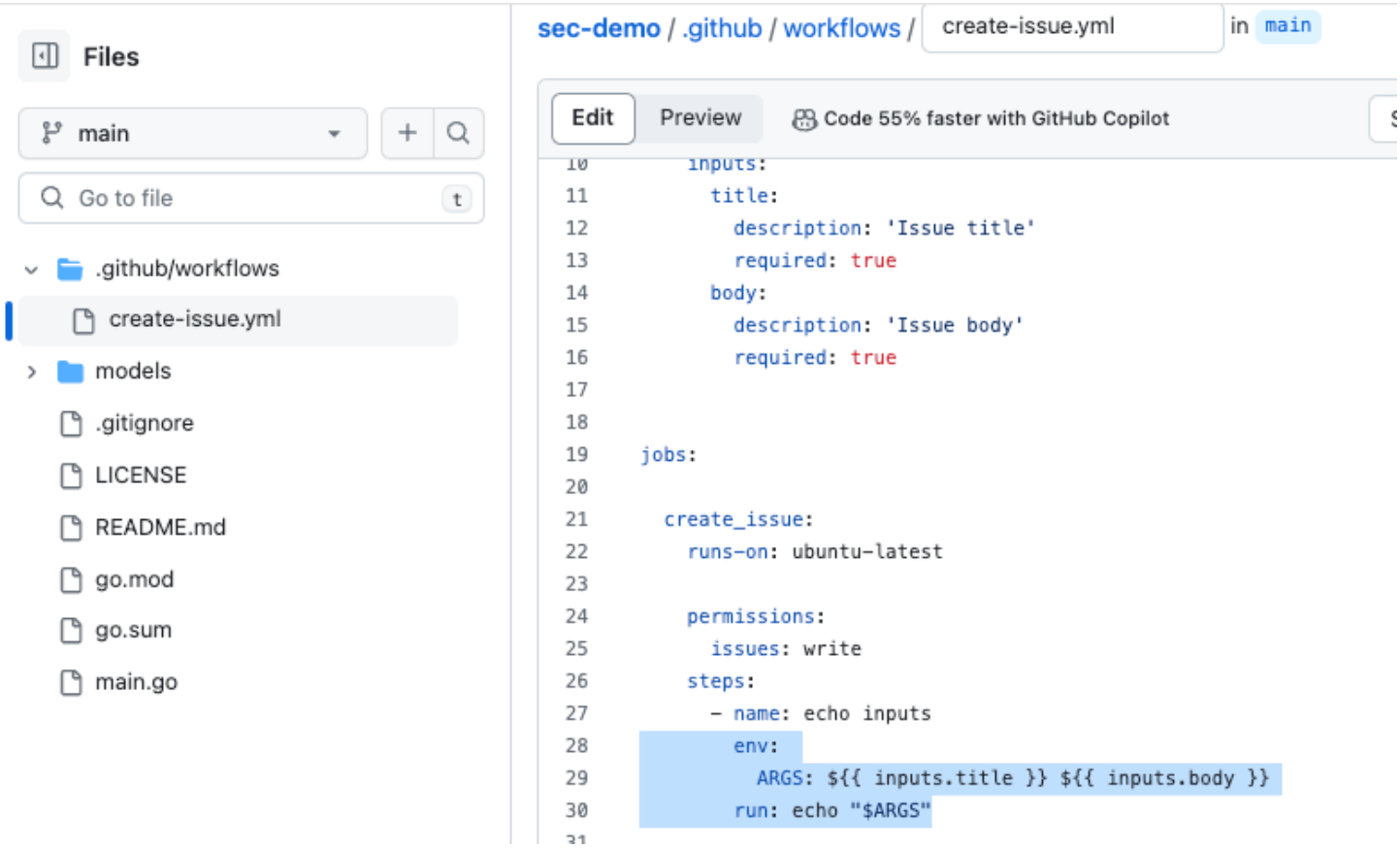
7. Now let's go back to the ***Code*** tab and fix the workflow. Click on the *.github/workflows/create-issue.yml* file to open it and click on the pencil icon to edit it.



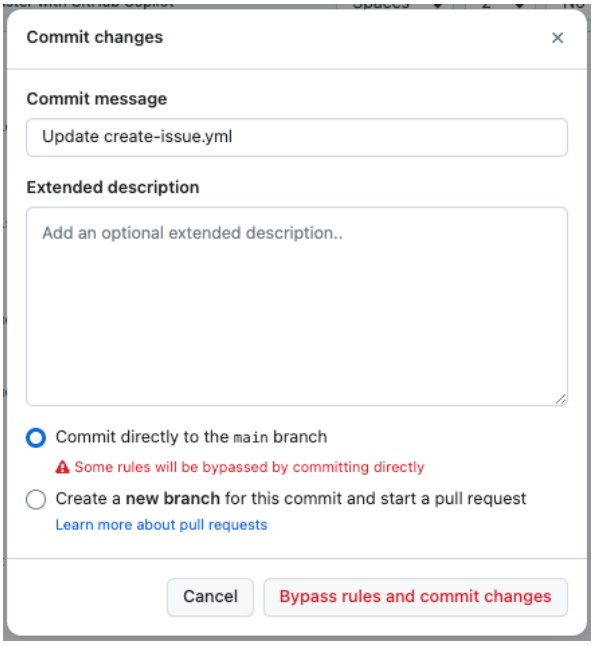
8. Replace the code at line 28 "*run: echo ${{ inputs.title }} ${{ inputs.body }}*" with this code:

```
env:
  ARGS: ${{ inputs.title }} ${{ inputs.body }}
run: echo "$ARGS"
```

