University of Amsterdam
System & Network Engineering

Trustworthiness of Cyber Infrastructure for e-Science

Authors:
Berry Hoekstra
bhoekstra@os3.nl

Niels Monen
nmonen@os3.nl

Coördinator:
Guido van 't Noordende
University of Amsterdam

August 6, 2010

Abstract

There are cases where the security of a Grid system has to be assured. The capabilities of the Grid systems to withstand attacks is an interesting aspect. Our research shows that it is possible to facilitate in assuring the robustness of the systems to withstand attacks. A way to quickly be able to get an overview of a system, is by creating a host description file that can be used to reason about the security of a system. We do this by matching the host description with a recent snapshot of the National Vulnerability Database, which contains publicly known vulnerabilities present in binaries and kernels. By focusing on the vulnerabilities, and specifically vulnerabilities that have an impact that involves privilege escalation, we can get a quick overview of the vulnerabilities of the system. A proof of concept shows that this can be done. The results of different scans show that even bleeding edge operating systems have vulnerabilities. The most critical vulnerabilities are located in the base kernel of the operating system and often have a high impact on the security of the system.

Contents

1 Introduction

1.1 Purpose

The “trustworthiness” of systems in a large-scale Grid system depends on many factors. A Grid consists of many different systems and users. Often, these systems are spread over multiple administrative domains. The data and computations running on these systems can be very privacy sensitive. If we take medical applications for example, a large degree of assurance is required that systems cannot be hacked. Hospitals are legally responsible for providing the highest possible degree of assurance that data concerning their patients remains well protected, which makes the above aspects very important in practice [1].

The purpose of this research project is to determine if there is a way to reason over the level of security of a remote system. Is it possible to determine how secure a system exactly is, and is there a way to determine how likely it is for a system to withstand attacks? If this is possible, this may help administrators to determine if systems are secure enough for certain jobs.

The purpose of the project is to answer the following questions:

> *Is it possible to reason about how secure a system is?*
> *Is it feasible to determine the robustness of a system to withstand attacks?*

1.2 Scope

In this paragraph, we define the scope of the project. It describes what we will do, and what we won’t do.

1.2.1 Do’s

The following points are within the scope of the project:

- Focus on different Linux distributions (Red Hat- and Debian-based).
- Research the best method to use vulnerability databases.
- Research possibilities to determine the robustness of a system.
- Create a proof of concept that can analyze a system based on version information.
- Focus on version information.

1.3 Approach

To get proper project results, we took the following approach:

- Research existing methods for security classification.
- Research methods for extracting vulnerabilities from vulnerability databases.
- Setup a test environment on Xen consisting of several different Linux distributions.
- Research binaries and libraries that are the most vulnerable.
- Discuss vulnerabilities that introduce privilege escalations.
- Create a proof of concept that can generate host descriptions.
- Perform tests on different distributions with different patch levels using the proof of concept.
- Analyze the test results.

2 Research

2.1 Research goal

The goal of the project is to study if it is possible to determine if there is a way to reason over the robustness of a system. We will analyze how we can construct machine-readable descriptions of systems in such a way, that it becomes possible to understand what the state of a system is. We are particularly interested in the system's ability to withstand attacks. To achieve this goal, we created a proof of concept that is able to do the above by performing different tasks. In the end, the proof of concept is able to provide us with more information about the system by providing a matched list of the vulnerable binaries on a system. An administrator may then able to reason over the system and determine if it is secure enough for certain applications. If some reported problems are unclear, it is possible to zoom in on certain points to acquire more information. All this can help us to reason about the security of a system.

2.2 Lab setup

In our lab environment, we have two servers available for testing. The processors powering the servers are equipped with support for virtualization. We configured both servers with an installation of the server edition of Ubuntu 9.04 and configured both installations with a Xen kernel. This enables us to run multiple virtual machines on top of the physical hardware. To do the testing and development, we created multiple virtual machines which are represented in the visualization below.

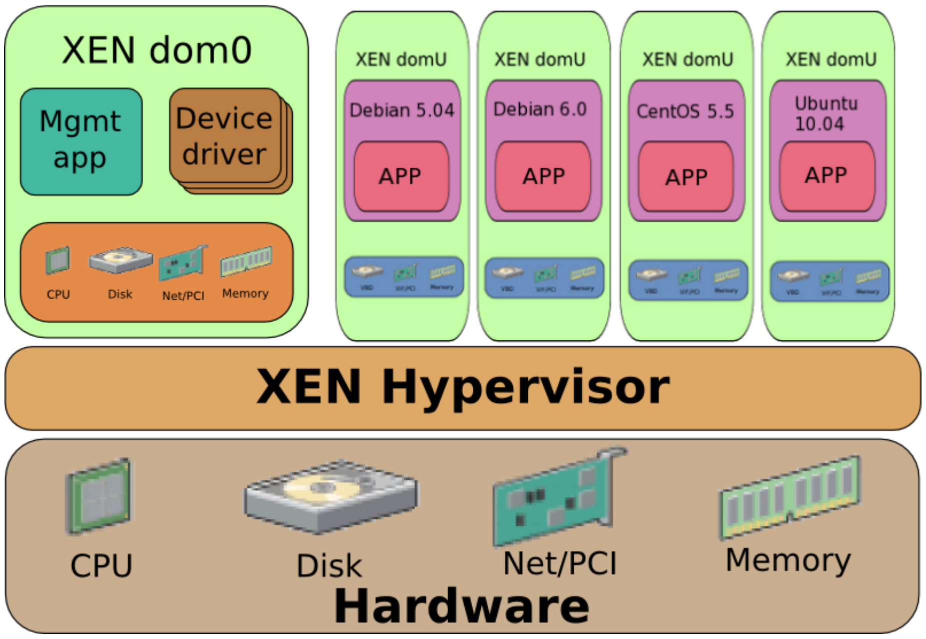

Figure 1: Xen setup [2].

