Travis Hill Staged Taint Analysis for Docker Actions 12/3/2024

1. Introduction

This report presents a novel framework for performing taint analysis on GitHub workflows, specifically focusing on identifying taints within flows that utilize Docker actions. This is a standalone tool written in Python and only provides support for Docker actions. Both composite and JavaScript actions have existing tools, such as ARGUS, that demonstrate the success of taking a staged taint analysis approach. As a result, this tool was heavily inspired by the ARGUS taint analysis engine methodology, adopting the staged analysis approach and utilizing taint summaries. The goal of this project was to see if adopting the approach in ARGUS was feasible for docker actions and not just composite or JavaScript ones.

2. Background

2.1) The ARGUS Methodology

Three core concepts of the ARGUS methodology were adopted to be used in this tool: the workflow intermediate representation (WIR), the taint summary creator, and the static taint analysis.

The workflow intermediate representation is a data structure used for taint analysis, effectively storing the execution flow of the entire workflow. It's structured similarly to a nested dictionary, allowing easy content access. When the WIR is generated, it scans each job and captures its execution types, dependencies, the data used, and where it goes. Essentially any interactions between a job and their respective steps are stored. By storing this information in a simplistic and easy-to-access manner, the WIR simplifies the static analysis process and speeds up processing time. This step is executed first before any analysis is done. Refer to Figure 1 to see a WIR representation of a GitHub workflow.

During taint summary creation, ARGUS scans the repository for any use of GitHub actions in the main workflows. Once an action is identified, they use CodeQL to analyze the actions and identify any taint sinks and taint sources, as well as keep track of the data flows. Once analysis is completed for the action, they store all the collected data in a database in WIR format for future use. This process only needs to be performed once, as once the summary is stored in the database it can be re-used and referenced in any analysis for any workflow if the same action is identified. The database is not specific to a singular workflow, the more actions that are analyzed, the more comprehensive the

database will become. This allows for efficient reuse of summaries for taint analysis, as repeated analysis only needs to be performed if an update to the action is made.

