Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH ARTICLE Open Access

Increasing virtual machine security in cloud environments

Roland Schwarzkopf*, Matthias Schmidt, Christian Strack, Simon Martin and Bernd Freisleben

Abstract

A common approach in Infrastructure-as-a-Service Clouds or virtualized Grid computing is to provide virtual machines to customers to execute their software on remote resources. Giving full superuser permissions to customers eases the installation and use of user software, but it may lead to security issues. The providers usually delegate the task of keeping virtual machines up to date to the customers, while the customers expect the providers to perform this task. Consequently, a large number of virtual machines (either running or dormant) are not patched against the latest software vulnerabilities. The approach presented in this article deals with these problems by helping users as well as providers to keep virtual machines up to date. Prior to the update step, it is crucial to know which software is actually outdated or affected by remote security vulnerabilities. While these tasks seem to be straightforward, developing a solution that handles multiple software repositories from different vendors and identifies the correct packages is a challenging task. The Update Checker presented in this article identifies outdated software packages in virtual machines, regardless if the virtual machine is running or dormant on disk. The proposed Online Penetration Suite performs pre-rollout scans of virtual machines for security vulnerabilities using established techniques and prevents execution of flawed virtual machines. The article presents the design, the implementation and an experimental evaluation of the two components.

Introduction

Infrastructure-as-a-Service (IaaS) Clouds [1] and virtualized Grid computing are based on the idea that users build individual virtual machines as execution environments for their tasks, allowing them to provide the required software stack without having to deal with Cloud or (multiple) Grid site administrators [2].

While the use of virtual machines is beneficial for service and infrastructure providers (users and providers in the Cloud nomenclature), by lowering the costs for the former and improving utilization and management capabilities for the latter, there are also some drawbacks. Since virtual machines are cheap and easy to create, users tend to create distinct virtual machines for different tasks. Users can branch new virtual machines based on old ones, snapshot machines or even rollback machines to a previous state. While these features provide great flexibility for users, they pose an enormous security risk for providers. A machine rollback, for example, could reveal an already fixed security vulnerability [3]. What makes the task of keeping the software stack up-to-date even more time-consuming is the the increasing number of virtual machines, a phenomenon called virtual machine sprawl [4].

More problems arise because some of the virtual machines are likely to be dormant (not running) at some point in time. These virtual machines cannot be easily kept up-to-date, because typically this would require the virtual machines to be started, updated and shut down again, which is not only time-consuming, but may also be a tedious process. Different solutions [4-6] have been developed to solve the maintenance problem of (dormant) virtual machines. While these solutions can be used to update dormant machines, they suffer from a potential compatibility problem. They "forcibly" install updates, either by changing an underlying layer [5] or by replacing files [4,6], and there is no guarantee that the updates can be safely applied and that they are compatible to the software stack and the configuration of all affected virtual machines.

Moreover, all of these solutions lack the ability to properly identify which applications are truly outdated. Since

*Correspondence: rschwarzkopf@mathematik.uni-marburg.de
Department of Mathematics and Computer Science, University of Marburg, Hans-Meerwein-Str. 3, D-35032 Marburg, Germany

this information is a prerequisite for the actual update process, it is a crucial step in the process of keeping (dormant) virtual machines in a Cloud or a virtualized Grid computing environment up-to-date. While such a check is easy to perform for running virtual machines, because of the commonly used package management systems on Linux platforms and automatic update facilities on Windows platforms, it is again a problem with dormant virtual machines. Even if virtual machines are kept up to date, the installed software might still contain design flaws or software vulnerabilities not fixed with the latest update. Thus, only checking for updates alone is not sufficient. Furthermore, machines used in a public IaaS environment are subject to external attacks, i.e., they might be a selected or random target chosen by scripts. Therefore, it is indispensable to continuously analyze the used virtual machines and take proactive countermeasures such as patching the revealed flaws.

In this article a combined approach that checks for software updates and scans virtual machines for known security vulnerabilities is presented. The first component called *Update Checker* is proposed to check a potentially huge number of Linux-based virtual machines for the necessity of updates. Since the Update Checker copies the information about installed packages to a central database, the check can be executed on the central instance without booting the virtual machine beforehand and shutting it down afterwards, which is the most time-consuming part of checking for updates of a virtual machine. Thus, the check is independent of the status of the virtual machine (running or dormant). Both apt/dpkg and yum/rpm are supported and therefore all major Linux distributions. The solution allows easy checking of all registered virtual machines, returning either the number of available updates or details about each of the available updates. The second component called *Online Penetration Suite* (OPS) is proposed to perform periodic or pre-rollout online-scanning of virtual machines. While periodic scans can be done in idle times, pre-rollout scans are executed before machines go live, delaying the start of a machine but using the latest version of the scanners for up-to-date results. Virtual machines are scanned for software vulnerabilities, using a combination of well-known security products.

Furthermore, the proposed solutions can inform the owners about relevant findings via e-mail. Using an API, other management tools can utilize the results. To leverage existing software, our proposal is based on the Xen Grid Engine (XGE) [2] and the Image Creation Station (ICS) [7] introduced in previous publications. The XGE is a software tool to create either virtualized Grid environments on-demand or to act as a Cloud IaaS middleware. The ICS offers an easy way for users to create, maintain and use virtual machines in the previously mentioned environments. An exemplary integration into the ICS, marking virtual machines that contain obsolete packages in virtual machine lists and providing details about available updates in detail views, and the XGE, preventing virtual machines containing obsolete packages from being started, is provided. The OPS scan process is triggered either by the ICS as a periodic maintenance operation or, if the additional overhead is acceptable, by the XGE as a pre-rollout check that might prevent a virtual machine from being started. As an alternative to preventing virtual machines from being started, those virtual machines can be started as usual and the owner is informed that his/her running machine is potentially unsafe. This can help administrators by giving them an overview of their dormant virtual machines, but also users without experience in the area of system maintenance (e.g. scientists that build custom virtual machines to execute their jobs), by making them aware of the problem.

