Security of Public Continuous Integration Services

Volker Gruhn Christoph Hannebauer Christian John

paluno – The Ruhr Institute for Software Technology
University of Duisburg-Essen
{volker.gruhn | christoph.hannebauer}@paluno.de, {christian.john}@stud.uni-due.de

ABSTRACT

Continuous Integration (CI) and Free, Libre and Open Source Software (FLOSS) are both associated with agile software development. Contradictingly, FLOSS projects have difficulties to use CI and software forges still lack support for CI. Two factors hamper widespread use of CI in FLOSS development: Cost of the computational resources and security risks of public CI services. Through security analysis of public CI services, this paper identifies possible attack vectors. To eliminate one class of attack vectors, the paper describes a concept that encapsulates a part of the CI system via virtualization. The concept is implemented as a proof of concept.

Categories and Subject Descriptors

K.6.5 [**MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS**]: Security and Protection—*Invasive software, Unauthorized access*; D.2.3 [**SOFTWARE ENGINEERING**]: Testing and Debugging—*Monitors*

General Terms

Security, Performance

1. INTRODUCTION

Over the course of the last fifteen years, Continuous Integration (CI) has become an important practice in software development. Developers applying this practice commit their code modifications to the main code repository at least every day. Every modification that compiles and passes the tests should be committed. A dedicated integration machine continuously compiles and tests the current versions of the source code. This way, integration problems are detected early. Thus, integration problems are easier resolved as they do not have time to grow. [15, 13]

Continuous Delivery is an extension to CI. In the Continuous Delivery approach, the CI system builds the release versions of the software products that will be shipped to the customers [21]. Current trends push this practice even further and propose to integrate even uncommitted code modifications [20].

However, in the case of Free, Libre and Open Source Software (FLOSS), the advent of CI did not have a measurable impact on development practices [11]. Antoniades et al. does not list CI systems as common tools used in FLOSS development [3]. This is one of the indications that CI is or at least was not a common practice in FLOSS development. This seems surprising at first, since both CI and FLOSS development are commonly associated with agile software development [26].

Hundreds of thousands of FLOSS projects do not maintain their own infrastructure and rely on software forge services like SourceForge[1] and github[2]. These software forge services provide a broad range of software development tools like Version Control System (VCS)s, bug trackers, and web site and download hosting. They do not usually provide CI systems, though. Therefore, maintainers of FLOSS projects have to maintain CI systems on their own servers if they want to use CI. Small FLOSS projects may rely completely on software forge services for their server-based tools. Thus, setting up and maintaining a CI system on their own servers is a major effort. This is presumably a major fact restraining the spread of CI among FLOSS projects.

Consequentially, large FLOSS projects that maintain their own infrastructure anyway hesitate less to use CI: For example, Mozilla[3] uses CI techniques like automated testing [28]. These CI systems integrate and test modifications committed to the FLOSS project's VCS repositories automatically. Hence, even for those large FLOSS projects, CI is restricted to core developers with access to the project's VCS repositories.

After Deshpande's and Riehle's study about lacking impact of CI on FLOSS [11], a couple of organizations started offering CI as a service in the cloud. Examples include FaZend[4], Travis CI[5], tddium[6], and CloudBees BuildHive[7]. Some of

[1] `http://sourceforge.net/`
[2] `https://github.com/`
[3] `https://www.mozilla.org/`
[4] `http://www.fazend.com/`
[5] `https://www.travis-ci.org/`
[6] `https://www.tddium.com/`
[7] `https://buildhive.cloudbees.com/`

these services offer free usage plans to FLOSS projects and even integrate into software forge services like github. However, some of the named CI services place restrictions on usage in case of the free FLOSS usage plans [9]. Similarly to the CI systems of large FLOSS projects like Mozilla, only the core developers of each FLOSS project may use its CI systems.

A multitenant CI system is a CI system that (i) hosts multiple projects and (ii) has multiple users each of whom shall access only some of the projects. These users are called tenants of the multitenant CI system.

This paper discusses what currently limits the widespread adoption of CI practices in FLOSS projects in Section 2. Two major reasons are identified: Resource costs and security vulnerabilities specific to multitenant CI systems. The remaining paper focuses on the second problem of public CI services, the security vulnerabilities in multitenant CI system. The state of research on software security analysis is presented in Section 7. Section 3 systematically analyzes possible attack vectors on CI systems. Section 4 proposes a solution that counters some of the attack vectors identified in the preceding section. In Section 5, a sample attack demonstrates the validity of the proposed solution. The limitations of the proposed solution are discussed in Section 6. The results are summarized in Section 8.

2. CURRENT LIMITS OF FLOSS CI

This section discusses two reasons that limit the widespread usage of CI systems among FLOSS projects, resource costs and security.

2.1 Resource costs

Firstly, CI systems need more computational resources than other software development tools. This is because CI systems do not only download every patch, but also have to compile all new versions of the source code and have to run a possibly large array of tests. These operations constitute a build job. Each committed source code modification of a project leads to a new build job in the CI system.

According to Moore's Law, transistor count and therefore computing performance doubles every 18 months. Assuming that computing performance is the dominating factor of costs for resource requirements of a build job, Moore's Law reduces the costs of each build exponentially over time.