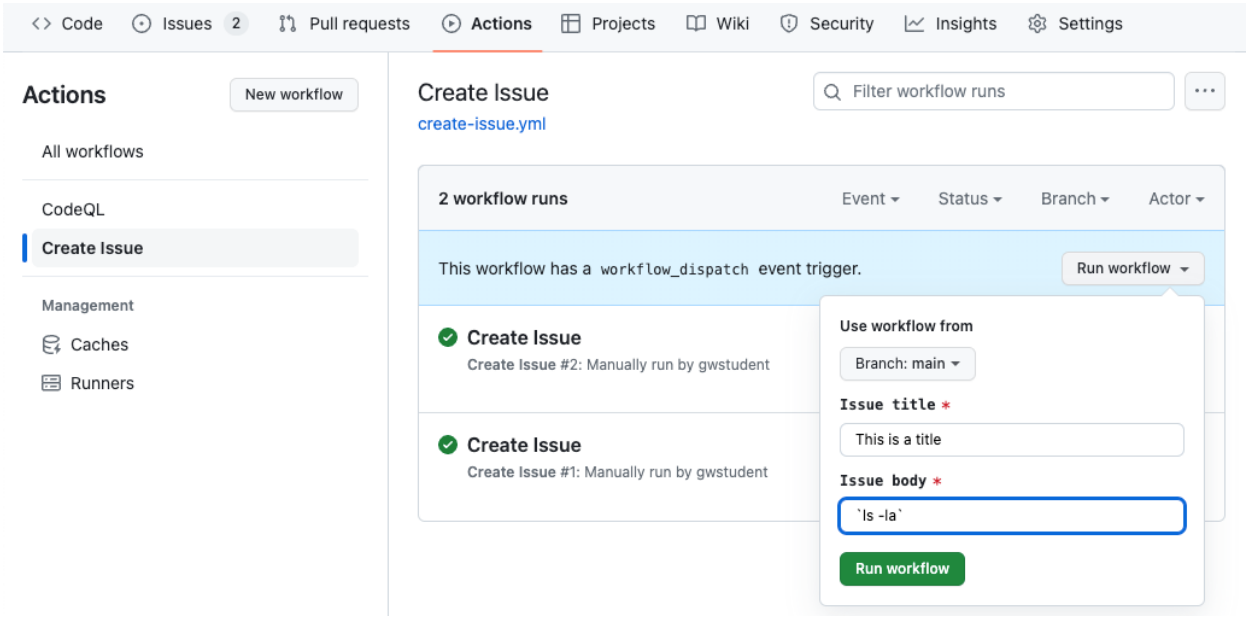
Be sure that it lines up as shown in the next screenshot.



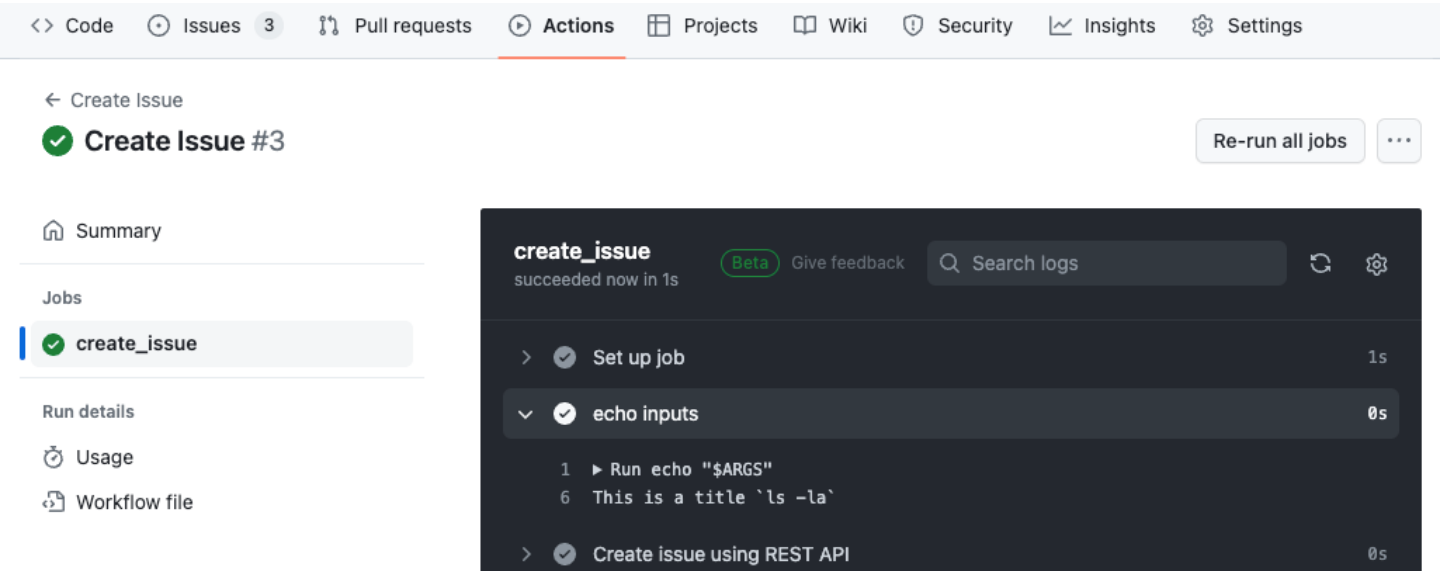
9. After making the change, click on the ***Commit changes…*** button and commit the change directly to the main branch. Notice the warning message because of the bypass we put in place. Click on the ***Bypass rules and commit changes*** button when ready.