The operating systems are updated to the latest patch level that is available from the repositories. We chose to use different Linux distributions that have different release dates, namely:

- Debian 5.04; kernel 2.6.26-2-686; May 12, 2010
- Debian 6.0; kernel 2.6.35-3; June 12, 2010
- CentOS 5.5; kernel 2.6.18-194.3.1.el5; May 13, 2010
- Ubuntu 10.04; kernel 2.6.32-22; June 3, 2010

This way we can compare the test results of the different distributions which have some time apart between the releases. More details on the installed operating systems can be found in the Appendix.

2.3 Related work

There isn't much related research specifically for our subject. The research that is related to this project is very scarce. However, there are several cases that are related to our project.

2.3.1 Vulnerability standardizations

Since we are looking into vulnerabilities, there might be some previous work available that has been done before and is related to this research. It appears there are standardizations available for making security measurable. The National Institute of Standards and Technology (NIST) has designed standards for this purpose [3]. For example, the Common Vulnerabilities and Exposures (CVE) standard, which is a dictionary of publicly known vulnerabilities and exposures [4]. The Common Platform Enumeration (CPE) standard is a standardized description of systems, platforms and packages [5].

2.3.2 Scoring systems

The Common Vulnerability Scoring System (CVSS) is a standard that is designed to assign a scoring to vulnerabilities. The scoring is designed to measure the impact and severity of a vulnerability. We can use this specification to measure a system. The score is calculated by using several variables [6], which we will discuss in Section 4.1.2. Interesting research done on the CVSS scores can be found in papers "An Analysis of CVSS Version 2 Vulnerability Scoring" [7] and "Vulnerability Scoring for Security Configuration Settings" [8] both written by Karen Scarfone and Peter Mell. In the second paper, Scarfone and Mell describe how the CVSS standard can be adapted for use with security configuration settings. They extracted parts of the CVSS specification that could be used to specify a new specification that uses security configuration issues specified in the Common Configuration Enumeration (CCE) database, instead of the vulnerabilities from the Common Vulnerabilities and Exposures(CVE) database. Because a lot of security issues are related to misconfigurations, the approach they take in the paper might be helpful with determining security scoring based on the configurations.

2.3.3 Central management systems

Organizations that have many systems running within their network are able to reduce their management costs by deploying a central management system for deploying patches to many systems.

We found multiple central management solutions for both Debian- and Redhat-based systems. The solution for managing Debian-based systems that we found was the most evolved is called ”Puppet” [9]. This product supports different Linux operating systems like Debian and Redhat.

A similar system that is specifically designed for Redhat-based systems is called “Spacewalk” [10]. This product is also recently supporting Debian-based distributions.

If such a central management system is deployed in a large infrastructure, it can manage all other systems by providing an automated update mechanism. If all systems are consistent and up-to-date, the risk of having vulnerable systems running within your network will likely decrease. We found both management systems don’t support integrated vulnerability checking, but instead make use of the default repositories of the operating vendor to update the systems. If a new version of a package is available it will raise an “alert” and provides an opportunity to update the connected systems. If no patches are available to obviate the vulnerabilities, no “alerts” will be raised. We can conclude that central management solutions might be useful to keep an environment patched, but found it not to be very useful to evaluate vulnerabilities of a system.

3 System classification

3.1 Determining the state of a system

To determine the security state of a system, we can look at the different aspects that may have influence here. Depending on these aspects, we can determine the security state, and classify the system as (in)secure.

3.1.1 Vulnerable binaries

Systems need binaries to perform certain jobs. Binaries can introduce problems or introduce security issues on a system and can cause, for example, privilege escalation that may result in normal users gaining administrative rights. We think that this is a good starting point to determine the security state of a system.

Binaries that cause security issues can introduce vulnerabilities in systems. Vulnerable binaries and libraries are in most cases reported. They end up in a database that contains all known vulnerabilities with their respective information.

Before a vulnerability is included in a vulnerability database, it first has to be discovered. In most cases, this is done by (security) researchers all over the world, but it is also common for a vulnerability to be discovered by incident.

When a vulnerability is found, the people that discovered it can provide a proof of concept on how to exploit the vulnerability. This proof of concept can help developers to solve the issue with the binary by creating a patch for it. The patched binary is labeled with a new version number so administrators can distinguish the vulnerable binary from the patched binary. A raise in the version number is not always the case, it is also common to include a patch level in the version notation.

3.1.2 Listing the binaries

By creating a list of the binaries and libraries that are locally installed on a system, and request the version number from each in the list, we can create a database that contains all binaries and libraries with their respective versions. We will call this the "host description". This host description can be compared to a vulnerability database and be useful for administrators. For the administrator to compare the list by hand to a vulnerability database can be very time-consuming, so it is desirable to have a compiled list at hand to compare with. Administrators can export the host description to easily evaluate the system remotely and sign off the list if the system is found to be secure enough for certain jobs.