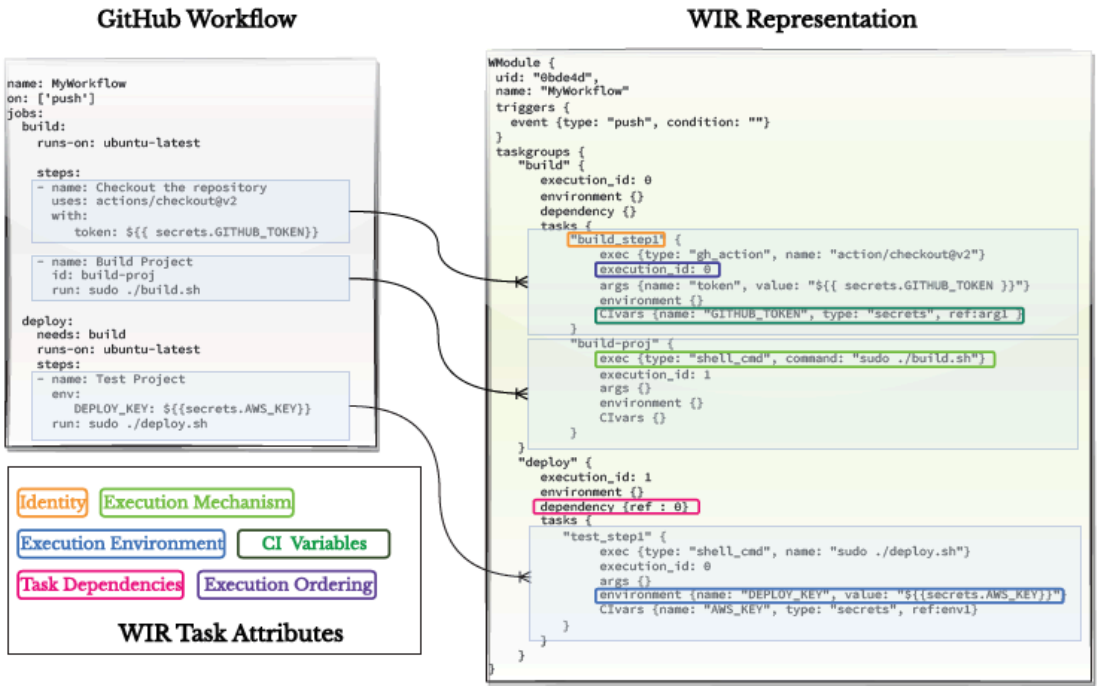

Figure 1) Shows the format of a WIR. Taken from ARGUS

The static taint analysis is fairly straightforward; it is a program that examines code for any taint sources and sinks and marks them accordingly, keeping track of data flows. The main novelty with ARGUS' approach is that it utilizes taint summaries from the taint summary creator. Whenever an action is used in a workflow, it refers to the taint summary database and uses the WIR to easily identify taint sinks/sources in the action, rather than re-analyzing the entire action once again. This drastically reduces processing time and wastes less computation resources.

The tool in this paper adopts all three of these ideas for improved performance and reliability.

2.2) What is considered tainted?

In a broad sense, tainted data refers to any data that originates from an untrusted or user-controlled source. In the context of GitHub Workflows, this tainted data commonly originates from data that is generated by a user triggering a GitHub event. ARGUS has defined what meets the criteria for taint sources, propagation, and sinks; this tool adopts the same criteria for its taint-tracking identification.

Taint sources are defined as data that originated from a GitHub event and is actively used within a GitHub workflow. For example, a user could post a comment on GitHub. Since the comment is user-defined information, it should be treated as if it was tainted. User-defined information can also expand to branch names, issue titles, commit messages, pull request contents, code review comments, etc. More examples can be found below.

github.event.issue.title	github.event.head_commit.message
github.event.issue.body	github.event.head_commit.author.email
github.event.discussion.title	github.event.head_commit.author.name
github.event.discussion.body	github.event.head_commit.committer.email
github.event.comment.body	github.event.workflow_run.head_branch
github.event.review.body	github.event.workflow_run.head_commit.message
github.event.pages.*.page_name	github.event.workflow_run.head_commit.author.email
github.event.commits.*.message	github.event.workflow_run.head_commit.author.name
github.event.commits.*.author.email	github.event.pull_request.title
github.event.commits.*.author.name	github.event.pull_request.body
	github.event.pull_request.head.label
	github.event.pull_request.head.repo.default_branch

Figure 2) Outlining taint sources in GitHub workflows. Taken from ARGUS

Taints propagation occurs as the data goes through the workflows via environment variable definitions or being passed directly into actions as input. Any variables that interact with a taint source or other tainted variables shall be considered tainted. Taint sinks are where this data is used in execution. Since ARGUS did not give an explicit definition of what a taint sink is for Docker actions, since it lacked support, a taint sink in this case refers to how the tainted variables are used within the Dockerfile.

2.3) Overview of GitHub Workflows and Docker Actions

Before going into the methodology of how this tool works, one should understand the data it is supposed to interact with.

As mentioned in the introduction, the tool is meant for GitHub workflows. A workflow in this context is an automated process that executes jobs, also referred to as tasks, outlined by a YAML file located in a special directory within a GitHub repository. A repository can have multiple workflows, each containing different jobs. A GitHub event occurring is what causes a workflow to execute. An event in this case means an activity originating from GitHub. Pull requests, committing to a repository, merging branches, etc. all fall under this category.

A job is a collection of steps in a workflow, and there can be multiple jobs per workflow. Each job is executed in order and may be dependent on other jobs executing successfully. The steps in a job define the metadata needed to execute a specific job. At a bare minimum, this will include the type of execution that is being performed and the variables that should be used, if any. The type of execution is typically either a command shell or a GitHub Action.