10. At this point, another workflow run will have kicked off, but without any parameters. We can prove out if the changes fixed the potential script injection. Click back on the ***Actions*** tab, select the ***Create Issue*** workflow from the left, and then fill in the *Run workflow* dialog as before. This time the command should not be executed but simply echoed out.



11. After the workflow run is completed, you can drill into the logs and verify the arguments were echoed correctly and not executed.

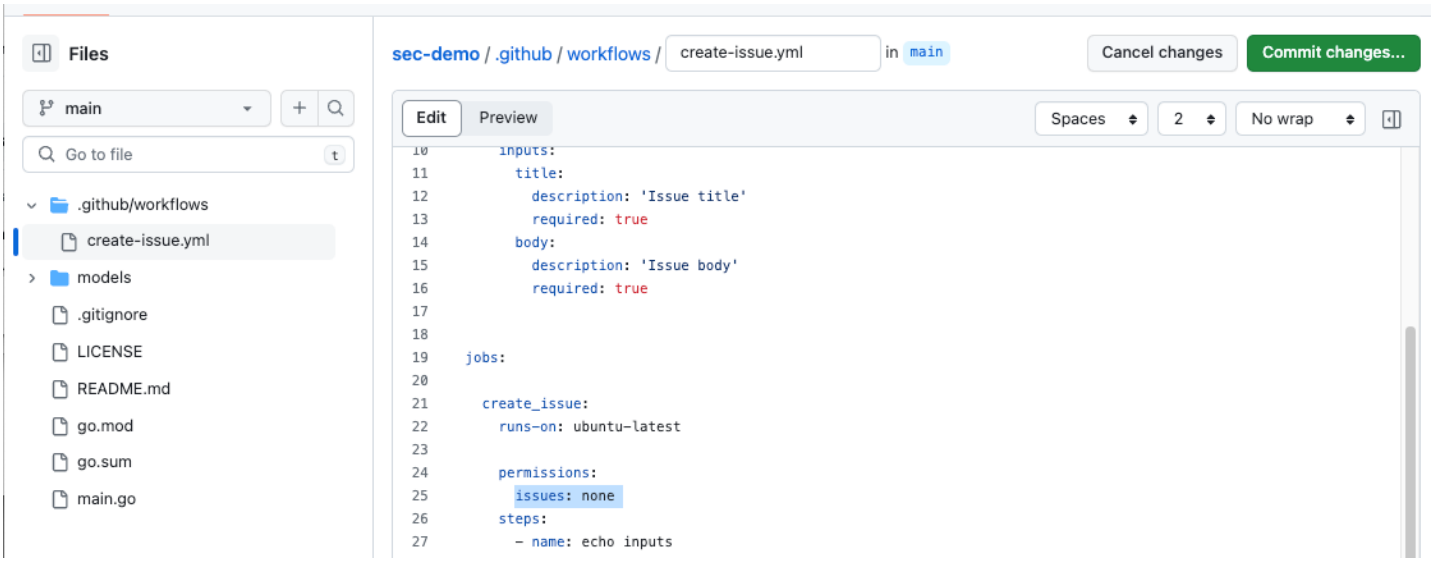


END OF LAB

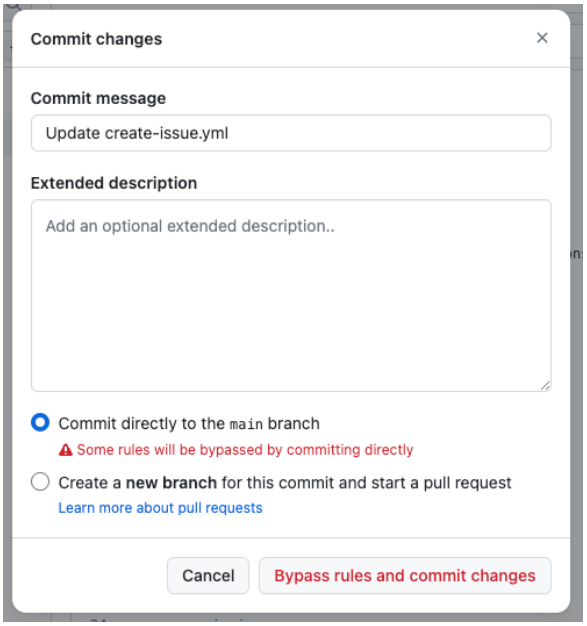
Lab 7 – Using a personal access token with a secret in a workflow