Some binaries are more popular, security critical or scrutinized then others, so for these binaries, it is more likely that vulnerabilities are to be found. While analyzing vulnerability databases, we noticed that for some binaries, vulnerabilities occur more often. Examples of binaries that often have issues are:

- sudo
- openssl

While researching the behavior of the local binaries while requesting the version numbers, we found multiple binaries that don't support the common switches to request the version numbers that we discussed earlier. This can result in unusable output. Some examples of binaries where these issues occur are:

- crontab
- hexdump
- nslookup
- update-grub

If we take the "crontab" binary for example, it will output the following when requesting the version number:

Crontab output

```
crontab: invalid option -- -
crontab: usage error: unrecognized option
usage:        crontab [-u user] file
        crontab [-u user] { -e | -l | -r }
                (default operation is replace, per 1003.2)
        -e       (edit user's crontab)
        -l       (list user's crontab)
        -r       (delete user's crontab)
        -i       (prompt before deleting user's crontab)
```

This output is unusable because there are no version numbers outputted. However, there are other ways to still figure out the version number. We tried to request the version numbers with the tool "strings". The "strings" tool enables us to print all the ASCII text found in a binary. We found that by using this tool, it is not very likely to find a version number (see Section 5).

3.1.3 Vulnerability listing and matching

The next step is to take a vulnerability database and provide an easy way to compare the local list with the list of vulnerable binaries from the vulnerability database that can compromise the security of a system. If we are able to match the binaries on the local system that are vulnerable with an entry in the vulnerability database, we can create another list or database that contains all the vulnerable binaries on a system. A visual representation can be seen below:

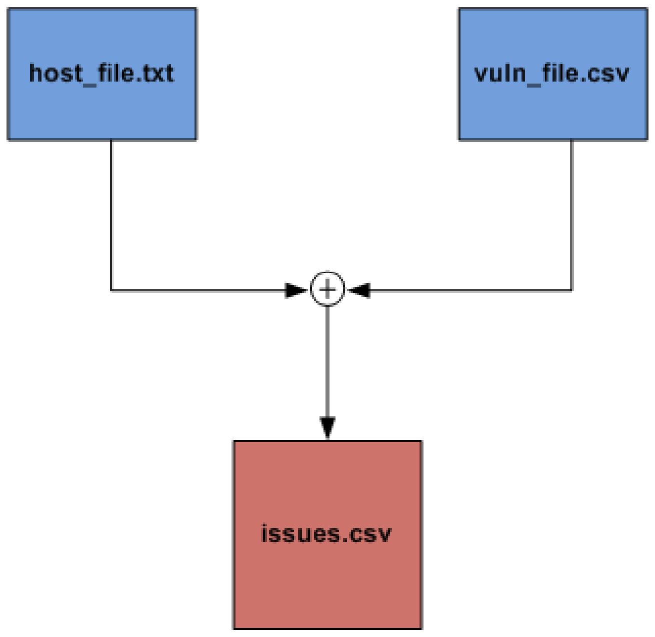

Figure 2: Evaluating a system.

With this knowledge, we may be able to classify the robustness of a system. In Section 5 we discuss a proof of concept with the above functionality. Limitations of describing classifications are discussed in section 5.5.

4 Vulnerability Databases

Vulnerability databases are the main source for information on older and more recent vulnerabilities. Online, there are different vulnerability databases to be found. Every database has different characteristics and is maintained by a different community. The databases comply to specific rules. Apart from these community-led databases, there are also commercial vulnerability databases available such as Secunia [11] and Vupen [12]. In this project, we will only focus on the free databases.

The databases can consist of multiple vulnerability feeds that are aggregated together. The advantage of an aggregated database is that it contains more vulnerability entries and it filters out the duplicate entries which wouldn't be the case if the databases are used separatly.

The databases export vulnerability information in different formats. The availability of the database in different formats can be convenient for different uses.

4.1 National Vulnerability Database

A leading vulnerability database is The National Vulnerability Database (NVD). The NVD is a database that contains vulnerability management data and information that is based on a range of standards.[13] [14] The database is sponsored by the department of Homeland Security of the U.S. government. It includes:

- Security checklists.
- Security related software flaws.
- Misconfigurations.
- Product names.
- Vulnerability impact scores.

We focus on the product names to determine the system binaries and the vulnerability impact scores to show the impact.

Also, the NVD has several databases aggregated into their own database. Some examples of those databases are:

- www.secunia.com
- www.vupen.com
- xforce.iss.net
- www.securityfocus.com

- www.osvdb.org

Because of this aggregation, NVD is the most complete database currently available. Another positive asset is the fact that you don't have to deal with duplicate vulnerabilities as these are filtered out by NVD. The NVD is providing its database in the XML format.

The NVD is using standards to describe the different vulnerability entries. These standards are all part of the "Making Security Measurable" initiative. This is a collections of standards created to make it easier to manage and measure security [15].

4.1.1 Common Vulnerabilities and Exposures

The Common Vulnerabilities and Exposures (CVE) standard provides a dictionary of publicly known information security vulnerabilities and exposures [4].

Each CVE Identifier includes the following [4] [16]:

- CVE Identifier number.
- Indication of "entry" or "candidate" status.
- Vulnerability description.
- References.