However, the total number of FLOSS source code lines doubles every 15 months [12]. The number of added FLOSS source code lines and similar measures double at the same rate, as they are derivations of the number of lines and derivations of exponential functions are exponential functions with the same basis again. Therefore, no matter what specific measure accounts most for the cost of CI maintenance, the described measure may be used to estimate the growth of computing performance required to provide CI services to all FLOSS projects. Hence, this required computing performance also doubles every 15 months. Taking these facts together, the following formula specifies the cost $C(t)$ to provide CI services to all FLOSS projects at a specific date, where t is the number of months lapsed since some fixed start date until that specific date, $\rho(t)$ is the computing performance required to provide CI services to all FLOSS projects at the specific date, and $\alpha(t)$ is the computing performance available for a constant amount of money:

$$C(t) = \frac{\rho(t)}{\alpha(t)} = \frac{2^{\frac{t}{15}}}{2^{\frac{t}{18}}} = 2^{\frac{t}{15} - \frac{t}{18}} = 2^{\frac{t}{90}}$$

This corresponds to a yearly cost increase of 9.7 percent. It is therefore possible that free usage plans of CI services for FLOSS projects will decline for economical reasons. The same argument still applies if CI services shall be provided only to a constant fraction of FLOSS projects, as this would not change the rate of cost changes.

One might question the assumption that computing power is not the dominating factor of costs for resource requirements of a build job. In this case, the general argument still applies, as variations of Moore's Law also apply to related resources. For example, hard disk space grows at a similar rate as computational power, with data density currently doubling about every two years [23]. This is still exponentially, but slower than computing power, so the yearly cost increase would be higher.

Accordingly, the same reasoning also applies to other software development tools like VCS. Assuming that the cost of VCSs is largely determined by hard disk prices, the cost of VCSs for all FLOSS projects increases at a yearly rate of about 23 percent. Thus, free usage plans for these services on software forges may become economically unfeasible in the future.

However, it is unknown for how long the exponential growth of FLOSS sustains. The growth may become subexponential before the economical wall for software development described above is reached. Additionally, with the growth of FLOSS in terms of source code, public interest in FLOSS also grows. This eventually results in increased funding for FLOSS projects and their maintenance. This may even out the increased costs of maintenance.

2.2 Security of public CI systems

Secondly, CI systems are more vulnerable to attacks and misconfiguration than the software development tools currently offered by software forges: In contrast to the other software development tools, build jobs require the CI systems to execute code that developers of the FLOSS project provide. This custom code may cause errors starting from unnecessary resource consumption over CI system outages to espionage, denial-of-service-attacks, virus bridgeheads, and similar malicious purposes.

For multitenant systems, it's even worse, as one malicious tenant may attack other tenants via the CI system. If future CI services want to provide CI services also for prospective developers instead of only the core developers of each FLOSS project, it would be even easier for a malicious attacker to gain access to the CI systems.

There is a particularly dangerous example of an attack on CI systems, a variation of the trusting trust attack on compilers [39]: An attacker could modify the CI system in a way such that software artifacts produced on those system would

be infected with a virus. For Continuous Delivery systems, infected versions of the software product would spread to end-users. These infections are insidious, as even a paranoid end-user who carefully reviews the software product's source code would miss the malicious code parts that the virus infection induces. Eventually, the infected CI system would build a new infected version of its own software that hides all traces that an infection has ever happened.

3. SECURITY ANALYSIS OF CI SYSTEMS

This section systematically analyzes attack vectors that malicious tenants of a multitenant CI system may use to attack the CI system. Of course, all tenants may easily compromise their own build on the CI system and potentially build malicious software with the help of the CI system. However, this is not a security risk, as these malicious tenants can also build malicious software without a public CI system. Thus, behavior is considered an attack only if the behavior either damages the CI system as a whole or compromises builds of other tenants.

The security analysis describes general attack vectors of CI systems, in contrast to actual flaws of specific implementations of CI systems. Thus, the attacks described cannot directly be used to attack a concrete CI system without modification. An attack on a concrete CI system must use one of the attack vectors determined in this section, though, but the attack still has to exploit a flaw in the implementation or configuration of the CI system.

As explained, the analysis applies to CI systems in general. Still, the CI system Jenkins[8] serves as an example and guides the line of argumentation. Jenkins-stats counts 57,334 Jenkins installations [5]. Hence, Jenkins is a common representative of a CI system. The concepts explained are common among implementations of CI systems, although the wording may be different for other specific implementations.

Langweg and Snekkenes provide a taxonomy of attacks on software applications. Earlier taxonomies of attacks [24] focus on whole operation systems. They argue that every attack on a software application has a location, a cause, and an impact: The attack needs an attack vector through which the data is transmitted to the target software application, which they call location. Further, the attack must exploit a fault in the software or configuration, which they call the cause of the attack. Finally, the attack allows unauthorized operations that defy one or more of the security attributes confidentiality, integrity, and availability [4]. They call these unauthorized operations the impact of the attack. [25]

The analysis in this section systematically examines the data inputs to CI systems. The analysis checks which data inputs can serve as attack vectors. In the taxonomy of Langweg and Snekkenes, these attack vectors are locations of attacks. For each attack vector, the possible impact of the attack is discussed. As described above, Langweg and Snekkenes assign attacks a third property, the cause. However, causes are specific to the implementation of a CI system and therefore, the analysis in this section looks at causes only as far as they are relevant for CI systems in general.

[8] `https://jenkins-ci.org`

3.1 Attacks using the web interface

Even in a CI system with only a single tenant, there may be several projects configured. These projects define among other things which source codes should be loaded and built, which events trigger a new build job, and what artifacts shall be archived after each build job. Different projects may depend on each other and form a hierarchy.

In a multitenant CI system, these projects separate user accounts from accessing other tenants' resources. Users of the multitenant CI systems can access their own projects but not the projects of others. CI systems may allow a much more fine-grained configuration of access rights on a sub-project level, for example Hudson [31] and Jenkins [38].

Security flaws in CI web interfaces have proven to be usable attack vectors [37, 36]. Web services may also open additional attack vectors. However, these are problems of web applications in general and not specifically CI systems. Therefore, this class of problems will not be analyzed more deeply in this paper.