GitHub actions are essentially reusable workflows that execute their own jobs and steps using the environment information from the main workflow. Some workflows may perform repeated tasks, for example creating a pull-request, which entail the same jobs and steps needed for execution. Rather than rewriting this code in every workflow where the task is used, the GitHub action is referred to instead and the required information is passed in as parameters. This reduces repetitive code, making the workflow easier to read and manage. Docker actions are simply a subset of GitHub actions that interact directly with Docker containers. These container images may be pulled directly from Docker Hub or from a local Dockerfile.

However, due to the arbitrary nature of docker actions, this makes taint analysis much harder to perform. Due to the wide range of binaries used in docker containers, there is an extremely high level of abstraction when it comes to data flows. For example, a docker container could start up an image of Ubuntu while copying a Python script from the base machine into the image environment and execute the script using the variables passed in, which can then cause an entirely different task to start executing in the image. Attempting to perform static analysis on such a dynamic process is extremely difficult and will most likely not cover the entire scope of the taint propagation.

This project acknowledges this difficulty and does not perform exhaustive taint analysis on Docker actions, as that would be an extremely time-consuming process and heavily recursive. Instead, the tool performs taint analysis on the GitHub workflow, tracking any taints from the main workflow and any propagation into the Docker actions, and even looks into potential sinks in the Dockerfile itself. This targeted approach combined with the design methodology inspired by ARGUS provides a practical and resource-conscious approach to Docker action analysis.

3. Methodology

The workflow of the staged docker taint-analysis tool proposed in this work is built upon three main files: generate_action_summary.py, wir_generator.py, taint_analysis.py, and main.py. Each is executed in their order of mention, or “stage”, and the results returned

from each file, execution are used for the taint analysis and the final summary display. This process heavily reflects the ARGUS methodology. See Figure 3.

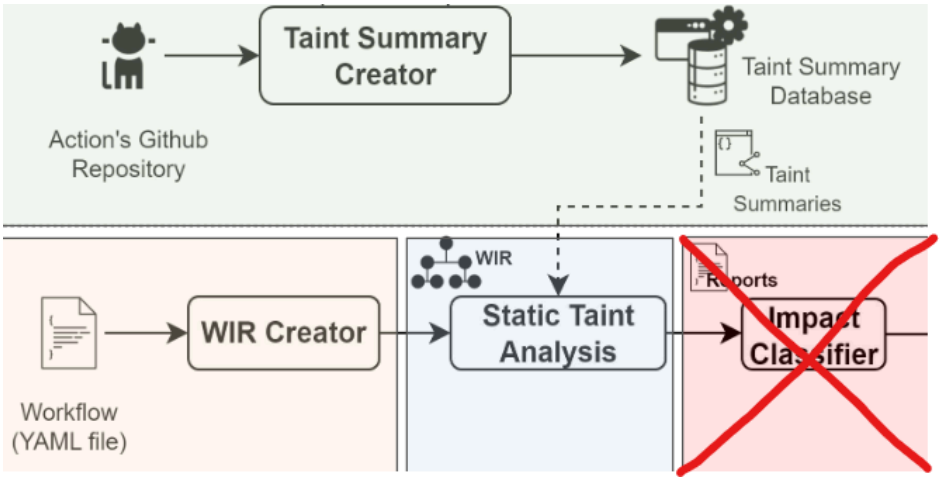

Figure 3) The ARGUS methodology modified to reflect what this tool is doing.

No impact classifier is added to this tool because one already exists with ARGUS that is fairly universal with the WIR format. Remaking it for one specific action rather than all actions is not worth the time investment. The other parts of the methodology are adapted due to ARGUS proving that they improved accuracy while reducing computational-resources, proving an effective way to approach the taint analysis balance problem.

3.1 generate_action_summary.py

This is the first step in the taint analysis process. It scans any action files located in the actions folder of the tool repository and loads them as a YAML object in Python. It then parses the action metadata for its inputs and outputs, marking them as potential taint sources and sinks. This is done through generic looping through the keys of the YAML file and storing any information in the WIR format as a summary. Information about the local docker container in the runs section will be collected if it exists, such as its passed-in arguments and image name, and added to this summary.