The article is organized as follows. The next section presents the proposed design. Then, its implementation is discussed, followed by the presentation of experimental results. Afterwards, related work is discussed. The final section concludes the article and outlines areas for future research.

Design

The following sections present the design of the proposed approach. The first section outlines the Update Checker, a solution for checking for updates in virtual machines. The second section describes the Online Penetration Suite, an approach for online-scanning virtual machines for known software vulnerabilities.

Update checker

Since the primary goal of the Update Checker is detecting obsolete software in (dormant) virtual machines, the term virtual machine is used throughout this article. Nevertheless, the solution is applicable to physical machines as well.

The concept of the Update Checker is to build a central database that contains all the information required for the task of checking for updates. This includes the list of installed packages, including the exact version of the installed package as well as the list of repositories that are used for each virtual machine. This information has to be imported into the central database when the virtual machine is first registered, and updated after each change of the virtual machine, i.e., after the installation of new software or the update of already installed software.

Since the Update Checker is not targeted at a single Linux distribution (compared to, e.g., Landscape for Ubuntu [8]), at least the two prevalent software management solutions are supported: apt/dpkg, used for example

in Debian and Ubuntu, as well as yum/rpm, used for example in Red Hat and Fedora as well as SuSE. Both solutions use a specific package database format as well as a specific repository format. While apt/dpkg uses the same plaintext file format both as package database and as repository database, yum/rpm uses a Berkeley database as package database and an XML file as repository database. Nevertheless, this has no influence on the structure of the database used to store the required information, since both systems have the concept of distinct package names and a consistent versioning scheme in common.

The design of the solution is shown in Figure 1. There are specific importers for the package databases and for the repository databases of the different software management solutions. This makes the Update Checker easily adaptable to other software management solutions. Information about the installed packages of a virtual machine is stored in the Package DB. Metadata about the VM, i.e., the time stamp of the import, the repositories used, etc., is stored in the Metadata DB. Information about the available packages on the different repositories is stored in the Repository Cache. When invoked, the Update Checker takes the information from these databases and the Repository Cache and matches installed and available packages to detect obsolete software and stores the results in the Result Cache.

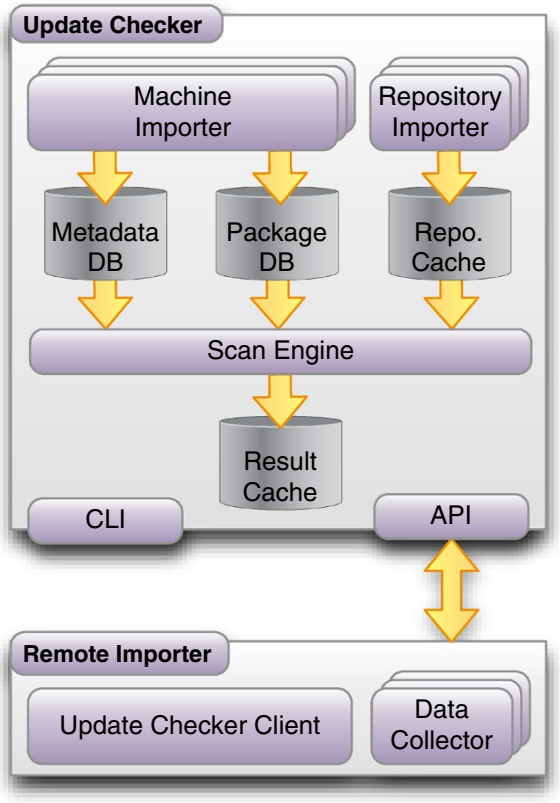

Figure 1 Update Checker architecture. The architecture of the Update Checker.

When a query for the state of one or more virtual machines is issued, the Update Checker first checks to see if the result of that query is already available in the Result Cache and returns the cached result if it is not obsolete. Cached results are considered obsolete after a configurable amount of time, depending on factors such as the frequency of updates or the need for security. Otherwise, it checks if the package lists of all repositories assigned to the virtual machine are available in the Repository Cache and not obsolete, i.e., the configured validity period has not yet expired. If this is not the case, the package lists are downloaded from the software vendor's repository, parsed and stored in the Repository Cache for future use. When using the Repository Cache instead of the real repositories, there is the chance that the Update Checker fails to identify an outdated package. Nevertheless, the Repository Cache is very useful for checking many virtual machines and by using a small validity period, the risk can be minimized. Finally, the actual check of the virtual machine is started, comparing the version of each installed package with the version available at the repository. Information about outdated packages is then stored in the Result Cache, so that subsequent queries regarding the same virtual machine can be answered faster.

To help the user to judge whether the identified outdated software poses a risk to the virtual machine, the Update Checker infers information about the priority of an update. Unfortunately, there is no common way to do this for multiple distributions. As a first approach, the source repository of the updated packages is evaluated, since distributions like Debian or Ubuntu use special repositories for security updates. The source of an update can therefore be used as a hint of its significance.

The Update Checker allows to query for the number of available updates for a single or multiple virtual machines as well as for details about the outdated packages and available updates for a single virtual machine. The former query allows a good estimation of the state of the virtual machine, where zero means the virtual machine is up to date, while a number greater than zero means that there are updates available. If significance information is available, individual numbers for each level of significance as well as the sum of the numbers are returned. This can either be used in situations where an overview over a number of virtual machines is required, e.g. , a list of virtual machines in a management tool like the ICS, or as a status check for a specific virtual machine, e.g., before it is started by the XGE.

Since the availability of updates itself allows no judgment about the threat resulting from the outdated packages, even when significance information is available, the

latter query allows a detailed examination of the status of a virtual machine, by giving a list of outdated packages. This allows the user of the virtual machine to do a threat analysis based on the outdated packages and decide whether immediate action is required or not. The described functionality is used as an example of the integration of the Update Checker with other components. The complete solution is shown in Figure 2.