4.1.2 Common Vulnerability Scoring System

Another standard that is used by the NVD is called the Common Vulnerability Scoring System (CVSS). This standard provides a score to the security impact and severity of a vulnerability. The assigned score helps to determine how urgent a vulnerability is and what priority of response is needed. CVSS can help with this by communicating the base-, temporal- and environmental-properties of a vulnerability [6]. The base-properties are the characteristics of the vulnerability that are constant with time and across user environments. The temporal-properties are the characteristics that can change over time. Such characteristics are for example: the confirmation of the technical details, the remedation status, and the availability of exploit code. The environmental-properties are the characteristics of a vulnerability that are associated with a user's IT environment.

NIST [3] calculates the CVSS score with some variables. These variables are (from the CVSS website):

- Base-properties:

Related exploit range (AccessVector) This defines if the vulnerability can be exploited from the Local machine (lowest score), Adjacent network and Network (highest score).

Attack complexity (AccessComplexity) This defines how hard it is to exploit this vulnerability, with Low (highest score), Medium, High (lowest score).

Level of authentication needed (Authentication) This defines if an attacker has to be authenticated with the system. The options for this variable are: None (highest score), Single Instance and Multiple Instances (lowest score).

Confidentiality Impact (ConfImpact) This defines if confident data could be accessed by using this vulnerability. The possible values are: None (lowest score), Partial and Complete (highest score).

Integrity Impact (ItegImpact) This defines if data could be changed after using this vulnerability. The possible values are: None (lowest score), Partial and Complete (highest score).

Availability Impact (AvailImpact) This defines if the system could become unavailable by using this vulnerability. The possible values are again: None (lowest score), Partial and Complete (highest score).

- Temporal-properties:

Exploitability (E) This defines if there is public available easy-to-use exploit code. The possible values are: Unproven (lowest score), proof-of-concept, functional and high (highest score).

Remedation Level (RL) This defines if, and what kind of fix, is published. The values that are possible are: official fix (lowest score), temporary-fix, workaround, and unavailable (highest score).

Report Confidence (RC) This defines the degree of confidence in the existence of the vulnerability. The options are: Unconfirmed (lowest score), uncorroborated, and confirmed (highest score).

- Environmental-properties:

Collateral Damage Potential (CDP) This defines the potential for loss of life or physical assets through damage or theft of property or equipment. The possible values are: none (lowest score), low, low-medium, medium-high, high (highest score).

Target Distribution (TD) This defines the proportion of vulnerable systems. It is meant to give an approximation of the percentage of systems that could be affected. The values that are possible are: None (lowest score), low, medium, high (highest score).

Security Requirements (CR, IR, AR) This defines the importance of the affected IT asset, measured in terms of confidentiality, integrity and availability. The possible values are: Low (lowest score), medium, high (highest score).

The formula they use to calculate the score is as followed:

- Base:

Base formula

```
BaseScore = round_to_1_decimal(((0.6*Impact)+(0.4*Exploitability)-1.5)*f(Impact))
Impact = 10.41*(1-(1-ConfImpact)*(1-IntegImpact)*(1-AvailImpact))
Exploitability = 20* AccessVector*AccessComplexity*Authentication
f(impact)= 0 if Impact=0, 1.176 otherwise
AccessVector     = case AccessVector of
                        requires local access: 0.395
                        adjacent network accessible: 0.646
                        network accessible: 1.0
AccessComplexity = case AccessComplexity of
                        high: 0.35
                        medium: 0.61
                        low: 0.71
Authentication   = case Authentication of
                        requires multiple instances of authentication: 0.45
                        requires single instance of authentication: 0.56
                        requires no authentication: 0.704
ConfImpact       = case ConfidentialityImpact of
                        none:             0.0
                        partial:          0.275
                        complete:         0.660
IntegImpact      = case IntegrityImpact of
                        none:             0.0
                        partial:          0.275
                        complete:         0.660
AvailImpact      = case AvailabilityImpact of
                        none:             0.0
                        partial:          0.275
                        complete:         0.660
```

- Temporal:

```
Temporal formula
TemporalScore = round_to_1_decimal(BaseScore*Exploitability
*RemediationLevel*ReportConfidence)
Exploitability   = case Exploitability of
                        unproven:            0.85
                        proof-of-concept:    0.9
                        functional:          0.95
                        high:                1.00
                          not defined:          1.00
RemediationLevel = case RemediationLevel of
                        official-fix:        0.87
                        temporary-fix:       0.90
                        workaround:          0.95
                        unavailable:         1.00
                        not defined:         1.00
ReportConfidence = case ReportConfidence of
                        unconfirmed:         0.90
                        uncorroborated:      0.95
                        confirmed:           1.00
                        not defined:         1.00
```

- Environmental:

```
Environmental formula
EnvironmentalScore = round_to_1_decimal((AdjustedTemporal+
(10-AdjustedTemporal)*CollateralDamagePotential)*TargetDistribution)

AdjustedTemporal = TemporalScore recomputed with the BaseScores Impact sub-equation
replaced with the AdjustedImpact equation

AdjustedImpact = min(10,10.41*(1-(1-ConfImpact*ConfReq)*(1-IntegImpact*IntegReq)
                 *(1-AvailImpact*AvailReq)))

CollateralDamagePotential = case CollateralDamagePotential of
                                 none:            0
                                 low:             0.1
                                 low-medium:      0.3
                                 medium-high:     0.4
                                 high:            0.5
                                 not defined:     0
TargetDistribution        = case TargetDistribution of
                                 none:            0
                                 low:             0.25
                                 medium:          0.75
                                 high:            1.00
                                 not defined:     1.00
ConfReq                   = case ConfReq of
                                 low:             0.5
                                 medium:          1.0
                                 high:            1.51
                                 not defined:     1.0
IntegReq                  = case IntegReq of
                                 low:              0.5
                                 medium:           1.0
                                 high:             1.51
                                 not defined:      1.0
AvailReq                  = case AvailReq of
                                 low:              0.5
                                 medium:           1.0
                                 high:             1.51
                                 not defined:      1.0
```