If a Dockerfile does exist, then the Dockerfile is loaded from the ./docker folder in the tool repository using the collected name to accurately search for the proper file. It is loaded as a regular file object in Python. Python regular expressions are then used to find the taint sinks and sources in the docker file. Currently, sources are identified by environment variable declarations while sinks are identified by the CMD declaration. These sinks/sources are then added to the existing summary WIR object that contains the action summary.

The final action summary is then added to the dictionary field in the taint_summary class, using the action name as the key, to be easily accessed during the taint analysis process.

Refer to:
https://github.com/twh9811/759Project/blob/main/src/generate_action_summary.py

3.2 wir_generator.py

This is the second step in the analysis process. This scans workflows located in the ./workflows directory of the tool repository and generates a workflow intermediate representation of the workflow. The WIR class is the core data structure that holds the parsed information of the workflow, containing the workflow name and a dictionary of jobs, where the key is the name of each job and the contents are a WIR representation of the steps of the job.

Parsing the workflow is similar to generate_action_summary.py, loading the workflow into Python as a YAML object. It iterates over each key in the file, which reveals the metadata of the job. Any environment declarations, dependencies, and each step of the job are captured. The details of the steps include the name of the step, the execution mechanism, and the arguments used with the execution mechanism. These details are organized into dictionaries that follow the WIR representation.
Refer to: https://github.com/twh9811/759Project/blob/main/src/wir_generator.py

3.3 taint_analysis.py

The Dcker_Action_Taint_Analysis class contains the logic for performing taint analysis on the WIR. When the class is initialized, it takes in a WIR of the workflow being analyzed and the taint summary object. The WIR contains all the details of the GitHub action workflow needed for analysis and the summaries object contains the predefined taint summaries for Docker actions that may be used in the workflows. It also contains a set of tainted variables and a flow-tracking dictionary to keep track of taints, and taint propagation.

The perform_analysis method performs the actual taint analysis. It iterates through the jobs and their tasks of the WIR object, looking at the organized meta-data. The initial taint sources are deemed any variable in the command-ingestion variable (CIvar) section of the WIR that references github. If a CIvar meets these criteria, it is added to the taint set and as a key to the taint propagation dictionary. The value in this dictionary

is then set to an empty list. Any values in the list indicate that the taint source has interacted with the variable in the order they are added in.

For each task in the job, special attention is paid to the execution type. If the task uses a Docker action, identifiable by the WIR declaration of the execution mechanism, then it extracts the precomputed taint summary from the summary object. Recall the information within the summary contains all the taint information of both the action workflow file as well as the docker file. This information is used to propagate taint from the sources or add new sources to the set. The flow-tracking dictionary is updated accordingly based on the WIR being able to capture the flow of variables.

Refer to: https://github.com/twh9811/759Project/blob/main/src/taint_analysis.py

3.4 main.py

The main file simply ties everything together. It generates the taint summary object and WIR object by calling the generate_action_summary.py and wir_generator.py files respectively. It then calls taint_analysis.py to perform the analysis using the summaries on the passed-in WIR object. Once the analysis is done, the tainted variables and taint flow are returned and parsed to be displayed legibly in a summary format. It displays the analysis summary in the terminal where the main file is executed as well as writes the summary to a file located in the ./results folder in the tool repository.

Refer to: https://github.com/twh9811/759Project/blob/main/src/main.py

4. Novelty Illustration

The novelty in this approach is that it can successfully identify taints in docker actions while utilizing the newly proposed staged taint analysis techniques proposed in ARGUS. It takes state-of-the-art techniques, such as the workflow intermediate representation and taint summaries, and expands upon their known capabilities; demonstrating even further depth and potential for static taint analysis. It can successfully track taint throughout GitHub workflows, their actions, and even into the Dockerfile itself, which the original authors of ARGUS did not introduce support for at the time of writing. Although the tool itself is not fully fleshed out and missing functionality in some regards, it proves that despite its difficulty, docker action taint analysis is possible. With more time and effort, a plugin for the main ARGUS taint analysis engine can be created to form a fully fledged taint analysis tool that supports all types of GitHub actions.

