Securing a Deployment Pipeline

Len Bass*, Ralph Holz*, Paul Rimba*†, An Binh Tran*, Liming Zhu*†
*Software Systems Research Group, NICTA
†School of Computer Science and Engineering, UNSW
Sydney, Australia
Email: {firstName.lastName} @nicta.com.au

***Abstract*—At the RELENG 2014 Q&A, the question was asked, "What is your greatest concern?" and the response was "someone subverting our deployment pipeline". That is the motivation for this paper. We explore what it means to subvert a pipeline and provide several different scenarios of subversion. We then focus on the issue of securing a pipeline. As a result, we provide an engineering process that is based on having trusted components mediate access to sensitive portions of the pipeline from other components, which can remain untrusted. Applying our process to a pipeline we constructed involving Chef, Jenkins, Docker, Github, and AWS, we find that some aspects of our process result in easy to make changes to the pipeline, whereas others are more difficult. Consequently, we have developed a design that hardens the pipeline, although it does not yet completely secure it.**

***Keywords*—supply chain, continuous deployment, DevOps**

I. Introduction

Securing the supply chain for software has become a matter of increasing concern [1], [2]. A deployment pipeline constitutes the last portion of the supply chain for software. A deployment pipeline retrieves portions of the final system from a variety of sources, builds the application from those inputs, packages and tests that build, and places the application package on a number of physical or virtual machines. Any of these stages can include vulnerabilities, e.g., retrieved portions of the system may not be the desired ones; building, packaging or deployment may have been corrupted.

Securing a pipeline is complicated by the wide variety of tools involved in the pipeline, where each one comes with their own security mechanisms. Our pipeline utilizes Chef, Jenkins, Docker, Github, and AWS. Because of space and complexity limitations, our examples focus on Jenkins. The points we make, however, are applicable to all of the tools in our pipeline and, indeed, to any deployment pipeline.

What we present is very much a work in progress, and consequently we are presenting a snapshot of our status at this point in time. Our contributions, at this point, are: a discussion of what it means to "subvert a pipeline", an engineering process that could produce a trusted pipeline within certain constraints, and an application of that process to a real world pipeline that does not satisfy those constraints yet but that results in a hardened although not entirely trusted pipeline.

II. Security and Trust

We first discuss how a pipeline might be subverted. We identify three distinct scenarios for subverting a deployment pipeline. The first one is when the image that is deployed is not a valid image. That is, the specification of the build is incorrect, the build itself does not follow the specification, or the image deployed is not the same image that was built. The second one is when an image is deployed without going through the complete pipeline. That is, an image is available for consumer interaction without passing through the checks inherent in the pipeline. The third one is when the production environment is accessible from a different environment. That is, machines in another environment (e.g., test, development) have direct access to the production environment when they should not. In this paper, we focus on the first scenario.

Our ultimate goal is to make our pipeline trustworthy. We use this term in the sense that we have assurance our pipeline is secure against attack and cannot be made to behave in a way that is not the intended one (for the purpose of this paper, we assume that, in the absence of attacks, the software adheres to its specification). There are several ways of generating such trustworthiness. These include security testing, static analysis, and formal verification (including model checking). The level of trustworthiness generated by these techniques varies, with security testing being the lowest and formal verification being the highest. Although formal verification provides the highest level of trustworthiness, it is very costly. The current state of the art of formal verification is up to about 5K lines of code.

In the context of our own pipeline, we start from the fact that Jenkins has had several vulnerabilities in the past [3], noting that a monolithic design means that a breach in any code part means attackers can gain the privileges of the entire process and thus directly gain control over other code parts, too. One of our goals is thus to find mechanisms that will harden a pipeline against attack. The result is a pipeline where the attack surface of the code base is reduced, in the sense that many parts of the code base have no access to other critical parts and thus cannot subvert the pipeline even when breached.

III. Threat Model

One question to consider when attempting to secure any software system is how powerful the attacker is that we want to defend against. This is not a straightforward question: consider, e.g. that the security of the pipeline depends on host security (a system's security) as well as network security (the security of the communication channels).