If we use these formulas in an example (CVE-2002-0392, a vulnerabily of Apache), we get the following calculation:

Example CVE-2002-0392

```
----------------------------------------------------
BASE METRIC                 EVALUATION         SCORE
----------------------------------------------------
Access Vector               [Network]         (1.00)
Access Complexity           [Low]             (0.71)
Authentication              [None]            (0.704)
Confidentiality Impact      [None]            (0.00)
Integrity Impact            [None]            (0.00)
Availability Impact         [Complete]        (0.66)
----------------------------------------------------
BASE FORMULA                              BASE SCORE
----------------------------------------------------
Impact = 10.41*(1-(1)*(1)*(0.34)) == 6.9
Exploitability = 20*0.71*0.704*1 == 10.0
f(Impact) = 1.176
BaseScore = (0.6*6.9 + 0.4*10.0  1.5)*1.176
                                           == (7.8)
----------------------------------------------------

----------------------------------------------------
TEMPORAL METRIC             EVALUATION         SCORE
----------------------------------------------------
Exploitability              [Functional]      (0.95)
Remediation Level           [Official-Fix]    (0.87)
Report Confidence           [Confirmed]       (1.00)
----------------------------------------------------
TEMPORAL FORMULA                      TEMPORAL SCORE
----------------------------------------------------
round(7.8 * 0.95 * 0.87 * 1.00)            == (6.4)
----------------------------------------------------

----------------------------------------------------
ENVIRONMENTAL METRIC        EVALUATION         SCORE
----------------------------------------------------
Collateral Damage Potential [None - High]  {0 - 0.5}
Target Distribution         [None - High]  {0 - 1.0}
Confidentiality Req.        [Medium]           (1.0)
Integrity Req.              [Medium]           (1.0)
Availability Req.           [High]             (1.51)
----------------------------------------------------
ENVIRONMENTAL FORMULA            ENVIRONMENTAL SCORE
----------------------------------------------------
AdjustedImpact = min(10,10.41*(1-(1-0*1)*(1-0*1)
         *(1-0.66*1.51))                  == (10.0)
AdjustedBase =((0.6*10)+(0.4*10.0)1.5)*1.176
                                          == (10.0)
AdjustedTemporal == (10*0.95*0.87*1.0)     == (8.3)
EnvScore = round((8.3+(10-8.3)*{0-0.5})*{0-1})
                                   == (0.00 - 9.2)
----------------------------------------------------
```

The main shortcoming of this scoring system is that it is developed for regular systems, like workstations or normal servers. This means that CVSS is network and driver biased as the named systems are. The CVSS specification uses the CVE specification for vulnerabilities. CVE is not designed specifically for systems that are running in a Grid infrastructure. These systems are shielded from the network, and only load specific drivers that are needed for the base system, so no media or bluetooth drivers are needed for example. The CVSS score is not optimized for Grid systems.

Just like the research done by Scarfone and Mell [8] where they adjusted the CVSS score for system configurations (CCE), instead of vulnerabilities (CVE), we can adjust the CVSS specification to adapt to grid systems. By adjusting and optimizing, or leaving out some parameters, we can adjust the specification to Grid systems. For example, if we assign a lesser score to availability and a higher score to integrity and privacy, we have a score more focussed on the data, instead of the availability of the system. These are just small examples of what we think can be adjusted to the current specification.

We still think the current scoring specification is useful, because it can give a overview of how critical a vulnerability is. A now high scoring vulnerability won't be a low scoring vulnerability on Grid systems.

4.1.3 Common Platform Enumeration

The Common Platform Enumeration (CPE) standard provides a standardized descriptions of systems, platforms and packages [5].

The standards discussed above are helpful to create a generic vulnerability database like the NVD. By making use of such standards, it is easier to provide consistent database descriptions.

4.2 Open Source Vulnerability Database

The Open Source Vulnerability Database (OSVDB) is an independent and open source database created by the community. According to the OSVDB website, their goal is to provide accurate, detailed, current, and unbiased technical information [17].

The OSVDB also makes use of some standards such as the CVE standard to link vulnerabilities to a unique CVE ID. The database of OSVDB is provided in multiple different formats, like XML, CSV, SQLite and MySQL. This can facilitate developers if a certain type of format is desired.

4.3 NVD vs. OSVDB

As discussed earlier in this report, we decided to limit our research to only one vulnerability database. To determine what database to use, we compared the pro's and con's from both databases against each other.

As of the 22nd of June, the NVD database contains almost 43,000 vulnerabilities. This comes down to an average of 15 new vulnerabilities every day. In contrast, the OSVDB has 64,647 vulnerabilities, which makes OSVDB the database with the most vulnerabilities. However, in our eyes the NVD is more useful than the OSVDB because the standards that NVD uses for describing the database can be used to look for certain notations. Because standards are used, the notations are the same in all cases, so they can be easily distinguished by our proof of concept. Also, the NVD includes vulnerabilities from the OSVDB.