Two different interfaces are provided by the Update Checker: a command line interface (CLI) and an API for use by other software. The former can be used, when an administrator manually wants to execute an update check or register a virtual machine. The latter is provided for other tools like the ICS or XGE, allowing them to easily access the status information. This interface is provided using the language-independent protocol XML-RPC [9], to be available to tools written in any language.

The Update Checker can also be configured to run the checks at regular intervals, e.g., daily or weekly. This speeds up queries by other tools, because the information is already available. Users can be informed about obsolete software in their virtual machines via email. Additionally, administrators can also be informed about all virtual machines using obsolete software, to get an overview of the security of all virtual machines running on their infrastructure.

To ease the registration of virtual machines, the remote importer is provided (see Figure 1). It uses software management solution specific Data Collectors to gather the information required for the Update Checker, sends it to the machine the Update Checker is running on and triggers the registration process.

It might seem cumbersome to manually re-register virtual machines after every change, but with the remote importer it is merely a single command. Furthermore, it can be easily automated when software for management and maintenance of virtual machines is used.

Online penetration suite

This section presents the Online Penetration Suite (OPS) to scan an arbitrary number of virtual machines for security vulnerabilities utilizing multiple security scanners. The OPS combines and interprets the different results and generates a machine-readable and a human-readable report. Furthermore, the OPS is able to manage (start, stop, migrate, etc.) virtual machines if necessary. This allows automatic testing of virtual machines in a virtualized infrastructure to detect known security vulnerabilities. Once the vulnerabilities are known, the administrators and users can fix them to protect their systems with respect to unwanted attacks.

Architecture

The OPS is divided into two parts: the *logic* part, containing the flow control and the report generator, and the *backend* part, operating the registered vulnerability scanners and the virtual machines. The architecture of the OPS is shown in Figure 3, containing two adapters for OpenVAS [10] and Nessus [11].

The *OPS Logic* module controls the processes of the OPS. It configures the security scanners, boots the virtual machines to test (if required) and starts the actual scans. Since the vulnerability scanners are basically third-party products with individual characteristics and modes of operation, they are abstracted by *Adapters* that hide the differences and provide an unified interface to start and monitor the vulnerability scanners. They allow the OPS not only to start the actual scans, but also to watch the scanners during the execution to detect any failures and react accordingly.

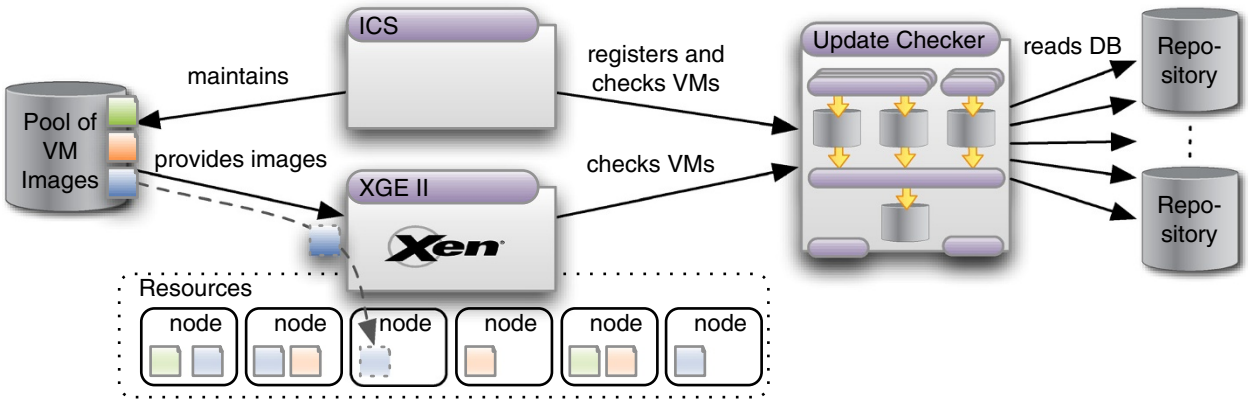

Figure 2 Usage scenario. The architecture of a complete system for virtualized Grid computing, consisting of the ICS, the XGE and the Update Checker. The figure shows the XGE deploying and starting a virtual machine, after the Update Checker has attested the virtual machine as being up-to-date.

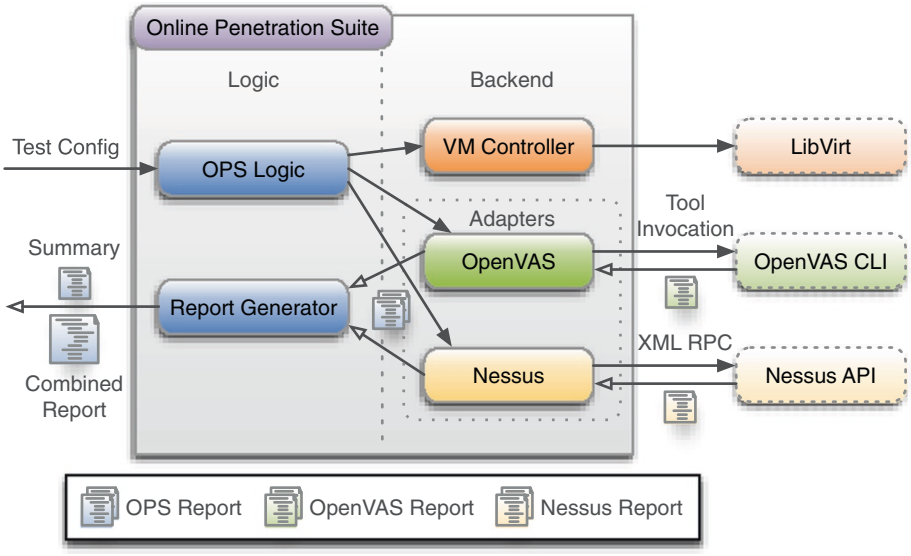

Figure 3 Online Penetration Suite architecture. The architecture of the Online Penetration Suite.

