The risk of CI/CD pipeline poisoning via CodeBuild

On the intricate challenges of setting up a secure CI/CD pipeline
In AWS, developers are typically granted the CodeBuild StartBuild action, either via customer managed policies or through the AWS managed policy for the developer role. However, this action can allow developers to bypass existing security controls and provides a, perhaps, unforeseen vector for exfiltrating application secrets, tampering with the application and, potentially, taking full control of the deployment servers by executing commands using elevated privileges. And due to the shared responsibility model, this is mostly an AWS customers’ challenge. In this article, we describe the problem and potential impact and give suggestions for improvement.

Asier Rivera Fernandez  
Cybersecurity researcher  
and senior associate at PwC Belgium  
asier.rivera.fernandez@pwc.com  
Connect with me on LinkedIn
Software is a core part for any modern enterprise. Recent events have, yet again, shown that the security of the Software Development Life Cycle (SDLC) environment is becoming an important risk factor for enterprises that develop software. Due to their complexity and diversity, development environments are difficult to secure properly, and as attackers are looking for the weakest link in the protection of a company, the SDLC environment might become the next big thing for them to attack interesting organisations (or their clients).

The security of SDLC environments is based on the existence of multiple security controls tailored to the needs of the specific environment. In modern development environments, a Continuous Integration (CI) and Continuous Delivery (CD) pipeline is a core part of the overall development practice. The security of a CI/CD pipeline is tightly coupled to the infrastructure and environment in which the pipeline is deployed.

In the case of AWS-based CI/CD pipelines, it is key to understand and fine-tune the services provided by AWS to help you meet your organisation's needs. Cloud services often cover many use cases with the purpose of fitting different customers and their specific needs. However, this can lead to functionalities that do not align with the security needs of all enterprises. These functionalities may lead to risks which are not simple to identify or mitigate. Identifying these risks requires an in-depth understanding of the technologies, configurations, interactions and functionalities, not only at component level but also as part of a CI/CD pipeline. Therefore, we have performed research and threat analysis to gain a deeper knowledge over cloud-based CI/CD pipelines implemented in AWS.

In this article, we describe a potential threat that can allow developers to exfiltrate secrets and influence the application developed within CI/CD-driven SDLC environments deployed via AWS provided services. The structure of the article is as follows. The Background section provides the information about the AWS services required for a good understanding of the threat we cover in this article. Next, in the Threat description section, we further elaborate on explaining the potential threat that can occur and be abused in CI/CD pipelines based on AWS services. In addition, we describe a Proof of Concept implementation that demonstrates a practical approach for the threat exploitation. In the Discussion section, we provide our insights on the response of AWS to our notification of this threat analysis and the responsibility of both parties, AWS and their customers. At the end, we summarize the content of the document and include recommendations that can help reduce or mitigate the threat.

1 https://www.crowdstrike.com/blog/sunspot-malware-technical-analysis/
In this article, we focus on the three main services used for Continuous Integration (CI), Continuous Delivery (CD) and CI/CD orchestration in AWS: CodeBuild, CodeDeploy and CodePipeline, respectively. The image below describes the task of each of these three services and shows the way in which they interact with each other from a high-level view. The subsections below provide the information we have gathered from our analysis of these three services, including the information required for the comprehension of this article and infrastructure diagrams that showcase the components. For a more in-depth explanation of AWS related topics, we refer the reader to the official AWS documentation.
**CodeBuild**

The CodeBuild service offers a simple solution to configure, provision, manage and scale-build servers and environments. The computing power is supported by container instances that focus on running jobs, such as code compilation, code testing, and code analysis. The key features of CodeBuild are the simplicity of integration in existing and new CI/CD pipelines and the flexibility provided through an AWS managed service. CodeBuild is a core service for AWS based pipelines, but it also requires the intervention of other AWS services, such as Elastic Container Service (ECS) and Elastic Container Registry (ECR) for the provisioning of the computational power via container instances, Simple Storage Service (S3) for the storage of source code packages and application artifacts, and Secrets Manager for the access to passwords and configuration parameters.

In order to perform its service, the CodeBuild container instance requires access to resources. This is achieved by providing the instance with access rights to the source storage services, which can be S3 buckets or version control repositories, such as CodeCommit and GitHub. The access rights can be provided via an IAM role for AWS services and via access credentials for non-AWS service providers. The IAM role, a so-called service role, is applied to the container instance and is configured at CodeBuild project level. The access credentials are also configured at project level; however, these are defined as part of the source service configuration. The access credentials can be configured in the form of an access token or OAuth credentials and its storage is managed by CodeBuild within AWS.