With the exception of our Deployer (defined later) and the actual cloud running the image, we assume that all our components run on a single machine. We consider

- A remote attacker attempting to exploit a component in the build environment that is directly accessible from outside of the environment. If successful, an attacker can gain the privileges of the process. We do not consider further privilege escalation (to administrative rights) – this would trivially compromise all processes on the machine.
- In the spirit of the authors of [4], a remote attacker attempting to exploit a component indirectly and without direct network access to the build environment. This can be done by a seemingly benign and unnoticeable change in a third-party repository, with code fetched from there as part of the build process. The actual code is not malicious itself for the third-party but does introduce an exploit which can, as above, allow a compromise of the entire process on the machine, with new privileges for the attacker.
- Attacks on the network links on the public Internet, i.e. on the connections between our machine, third-party repositories, Deployer, storage, and cloud.

We do not investigate the direct security problems of fetching code from third-party repositories where it is not possible to tell if the code has been tampered with by an attacker. These constitute a class of challenging problems of their own. One particular direction worth exploring are deterministic builds, as they are currently being introduced in, e.g., the Debian GNU/Linux distribution. We also explicitly exclude one environment, which we cannot control: the cloud where the image is deployed. Here, we assume this environment to be trustworthy. Furthermore, we assume that the compiler is correct.

IV. Identifying Security Requirements

A pipeline is defined by a number of steps, each of which consists of a number of actions to be carried out. There is a logical ordering on the steps and the actions. For simplicity, we assume there are no branches in the pipeline, i.e. it can be viewed as actions being carried out in sequence, and actions are grouped logically into steps. Every action can be described by a specification, and every action has a well-defined outcome. Our goal is to harden the pipeline's integrity, which we understand here to mean three goals are achieved:

1) The order in which actions are executed cannot be changed by anyone without the appropriate credentials.
2) Only someone with the appropriate credentials can change an action such that it does not conform to the activity described in its original specification.
3) The outcome of an action is exactly as described in the (most recent) specification.

Note that this definition is concerned with integrity only. At this stage, we disregard issues like leaking sensitive information to an attacker – except if this leak leads to the attacker obtaining such credentials that are necessary to violate one of the goals above. In particular, this definition means that we need to prevent any attacker from changing any of the code or executables that are used to carry out an action. Furthermore, our third goal implies that the specification of the actions must be protected.

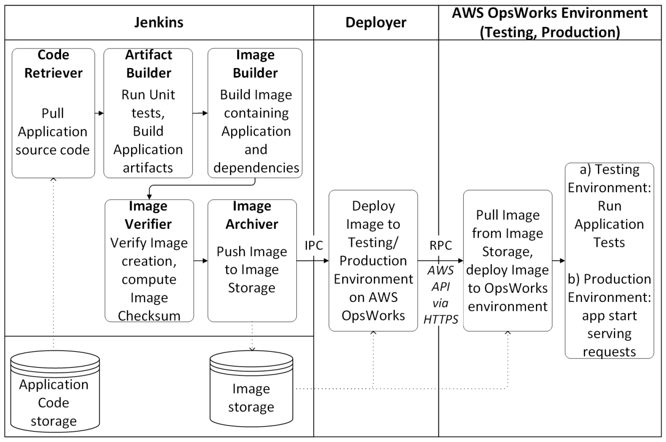
Jenkins
Deployer
AWS OpsWorks Environment (Testing, Production)
Code Retriever
Pull Application source code
Artifact Builder
Run Unit tests, Build Application artifacts
Image Builder
Build Image containing Application and dependencies
Image Verifier
Verify Image creation, compute Image Checksum
Image Archiver
Push Image to Image Storage
IPC
Deploy Image to Testing/ Production Environment on AWS OpsWorks
RPC
AWS API via HTTPS
Pull Image from Image Storage, deploy Image to OpsWorks environment
a) Testing Environment: Run Application Tests
b) Production Environment: app start serving requests
Application Code storage
Image storage

Fig. 1: Jenkins' current activities in deploying an image

V. Process of Securing the Pipeline

Scenario 1, as defined in Section II, is about the integrity of the image that is created and deployed. In our originally implemented pipeline, this is a responsibility of Jenkins. Jenkins retrieves elements that go into the image, builds them according to a specification, saves the created image, tests the created image, and subsequently invokes a Deployer and AWS to place the image into the desired environment – either a testing or a production environment. We show these activities in Figure 1. Although this pipeline is based on Jenkins and deploys into AWS, the steps are generic and must be performed by any Continuous Integration/Deployment tool suite. We will use Figure 1 to exemplify both our process and our hardening recommendations.