3.2 Attacks using the build process

Executing a build job in a CI system comprises the four steps

1. **VCS checkout**, the build server downloads the source code from the VCS,
2. **Build preparations**, the build server sets up the build environment,
3. **Builder runs**, the build server compiles the source code to produce build artifacts, and
4. **Notification**, the build server notifies the master server about the results of the build.

The concept of these steps borrows from Prakash's presentation of build steps in the CI system Hudson [30]. There are differences in Prakash's build steps and the build steps shown here, as Prakash's build steps describe a build job from an software architect's perspective instead of a software security analyst's.

The build server does not execute these steps in strict order. Instead, some steps overlap. For example, *Notification* starts with the *VCS checkout* already. Each step introduces possible attack vector on the CI system. The following paragraphs discuss the attack vectors for each of these steps.

VCS checkout. In this step, the build server downloads the latest version of the source code from the configured VCS. The downloaded source codes usually also contain build scripts like Makefiles [16], shell scripts or Visual Studio project files [27]. These files are not actually compiled into the produced build artifact to be delivered to the end-users. Instead, they support the build process with tasks like pre-processing the source code to target a specific execution platform or distinguishing between wanted and unwanted optional modules for a specific build.

This step already involves two possible attacks.

First, the VCS may include external resources. In the VCS Apache Subversion (SVN)[9] for example, a directory under source control may contain the property `svn:externals` to specify additional repositories to include in the checkout [10, Externals Definitions in Chapter 3]. As another example, the VCS git[10] contains a similar feature called submodules [7, Chapter 6.6].

These VCS features may instruct the build server to execute Hypertext Transfer Protocol (HTTP) requests to other servers. Therefore, these requests may

- access data on other web servers with host-based access restrictions and disclose it to the attacker,
- execute web service commands with host-based access restrictions on other web servers, or
- consume network bandwidth on the build server and target other servers in a denial of service attack.

Second, the downloaded source code may contain symbolic links. Attackers may forge symbolic links to files on the build server outside of the directory intended for the project. Subroutines of the CI system with high privileges may thereby be tricked to access files on the build server with elevated access rights. In other programs, there have already been attacks that exploit symbolic links [22].

Build preparations. In the *Build preparations* step, the build server configures the build environment for the build job. For example, a configure script may prepare files for a specific target platform or a batch process may copy dependencies to locations required for the builder runs. The Build preparation step may require execution of arbitrary code, as requirements varies among projects and it is therefore not possible to offer only a closed set of preparation options.

After the builder runs, packaging the artifacts may also require the execution of arbitrary code within some post-build steps. From a security perspective, these post-build steps are similar to the *Build preparation* step. Therefore, they are also discussed here.

Build preparation steps can be configured directly in the CI system. For example, Jenkins allows shell commands to be configured directly via the web interface. An example of a malicious command is shown in Figure 1.

There are obvious countermeasures against this attack: The process executing the build may have only restricted user rights which suppresses malicious parts of the code. Alternatively, the CI system may completely forbid its tenants to configure *Build preparation* steps. Automated analysis of build steps may also prevent malicious build steps to be executed.

[9] `https://subversion.apache.org/`

[10] `http://git-scm.com/`

Execute shell

Command `rm -f -v /`

See the list of available environment variables

Figure 1: Screenshot of Jenkins' web interface. A build step is configured to delete all system files.

It is not sufficient, however, to only restrict build steps configured directly in the CI system. The build scripts loaded from the VCS with the source code can also contain arbitrary code that is executed during the build. FLOSS projects often use Makefiles [16] which allow execution of arbitrary commands. Other types of build environments also provide this possibility. Figure 2 shows how to configure a malicious deletion command in a Visual Studio project file. The CI system Jenkins can be configured to call the application MSBuild to execute these Visual Studio project files and then the build server also executes the commands configured there.

Builder runs. The build server compiles the source codes and links all libraries in this step. The output are the software artifacts of the project. The output also comprises executable tests that the developers have programmed. The build server executes these tests and adds the test results to the output. Plugins may allow the tests to be distributed to special test servers. This allows tests to be run on multiple platforms.

An attacker may include malicious behavior in the source code. Of course, the build server does not execute this malicious code directly, as it is only compiled and linked at first. However, the build server executes tests and therefore indirectly also the software product. This way, the attacker can use the build server to start denial of service attacks or access restricted web resources. An attack has additional impact if the user account running the builder run was given too much rights or if the attackers use a security flaw to elevate their rights. In these cases, the attackers could shut down the build server. As a more sophisticated attack, they could modify the build server to compile or link virus code into all software artifacts build on the server. On a multi-tenant system, the software artifacts of other tenants would then get infected with malicious code.

As explained in Subsection 2.2, this may lead to a variation of Thompson's trusting trust attack [39]: When the CI system builds a new version of its own software, the compromised build server may inject malicious code in the CI system software. If this is thoroughly executed, this attack is very difficult to detect, as the CI system may remove almost all traces that the attack has happened.

In a standard installation of Jenkins with default settings on

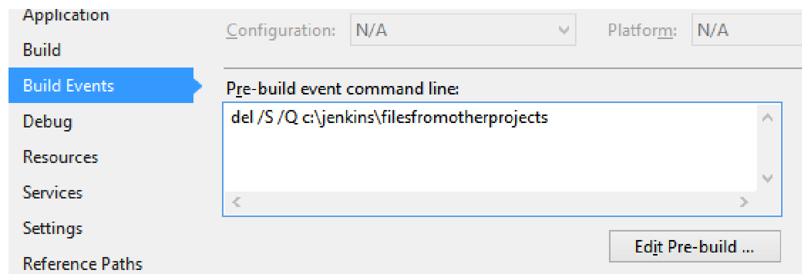

Figure 2: Screenshot of Microsoft Visual Studio 2012. A build step is configured to delete files from other projects.

a Linux Debian server, a builder run may execute commands on the build server with administrator rights. Thus, the attacks mentioned above would apply to such a system.