Every time a CodeBuild build execution is started, CodeBuild requests ECS to deploy a new container instance with a container image that is configured at project level. ECS will then set up and deploy a new container with the target container image, which can be provisioned by ECR service or another non-AWS container image registry service. Once the container is ready to be used, the target source package will be downloaded into the container instance, which includes the resources required to perform the job. These resources can be source files, compiled libraries, applications and others. In our overview diagram below, we named this process the **Provision task**.
Once the container has been made ready, the next step of the CodeBuild execution is to perform the actual job it was created for. In this article, we focus on the code compilation task. The container executes the given commands to perform the job. These commands and other related settings are provided to the container via the BuildSpec file, which can be defined as part of the project configuration, can be provided within the source package, or can be available in an S3 bucket.

The BuildSpec file is a key resource for CodeBuild, since it determines the environment variables and secrets to be made available to the container. To provide a secure secret provisioning functionality, CodeBuild supports the use of the Secrets Manager and System Parameters Store services provided by AWS. The commands included in the BuildSpec file can be distributed over different phases and provide the customers with a flexible functionality to implement a job flow that fits their needs. The BuildSpec file also specifies the locations in which the container will store the output resources, such as artifact resources and test reports. There exist other parameters and settings that can be set in the BuildSpec file, but they will not be covered in this article.

After the execution of the commands in the BuildSpec file, CodeBuild will package and upload the artifacts and reports generated by the container. These packages are uploaded to the storage location specified in the CodeBuild project configuration. The artifact package resulting from a build execution will include all resources required for the deployment of the new version of the application. These artifact packages are further analyzed in the following CodeDeploy subsection. The build execution will then be finalized and completed if no errors were encountered. We have named this process the Build task.

Diagram illustrating the steps taken by CodeBuild to perform a build execution.

The table below summarizes the tasks that CodeBuild performs in order to provide the code compilation, test, and analysis services. The table also includes information about the activity description, actors that intervene in the task, and outcome resources generated during the task execution.
The CodeDeploy service focuses on the management and orchestration of the deployment of new versions of applications. CodeDeploy provides a simple manner to identify, group and reach out to target services that are deployed in, or connected to, the AWS cloud environment. In order to perform the deployment, CodeDeploy requires access rights to list, describe and interact with the target services. These access rights are provided via an IAM role that is referred to as the service role. This is similar to the approach used in CodeBuild.

CodeDeploy supports three types of automated deployments: ECS deployments, Lambda deployments and server deployments, which can be performed in Elastic Compute Cloud (EC2) or On-premise instances. All three deployment types use an AppSpec file. However, the content of the AppSpec file is different for each deployment type. In this article, we focus on the server deployment type, which we further describe below.

The AppSpec file used for server deployments is stored in the artifact package together with other resources required for the installation of the new application version, such as executable, configuration, media and script files. This version of the AppSpec file contains three major sections: files, permissions and hooks. These sections determine the location to copy the resource files, the permissions to be set to the resource files, and the commands or script files to be executed, respectively. Therefore, the AppSpec file is a key component during the deployment, similar to the BuildSpec file for CodeBuild.

CodeDeploy identifies target server instances, provides the location of the new artifact package to the CodeDeploy Agent, and monitors the deployment process. However, CodeDeploy delegates the actions performed within the server instances to the CodeDeploy Agent. Therefore, CodeDeploy does not require any access rights to execute commands within the server instances. Once a CodeDeploy Agent receives the information for a new deployment, it downloads the target artifact package. The Agent then extracts the files from the artifact package and follows the AppSpec file definition. As a result, the extracted files are copied with the permissions specified in the AppSpec file, and the commands and script files indicated in the hooks section are executed.
CodePipeline focuses on the orchestration of a CI/CD pipeline, including version management, the interaction and communication with other services through the pipeline, and the monitoring of running and finalized pipeline executions. The CodePipeline project is composed of stages that contain actions. Actions represent the invocation of another service provider to perform a job, such as code compilation, code testing, and manual approval request. The combination of stages and actions provide customers a flexible and visual manner to create, manage and run CI/CD pipelines.

CodePipeline also manages the storage of the resources generated in the actions that compose the pipeline, such as artifact packages, test reports and others. This is achieved via an S3 bucket that is configured at project level. In addition, it is possible to specify the input resources that each action needs to receive. For example, a build execution requires the source package whereas a unit test execution may require the artifact package previously generated by the build execution. Nevertheless, the service executing the action must have access rights to the target input and output resource locations. For example, a build execution must have access to the S3 bucket indicated by CodePipeline for the storage of the input and output resource.