For a scan, the OPS needs two input parameters: the names of the target virtual machines and the name(s) of one or more vulnerability scanners. If no scanners are provided, the OPS chooses all scanners by default. A name uniquely identifies a virtual machine and allows the OPS to obtain further information like the IP and MAC address, path to the disk image(s), etc.

The *Report Generator* module collects the reports from the different scanners and generates the final result: a summary, containing the number of detected vulnerabilities categorized by a risk factor, and a combined report, containing the results from the security scanners in a unified format. To enable the Report Generator to analyze and understand the reports, the adapters have to convert the reports from the native format of the scanner to the unified OPS format.

The *backend* part of OPS consists of adapters to the required tools and libraries. It provides a module to control virtual machines using the libvirt [12] library as well as the vulnerability scanner adapters. Currently, the OPS supports two different scanners: OpenVAS [10] and Nessus [11], both well-known and established security-products.

Running vulnerability scans

OpenVAS is built as a client-server-architecture. The server is divided into three parts: administrator, manager and scanner. All clients communicate with either the manager or the administrator that both call the scanner. The OPS uses `omp`, a tool from the OpenVAS command line client for interaction. In order to guarantee a seamless scan, some of the countless options of OpenVAS are preset by the OpenVAS adapter module using a configuration file. This prevents the user from choosing wrong options that could possibly lead to false results. Nevertheless, by modifying the adapter configuration file it is possible for an administrator to enable/disable tests or set/unset options.

Nessus, being the ancestor of OpenVAS, is also built as a client-server-architecture. To control it, an XML-RPC interface is used. Nessus needs a number of parameters to start the scan process: the IP address of the server, authentication data and a scan configuration. Similar to the OpenVAS adapter, the Nessus adapter module presets a number of options to guarantee a seamless scan process.

Structure of the reports

The combined report generated by the *Report Generator* is hierarchically divided into several parts. It starts with a summary of all reports and contains the results of each scanner structured by each tested virtual machine. Finally, the machine-specific report contains the vulnerabilities of this host. This includes a detailed description of the vulnerability, the severity level and if applicable, port number and transport protocol. The following paragraph shows an excerpt of a report:

```
<vulnerability>
<title>Microsoft Outlook SMB
Attachment
Remote Code Execution Vulnerability
(978212)</title>
<port>general/tcp</port>
<risk_factor>HIGH</risk_factor>
<description>
```

```
Overview: This host has critical
 security
update missing according to Microsoft
Bulletin MS10-045.
[...]
CVE : CVE-2010-0266
BID : 41446
</description>
</vulnerability>
```

Implementation

In this section, the implementation of the Update Checker and the OPS is outlined.

Update checker

This section describes important parts of the implementation of the Update Checker, working from the top to the bottom of Figure 1. First, the machine and repository importers and their sources of information are described using the Debian Package Manager (dpkg) and the Advanced Packaging Tool (apt) of Debian and its derivates as an example. Afterwards, the internal databases and caches, the Scan Engine and the different interfaces are described. This section is concluded with details about the remote importer and the integration with other components. Further implementation details can be found in a previously published paper [13] of the authors. The implementation of the Update Checker has been done using the Ruby programming language.

Machine importer

A machine importer is responsible for importing the list of installed packages and enabled repositories of a machine into the Package DB and Metadata DB, respectively. This information is collected from the package database, that keeps track of installed packages, versions, files belonging to each package, etc., and from the configuration files of the software management solution.

The package database of dpkg is stored in `/var/lib/dpkg` and consists of several text files, of which the file `status` is of particular interest, because it contains the metadata for each package that has ever been installed on the system. For each package it contains about a dozen key-value-pairs, of which three are required to extract the information: `Package`, which contains the package name, `Status`, which contains the state of the package (installed or not installed), and `Version`, which contains the exact version of the package. The following snippet shows the parsed parts of a dpkg package management database entry:

```
Package: openssh-server
Status: install ok installed
Version: 1:5.1p1-5
```

The repositories used by apt are stored in `/etc/apt/sources.list`. This file contains multiple definitions, one per line, in the following format:

```
deb ROOT ARCHIVE COMPONENT
(COMPONENT...)
```

The meaning of these fields is explained in the next section. They are required to build the URL for the actual repository that is required to load the list of available packages.

Repository importers

A repository importer is responsible for importing the list of available packages in a repository into the Repository Cache. This information is gathered from the repository database of the software management solution. The repository database of an apt repository can be found using the following URL that is built using information from the fields in the config file.

```
ROOT/dists/ARCHIVE/COMPONENT/ ⤸
   binary-ARCHITECTURE/Packages.TYPE
```

The *ROOT* field contains the root URL of the repository or mirror. The next two fields partition the repository: Debian and Ubuntu use *ARCHIVE* to divide the repository by the release (e.g. *stable* or *testing*) and *COMPONENT* to divide by license type and level of support (e.g. *main*, *contrib* or *non-free*). The last two fields specify the system architecture and the compression format of the repository database.

The repository database uses the same format as the package database of dpkg. Thus, parsing can be done using the same technique.

Internal databases and caches

The Package DB is used to store a name-version-pair for each installed package on every machine. Its counterpart is the Repository Cache that stores a name-version-pair for each available package on every repository. Initially, it was planned to store this information in a database. Unfortunately, importing a virtual machine or updating the list of available packages of a repository was very slow using this technique. As a faster alternative, a hash encoded in JSON [14] was chosen, written to an individual file per virtual machine or repository, respectively. This was faster by a factor of more than 23 when measured for the import of two Debian repositories (2.16 sec using the hash versus 50.02 sec using the database). The equivalent to the database snippets shown above in the internal format is the following:

```
...,"openssh-server":"1:5.1p1-5",...
```

Information about outdated packages is stored in the Result Cache. It stores name-old version-new version-priority-quadruplets in a JSON encoded list, written to an individual file per virtual machine.

The Metadata DB stores a list of all registered virtual machines and repositories as well as the mapping between them. Furthermore, it stores the names of all files that build the Package DB, Repository Cache and Result Cache, together with an expiration date for each file of the two caches.