There are multiple countermeasures against the attacks listed above. Obviously, the account running the builder runs should be granted only minimal rights. Additionally, the build server might scan the source code for known malicious code fragments. However, it is a common practice to store library dependencies in binary form next to the dependent source code. An attacker may compile the malicious code into one of these libraries, so that the source code compiled on the build server is free of malicious parts, although the tests indirectly execute the malicious code in the library. Thus, the build server would need to take the extra step and also scan the binary libraries. A wide range of commercial programs exists for the task of detecting malicious binary files. However, malware detectors are unreliable especially when detecting new malware or new variations of existing malware [8].

Section 4 presents another approach to counter the attacks emanating from this step of the build. This approach works even if all of the countermeasures above are useless or not even applied. Of course, it is still better to have multiple lines of defense and also implement the measures mentioned above where possible.

Notification. The build server reports the current status and eventually the results of the build back to the master server. In Jenkins, this includes every line of output to the console. After the build, the build server may transmit test results and build artifacts to the master server.

If attackers take control over the build server in one of the preceding steps, they may modify the data transmitted back to the master server. There is a weaker and a stronger form of an attack resulting from these modifications.

Some CI systems display test results or other result data in a web browser. Jenkins is an example, if it is extended with appropriate plugins. In the weaker form of the attack, the attackers add executable code like JavaScript code to the results. In this case, the attackers have to find a flaw in the master server's web interface to circumvent the security mechanisms that suppress execution of such executable code. If other tenants open the result data with their browser, the malicious executable code may act within the security context of these other tenants from within the web browser. Thus, the malicious executable code may change the configuration of other tenants' projects in the CI system. This may enable the malicious code to reproduce itself to other projects on the CI system.

The stronger form of the attack uses the communication channel from a compromised build server back to the master server. The build server uses a flaw in the communication protocol to take over the whole master server on behalf of the attackers. Obviously, this would compromise the whole CI system with all projects from all tenants.

4. SECURE BUILD SERVERS

As shown in Section 3, users of multitenant CI systems may attack each other through the resources they share in the CI system. More specifically, attackers may modify the behavior of build servers to affect succeeding build jobs of other projects on the same build server. Attackers have to use security flaws in the source code or configuration of the CI system to accomplish their attack. While the maintainers may fix each of these security flaws when they become aware of them, the CI system is always susceptible to zero-day exploits and the maintainers' inattentiveness. This section describes an approach that inhibits attacks from one project to other projects built on the same build server. At the same time, the approach does not require a secure configuration of the build servers.

Through the attacks described in the steps *Build preparations* and *Builder runs* in Section 3, attackers can prepare a malicious build job that compromises the build server of the CI system that builds this malicious build job. The attackers may then use the compromised build server for their purposes. On a multitenant CI system, build jobs of projects from other tenants will be scheduled to the same build server. Attackers may compile malicious code into the artifacts produced in the build jobs of those projects. These infected software artifacts may act as Trojan horses and compromise the computers of end-users of the other projects. The build processes and infections are depicted as a Unified Modeling Language (UML) state machine diagram in Figure 3. Note that the CI system maintainers and tenants of the CI usually cannot distinguish between the states *Clean system* and *Compromised system*. In this paper, a Default Build Server is a build server as described above, with the concept described later in this section not applied.

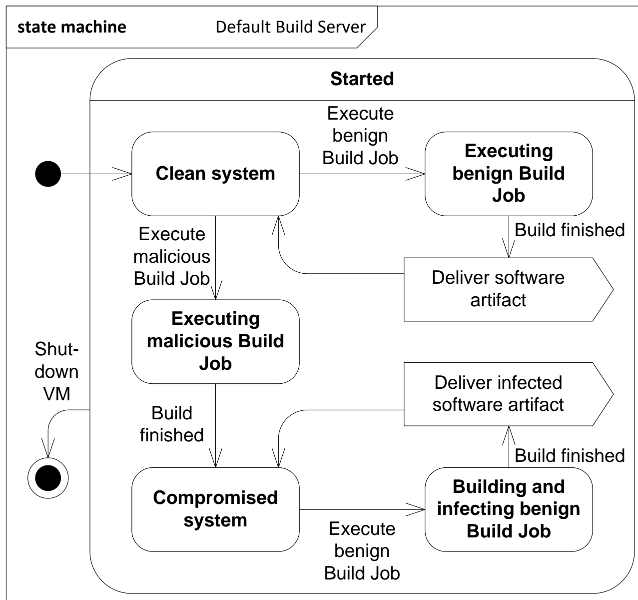

Figure 3: UML state machine diagram accounting for attacks that infect a Default Build Server.

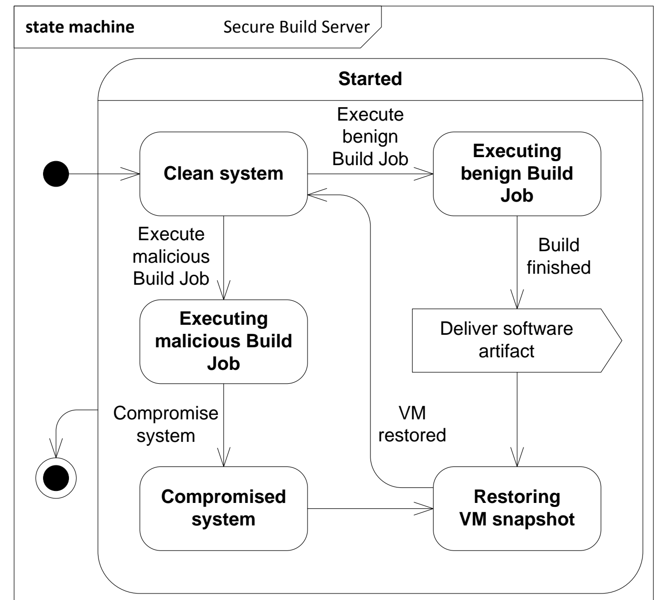

Figure 4: UML state machine diagram of an attack on a CI system using the proposed solution.