To perform this orchestration, CodePipeline also requires access rights to invoke, and interact with, the service providers used as part of the pipeline. These service providers can be AWS services, such as CodeCommit and CodeDeploy, or non-AWS services that are deployed in the cloud or are publicly available via the internet, such as BlazeMeter and GitHub Enterprise. The access rights for AWS services are granted via an IAM role that is assigned to the CodePipeline project, also called service role. The access rights for non-AWS services is supported by access credentials that are configured when creating a pipeline action of this type.

In this article, we further describe the service provided by CodePipeline by covering four generic pipeline stages. These stages represent a CI/CD pipeline configured to integrate and deploy a new application version that targets EC2 instances. Other stages and actions can be added in specific pipelines implemented for enterprises. However, we focus on the description of the general use case scenario for CodePipeline. We have named these four stages Source, Build, Others and Deploy, and we further describe them individually below.

**Source stage**

Every pipeline execution is identifiable via a unique ID and can be started manually or via an event originated from the source repository, such as a new code being merged to the master branch. CodePipeline supports repository servers, such as CodeCommit, GitHub enterprise and BitBucket, or other storage services, such as S3. During this stage, CodePipeline asks the source service (CodeCommit in this scenario) for the source code base from the target repository. This source code is added to the source package, which is then stored in the S3 bucket selected for the pipeline. The names of the packages generated through the pipeline are randomly generated by CodePipeline for each pipeline execution. These random names allow CodePipeline to manage the package versioning.
**Build stage**

Commonly, the next step in CI/CD pipelines is to compile the source code into application executables and configure other resources, such as media files and deployments scripts. To do so, CodePipeline delegates this task to another service, such as CodeBuild. As represented in the following diagram, CodePipeline can start a new build execution for the CodeBuild project specified in the pipeline action. CodePipeline also indicates to CodeBuild the location in which the input resource, the source package, is available and the location in which CodePipeline expects CodeBuild to store the output resource, the artifact package in this scenario.

Once the build execution is started, the process will follow a normal CodeBuild build execution as described earlier in the CodeBuild subsection. Once the build execution is completed, CodePipeline will set the action as completed, if no errors were encountered, and the pipeline execution will continue.

Diagram illustrating the actions followed by CodePipeline to perform a common Build stage.
Others stage

In this stage, we group all other activities that can be required in a CI/CD pipeline that do not specifically perform code integration and delivery covered in the previous and next stages. This stage can include activities, such as code unit testing and scans, SAST and DAST analysis, manual approvals and other actions. Nevertheless, from the point of view of CodePipeline, these actions will follow a generic workflow. This workflow is represented in the diagram below and includes three steps:

- Start the action: CodePipeline notifies the service provider that a new job is required and provides the location of the input and output resources to be used by this action.
- Wait for the action: CodePipeline waits for the service provider to perform its job and monitors the status of this and other actions that may be executed in parallel.
- Finalize the action: CodePipeline expects to receive a result notification from the service provider that allows CodePipeline to determine whether the job was successful or not.

Diagram illustrating the actions followed by CodePipeline to perform a common Others stage.
Deploy stage

In common pipeline implementations, the execution continues with the deployment of the new version. Once more, CodeDeploy orchestrates the action initialization, resource management and monitoring tasks. However, CodePipeline delegates the execution of the deployment to another service provider. In this scenario, CodeDeploy executes the deployment job as previously described in the CodeDeploy subsection. However, in the diagram available below, we provide a high-level overview of the deployment job in the context of the CI/CD pipeline. The last task for CodePipeline is to monitor and finalize the pipeline execution once the deployment job is finalized.

Diagram illustrating the actions followed by CodePipeline to perform a common Deploy stage.

The table below summarizes the stages described above. The table also includes information about the activity description, actors and outcome resources generated during the stage.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Activities</th>
<th>Actors</th>
<th>Outcomes</th>
</tr>
</thead>
</table>
| Source | - Collect source code into the source package.  
- Store source package with random name. | CodePipeline.  
CodeCommit or other repository solution. | Source package. |
| Build | - Specify source and artifact package location and random names.  
- Start build execution with source and artifact packages information.  
- Compile and configure source code.  
- Prepare and upload an artifact package. | CodePipeline.  
Artifact package. |
| Others | - Run unit tests.  
- Run scans.  
- Request manual approval.  
- Other activities. | CodePipeline.  
CodeBuild.  
Other services and tools. | Test results.  
Scan results.  
Approval report. |
| Deploy | - Specify artifact location and random name.  
- Start deployment execution.  
- Identify targets and notify targets.  
- Download and install the artifact package.  
- Configure new tasks or functions. | CodePipeline.  
CodeDeploy.  
CodeDeploy Agent.  
EC2 and On-premise instances. | CodeDeploy execution.  
New application version installed in target instance. |
During our analysis of the actions that can be performed using the three services described in the Background section, we identified a threat for the security of the CI/CD pipelines implemented via AWS services. Indeed, our analysis revealed that developers who are allowed to use the StartBuild action can gain access to secrets and environments that they were not authorized for nor expected to access. We have taken this as a working hypothesis and will explain and demonstrate this in the remainder of the section.
**Threat actor**