Scan engine

In this component, the actual identification of outdated packages takes place. Whenever a query for available updates of a virtual machine is submitted and there is no current result in the result cache, the Update Checker first determines the required repositories using the Metadata DB. If the repository cache does not contain current versions of the required repositories, a repository importer is used to update the cache. Afterwards, the list of installed packages is retrieved from the Package DB and the version of each package is compared with the version of that package stored in the repository cache. Outdated packages are stored in the result cache with installed and available version, so that subsequent queries can be handled faster. Finally, the number of outdated packages or the list of outdated packages is returned to the issuer of the query.

One particular problem discovered during the implementation of the Update Checker is the format of the version numbers used by the different package management systems or distributions, respectively. While most of the distributions use versions composed of the fields epoch, version and release, there are subtle differences between the distributions, e.g., separators, format of the release field, etc. Even the versionomy gem, a Ruby library especially designed for version comparisons, failed to correctly compare Debian version numbers.

One possibility is the use of the dpgk binary which provides an option to compare versions. This is very slow, since each comparison requires forking a new process. A Ruby library named dpkg-ruby implements version comparison using a native library. An old version of this library contains a Ruby-only version of the version comparison. Although slower, this solution is preferred to be independent of native libraries. By using an additional string comparison beforehand, performance losses can be cut down. Except for some minor tweaks, this version comparison library worked with all version numbers that were encountered in Debian and Fedora.

A daemon is used to provide some automation. All virtual machines can be checked for updates automatically at regular intervals. As described above, this frequently updates the cached repository databases and caches the results for all virtual machines. Queries using the API or the command line interface can then be served from the cache, requiring almost no time (only a file has to be read). The daemon also allows to notify users by email about outdated packages in their virtual machines. Additionally, the daemon can be configured to send emails about the status of all virtual machines to administrators.

Online penetration suite

The Online Penetration Suite is implemented in the Java programming language. Virtual machines are controlled using the Java binding of the libvirt library, the Nessus scanner is invoked using the Apache XML-RPC library and the reports of the vulnerability scanners are processed and converted using the Java API for XML Processing (JAXP).

Depending on the test configuration specified via the command line, the OPS frontend selects the required vulnerability scanners, starts their server components (if required), boots the virtual machines to scan (if they are not running already) and finally initiates and monitors the actual scan processes. All of these operations are hidden behind an interface that is implemented by the adapters, making the OPS easily extensible with new scanners. Since the report generation process is based entirely on reports in the unified OPS format, no vulnerability scanner dependent code is required for this step in the frontend.

The adapters use different techniques to control and monitor the actual vulnerability scanners. OpenVAS provides a command line interface, so its adapter needs to create a test configuration in the form of an XML file and pass it as an argument to the `omp` binary. Monitoring of OpenVAS requires analyzing the output of its client. For Nessus, the provided XMLRPC API is used. It contains methods to start and monitor the actual scan process. Both adapters contain code to convert the proprietary report formats into the unified OPS format.

Experimental results

The following section presents an evaluation of the presented components.

Update checker

Measurements have been conducted to evaluate the Update Checker on an Intel Xeon E5220 machine with 1 GB memory. The first measurement is a local measurement testing all components of the Update Checker, i.e., machine import, repository import and update checking. Three Debian and three Fedora virtual machines have been used in this test, with varying numbers of installed packages and enabled repositories. Each test has been executed 20 times and average values have been calculated. The results are shown in Table 1.

In the first part of this evaluation, the different machine importers were tested. All required files were copied to the

Table 1 Update Checker component benchmark

Distribution	Installed packages	Machine import	Repository import	Update import
Debian	563	0.04 secs	2.39 secs	0.44 secs
Debian	867	0.06 secs	2.80 secs	0.44 secs
Debian	1493	0.07 secs	2.68 secs	0.78 secs
Fedora	591	0.03 secs	13.59 secs	0.38 secs
Fedora	1063	0.04 secs	14.84 secs	1.00 secs
Fedora	2159	0.05 secs	15.38 secs	2.10 secs

Benchmark of all individual components of the Update Checker.

machine the test was executed on prior to the evaluation, thus no network communication is involved. Furthermore, before the measurement `rpm -qa` was executed on the source machine to generate a list of installed packages including their version. This is required to work around incompatibilities (i.e., the rpm binary on Debian squeeze could not read the rpm database of a Fedora 15 installation).

The growing import times can be explained with the growing number of installed packages that must be parsed.

The second part of the test measured the time required to download and parse all repository databases for the virtual machines (each machine had between 2 and 4 repositories configured) without using the repository cache. The times measured are thus artificial and are only of little relevance for actual usage, but allow evaluating the repository import and update checking. While the times for the Debian machines are quite stable, the increase of the time for Fedora is caused by the number of repositories used (2, 3 and 4, respectively). The very bad performance of the Fedora repository import is caused by the use of XML in the repository database.

The last part of the test evaluates the algorithm that actually checks for updates. Again, the increase in the times is caused by the growing number of packages. The reason for the worse results for Fedora are probably the longer and more complex version numbers used in Fedora, making the comparison harder and more time-consuming.

The measured values are promising. Checking for updates is a very fast process with the Update Checker. Because of the individual files used for the Package DB and Repository Cache, we do not expect performance degradation when the number of virtual machines increases. The relatively long time required for importing yum repositories is compensated by the repository cache, that results in every repository being downloaded and parsed only once during the configurable validity period of the cache.

To evaluate the influence of the repository cache, another measurement has been conducted that represents a more realistic scenario: checking all imported virtual machines for updates. The six machines from the last measurement were checked at once, taking advantage of the repository cache. The experiment was repeated 20 times and the average times are shown in Figure 4. The results indicate that the repository cache is very effective in cutting down the time required to check multiple virtual machines for updates.

To evaluate the scalability (and applicability for physical machines) of the Update Checker, 115 physical nodes from our compute cluster were imported. All machines were checked at once using the repository cache. The experiment was repeated 20 times and the time required to check all virtual machines was calculated. The results shown in Figure 5 provide evidence for the scalability of the Update Checker. The average check time was 34.53 seconds for all 115 machines, that is 0.30 seconds per machine.