The differences between both databases are the formats they are provided in. We chose to use the NVD database, because of the machine-readable descriptions of the vulnerabilities, but NVD does not provide the database in a CSV format, but XML instead. We prefer CSV because it is easier to script with BASH, however, we can work around the "limitation" of the XML format by parsing it to CSV format first.

5 Proof of concept

In the previous chapters we discussed the methods that can be used to create a host description. To be able to test the findings, we created a proof of concept. The proof of concept is a prototype that can show us if it is feasible to actually create usable output that can be helpful to classify a system's security settings.

The download location to the proof of concept is included in the Appendix.

5.1 Functionality

Using the findings of our research, we know what functions the proof of concept should have. In the following list states the core functionality that should be included.

- Generate a list of local binary versions.
- Generate a list of known vulnerabilities from the NVD.
- Generate a list with vulnerable binaries for the scanned system by matching the lists.
- List all the binaries with amount of vulnerabilities and scores.

The proof of concept should check if local binaries are patched for certain known security issues. These security issues are fetched from the National Vulnerability Database and parsed to a CSV file for easy matching. If a binary's version matches with a version from the database that is known to be vulnerable, a hit is generated on the screen. The final result will be a list with all the vulnerable binaries on the system.

The proof of concept is also easily adaptable to other vulnerability databases, if they are parsed to the CSV format that is used by the prototype.

5.2 Version numbers

The first function of the proof of concept is able to determine the versions of the local binaries on the system. There are multiple possibilities to determine the version numbers.

The easiest way is to request the version numbers from the local package managers. For Red Hat-based operating systems, this can be done by giving the following command.

```
$ rpm -qa
```

For Debian-based systems, the following command has to be used.

```
$ dpkg --list
```

We can distinguish these operating systems with files that are only on one of them. For Red Hat-based operating systems, this is the file "/etc/redhat-release", and for Debian-based "/etc/debian_version".

This method can be useful if systems only have packages installed that are included in a repository. However, it is safe to assume that systems in a grid have custom software installed that isn't available through a package manager. It will depend on the software if vulnerabilities are included in the NVD.

Most binaries support switches to request the version number. By running the binary with one of the switches below, it will output the version number. The following switches are the most common ones.

- –version
- -version
- –help
- -?
- -v
- -V

We started off with analyzing the behavior of the binaries when using these switches. We found cases where a lot of unusable information is printed on the screen when requesting the version numbers. Other unwanted behaviour is the printing of information on output streams other than the default “stdout”. [18] Luckily, non-default output streams can be redirected to the default “stdout” one by using the following.

```
2>&1
```

Also, if we use the “-V” or the “-v” switch, some binaries will run in verbose mode and will not exit. To prevent the proof of concept from hanging while requesting the version number, we decided to put these binaries on a blacklist. The one running the system has to decide if manual checking has to be done.

Although most binaries will output the version number, some don’t. To be able to still find out what the version is, we started to analyze them more thoroughly. The tools we used are the following.

- strings
- nm

Most of the time, these tools aren’t installed by default. That isn’t a problem though, because with these tools can’t find the version number most of the time. For example, strings most of the times outputs “version %s”, which is a variable pointing to one of the many numbers found by using strings. This is very unlikely to match up. Nm can only check object files (libraries) for symbols. However, this doesn’t output the versions.

5.2.1 Version notations

When we request the versions and isolate the lines with the version output in them, we noticed that the notations of the binary versions can differ greatly. It seems that every developer maintains its own version notation. This makes the detection and isolation of the version numbers more difficult. The example below shows the different version notations we found while researching this on our test set-up.

Version notations
```
acpid-1.0.8
apropos 2.5.2
at version 3.1.9
clear_console: Version 0.1
chvt: (console-tools) 0.2.3
base64 (GNU coreutils) 6.10
chrt (util-linux-ng 2.13.1.1)
GNU bashbug, version 3.2.39-release
Debian dpkg-architecture version 1.14.29.
apt 0.7.20.2 for i386 compiled on Apr 20 2009 21:52:36
GNU addr2line (GNU Binutils for Debian) 2.18.0.20080103
Debian ‘dpkg’ package management program version 1.14.29 (i386)
```

This example shows that the version numbering is not very consistent. We can only make an exact match with the vulnerability database entries if we are able to extract all version numbers in the right way. To do this, we make use of regular expressions.

5.2.2 Version number extraction

Like we discussed in section 3.1.2, not all binaries support the extracting of the version numbers. This can be done with the “strings” tool. However, we were unable to locate the version using this tool.

```
$ strings /usr/bin/crontab | grep version
# (Cron version -- %s)
```

The “%s” part in the output is a variable that is defined somewhere in the code of crontab. It is difficult to look for the version number this way, because there are more numbers present in the code than only the version number.

Another thing to take into account is what version of the Linux kernel is running on the system. The Linux kernel is often updated and patched because new vulnerabilities are discovered. One simple task to take this with the list of local binaries is to check the kernel version. This can be done with the following command:

```
$ uname -r
2.6.26-2-686
```

5.3 XML parsing

The vulnerability database we chose to use is the National Vulnerability Database. The database is provided on the NVD website as an XML file. There are databases available from 2002 up to now. These databases are provided in XML.