As explained, there is no generic method to prevent attackers from compromising the build server that executes build jobs of a malicious project. However, a build server can be considered in the state *Clean system* if it has not previously executed any build jobs. Since build servers executing build jobs for projects owned by attackers do not need to be in the *Clean system* state, this criterion is met if every project has its own dedicated build server. As the master server controls all build servers and therefore must not be compromised, the master server must not be a build server. This approach is infeasible for CI systems responsible for too many projects to have one build server for each of these projects. Public CI systems that offer CI services for FLOSS projects therefore cannot use this approach.

However, the source code of most FLOSS projects changes seldomly and only in a few FLOSS projects, the source code changes often: The FLOSS project meta-repository Ohloh measures activity on FLOSSprojects. As of 2013-03-14, Ohloh has 582,716 FLOSS projects in its database. With the FLOSS projects sorted by Ohloh's activity measure, the projects at the 0.1, 1, and 10 percentile are *libgit2*, *kfusion*, and *chain-pattern*, respectively. These projects had 2279, 60, and 0 commits within the last twelve months, respectively. [6]

Thus, most of the dedicated build servers would be idle most of the time, even for projects with very high activity. For example, libgit2 would also cause only about 6.23 builds per day. Accordingly, the following approach uses virtualization to prevent builds from different projects on the same machine. At the same time, it does not require dedicated servers for every project.

The concept of a Secure Build Server extends the Default Build Server as described above. A Secure Build Server is a Virtual Machine (VM). After every build, a plugin on the master server restores the VM of the build server to an original clean state. To improve performance, the VM uses a virtualization environment that supports snapshots, for example VirtualBox [29, Chapter 1, Snapshots]. Restoring a snapshot allows the virtualization environment to quickly discard all changes made since the start of the VM. Similarly, Amazon's cloud services allow using a so called local instance store to start machines from a specific snapshot [2]. If an attacker schedules a malicious build job, the build server will build the malicious build job and the build server will be compromised, but the build server will be reset before it executes any other build jobs. Build jobs of other projects therefore cannot be infected anymore. Figure 4 shows a UML state machine diagram of such a Secure Build Server.

4.1 Proof of concept

This subsection describes a proof of concept realization of a Secure Build Server. The realization uses the CI system Jenkins. The virtualization environment VirtualBox[11] hosts the VMs of the build servers. A modified version of the Jenkins plugin VirtualBox Plugin[12] controls these VMs of the build servers.

The modified version of the VirtualBox Plugin forks from the beta version branch "snap"[13] that supports VirtualBox snapshots. An additional modification ensures that the build server VMs restart after every build job and thereby restore a specific snapshot. The modified version developed for this paper can be downloaded from github[14].

[11] https://www.virtualbox.org/
[12] https://wiki.jenkins-ci.org/display/JENKINS/VirtualBox+Plugin
[13] https://github.com/jenkinsci/virtualbox-plugin/tree/snap
[14] https://github.com/paluno/virtualbox-plugin/tree/snap

4.2 Resource considerations

As explained in Section 2, computational resources and security are two major problems limiting widespread CI use for FLOSS projects. The approach presented in this section increases security, as it prevents all attacks that use the build server to attack other tenant's builds. However, this is a trade-off, because the CI system has to restore a VM for every build. This subsection measures the impact on performance and therefore computational resources to take this trade-off into consideration.

The time needed to execute a build job depends on properties of the project: The build tools, the number of lines of source code, and the programming language, for example. Another important factor is the performance of the hardware the build server runs on. The project and the hardware are both implementation specific and therefore will not be regarded here. Hence, this analysis disregards the time of the actual Build Job's execution. Instead, the analysis compares the start up times of a Default Build Server and a Secure Build Server as described above.

The performance tests ran in the test environment described in the last subsection. The Jenkins master server runs on a Linux VM on a physical Windows system. Two Jenkins slaves representing a Default Build Server and a Secure Build Server run as VMs on another physical system, which runs Linux. Both physical systems have a four core Intel Atom 330 Central Processing Unit with 4x1.6 GHz, 2 GB physical memory, and a 250 GB SATA hard drive with 7200 RPM.

In the performance test, two build jobs are issued simultaneously on the Jenkins master server. There is a delay of about three seconds between the starts of the two build jobs, as they were issued manually via the web interface. Figure 5 shows a UML timing diagram where this issuance on the Jenkins master server is at time 0 seconds. After between 20 to 28 seconds, the VirtualBox host starts resuming or restoring the VMs. The Default Build Server needs about 58 seconds after the build job was issued to resume and start executing the build job. The Secure Build Server needs about 140 seconds to restore the original snapshot and then start executing the build job.

The concept of a Secure Build Server described in this section has no influence on the time needed to execute a build job for the first time. However, a Secure Build Server needs more time for subsequent build jobs, as the Build Server cannot store intermediate data of the build job. For example, while a Default Build Server only loads the modifications of the source code since the last build job from the VCS, a Secure Build Server has to download the whole source code for every build job. Additionally, a Default Build Server may reuse intermediate files from the last build job. Examples are object files for unmodified source code files in the programming language C. The performance decrease of a Secure Build Server in practice may therefore be greater than the pure start up times of the Build Server as analyzed in this subsection.