In our threat analysis, we decided to focus on the developer role as the primary threat actor. To accomplish this, we have evaluated the AWS managed policies. These policies are created and managed by AWS to provide permissions for many common use cases and job roles. AWS recommends that their customers use these policies in favor of inline policies in most cases and to create customer managed policies based on existing AWS managed policies. The CodeBuild Developer access policy grants the StartBuild action to developers. We further describe this action in the Attack Vector subsection later in the article. We understand that, based on AWS recommended policies, developers should be allowed to use the StartBuild action. It is also important to note that the description of this policy states that the user will not be provided administrative access rights.

**Threats**

In this article, we describe the potential risks resulting from the misuse of the StartBuild action by rogue developers, or malicious attackers on behalf of AWS developers. The results of the analysis include different cases for the use of the StartBuild action that can represent a risk for the enterprise IT infrastructure and CI/CD pipelines. Below, we describe three threats that we identified and can affect the customers of AWS.

**Exfiltration of secrets**

The protection of sensitive and secret information within a secure SDLC environment is crucial to enforce a hardened configuration and access control within the development and deployment environments. This information can include confidential information or secret keys that can be used for other harmful actions if they are exposed to malicious actors. We evaluate whether a developer could exfiltrate sensitive information and secrets that should not be accessible by developers in a secure SDLC environment.

**Application tampering**

A secure SDLC environment should not allow any user to alter the application in an unauthorized or undetected manner. This is enforced to stop actors from being able to add malware or backdoor functionalities to the legitimate application. We evaluate whether an AWS developer could bypass security mechanisms implemented in secure SDLC environments and modify the developed application in an uncontrolled manner.

**Execution of privileged commands in deployment server**

Segregation of duties is a basic risk management principle that helps protect against a single actor being able to perform a complete sensitive action or set of actions. In the secure SDLC environment, this principle is applied by means of ensuring that the actions required in the SDLC environment are distributed and can only be performed by specific actors. For example, developers should not have high-level privileges in deployment environments since they already have access to the source code used to build the application. However, we test whether a developer can potentially run privileged commands in the deployment systems.

**Attack vector**

The threat analysis we performed on the CodeBuild actions particularly, and on the complete AWS CI/CD pipeline implementation services, helped us notice that the StartBuild action can serve as an attack vector for developers to gain access to secrets and environments that they were not authorized nor expected to access. The StartBuild starts a new execution of a specific project within the CodeBuild service. This action supports a large list of parameters that were evaluated as part of our threat analysis. However, in this article, we cover two parameters that can potentially be abused and lead to a privilege escalation risk for the enterprise SDLC environments and IT infrastructure. To understand how this threat can manifest itself, we explain the buildspecOverride and artifactsOverride parameters.

---

buildspecOverride parameter

The official description of this parameter states that it allows one to define a new BuildSpec file to be used during the started CodeBuild execution. This parameter supports three definition options: inline definition of BuildSpec file, filename of the BuildSpec file provided within the source code base, and S3 ARN for a BuildSpec file stored in an S3.

The potential risk lies in the fact that you can define the BuildSpec file in an arbitrary manner, which also affects the commands executed in the container instance. As a result, it allows the user to determine what the container executes and how it manages the resources.

artifactsOverride parameter

Similarly to the previous parameter, the artifactsOverride parameter allows users to modify a CodeBuild project configuration value in an arbitrary and controllable manner. This can be used to determine the location in which CodeBuild will store the output resources resulting from the execution of the container instance.

The user can export the artifact package into an S3 bucket owned by the user. Also, the user can set an existing artifact package as destination, even if the user has no access to the existing artifact package, and use CodeBuild to override the existing artifact package with a modified artifact package. Therefore, this parameter represents a risk with regard to data exfiltration and legitimate data integrity as we further demonstrate later in this article.

Attack scenarios

In this subsection, we further introduce three attack scenarios that further describe the threats previously listed in this article. The scenarios are ordered based on the complexity of the attack vector, least complex first. Moreover, we have described some of the security mechanisms that are affected within the different areas of the IT security landscape of an enterprise. The first scenario focuses on the exfiltration of secrets managed within the pipeline. The second scenario focuses on application tampering for injection of malicious functionalities. The third scenario focuses on the execution of privileged commands in the target server instance. In all scenarios, we consider developers that are allowed to perform the StartBuild action in the CodeBuild project, independently or as part of a CI/CD pipeline.