Purpose: *In this lab, we'll see how to encapsulate a personal access token in a secret and use that in a workflow.*

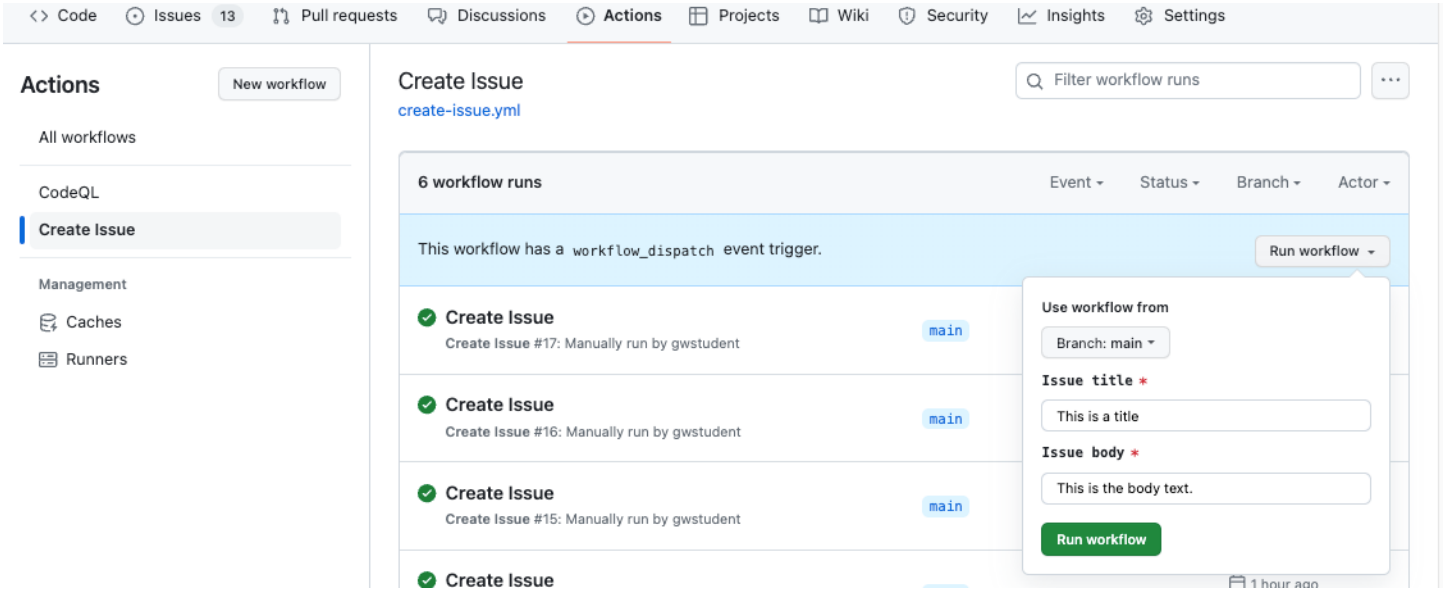
1. The workflow to create a GitHub issue that we used in the last lab authenticates using the "built-in" GITHUB_TOKEN. It must have permissions to execute the necessary code. Let's see what happens if we remove that permission. Edit the file *.github/workflows/create-issue.yml* and change the line under *permissions:* from *issues: write* to *issues: none*. **Commit your changes** to the **main** branch when done.



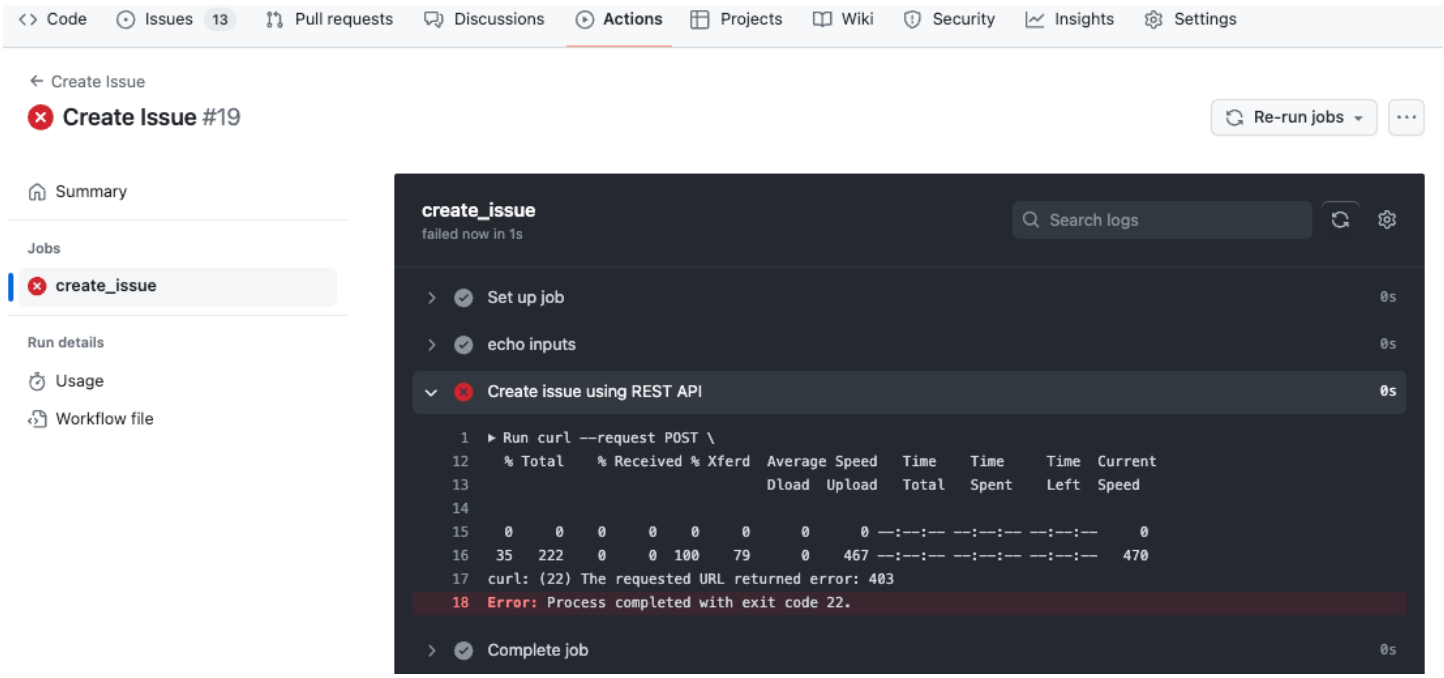
You will need to bypass the rules to commit the changes again.