5. VALIDATION

This section describes the validation of the approach described in Section 4. First, an attack is devised that modifies a build server such that all later builds on this server produce modified versions of software artifacts. The attack is sufficient to inject malicious code into these modified versions. However, an actual infection is not necessary for a proof of concept and will not be part of the attack. Second, the devised attack executes on a Default Build Server in the test environment described in Section 4. This shows that the Default Build Server is vulnerable to the attack. Third, the devised attack fails on a Secure Build Server as defined in Section 4. This shows that the approach successfully prevents these kinds of attacks.

5.1 Attack scheme

As an example scenario, the CI system consists of a standard Jenkins installation with one master server and two managed slave servers. All systems run on a minimal Debian Linux environment. The master server connects to the slave servers via Secure Shell (SSH) tunnels. One slave server is a Default Build Server, the other slave server is a Secure Build Server. The master server only dispatches build jobs but is not a build server itself.

We demonstrate the attack on a fictive FLOSS project ABC that we set up for this validation in the environment described above. Any real FLOSS project would have also worked, as the weakness relies on the configuration of the CI system alone. ABC's SVN repository resides on servers with the domain name `svn.example.com`. The CI system also hosts a second project named Mallet. Unknowingly to the CI system and the ABC maintainers, Mallet tries to inject malicious code into the build artifacts of project ABC.

Mallet's strategy exploits the fact that the build server performs all build jobs. Mallet redirects access to ABC's SVN repository to its own SVN repository that contains modified code. Mallet achieves this with a modification of the `hosts` file on the build server. A post-build steps of Mallet's build adds an entry for `svn.example.com` to the `hosts` file with an Internet Protocol (IP) address that Mallet controls, in this example `192.168.21.59`. This post-build step consists of the following line of shell script:

```
echo '192.168.21.59 svn.example.com' >> /etc/hosts
```

5.2 Attack on a Default Build Server

After Mallet's build, the file `hosts` contains an entry claiming that the host name `svn.example.com` resolves to the IP address `192.168.21.59`, a server that Mallet controls. When the build server builds ABC the next time, the build server tries to fetch the freshest source code for project ABC. ABC has an SVN repository configured on the server `svn.example.com`, so the build server resolves this host name. As the `hosts` file has a higher priority than Domain Name System (DNS), the build server does not resolve the correct IP address. Instead, the Default Build Server resolves the IP address `192.168.21.59` that Mallet configured in the hosts file.

Thus, the Default Build Server does not download the actual ABC source code from `svn.example.com`, but instead some modified version from Mallet's server. Mallet might have configured the server to forwards requests for the SVN repository on `192.168.21.59` to the actual SVN repository on

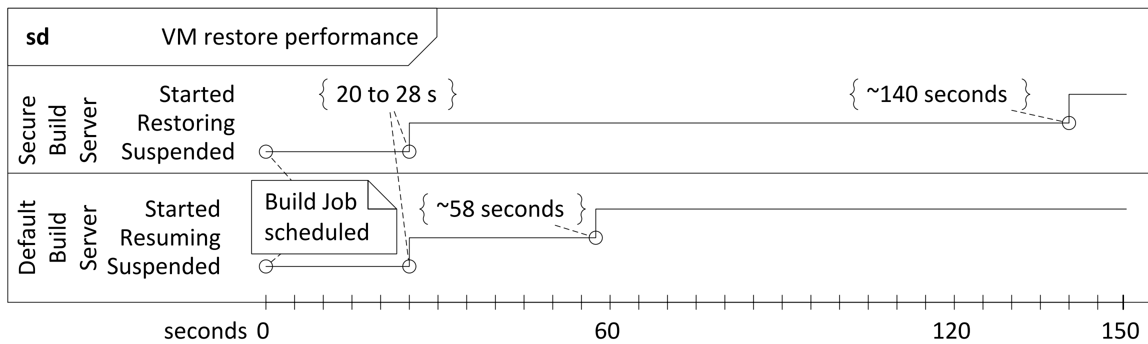

Figure 5: UML timing diagram of the VM start up latency in the test environment.

svn.example.com. In this case, changes on svn.example.com still propagate to the software artifacts build on the CI system. Mallet's server might add malicious source code while forwarding the original source code. Thus, the modification would be difficult to detect for the ABC developers, as it does not show in the logs. In the actual implementation of this proof of concept, Mallet's server just delivers different source codes instead of actual malicious source code, as it is obvious that delivering malicious source code is easily possible.

5.3 Attack on a Secure Build Server

Mallet now launches the same attack on a CI system that implements the approach described in Section 4, a Secure Build Server. Contrary to the previous course of events, the Secure Build Server resets the VM of the slave server after Mallet's build job. This restores the original `hosts` file and reverts Mallet's changes. When building ABC afterwards, the build server resolves `svn.example.com` to the correct IP address from DNS and loads the unmodified ABC source code. Thus, Mallet's attack was unsuccessful.

There are more obvious and direct countermeasures to prevent the devised attack. Running the build jobs on the build server in a restricted user account that has no access to the `hosts` file suffices already. However, the attack may be refined to outdo these countermeasures again, like combining the devised attack with a rights elevation attack. Of course, there are also countermeasures against these rights elevation attacks. Additionally, different but similar attacks to the shown attack also have obvious and direct countermeasures, like fixing a security flaw that allowed attackers to take over the build server. However, specific countermeasures only work against specific attacks. The devised attack is therefore only a trivial representative of a larger class of attacks, each of which has a specific countermeasure. The approach described in Section 4 prevents all attacks in this larger class.

6. LIMITATIONS

The Secure Build Server concept described in Section 4 has limitations that this section will discuss.