To demonstrate its novelty in practice, look at the following sample workflow below. All the taints, propagations, and sinks are clearly marked with comments. The main workflow gets input from the title of a GitHub pull request and is triggered when a pull-request is made. The title is stored in two variables, which are then passed into a GitHub action for further processing.

```yaml
name: "Taint Analysis Simple Workflow"

on:
  pull_request:

jobs:
  taint-test:
      runs-on: ubuntu-latest
      steps:
      - name: Extract and Process Data
      id: extract_process
      # Taint Sink
      uses: /extract-and-process-action@v1
      with:
            # Taint Source
            input_text: ${{ github.event.pull_request.title }}
            user_input_var: ${{ github.event.pull_request.title }}
```

File 1: The main workflow file

The GitHub action uses the input from the main workflow to start builds a docker image and then runs it using the input as parameters.

```yaml
name: "Extract and Process Data"
# IMPORTANT NOTE, VARIABLE NAMES MUST MATCH BETWEEN WORKFLOW AND ACTION
DEFINITIONS
inputs:
  # Taint Propagation from workflow
  input_text:
      description: "Input text to extract and process"
      required: true
  # Taint Propagation from workflow
  user_input_var:
      description: "Input text to extract and process"
      required: true
```

```
runs:
  using: "docker"
  # Taint Sink
  image: "Dockerfile"
  args:
      # Further Taint Propagation
      - ${{ inputs.input_text }}
      # Further Taint Propagation
      - ${{ inputs.user_input_var }}
```

File 2: The Docker Action

The Dockerfile itself takes the given input, stores it as environment variables further propagating the taint and then uses these tainted variables in an echo command which will execute when the container is run.

```
FROM alpine:latest

# Renaming to demonstrate flow tracking
ENV RENAMED_INPUT_TEXT="${input_text}"
ENV RENAMED_USER_INPUT_VAR="${user_input_var}"

# Taint Sink. Tainted vars used in execution
CMD echo "${RENAMED_INPUT_TEXT}" && echo "${RENAMED_USER_INPUT_VAR}"
```

File 3) The Dockerfile

When this workflow is analyzed by the tool, it gives the summary seen below. It successfully identifies all the tainted variables found within the workflow. It also successfully tracks the taint-flow between the two variables, properly identifying which tainted variable tainted which “normal” variable and even distinguished which echo command the tainted variable was used in. This proves that on a Dockerfile level, taint can be identified in Docker actions and successfully tracked by taking a staged static analysis approach utilizing taint summaries, just as ARGUS had.

```
Analyzing Workflow: Taint Analysis Simple Workflow
 Analyzing Job: taint-test
  Analyzing Task: Extract and Process Data
   Checking Taint Sources Based on Task Command Ingestion Args (GitHub
Events)
      Variable 'input_text' has been tainted directly from user input
```

```
(GitHub event)
      Variable 'user_input_var' has been tainted directly from user input
(GitHub event)
      Uses Docker Action: Docker Action: 'extract-and-process-action'
      Taint Summary Found: {'inputs': ['input_text', 'user_input_var'],
'docker_details': {'container_image': 'Dockerfile', 'sources':
{'input_text': 'RENAMED_INPUT_TEXT', 'user_input_var':
'RENAMED_USER_INPUT_VAR'}, 'sinks': ['echo "${RENAMED_INPUT_TEXT}"', 'echo
"${RENAMED_USER_INPUT_VAR}"']}, 'sinks': ['inputs.input_text',
'inputs.user_input_var']}
      Tainted Variable 'inputs.input_text' has been tainted by a tainted
source propagating to the Docker Action
      Tainted Variable 'inputs.user_input_var' has been tainted by a
tainted source propagating to the Docker Action
      Tainted Variable 'RENAMED_INPUT_TEXT' has been tainted by a tainted
source propagating to the Docker File
      Tainted Variable 'RENAMED_USER_INPUT_VAR' has been tainted by a
tainted source propagating to the Docker File

All Tainted Variables:
inputs.user_input_var
RENAMED_USER_INPUT_VAR
inputs.input_text
RENAMED_INPUT_TEXT
user_input_var
input_text

Tainted Flows:
Flow 1:
 Origin: input_text
      Propagation: inputs.input_text
      Propagation: RENAMED_INPUT_TEXT
      Propagation: echo "${RENAMED_INPUT_TEXT}"

Flow 2:
 Origin: user_input_var
      Propagation: inputs.user_input_var
      Propagation: RENAMED_USER_INPUT_VAR
      Propagation: echo "${RENAMED_USER_INPUT_VAR}"
```