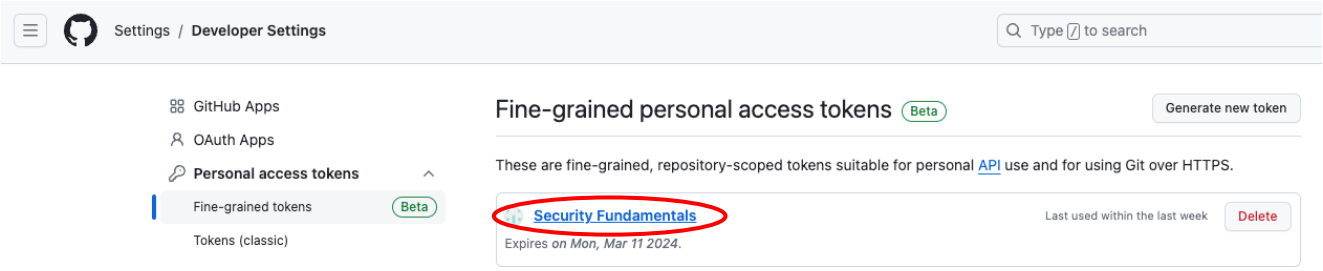
2. Go to the ***Actions*** tab, select the ***Create Issue*** workflow and run it as before.



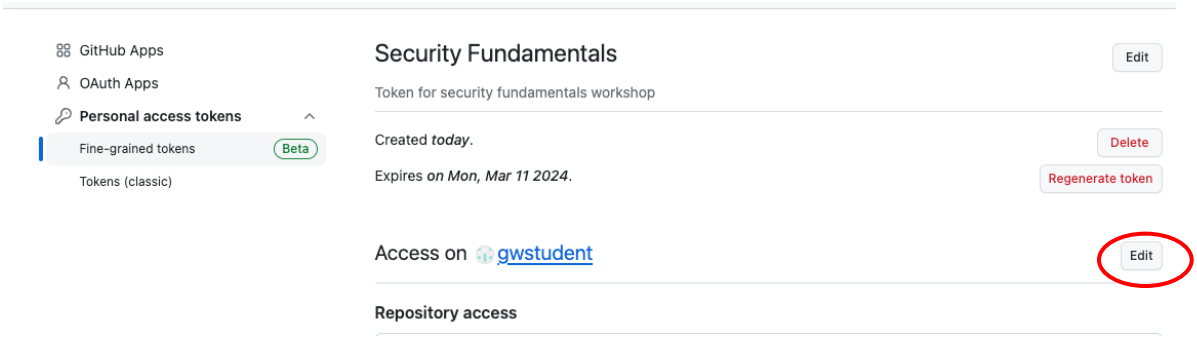
3. This run will fail because the token does not have permissions. You can see this by drilling into the logs for the job. Click on the commit message in the most recent run, select the *create_issue* job and expand the step to "Create issue using REST API". You'll see a 403 error.



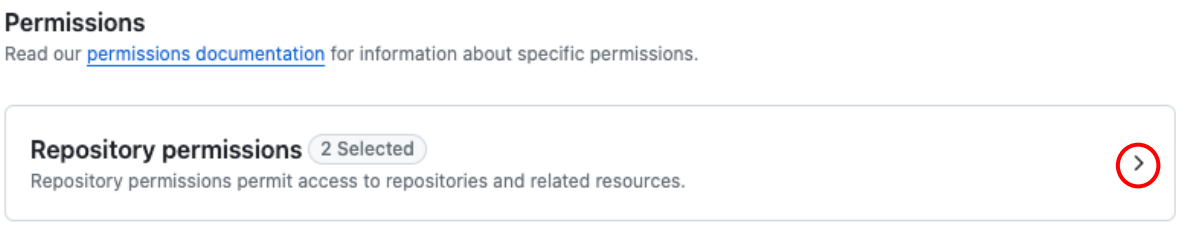
4. Let's update our personal access token (PAT) to use instead in the workflow. As you did in the first lab, go to your **Developer Settings** (or github.com/settings/apps). Then, under ***Personal access tokens***, select ***Fine-grained tokens***. Your token that you created in lab 1 should show up. Click on the name you gave it.