We decided to use a BASH script to create a proof of concept that is feasible during the time span of the project. However, it is difficult to parse XML using BASH. The database from the NVD is only provided in this format. However, it is easy to create a list local binaries, and match them against a CSV file with BASH. We decided to first parse the NVD XML database to a CSV format and later process this CSV formatted database using BASH.

XML parsing can be done in several ways, we chose to use Python to do this. We used the standard Python libraries to provide the XML parsing capabilities. This module is called “minidom” [19]

First, we analyzed the NVD XML database for the most relevant tags. A complete list of the tags used in the NVD database can be found in the Appendix (Section 9.3). Out of of these tags we chose the tags that are describing the most useful information for us.

Tag	**Description**	**Why**
vuln:product	The specific software with version information.	Critical information.
vuln:cve-id	Defines the CVE-ID.	Unique ID is needed for identification.
vuln:published-datetime	Defines on which date the vulnerability is published.	Age of vulnerability
vuln:cvss	Within these tags the CVSS is defined.	Can be used for severity.
cvss:score	Defines the score.	Can be used for severity.
cvss:access-vector	Defines if the vulnerability is exploitable from the network or only local.	Ease of exploitation.
vuln:security-protection	Defines the type of privilege escalation (Admin/User/Other).	Privilege escalation is critical in Grid
vuln:summary	Summary of the vulnerability.	Needed information for description.

In the vulnerability summary, there can be parts of HEX code to describe a certain type of vulnerability or exploit. Our parsing script solves this by encoding all the data to a UTF-8 character set. The UTF-8 character set introduces no problems with the HEX codes.

To make sure the parsing of the XML database to a CSV file is performed in a consistent way, we tried to get some proof. After the parsing of the database was done, we first counted all the “cve-id” tags in the XML file and compare that to the lines parsed in the CSV file. As you can see in the results below, the numbers match. This way, we know that the parsing is done correctly.

```
$ grep "cve-id" nvdcve-2.0-2002.xml | wc -l
6711
$ cat vuldb02.csv | wc -l
6711
```

By counting every unique CVE-ID, we know how many vulnerabilities are described in the XML file. When the XML database is parsed to a CSV file by the proof of concept, every vulnerability is described on it’s own line. By counting the lines, we know how many vulnerabilities are in the file.

5.4 Matching the vulnerabilities

To find out which binaries and packages are vulnerable, the script matches the host description file with the parsed CSV file with all the known vulnerabilities. It does this by matching the binary or package name together with the version notation with the “vuln:product” tag, which is one of the tags that we chose (see Section 5.3).

In this stage, some of the XML tags are not interesting to use in the CSV file, and thus not needed. We chose to put only the useful information Every line has the following information:

- Binary or package name
- Version installed
- CVE ID
- Date of publishing
- CVSS score
- Access vector (local/network)
- Security protection (is privilege escalation possible)
- Summary of the vulnerability

After the matching of the system with the database is done, the result of this matching is a listing of the found vulnerabilities for the particular host. The results are written to another CSV file, which contains the found vulnerabilities for the system that was matched. In this file every vulnerability is specified like in the example that can be seen below.

Entry example
```
nano,2.2.2,CVE-2010-1160,2010-04-016T20:30:01.397-04:00,1.9,
LOCAL,NONE,GNU nano before 2.2.4 does not verify whether a
file has been changed before it is overwritten in a file-
save operation, which allows local user-assisted attackers
to overwrite arbitrary files via a symlink attack on an
attacker-owned file that is being edited by the victim.
```

This database is saved, and is used by the script to count the total amount of vulnerabilities together and to calculate the lowest, highest and average CVSS score. We can use this score to give a certain insight into the severity of the found vulnerabilities.

5.5 Limitations

This section describes some limitations of classifying a systems security state.

5.5.1 Descriptions

The databases that provide the vulnerability information may have different descriptions on particular vulnerabilities. As some databases use human-readable descriptions on vulnerabilities, other use machine-readable descriptions. To get a consistent representation of the state of security of a system, it is best to make use of machine-readable descriptions. With machine-readable descriptions, it is possible to create scripts that can look for particular keywords to determine if a vulnerability is critical, or maybe not so harmful.

To do this, we need vulnerability databases that have a consistent way of describing vulnerabilities. The standardization of such descriptions can help us to create the scripts we discussed earlier. This is also the main reason we chose to use the National Vulnerability Database (see Section 4.3).

5.5.2 Querying repositories

We discussed ways to determine security issues of systems. Another way to check for vulnerability fixes is by querying the software repository of the operating system's vendor. For example, if a system is running a Red Hat, or a Red Hat-based operating system like the CentOS distribution that we used, the Red Hat software repository is checked for software updates. If a vulnerability is known to the vendor, a fix is likely to appear in the repository. The system then notices a newer version of the binary than the one that is installed and will put out a notification on a newer version. The problem with this approach is that there might be so-called "zero-day" vulnerability or exploits that can be critical to a systems security status if the exploit is used on such a system. So, to get a consistent classification, this approach is not the right one to use.

5.5.3 Packets, binaries and libraries

The operating system is installed with the software that is desired. This software is often installed from a repository, but can also be installed by hand.

If it is the case that the software is installed from a software repository, it is often the case that the software is installed as a package. This package can contain multiple binary files that together are the software. When we use the software repository for the checking of vulnerabilities, we won't know what binaries are vulnerable, only what packages are installed.