Our idealistic process for hardening the security of the pipeline has the following steps:

1) Identify the security requirements for the pipeline.
2) Identify the trustworthy and untrustworthy components of the pipeline.
3) Repeat until all of the requirements have been satisfied OR can no longer decompose the untrustworthy components
   a) Model the interactions between the components
   b) Analyze the model to check whether it satisfies its requirements.
   c) Decompose untrustworthy components causing an unsatisfied requirement into a trustworthy and an untrustworthy portion.
4) Implement new trustworthy components and modify untrustworthy components to utilize the trustworthy components to perform sensitive operations.

This process is based on the idea that the actual building and deploying activities are small pieces of code that can

	Functionality	Access rights	Complexity/Criticality	Network connectivity	Typical impact of compromise
Jenkins	Does everything as previously, except for the components below	Jenkins' workspace; but none of the below	Very high / Low	Localhost only	Limited to workspace
Orchestrator	Triggers each step in sequence	Read on configuration files	Low / High	TLS with deployer	Can change order of steps
Code Retriever	Pulls source code from repository to local workspace	Read from repo, Write to own workspace; no Execute	Low / High	TLS with deployer	Pull wrong source code; modify source code
Unit Tester	Runs automated unit tests on source code	Read from Code Retriever's workspace; Write and Execute on own workspace.	High / Low	None	Cannot compromise integrity (but can halt pipeline)
Artifact Builder	Builds deployable artefact from source code	Read from Code Retriever's workspace; write and execute on own workspace	High / High	Accesses third party software	Injection of malicious third-party code
Image Builder	Build VM image	Read from Artifact builder's workspace. Write on own workspace.	Medium / High	Accesses third party software	Replacement of image with malicious one
Image Verifier	Verifies image creation	Read from Image builder's workspace.	Very low / Low	None	Can halt pipeline
Image Archiver	Pushes image to storage	Read from Image builder's workspace. Write on Storage.	Very low / High	TLS with Storage	Replacement of image with malicious one. Leakage of credentials (Storage)
Deployer	Deploy image to testing/production	Read from Storage.	Low / High	TLS with Orchestrator. Access to production API.	Replacement of image with malicious one. Leakage of credentials.

TABLE I: Hardening recommendations for components

be encapsulated into trustworthy components and that the trustworthy components can mediate access to the actual building and deploying activities. More detail about the formal portions of this process can be found in [5]. As we will discuss later, some components are not easily decomposed. Hence this process will result in a secure pipeline if carried to completion or a hardened pipeline if completion is not possible. If all access to sensitive data or function is performed by a trustworthy component, and the actions of untrustworthy components with respect to sensitive data or functions are mediated by a trustworthy component, then the pipeline is secure. With large systems such as Jenkins or AWS, gaining trust in components cannot be done by code-level formal verification because it is too expensive to do. Understanding the vulnerabilities, however, allows measures to be identified that will reduce the attack surface of the pipeline and result in its hardening.

In our case, we used the Serscis Access Modeller (SAM) [6] to perform design-level formal verification in step 3.b of our process. The requirements are translated into assertions and SAM verifies that the current design with designated trustworthy and untrustworthy components satisfies those requirements.

We began by assuming that all of the components are trustworthy. Even in this case, there were potential vulnerabilities. We explore one of them to show how our process works.

The specification of Jenkins' activities is by means of a configuration file. These files are stored in human readable form. This means that an attacker could edit these files outside of Jenkins or its support systems. Because our model does not model the environment in which these files are stored, our analysis pointed out the possibility of the configuration files being vulnerable. Consequently, we modified our model to assume that the configuration files are encrypted and integrity-protected. To implement this change, we need a small trustworthy component to be created (not too difficult) and to modify Jenkins and its support systems to assume that the configuration file is protected (somewhat more difficult).

Once we have a verified design with the assumption that the components of the pipeline are trustworthy, we relax that assumption and make components untrustworthy. This exposes new vulnerabilities. We illustrate this again with an example.