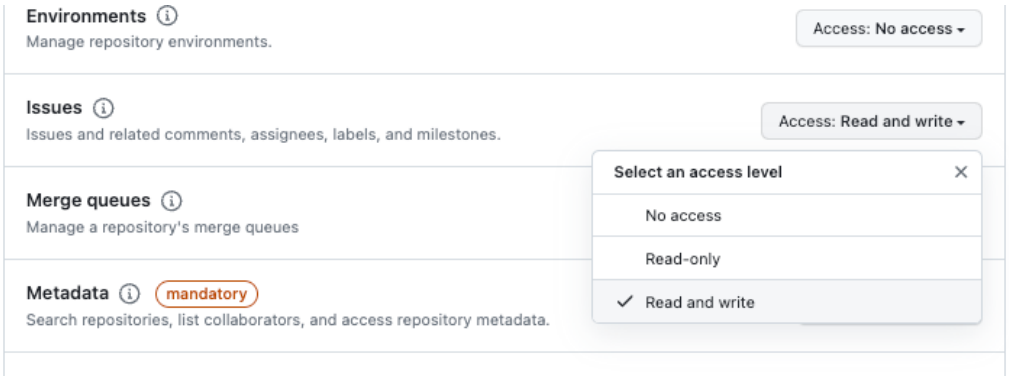
5. On the next screen, click on the ***Edit*** button.



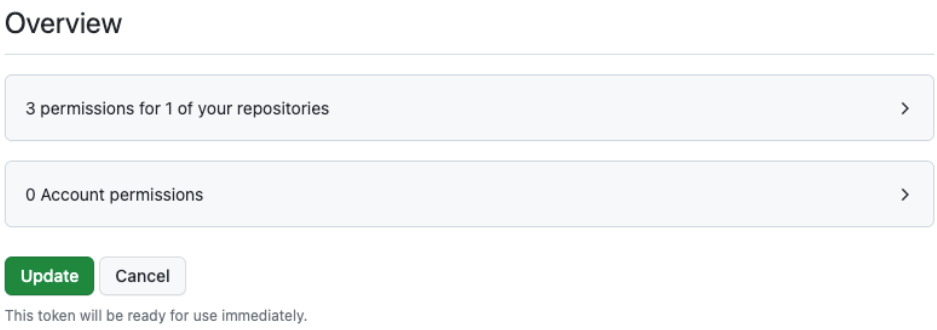
6. On the next screen, find the *Permissions* section, then click to expand the *Repository permissions* section.



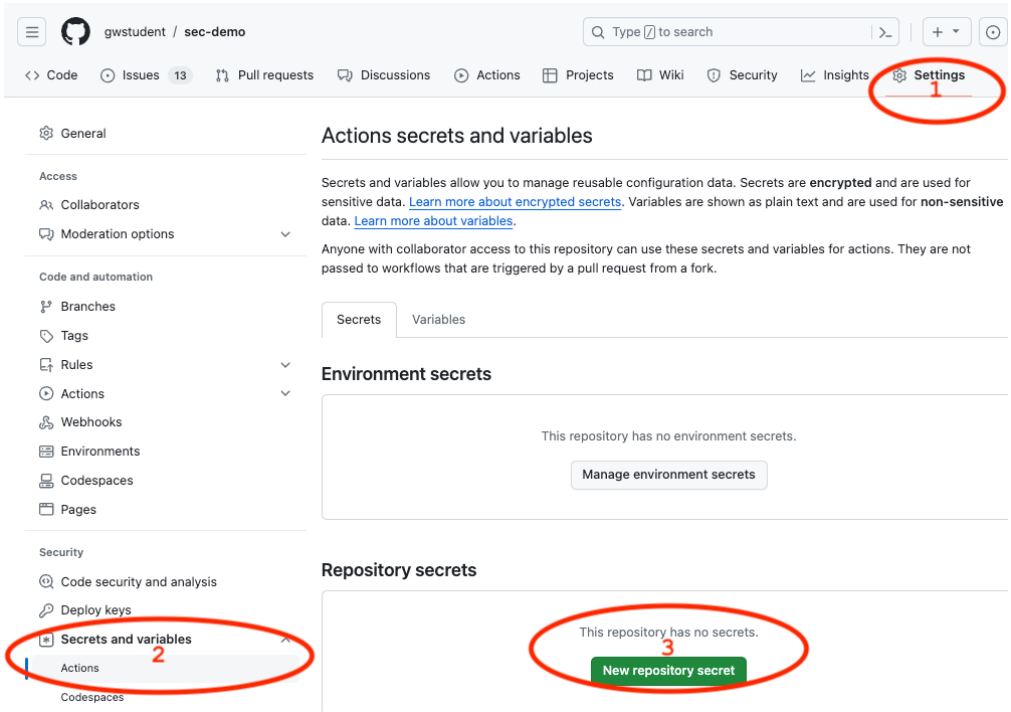
7. In the ***Repository Permissions*** section, find the ***Issues*** row and change the permissions to ***Read and write***.