The build jobs are encapsulated and cannot influence each other, as the VM acting as Secure Build Server is restored to a clean snapshot after every build job. There are two methods that attackers may use to bypass the encapsulation.

First, attackers can use a flaw in the VM code to escape the VM. This way, they can compromise the computers hosting the VMs. Subsequently, they can compromise other VMs or other parts of the computer infrastructure. [33]

Yet, VM escapes are not a problem specific to Secure Build Servers. In fact, if cloud services like Amazon EC2[15] host the Secure Build Servers, attackers might escape any VM that Amazon EC2 hosts to achieve the same goals as escaping a Secure Build Server. It would even be easier for the attackers to escape a VM that they themselves own, as they would have full control over these VMs without the need to find and exploit flaws in the CI system.

Second, the Paragraph *Notification* in Subsection 3.2 explained that a compromised build server could attack the master server of the CI system. Restoring a snapshot of the Secure Build Server would remove malicious code from the Secure Build Server, but the malicious code would survive on the master server. The compromised master server could easily compromise the Secure Build Server again after it restored the snapshot. The master server could compromise other build servers as well.

As explained in Section 2, the cost of computational resources is another problem of public CI systems. Subsection 4.2 showed that Secure Build Servers perform worse than Default Build Servers. In practical applications, the security benefit might not be worth the added costs of computational resources.

7. RELATED WORK

This section gives a summary of current research on security solutions based on virtualization.

Rosenblum and Garfinkel describe advantages and challenges in virtualization research. One section is also dedicated to security properties of VMs. Monitors running outside the VMs can analyze attacked system even down to the hardware level. Attacks are therefore more difficult to conceal. They also suggest to use VMs to separate environments or

[15]`https://aws.amazon.com/ec2/`

applications from each other. The solution in Section 4 may be seen as a special case of this general idea. [34]

They further elaborate that VMs with a dedicated purpose should only be allowed network access as far as necessary. This Guest OS Independence, as they call it, allows more fine-grained access control than solutions without VMs. This concept allows to prevent attackers from using Secure Build Servers for denial of service attacks. [17]

Price compares security advantages and disadvantages of virtualization. The list of advantages contains encapsulation as the possibility to run different applications without influencing each other. Disadvantages that implementors of the solution described in this paper should consider include rollback vulnerabilities: Administrators may install security patches on VMs like the Secure Build Servers. If the VMs restore snapshots without the security patches afterwards, the VMs are vulnerable again. This may happen unknown to the administrators, who consider the VMs patched. [32]

8. CONCLUSION AND FUTURE WORK

This paper argued that small FLOSS projects rely on software forge services instead of maintaining their own infrastructure for development tools. Therefore, more FLOSS projects could profit from CI techniques if software forge services offered CI systems as a service. Large FLOSS projects with their own infrastructure may speed up development if external developers could use their CI systems. However, public CI systems with multiple independent tenants suffer from two problems before they can be used extensively. First, CI systems come with high costs because of the required computation, bandwidth, and memory resources. Second, as all tenants execute their own code on the CI systems, tenants may cause malicious or non-malicious failures that affect other tenants.

A security analysis showed the attack vectors enabling failures to spread from one tenant's part of the CI system to the others'. One class of attack vectors starts on the build server that executes the tenant's own code. Section 4 detailed a concept that encapsulates build job via virtualization and snapshots, creating a Secure Build Server. The CI system resets the Secure Build Servers back to an original and uncompromised state after each and every build job. Thereby, Secure Build Servers prevent attacks that rely on multiple build jobs on the same build server, even if the attacks are not detected. A proof of concept implementation on the CI system Jenkins shows that the concept of a Secure Build Server can be realized.

8.1 Future Work

A pattern language is a concept to formally cover problems and their solutions of some engineering domain in a comprehensible and adaptable way. Ideally, each pattern in the pattern language identifies one invariant among a group of problems and their solutions [1]. Schumacher and Roedig suggest to specify security problems and their solutions as security patterns [35]. For example, Fernandez and Pan have specified authorization control as security patterns [14]. The problem and solution presented in Section 4 could be written as a pattern. Thereby, other use cases for encapsulation through virtualization may emerge. Furthermore, the relationship of the solution presented in this paper to other security solutions may come out more clear.

In our ongoing work, we study the concept of a Wiki Development Environment (WikiDE) that minimizes the contribution barrier to software development projects. A public CI system is an important part of a WikiDE. Nevertheless, FLOSS projects that do not employ a WikiDE can still take advantage of a public CI system. [19, 18]

Restoring an uncompromised state unconditionally after every build job reduces performance of the CI system. Hence, Secure Build Servers are a trade-off between performance and security. This trade-off can be improved in some cases: If one developer modifies the source code of one project multiple times, this causes multiple build jobs, one for every modification. A single build server can execute these build jobs without restoring an earlier state and without reducing security, as only one tenant of the CI system is affected.

Furthermore, this partially applies to dependency projects: Assume project A depends on project B, so project A uses some build artifacts of project B as input. If project B contains malicious code, the build process of project A can be compromised—independent from the fact whether the build server had executed a build job of project B previously: If the build server was uncompromised before building project A, it can be compromised through the build artifacts of project B that are used as input. This allows reusing a Secure Build Server to execute build jobs from project A even if it has executed a build job from project B before. However, this is a one-way trust and a Secure Build Server must not execute build jobs from project B after executing build jobs from project A.

Therefore, future research should develop strategies to schedule builds to Secure Build Servers, so performance is optimized while maintaining the same level of security as a Secure Build Server that restores a snapshot after every build job.

9. REFERENCES

[1] C. Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[2] Amazon Web Services, Inc. Amazon EC2 FAQs, Mar. 2013. `http://aws.amazon.com/en/ec2/faqs` [accessed 2013-03-28].