Another measurement was conducted to evaluate the import time of the virtual machines, when the remote importer is used. This involves gathering all required files, executing `rpm -qa` in the case of rpm based distributions, sending everything to the Update Checker and starting the import process. For each virtual machine, 10 imports were executed. The results are shown in Figure 6.

As expected, the amount of time the import process requires grows with the number of packages in the database. Generally, the import process is faster for apt/dpgk based virtual machines than for yum/rpm based virtual machines. The source of this problem seems to be the use of the rpm binary to extract the information from the database.

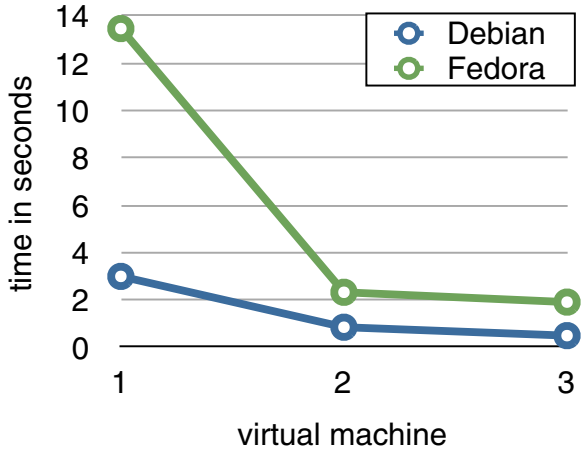

Figure 4 Update checking performance. Benchmark of the update checking process for multiple virtual machines using the repository cache.

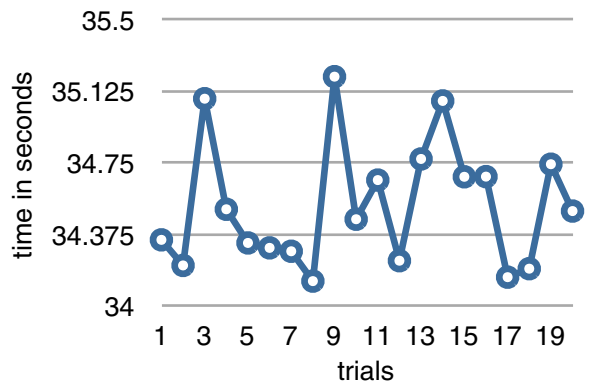

Figure 5 Results of the scalability evaluation. Benchmark of the update checking process for 115 machines using the repository cache.

Online penetration suite

The following section presents measurements related to the OPS. All tested systems are Xen domainU virtual machines running Debian Squeeze and located on Pentium IV systems with 1 GB memory. The OPS node is an Intel Xeon E5220 machine and 1 GB memory. All systems are interconnected with switched fast Ethernet.

The first experiment measures the total runtime of the OPS depending on the number of virtual machines. Figure 7 shows the results. The OPS used both vulnerability scanners in parallel while the number of target virtual machines was increased with every run. To get a robust mean, 100 trials were performed. Testing one virtual machine took 684 seconds on average, testing two machines took 859 seconds, testing three machines 1056 seconds, and it took 1279 seconds to test all four machines. Obviously, the measurement reveals that the runtime increases linearly with the number of tested systems. Furthermore, it reveals that it is more efficient to test multiple targets in parallel instead of scanning one after another.

In order to test the efficiency of the OPS, multiple tests against virtual machines running different versions of the Debian operating systems were conducted. The unpatched release version of Debian *Etch* (released April 2007), *Lenny* (released February 2009), *Squeeze* (released February 2011) and *Wheezy* (current unstable version) were used. The results of the tests are shown in Table 2.

The OPS successfully revealed a number of security vulnerabilities in all tested versions, including two high-risk flaws in each version. Debian *Etch* is the oldest release and contains the lowest number of vulnerabilities because it contains less features (in terms of installed services) than all other versions. Other flaws are related to the installed kernel version. The flaws appeared with newer kernel versions and thus, only in newer Debian versions.

Related work

The Cloud computing risk report written by ENISA [15] mentions the failure of customer hardening procedures as one of the research problems needed to be solved. Customers failing to secure the computing environment may pose a vulnerability to the Cloud infrastructure.

Automation of system administration, including system administration and updating systems is one of the relevant research topics mentioned in the Expert Group Report [16] created by the European Commission.

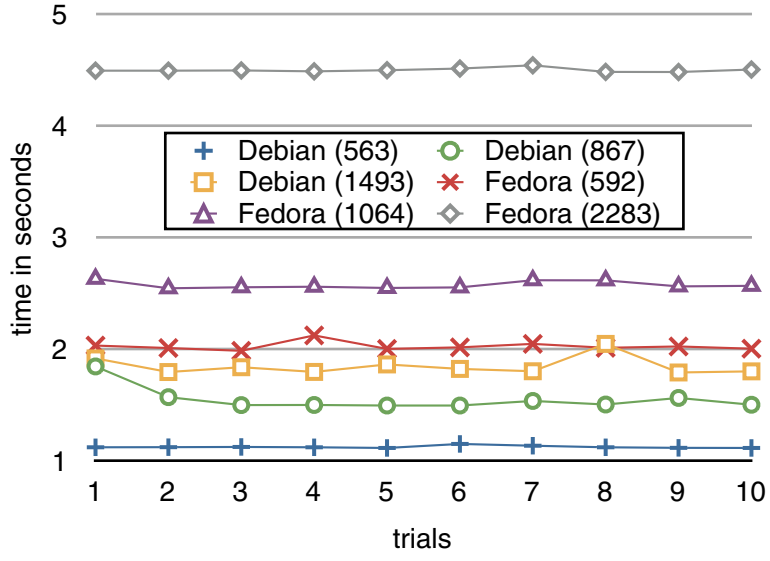

Figure 6 Remote import performance. Total time required to import each of the 6 virtual machines using the remote importer.