One of the vulnerabilities comes from the fact that the components "Artifact Builder" and "Image Builder" are untrustworthy. Once the image is constructed, we can verify its integrity by calculating a checksum, protecting the checksum, and verifying at the last step in the pipeline that the checksum is correct for the deployed image. The problem we have not yet solved is how to trust Artifact Builder and Image Builder.

Our process tells us to decompose these two portions into untrustworthy and trustworthy portions. There are three different functions we want to trust: the Artifact Builder and Image Builder retrieve the correct elements from the correct locations, and they only retrieve elements from specified locations. Furthermore, they compose these elements correctly.

The first two of these requirements can be done in a trustworthy fashion by means of cryptographic protocols and origin authentication. Note, however, that the correctness of the elements retrieved is outside of the scope of our pipeline.

What we currently do not know how to accomplish is to generate trust in the composition of the retrieved elements. This is the heart of the build process. There are many ways that one could subvert the actual build including adding non-specified components to the build. For the moment, we fall back to satisfying only the first two functions – leading us to call this "hardening" rather than "securing". This line of thinking led us to identify and restrict the input and output of each of the components and to identify those portions that could easily be made trustworthy. That is, we re-architected our pipeline.

VI. Result of Re-Architecturing

Table I shows the result of our re-architecting. The omnipotent component (e.g., Jenkins) has been re-architected to interact with a number of much smaller components, each with defined access rights and network connectivity permissions. The result is a deployment pipeline that is well-understood. Compromise of certain components can still damage the system to the point that the integrity of the pipeline is compromised. However, a significant amount of code is executed with fewer privileges; so that compromise of that portion cannot lead to compromise of the pipeline's integrity any more. The smaller code pieces lend themselves to inspection and (formal) verification much more readily.

In particular, we can see that several components that were previously untrustworthy are now either trustworthy or have very restricted access to resources. In the context of our threat model, this means an attack can either be stopped, thanks to verifiability of a smaller code base, or constrained, thanks to enforcing access rights between components. In future work, capabilities could provide an interesting way to implement such constraints for components. The Orchestrator, for example, needs very few privileges and needs to communicate only with one component over the network, the Deployer. It cannot, however, replace the image that the Deployer is going to read from Storage. If the Orchestrator is compromised, the attacker can only halt the pipeline. The components that still allow compromising the pipeline's integrity are the Code Retriever, Artifact Builder, Image Archiver, and Deployer. Components that have no network connectivity, e.g. the Unit Tester, do not need to be trustworthy and their impacts on security have been minimized.

VII. Discussion

We discussed how to harden one portion of the deployment pipeline – the portion performed by Jenkins in our current implementation. The portion we discussed, however, is only a small portion of the actual pipeline we have implemented.

We also only investigated one possible pipeline subversion scenario. We have identified three different scenarios and there are likely many others. Consequently, the actual securing of a deployment pipeline is going to be a complicated affair.

One aid in securing a pipeline is to architect the tools involved so that the critical functionality of the pipeline – a small portion – can be made trustworthy and the trustworthy functionality mediates access to the critical functionality. We have presented our initial thoughts on how to do this. Our plans are to continue exploring what it takes to architect and secure a deployment pipeline and this paper presents a snapshot of our initial thoughts.

Acknowledgment

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] R. J. Ellison, J. B. Goodenough, C. B. Weinstock, and C. Woody, "Evaluating and Mitigating Software Supply Chain Security Risks," CMU/SEI-2010-TN-016, May 2010.

[2] Improving Cybersecurity and Resilience through Acquisition. https://acc.dau.mil/CommunityBrowser.aspx?id=694372&lang=en-US.

[3] https://wiki.jenkins-ci.org/display/SECURITY/Home

[4] M. Perry, S. Schoen, and H. Steiner, Reproducible Builds, Talk at 31C3, Hamburg, Germany, December 2014, http://events.ccc.de/congress/2014/Fahrplan/events/6240.html

[5] P. Rimba, L. Zhu, L. Bass, I. Kuz, and S. Reeves, "Composing Patterns to Construct Secure Systems," unpublished.

[6] T. Leonard, M. Hall-May, and M. Surridge, "Modelling access propagation in dynamic systems," ACM Transactions on Information and System Security (TISSEC), vol. 16, no. 2, September 2013.