Exfiltration of secrets

A developer who can start a build execution can potentially exfiltrate secrets, such as production passwords, signing certificates and proprietary source code. In addition, the developer can potentially alter the original source code to introduce malware and system backdoor functionality within the target application.

The buildspecOverride parameter can be used by a developer to specify the commands executed by the container instance of CodeBuild. These commands can be used to collect the private resources that the container instance has access to via environment variables, secrets managers (Secrets Manager, System Parameter Store, etc.), storage services (S3, RDS, DDBB, etc.) and source code repositories (CodeCommit, GitHub, etc.). Next, the commands can potentially submit these secrets into a service provider under the control of the developer via the internet connection of the container instance. The following image illustrates this attack vector.

In addition, the developer can also use the artifactsOverride parameter to exfiltrate the secrets. This can be done by specifying a S3 bucket of choice with the parameter. As a result, CodeBuild stores the artifact package in the S3 bucket of choice including the secrets and resources that could be included in the artifact package as part of the legitimate behaviour of the build execution. The following image illustrates this attack vector.
As a result, the developer can exfiltrate secrets and intellectual property resources used during the application build process. This scenario makes ineffective the security tasks, such as secure secrets management and access control with regard to critical and sensitive intellectual property, performed to control and harden the access to sensitive and proprietary information managed within the SDLC.

Application tampering

A developer who can start a build execution can potentially specify a BuildSpec file definition that performs the legitimate build process while including backdoor and malware functionalities into the generated output artifact. This output artifact could be used for deployment or be delivered to clients and allow attackers to use the malicious functionalities.

The commands defined in the crafted CodeBuild declaration can attempt to modify the source code or append libraries to add malware or system backdoor functionalities. This altered code can be used to compile an application package that can then be used to replace the legitimate one. This altered application can potentially be used in servers, which would introduce malicious functionality that could be used by attackers.

As a result, the developer can tamper with the application to introduce malicious functionalities. This application can potentially be deployed in servers and introduce a new threat to the deployment environment of the enterprise or the customers, in the case that the application is published. This scenario can allow developers to bypass security controls implemented in the development phase, such as peer code review and the four eyes principle.
Execution of privileged commands in deployment server

A developer who can start a build execution can potentially run arbitrary privileged commands in the deployment server if the CodeBuild project is used as part of a CI/CD pipeline, for example, orchestrated by CodePipeline.

The buildspecOverride parameter can be used by a developer to specify the commands executed by the container instance of CodeBuild. These commands can be used to incorporate new scripts and modify the scripts included in the artifact package. These scripts are likely to be executed in the deployment server during the installation of the new version via the artifact package.

In CI/CD pipelines based on AWS services, the artifact package will include the AppSpec file used by the CodeDeploy Agent. Therefore, the AppSpec file can be altered within the CodeBuild container instance via the commands included in the BuildSpec file. In addition, the runas property of the AppSpec file allows the user to determine that the arbitrary commands and scripts are meant to be run as a privileged user (i.e., root) without any password or authentication being required to elevate the user. This can provide the developer full control over the system if the artifact is installed. The following image illustrates this attack vector.

As described in the Background section, the CodePipeline service sets random names for the input and output resources for every pipeline execution. These random names and the storage location are necessary for the exploitation of this attack scenario. Therefore, we have evaluated the CodeBuild and CodePipeline actions and identified those that can be used in order to collect this randomized information. We noted that a user can collect this information by asking CodeBuild or CodePipeline for information. Moreover, we identified that the CodeBuild Developer access policy, described in the Thread actor subsection, also grants the actions required to collect the randomized information via CodeBuild. Therefore, the developer can collect the randomized information and provide it within the artifactsOverride parameter to request CodeBuild to overwrite the legitimate artifact package.

As a result, the developer can potentially execute privileged commands in the deployment servers of the enterprise. This scenario can bypass security controls that are recommended to be implemented at the operations side of an SDLC, such as secrets management, system security, least privilege principle policy, and configuration hardening.

The following table summarizes information for the two options that can enable the collection of the CodePipeline randomized resource names. The table includes the actions required, the AWS managed policy that incorporates the actions required and the target actors for the AWS managed policies based on the policy description.