[3] I. Antoniades, I. Samoladas, S. K. Sowe, G. Robles, S. Koch, K. Fraczek, and A. Hadzisalihovic. Study of available tools. EU Framework deliverable, FLOSSMetrics Consortium, 2008.

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, 2004.

[5] D. Bartholdi. Total jenkins installations, Feb. 2013. `http://stats.jenkins-ci.org/jenkins-stats/svg/total-jenkins.svg` [accessed 2013-03-28].

[6] Black Duck Software. Ohloh, Mar. 2013. `http://www.ohloh.net/` [accessed 2013-03-14].

[7] S. Chacon. *Pro Git.* Apress, 2009.

http://git-scm.com/book [accessed 2013-03-28].

[8] M. Christodorescu and S. Jha. Testing malware detectors. *SIGSOFT Softw. Eng. Notes*, 29(4):34–44, July 2004.

[9] CloudBees Developer Resources. CloudBees FOSS Program, 2013. http://www.cloudbees.com/foss [accessed 2013-03-28].

[10] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subversion.* O'Reilly Media, 1.7 edition, 2011. http://svnbook.red-bean.com/ [accessed 2013-03-28].

[11] A. Deshpande and D. Riehle. Continuous Integration in Open Source Software Development. In B. Russo, E. Damiani, S. Hissam, B. Lundell, and G. Succi, editors, *Open Source Development, Communities and Quality*, volume 275 of *IFIP International Federation for Information Processing*, pages 273–280. Springer Boston, 2008.

[12] A. Deshpande and D. Riehle. The total growth of open source. In *Fourth International Conference on Open Source Software*, Sept 2008.

[13] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk.* Addison-Wesley Professional, 2007.

[14] E. B. Fernandez and R. Pan. A pattern language for security models. In *8th Conference on Pattern Languages of Programs (PLoP)*, 2001.

[15] M. Fowler. Continuous integration, May 2006. http://martinfowler.com/articles/continuousIntegration.html [accessed 2013-03-28].

[16] Free Software Foundation, Inc. Gnu make, July 2010. https://www.gnu.org/software/make/ [accessed 2013-03-28].

[17] T. Garfinkel and M. Rosenblum. When virtual is harder than real: security challenges in virtual machine based computing environments. In *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10*, 2005.

[18] V. Gruhn and C. Hannebauer. Components of a wiki-based software development environment. In *E-Learning, E-Management and E-Services (IS3e), 2012 IEEE Symposium on*, 2012.

[19] V. Gruhn and C. Hannebauer. Using Wikis as Software Development Environments. In *Proceedings of the 11th International Conference on Intelligent Software Methodologies, Tools and Techniques (SoMeT_12)*, Frontiers in Artificial Intelligence and Applications. IOS International Publisher, October 2012.

[20] M. L. Guimarães and A. R. Silva. Improving early detection of software merge conflicts. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, 2012.

[21] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Professional, 2010.

[22] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Computer Security Applications Conference, 1994. Proceedings., 10th Annual*, pages 134–144, Dec. 1994.

[23] M. Kryder and C. S. Kim. After hard drives—what comes next? *Magnetics, IEEE Transactions on*, 45(10):3406–3413, Oct.

[24] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws. *ACM Comput. Surv.*, 26(3):211–254, Sept. 1994.

[25] H. Langweg and E. Snekkenes. A classification of malicious software attacks. In *Performance, Computing, and Communications, 2004 IEEE International Conference on*, pages 827 – 832, 2004.

[26] A. M. Magdaleno, C. M. L. Werner, and R. M. de Araujo. Reconciling software development models: A quasi-systematic review. *Journal of Systems and Software*, 85(2):351 – 369, 2012.

[27] Microsoft Developer Network. MSBuild Project File Schema Reference, 2013. http://msdn.microsoft.com/en-us/library/5dy88c2e.aspx [accessed 2013-03-28].

[28] Mozilla Developer Network. Mozilla automated testing, Mar. 2013. https://developer.mozilla.org/en-US/docs/Mozilla/QA/Automated_testing [accessed 2013-03-28].

[29] Oracle Corporation. *Oracle VM VirtualBox User Manual*, version 4.2.10 edition, Mar. 2013. http://www.virtualbox.org/manual/ [accessed 2013-03-28].

[30] W. Prakash. Hudson execution and scheduling architecture, Nov. 2010. http://www.hudson-ci.org/docs/HudsonArch-Execution.pdf [accessed 2013-03-28].

[31] W. Prakash. Hudson security architecture, Dec. 2010. http://www.hudson-ci.org/docs/HudsonArch-Security.pdf [accessed 2013-03-28].

[32] M. Price. The paradox of security in virtual environments. *Computer*, 41(11):22–28, 2008.

[33] J. S. Reuben. A survey on virtual machine security. In *Fall 2007 Seminar of Network Security.* Helsinki University of Technology, 2007. http://www.tml.tkk.fi/Publications/C/25/papers/Reuben_final.pdf [accessed 2013-03-28].

[34] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38(5):39–47, 2005.

[35] M. Schumacher and U. Roedig. Security engineering with patterns. In *8th Conference on Pattern Languages of Programs (PLoP)*, 2001.

[36] SecurityFocus. Jenkins Multiple Cross Site Scripting and Directory Traversal Vulnerabilities, Mar. 2012. http://www.securityfocus.com/bid/52384/ [accessed 2013-03-28].

[37] SecurityFocus. Jenkins Multiple HTML Injection Vulnerabilities, Feb. 2012. http://www.securityfocus.com/bid/52055/ [accessed 2013-03-28].

[38] J. F. Smart. *Jenkins: The Definitive Guide.* O'Reilly Media, 2011.

[39] K. Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, Aug. 1984.