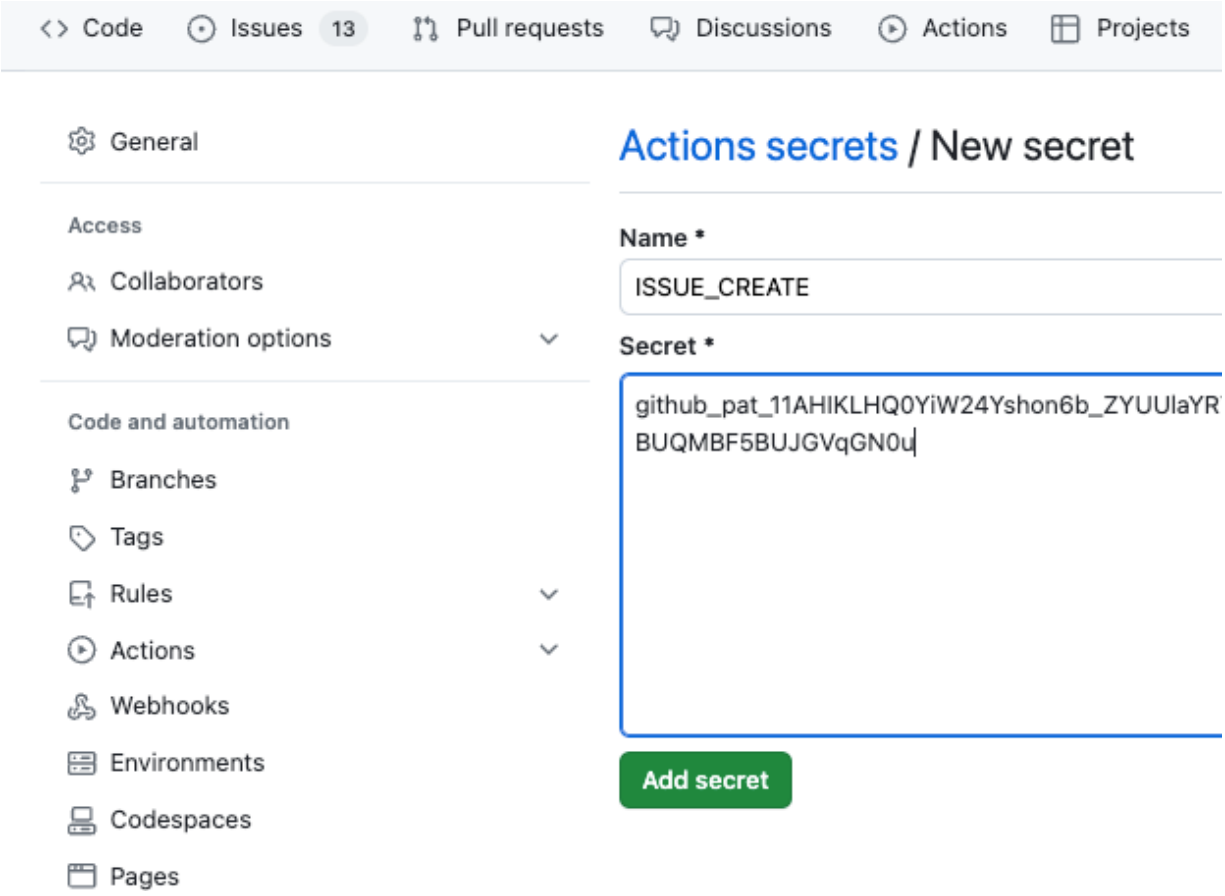
7. Scroll to the bottom of the page and click on the ***Update*** button to save your changes.



8. To use our token with the workflow, we need to store it in a secret for the action. Go back to the repository's ***Settings***, and on the left side, in the ***Secrets and variables*** section, select ***Actions***. Then click on ***New repository secret***.



9. On the screen for the new secret, enter a name like ISSUE_CREATE in the ***Name*** field. In the ***Secret*** field, paste the personal access token you previously generated. Then click on the ***Add secret*** button.