<table>
<thead>
<tr>
<th>Options</th>
<th>Activities</th>
<th>AWS Managed Policy</th>
<th>Target actors</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeBuild access</td>
<td>· ListBuilds.</td>
<td>· CodeBuild Developer Access.</td>
<td>· AWS developers.</td>
</tr>
<tr>
<td></td>
<td>· BatchGetBuilds.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodePipeline access</td>
<td>· ListActionExecutions.</td>
<td>· CodePipeline Read Only Access.</td>
<td>· Auditing and monitoring actors, could also be developers.</td>
</tr>
</tbody>
</table>

As a result, the developer can potentially execute privileged commands in the deployment servers of the enterprise. This scenario can bypass security controls that are recommended to be implemented at the operations side of an SDLC, such as secrets management, system security, least privilege principle policy, and configuration hardening.

The following table summarizes information for the two options that can enable the collection of the CodePipeline randomized resource names. The table includes the actions required, the AWS managed policy that incorporates the actions required and the target actors for the AWS managed policies based on the policy description.

<table>
<thead>
<tr>
<th>Options</th>
<th>Activities</th>
<th>AWS Managed Policy</th>
<th>Target actors</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeBuild access</td>
<td>· ListBuilds.</td>
<td>· CodeBuild Developer Access.</td>
<td>· AWS developers.</td>
</tr>
<tr>
<td></td>
<td>· BatchGetBuilds.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodePipeline access</td>
<td>· ListActionExecutions.</td>
<td>· CodePipeline Read Only Access.</td>
<td>· Auditing and monitoring actors, could also be developers.</td>
</tr>
</tbody>
</table>

As a result, the developer can potentially execute privileged commands in the deployment servers of the enterprise. This scenario can bypass security controls that are recommended to be implemented at the operations side of an SDLC, such as secrets management, system security, least privilege principle policy, and configuration hardening.

The following table summarizes information for the two options that can enable the collection of the CodePipeline randomized resource names. The table includes the actions required, the AWS managed policy that incorporates the actions required and the target actors for the AWS managed policies based on the policy description.

<table>
<thead>
<tr>
<th>Options</th>
<th>Activities</th>
<th>AWS Managed Policy</th>
<th>Target actors</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeBuild access</td>
<td>· ListBuilds.</td>
<td>· CodeBuild Developer Access.</td>
<td>· AWS developers.</td>
</tr>
<tr>
<td></td>
<td>· BatchGetBuilds.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CodePipeline access</td>
<td>· ListActionExecutions.</td>
<td>· CodePipeline Read Only Access.</td>
<td>· Auditing and monitoring actors, could also be developers.</td>
</tr>
</tbody>
</table>
In this section, we demonstrate step by step how an AWS developer can use the StartBuild action to exfiltrate secrets and sensitive information from the container instance and potentially execute privileged commands in the deployment server. To simplify the job of the developer, we have developed a tool called CodePipeline Poisoning Tester (CPPT). The CPPT tool will help automate the steps a developer has to perform to accomplish the attack.
The CPPT tool\(^3\) is composed of a serverless AWS based API endpoint (CPPT API) and a python script (CPPT script). The API is used for execution monitoring and demonstrating that data can be exfiltrated over the internet connection of the container instance. The script is responsible for collecting the information of the pipeline execution and adapting the arbitrary commands to be executed within the container instance. Since this tool does not aim to harm any of the instances or their resources, it will inject harmless commands. The CPPT tool's arbitrary commands only perform two harmless actions listed below:

1. **Create files called CPPTWasHere.\(^*\)**
   - The asterisk (*) is replaced by ‘container’ or ‘server’, depending on the instance in which the file is created.
   - These files allow the user to evaluate the poisoning process when the CPPT API cannot be used.
   - The file created in the container is also added to the poisoned artifact.
     - It also demonstrates that arbitrary files can be added to the artifact.
   - The file created in the server is written in the root (/) directory of the server.
     - It also demonstrates that root level commands can be executed.

2. **An HTTP POST request to the CPPT API**
   - Used to keep track of the poisoning process.
     - It also demonstrates that data can be exfiltrated.
   - The request helps the CPPT tool’s monitoring phase.

---

\(^3\) [https://github.com/AsierRF/CodePipeline-Poisoning-Tester](https://github.com/AsierRF/CodePipeline-Poisoning-Tester)
How the CPPT tool works

In this subsection, we describe the actions performed by the CPPT tool to demonstrate that a developer can poison a CodePipeline CI/CD pipeline by means of application tampering, exfiltration of secrets and execution of privileged commands in the deployment server. The following steps also represent the four generic pipeline stages to describe the scenario. The images provided below contain black arrows that denote the common pipeline execution orchestrated by CodePipeline, similar to the ones illustrated in the Background section, whereas the red arrows denote CPPT tool functionality during the specific stage.