File 4) The log summary generated by the tool

5. Comparative Analysis

Due to the inspiration the tool took from it, ARGUS is the baseline used when comparatively analyzing this tool. Since when ARGUS was evaluated it did not have Docker action support, it could not identify taint within any of the Docker actions in its dataset. This automatically means this tool is better comparatively when it comes to analyzing Docker actions. Utilizing the taint summaries and workflow intermediate representations allows for faster accesses and less computation time, meaning it is less resource intensive and faster than traditional static analysis engines and mostly comparable to engines like ARGUS.

However, in every other aspect, this tool does not compare to ARGUS. It was not thoroughly evaluated over millions of public repositories and thousands of public workflows. There was not a scalable, time-efficient way to approach this as they had in the paper, given the timeframe of required completion. As a result, the only tests done for this tool were done with a handful of personally made workflows. It is severely lacking in thorough testing and although it can analyze Docker actions, it is by no means more efficient overall than ARGUS. The accuracy has a high likelihood of being low compared to the high accuracy scores received by ARGUS when put through the same tests.

Another downside of this tool is that it is written in Python. ARGUS originally uses CodeQL as its main analysis language, which has inherent features meant to aid in static analysis. It does not rely on regular expressions like the Python approach, and instead uses a querying process. Overall, CodeQL is faster and less-resource intensive than Python, and has tools that allow for more comprehensive-flow tracking across files. However, due to not having the time to learn CodeQL this tool was written in Python. As a result, the ARGUS implementation is inherently more efficient in almost every regard and a better approach to the taint analysis, especially with the balance problem.

The final main downside is that this tool can only be used to analyze Docker actions. ARGUS has the functionality to analyze both composite and JavaScript actions. These two types of actions are what a majority of the GitHub workflows use. As ARGUS stated, 70% of their 2.8 million repositories contained these two types, while only 30% contained docker actions. The use case for a stand-alone tool like this is very specific and does not have the same broad application as ARGUS.

6. Conclusion

This report demonstrates the feasibility of performing taint analysis on Docker actions using a staged taint analysis approach. The standalone tool developed serves as a proof-of-concept, successfully detecting taint propagation through workflows and even into Dockerfiles. However, its accuracy and effectiveness are limited when compared to the baseline of ARGUS. If future work was to be done, the main focus would be re-designing the tool in CodeQL to utilize it's inherent taint-analysis capabilities, enhancing accuracy and processing times. This redesign would also enable integration with the main ARGUS taint analysis engine, creating a fully comprehensive tool to analyze GitHub workflows.

Works Cited

Muralee, Siddharth, et al. "{ARGUS}: A Framework for Staged Static Taint Analysis of {GitHub} Workflows and Actions." *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, https://www.usenix.org/conference/usenixsecurity23/presentation/muralee

“Creating a Docker Container Action - GitHub Docs.” GitHub Docs, 2024, docs.github.com/en/actions/sharing-automations/creating-actions/creating-a-docker-container-action

“Understanding GitHub Actions - GitHub Docs.” GitHub Docs, 2024, docs.github.com/en/actions/about-github-actions/understanding-github-actions.