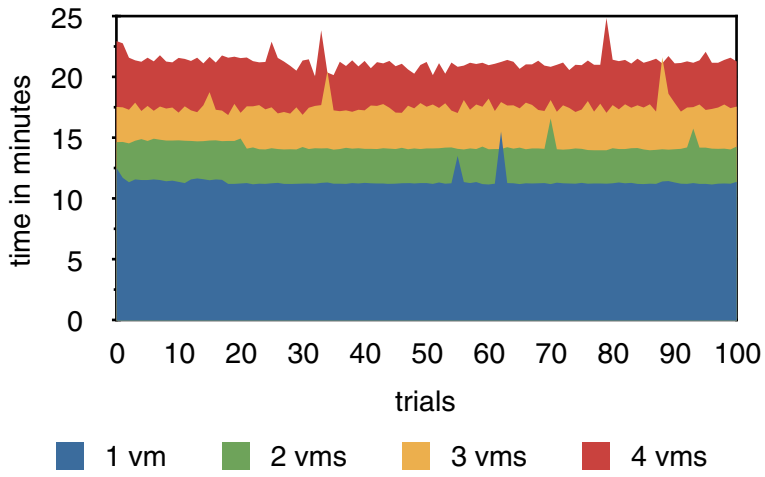

Figure 7 Results of the OPS evaluation. Benchmark of the runtime of the OPS depending on the number of tested systems.

An image management system, called Mirage, is presented by Wei et al. [6]. Mirage addresses security concerns of a virtual machine image publisher, customer and administrator. To reduce the publisher's risk, an access control framework regulates the sharing of virtual machines images. Image filters remove unwanted information (e.g., logs, sensitive information, etc) from images prior to publishing. The authors also present a mechanism to update dormant images and apply security updates. While Mirage offers a complete solution for virtual disk image maintenance, it lacks the features presented in this article. Mirage cannot show whether the packages in a system are outdated and work with multiple package management systems.

Based on Mirage, Reimer et al. [4] present the Mirage image format (MIF), a new storage format for virtual machine disk images. MIF solves the problem of *virtual machine image sprawl*, i.e., the complexity of maintaining disk image content that changes continuously due to cloning or snapshotting. MIF stores the disk image content in a central repository and supports searching, installing and updating applications in all images. By using a special storage device, disk images share common blocks and thus take up only a fraction of the actual disk space. Using MIF it is also possible to update packages on a system although the update procedure is quite complex. At first, it is quite unclear how the system determines whether there is a need for an update. Furthermore, the system needs a modified version of dpkg, thus, it is not usable with off-the-shelf installations or other package management solutions. The authors state that "the optimized Dpkg does not support some of Dpkg's features".

Table 2 OPS results for Debian

Distribution	Risk level *none*	Risk level *Low*	Risk level *medium*	Risk level *high*
Debian Etch	14	2	0	2
Debian Lenny	43	2	3	2
Debian Squeeze	44	2	3	2
Debian Wheezy	43	2	3	2

Number of security vulnerabilities the OPS detected in different versions of Debian Linux.

A system for unscheduled system updates, called AutoPod, was presented by Potter et al. [17]. AutoPod is based on system call interposition and the chroot utility and is able to create file system namespaces, called pods. Every process in a pod can be offline-migrated to another physical machine by using a checkpoint mechanism. Unfortunately, AutoPod is bound to Debian Linux and cannot be used with other package managers. Furthermore, it also updates a system automatically, which could lead to problems in case of an incomplete update. In contrast to the presented solution, AutoPod is based on chroot, which is known for having several major security flaws in the past.

Sapuntzakis et al. [18] developed a utility, called the Collective, which assigns virtual appliances to hardware dynamically and automatically. By keeping software up to date, their approach prevents security break-ins due to fixed vulnerabilities. While their approach allow updating whole virtual machine appliances, it does not allow the update of certain packages within the appliance. Furthermore, it is not possible to determine whether certain packages are outdated.

Layered virtual machines [5] can be used to solve the maintenance problem of dormant virtual machines. These machines are split up in different layers, such as a common base layer, containing a base system with some commonly required libraries and tools, an user layer containing

specific applications required by the user and potentially other layers. Besides benefits when it comes to storage and transfer of those virtual machines, considering shared layers that need to be stored and transferred only once and reused by many virtual machines, this architecture also helps with the problem of keeping machines up-to-date. Because a base layer is shared by many virtual machines, updating the base layer will affect all virtual machines built on top. Although not the complete software stack is affected by those updates, some of the most important parts of the system (e.g., the SSH libraries, which were affected by a serious bug in the Debian implementation back in 2008 [19]) can be fixed this way.

Canonical, the company behind Ubuntu Linux, offers a commercial product called Landscape [8]. Landscape can be used to manage Ubuntu (virtual) machines, including package management and monitoring. While Landscape is able to detect and update outdated applications within virtual machines, it can only handle the Debian package format and is not able to update dormant machines. However, Landscape can update outdated machines once they are live the next time.

SAVEly, a tool to check Amazon Machine Images (AMIs) for vulnerabilities was presented by Bleikertz et al. [20]. The authors construct an attack graph based on the security polices used in EC2. These policies are used to group machines while restricting the communication between them. Based on the graph, the authors use the OpenVAS scanner to check the AMI for remote vulnerabilities. Their approach is tightly coupled to Amazon's EC2 and cannot be used with other IaaS implementations or in virtualized Grid environments.

Yoon and Sim [21] present an automated network vulnerability assessment framework. It uses a combination of a scan manager, message relay server and scanners to check the hosts in a network for vulnerabilities. Their approach uses similar techniques as the ones presented, but it lacks the ability to work in a Cloud computing environment. It is neither able to control virtual machines, nor to instrument an IaaS solution like the XGE.