Source stage

The CodePipeline pipeline execution starts with the Source stage, as represented in the image below. A developer can have the CPPT tool script running in the local device. The CPPT script will start querying CodePipeline in order to gather information about the ongoing pipeline executions. The CPPT script can also be started after invoking the execution of the pipeline.
Build stage

The pipeline execution will continue with the Build phase, as represented in the image below. The CPPT script will continue querying CodePipeline to gather information regarding the random location in which CodeBuild will store the output artifacts for this pipeline execution. Once the output artifact is known, the CPPT script will request CodeBuild to start a new build with the properties listed below. In addition, once the StartBuild is successful, the CPPT script will POST the information to the CPPT API.

CPPT script properties when calling the StartBuild action:

- artifactsOverride: The artifact location being the same as the one received from CodePipeline.
- buildspecOverride: A CPPT crafted BuildSpec that will execute the harmless actions previously described.
**Others stage**

While the pipeline execution is performing other tasks, such as SAST analysis and/or manual approval, the build started by the CPPT script will be performed. As shown in the diagram below, the container used to build the artifact will follow the CPPT crafted BuildSpec definition, which will contact the CPPT API and create a file called CPPTWasHere.container. This demonstrates that the developer can exfiltrate data and modify the source code.

The CPPT crafted BuildSpec will also poison the output artifact package by adding a CPPT crafted AppSpec file and script (referred to as poisoned artifact package in the diagram). This poisoned artifact package will replace the original one created by the legitimate build execution requested by CodePipeline. In the background, the CPPT script will start querying the CPPT API to show the user a table to monitor the status of the poisoned sources and artifacts.

Diagram illustrating the actions followed by CPPT during a common pipeline execution Others stage.
Deploy stage

The deployment stage and its activities will execute normally. If the build execution triggered by the CPPT was finished by this time, the package downloaded by the CodeDeploy Agent will be the poisoned package artifact, as shown in the image below. As a result, the arbitrary commands injected by the CPPT tool will be executed within the deployment server. This demonstrates that the developer can potentially execute privileged commands and exfiltrate secrets from the server, such as AWS access keys, certificates and passwords.

In the background, the CPPT script will continue querying the CPPT API to show the user a table to monitor the status of the poisoned artifacts. The status of the different attempts can always be checked directly in the DynamoDB table created for the CPPT API via the CloudFormation template also provided within the tool repository.
In this section, we discuss the difficulty to apply security mechanisms and approaches when adopting cloud services for development. This discussion results, among others, from the response of AWS when we notified them of the results of our threat research.

**Extent of intended behavior**

We have notified AWS about our security concerns with regard to the actual implementation of the buildspecOverride and artifactsOverride functionalities. In their response, they stated that the behavior of the StartBuild action and the buildspecOverride and artifactsOverride parameters is intended behavior. We agree with the fact that the official documentation for the StartBuild action informs about the buildspecOverride parameter, which allows one to define a new BuildSpec file, and the artifactsOverride parameter, which allows one to change the storage location to be used by CodeBuild to store the output artifact. However, we think that the general message provided by the current documentation is not adequately reflecting the risk represented by the action and its parameters.

Indeed, the definition of the StartBuild action in the official documentation describes the functionality provided as: *Starts running a build*. However, we have demonstrated that the StartBuild action does not just allow the user to start a build but also allows one to modify the container runtime logic and output storage location. Next to influencing the build action in CodeBuild, the functionality provided via the buildspecOverride parameter can also have an impact on other services and resources that are not linked to CodeBuild when they would be integrated within a CI/CD pipeline, which would increase the difficulty of evaluating and mitigating the risk.

We understand that there may exist situations that require this type of functionality. For example, the buildspecOverride parameter allows one to test a new BuildSpec definition during a test run without altering the project configuration and potentially breaking the pipeline execution. Or, this functionality also extends the flexibility of CodeBuild to support the use of a single project that uses multiple BuildSpec files, which contain different configuration and build commands, to target different environments, such as Test, Acceptance and Production. However, we believe that the risk that this functionality can represent towards AWS customers should be further analyzed.

In our opinion, the functionality should be more restrictive with regard to the options supported for BuildSpec file declaration. This could be achieved, for instance, by updating the buildspecOverride parameter to only support BuildSpec file declarations that are included in the source code package. When these files would be included in the source code base, they are commonly covered by other security controls implemented in a secure SDLC, such as peer review and the four eyes principle. This restriction would still allow customers to cover the use cases described above while ensuring a more controllable and more secure SDLC environment.