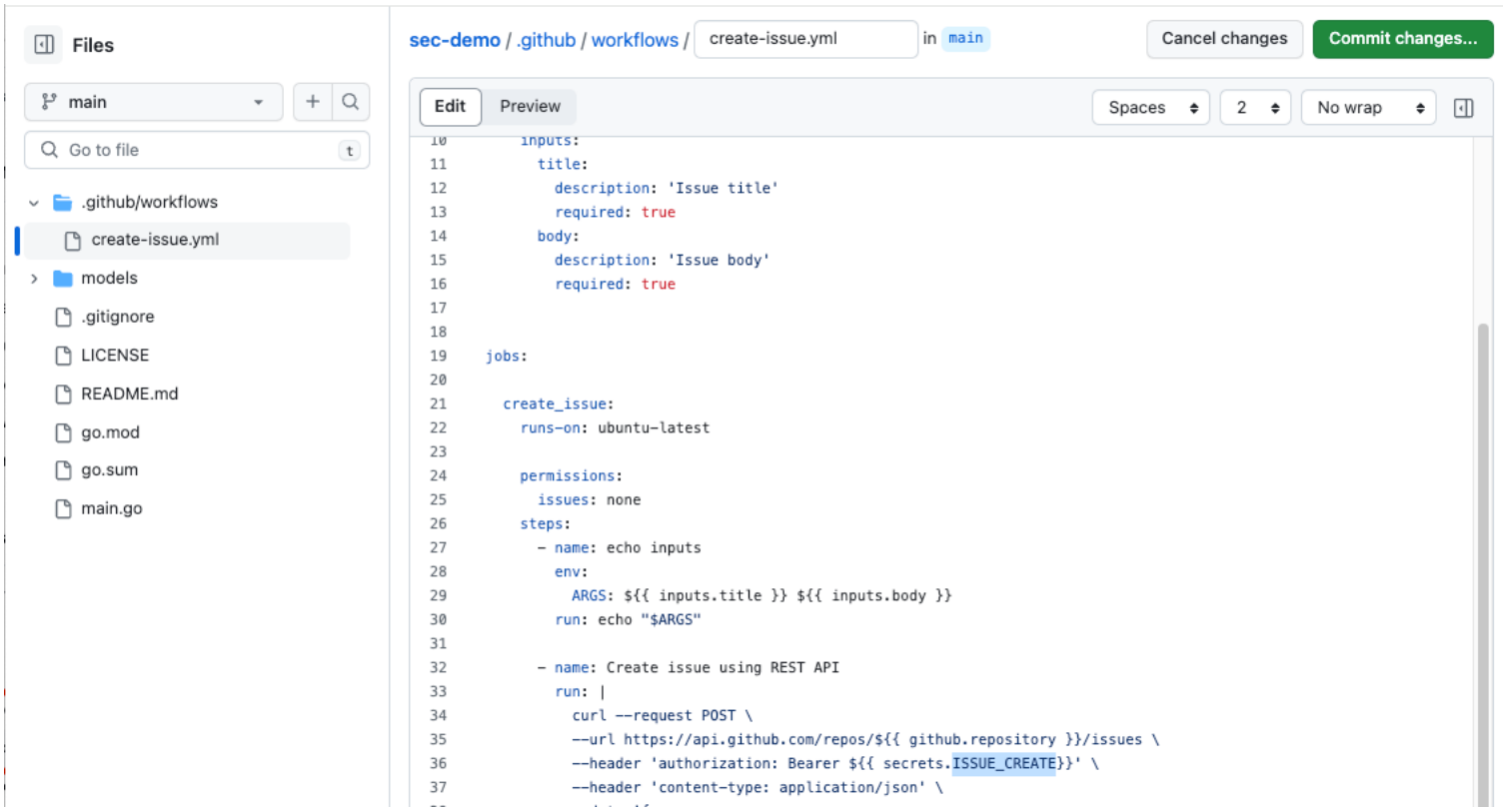


10. Now, we need to change the workflow to use the new token we create via the secret instead of the secret with the GitHub token. Edit the file *.github/workflows/create-issue.yml* and change the line (probably around line 35 or 36)

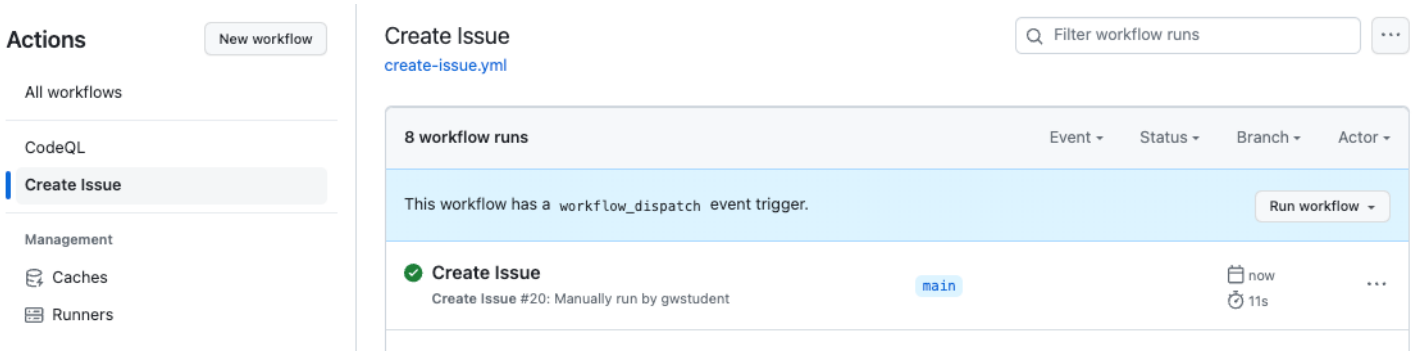
    --header 'authorization: Bearer ${{ secrets.**GITHUB_TOKEN**}}' \

    to

    --header 'authorization: Bearer ${{ secrets.**ISSUE_CREATE**}}' \



11. Commit the changes back to the main branch (bypassing the rules as before). Then run the workflow as before, from the ***Actions*** menu (*Create Issue* under *All workflows*). You should see that even though we took away the permissions, those were only for the GitHub token. The PAT allows the workflow to run as expected.



END OF LAB

That's all - THANKS