Conclusions

In this article, a new approach to increase the security of virtual machines in either virtualized Grid or Cloud computing environments has been presented. It is based on two components: a first component called Update Checker to identify outdated packages can check either running or dormant virtual machine images efficiently. It supports the two major Linux software management solutions, namely apt/dpkg and yum/rpm, and thus all major Linux distributions currently used in Grid or Cloud environments. Due to its flexible design, plugins for other software management solutions can be easily added. The use of multiple caches speeds up the check process, resulting in a time less than a second for a complete check of an average virtual machine. A second component called Online Penetration Suite scans virtual machines for software vulnerabilities using established security techniques. It can identify flaws in software components listening on the network. Both components are integrated into two already existing solutions (XGE and ICS) that leverage their capabilities to deny running too outdated machines or provide the user with the ability to update his or her machines.

There are several areas for future work. For example, the current implementation of the Update Checker only supports software installed using the package management systems of current Linux distributions. Nevertheless, there are cases where software is installed in other ways, either by compiling it manually or by installing software from binary packages that are not available in repositories. The idea of a generic framework with software specific plugins that can determine the installed version seems to be promising. Problems to solve are binaries without a version parameter and even more locating the software that was installed without using the package management system. Furthermore, the current approach to infer the significance of updates is a very basic approach. Comparing the list of outdated packages to the security advisories of the distribution, if available, seems to be promising. This would require distribution specific parsers for the advisories, since there is no unified advisory format, and manual configuration of the advisory sources for each distribution. The OPS currently controls two vulnerability scanners. In the future, it would be desirable to support a larger number of scanners.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
All authors contributed equally. All authors read and approved the final manuscript.

Acknowledgements
This work is partly supported by the German Ministry of Education and Research (BMBF) (D-Grid Initiative and HPC-Call) and the Hessian Ministry of Science and Art (HMWK).

Received: 30 January 2012 Accepted: 5 June 2012
Published: 17 July 2012

References

1. Armbrust M, Fox A, Griffith R, Joseph A (2009) Above the Clouds: A Berkeley View of Cloud Computing, Technical Report UCBEECS200928 53(UCB/EECS-2009-28). EECS Department University of California Berkeley
2. Smith M, Schmidt M, Fallenbeck N, Dörnemann T, Schridde C, Freisleben B (2009) Secure On-demand Grid Computing. J Future Generation Comput Syst 25(3): 315–325
3. Garfinkel T, Rosenblum M (2005) When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing. In 10th Workshop on Hot Topics in Operating Systems 121–126

4. Reimer D, Thomas A, Ammons G, Mummert T, Alpern B, Bala V (2008) Opening Black, Boxes: Using Semantic Information to Combat Virtual Machine Image Sprawl. In Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments 111–120. Seattle: ACM
5. Schwarzkopf R, Schmidt M, Fallenbeck N, Freisleben B (2009) Multi-Layered Virtual Machines for Security Updates in Grid Environments. In Proceedings of 35th Euromicro Conference on Internet Technologies, Quality of Service and Applications (ITQSA) 563–570. Patras: IEEE Press
6. Wei J, Zhang X, Ammons G, Bala V, Ning P (2009) Managing Security of Virtual Machine Images in a Cloud Environment. In Proceedings of the 2009 ACM Workshop on, Cloud Computing Security, CCSW '09 91–96. New York: ACM
7. Fallenbeck N, Schmidt M, Schwarzkopf R, Freisleben B (2010) Inter-Site Virtual Machine Image Transfer in Grids and Clouds. In Proceedings of the 2nd International ICST Conference on Cloud Computing (CloudComp 2010) 1–19. Barcelona: Springer, LNICST
8. Canonical Inc (2011) Ubuntu Advantage Landscape. http://www.canonical.com/enterprise-services/ubuntu-advantage/landscape
9. Winer D (2003) XML-RPC Specification. http://www.xml-rpc.com/spec
10. OpenVAS Developers (2012) The Open Vulnerability Assessment System (OpenVAS). http://www.openvas.org/
11. Tenable Network Security (2012) Nessus Security Scanner. http://www.nessus.org/products/nessus
12. Libvirt Developers (2012) Libvirt - The Virtualization API. http://libvirt.org/
13. Schwarzkopf R, Schmidt M, Strack C, Freisleben B (2011) Checking Running and Dormant Virtual Machines for the Necessity of Security Updates in Cloud Environments. In Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom) 239–246. Athens: IEEE Press
14. Crockford D (2006) The application/json Media Type for JavaScript Object Notation (JSON). http://www.ietf.org/rfc/rfc4627
15. ENISA European Network and Information Security Agency (2009) Cloud Computing Risk Assessment. http://www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-risk-assessment
16. Lillard TV, Garrison CP, Schiller CA, Steele J (2010) The Future of Cloud Computing. In Digital Forensics for Network, Internet, and Cloud Computing 319–339. Boston: Syngress
17. Potter S, Nieh J (2005) AutoPod: Unscheduled System Updates with Zero Data Loss. In Autonomic Computing, International Conference on 367–368
18. Sapuntzakis C, Brumley D, Chandra R, Zeldovich N, Chow J, Lam MS, Rosenblum M (2003) Virtual Appliances for Deploying and Maintaining Software. In Proceedings of the 17th USENIX Conference on System Administration 181–194. Berkeley: USENIX Association
19. Debian Security Advisory 1576-1 OpenSSH (2008) Predictable Random Number Generator. http://www.debian.org/security/2008/dsa-1576
20. Bleikertz S, Schunter M, Probst CW, Pendarakis D, Eriksson K (2010) Security Audits of Multi-tier Virtual Infrastructures in Public Infrastructure Clouds. In Proceedings of the 2010 ACM Workshop on Cloud Computing Security, CCSW '10 93–102. Chicago
21. Yoon J, Sim W (2007) Implementation of the, Automated Network Vulnerability Assessment Framework. In Proceedings of the 4th International Conference on Innovations in Information Technology 153–157. Dubai: IEEE

doi:10.1186/2192-113X-1-12
Cite this article as: Schwarzkopf *et al.*: **Increasing virtual machine security in cloud environments.** *Journal of Cloud Computing: Advances, Systems and Applications* 2012 **1**:12.