4 https://docs.aws.amazon.com/codebuild/latest/APIReference/API_StartBuild.html
Shared responsibility model

In their response, AWS also stated that granting access to the StartBuild action and the security of the CI/CD pipeline deployed in the AWS cloud environment is the responsibility of their customers. While we understand this reasoning, we believe that AWS as a cloud provider can further enhance the security posture of their customers by improving the information that customers have available. As we already mentioned, the official documentation, in our opinion, does not explain the risk that the StartBuild action can represent. Moreover, AWS recommends following their AWS managed policies, which grant this action to users with the developer role. This action could lead to a risk, as we already demonstrated, especially in SDLC environments that rely on the Segregation of Duties principle. Therefore, we think that customers can, and will, implement their CI/CD pipelines in AWS in an insecure manner, even when they are in line with the official documentation and recommendations.

At this moment, AWS customers can only completely deny the access to the StartBuild action to avoid the discussed risk or adapt their pipelines and create customized solutions to try and limit the impact of the risk. This may not suit the needs of all customers of AWS; however, they have no security mechanisms to help limit the access to the buildspecOverride and artifactsOverride functionalities in a granular manner.

Potential improvements

As part of our notification, we recommended AWS to implement security mechanisms to allow customers to limit the access to, and the functionalities of, the buildspecOverride and artifactsOverride parameters. These mechanisms could be implemented by (1) reducing or restricting the functionalities of the parameters to only allow the referencing of resources within a controlled environment, such as the source code repository or specific S3 buckets, (2) splitting the action into one action without support for the parameters and one privileged action with support for the parameters, or (3) implementing a more granular access control mechanism, i.e. via IAM tags, to limit the access to the parameters. These mechanisms can help IAM administrators to limit the access to the action and its parameters in a granular manner. However, these mechanisms will have to be implemented by AWS. As long as these mechanisms are not available to companies, the latter are put in a difficult situation in which they have to decide between security and functionality.

This threat analysis has clearly demonstrated that the adoption of cloud services brings new challenges for enterprises and their security teams. Enterprises need to be aware of their responsibilities with regard to the security in the cloud. The analysis of computing performance and cost-efficiency are common tasks performed when adopting cloud services. However, we believe that investing in training on cloud security should be a must in the list of tasks performed when adopting and developing cloud-based solutions. Clearly, identifying and mitigating these new threats requires a deep understanding of the cloud environment and the related technologies.
In this article, we have analyzed potential threats in setting up CI/CD pipelines by demonstrating that the overriding of the BuildSpec file via the StartBuild action can represent a risk of CI/CD pipeline poisoning for the SDLC environment, the CI/CD pipeline and the IT infrastructure of the AWS customers. We want to point out the potential danger of using the StartBuild action in order to ensure a proper security discussion when implementing a secure SDLC process and environment including CI/CD pipelines deployed via AWS services.

In our opinion, the actual implementation of the StartBuild action and its buildspecOverride and artifactsOverride parameters represents a privilege escalation and data exfiltration risk for four reasons:

- The developer gains access to the protected information managed by the container, such as application secrets, configuration parameters and signing certificates.
- The developer has the possibility to determine what is included within the output artifact that can potentially be deployed. As a result, the developer can execute arbitrary privileged commands in the deployment instance.
- The developer elevates his/her privileges within the AWS environment via the service role assigned to the container instance. The container instance service role may provide access rights that extend those originally granted to the developer role.
- The developer can determine the location in which the output artifact has to be stored to abuse the CodeBuild service and exfiltrate data into storage locations that are not owned or controlled by the enterprise.

We recommend AWS customers to not implement AWS managed policies without evaluating whether they fit their needs. Enterprises are responsible for the hardening of the CI/CD pipeline configuration and implementation. Consequently, they have to make the difficult choice between security and flexibility. At this time, AWS customers have no mechanism to limit the access to the buildspecOverride and artifactsOverride parameters other than denying access to the StartBuild action. Therefore, we recommend adding an explicit deny statement for the CodeBuild StartBuild action in the policies and roles assigned to users that are not intended to use this action.

In case the buildspecOverride and artifactsOverride features would not be required but developers would need to start build executions, we recommend implementing a serverless function that could be invoked by the developers. This serverless function could call the StartBuild action on behalf of the developer without allowing the buildspecOverride and artifactsOverride parameters to be used. However, if the buildspecOverride and artifactsOverride features would be required, we recommend isolating pipeline actions in different CodeBuild projects and limiting the user access to only those projects that do not manage secret or sensitive information.

Note that, while in this article we have focused on AWS services to prove our point, similar arguments can (probably) be made for other cloud providers or CI/CD tools that form the basis of modern software development practices. The analysis described in this article can be extended by evaluating other providers and deployment types. Moreover, a secure and cloud-based SDLC guidance and good practices framework can be of great use for the cloud and DevOps communities.