In the case of the libraries, we couldn't get the versions for each used libraries. We can get the used libraries by running "ldd". An example when running ldd on crontab we got the following output:

```
ldd /usr/bin/crontab
linux-gate.so.1 =>  (0x0011a000)
libselinux.so.1 => /lib/libselinux.so.1 (0x00a0e000)
libpam.so.0 => /lib/libpam.so.0 (0x009b6000)
libpam_misc.so.0 => /lib/libpam_misc.so.0 (0x00a33000)
libaudit.so.0 => /lib/libaudit.so.0 (0x0099b000)
libc.so.6 => /lib/libc.so.6 (0x00826000)
libdl.so.2 => /lib/libdl.so.2 (0x00808000)
libsepol.so.1 => /lib/libsepol.so.1 (0x009c6000)
/lib/ld-linux.so.2 (0x007e9000)
```

Now we know which libraries are used, we can check it with “nm”, however, this doesn’t output a precise version. So we found that it is very hard or sometimes not possible to request the version number of an installed library. However, we can request the version numbers of the libraries provided in by the package manager by requesting the version from a package manager like yum, rpm and dpkg. This is a workaround, though.

Another limitation of the proof of concept at this time is that to get the versioning information, we run the binaries. This means that the package runs under root. When someone places a malicious binary in one of the checked directories, it gets executed by the proof of concept. A solution would be to run the binary in a jailed environment, so it is isolated from the rest of the system.

5.6 Proof of Concept operation

The operation of the proof of concept is as follows:

First, we want to retrieve the current NVD databases in XML format. The vulnerabilities are divided per year, so there are 9 XML databases to download.

Next, we parse the downloaded XML files by using the parsing functionality of the proof of concept. This parses the XML databases to a CSV format for easier use with BASH. We end up with a CSV formatted database containing the most interesting information, as we discussed earlier. This can be used later on to match the system with the most recent known vulnerabilities.

Now, we scan the system with the proof of concept. All the version numbers of all the binaries are requested. Like discussed earlier, we can’t get the version of the used libraries, so we use the library packages which can be found in the package managers. After the system is scanned, we have a list of all binaries and libraries with the version numbers that belong to them.

Both the system file and the vulnerability file are matched. After matching is done, we know which binaries on the system are vulnerable. The total amount of found vulnerabilities is shown, but also the amount of vulnerabilities that may cause a privilege escalation.

The proof of concept calculates the amount of vulernabilities, together with the lowest, highest and average CVSS scoring for the vulnerabilities. This information is outputted on the screen or to a file for a Grid administrator to review. If something is found interesting, there can be zoomed in on the vulnerabilities.

We visualised the proof of concept in the figure below.

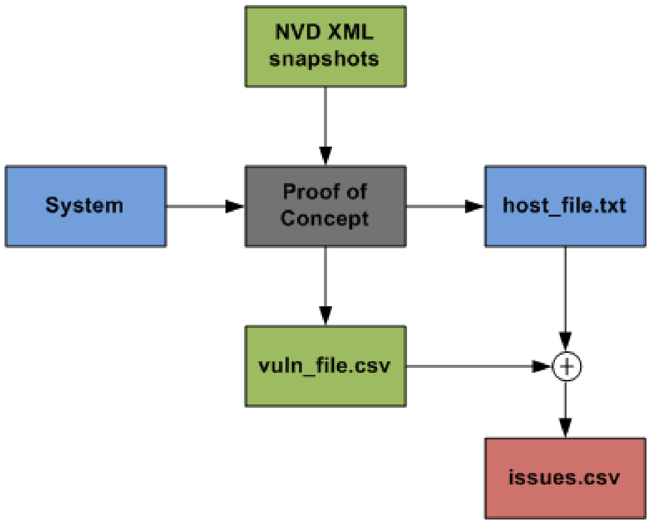

Figure 3: Proof of Concept visualisation

6 Results

In this chapter, the results of the project are discussed.

6.1 Proof of concept results

We used the proof of concept discussed in the previous chapter in our test environment to test the state of the systems. We adjusted the proof of concept based on our requirements. In the end the results show the following information on screen.

- Kernel version
- Total amount of found vulnerabilities
- Total amount of escalation vulnerabilities
- Amount of vulnerabilities listed per binary
- CVSS scoring per binary

On a side note, the CVSS scoring listed per binary, is divided in lowest, highest and average score. This way, the administrator can have some sort of indication that some vulnerabilities can have a larger impact than others.

We ran our test over different Linux distributions that each have different patch levels. The results can be seen on the following pages.

We think it is important to look at the amount of vulnerabilities, and how many vulnerabilities can cause a privilege escalation. This is because we want to focus on the most critical vulnerabilities that can pose a threat to the integrity or privacy of the data.

6.1.1 Debian Lenny

The base kernel of this Debian installation is about two years old (released on July 13 2008). The installed packages are updated to their latest version available in the repository.

Debian Lenny results
```
DEBIAN Lenny (Linux debian 2.6.26-2-686 #1
SMP Wed May 12 21:56:10 UTC 2010 i686 GNU/Linux)
TOTAL AMOUNT OF VULNERABILITIES FOUND: 153
TOTAL ESCALATION VULNERABILITIES FOUND: 9
VULNERABILITIES PER BINARY/PACKAGE
    122 kernel 2.6.26
      9 python 2.5.2
      3 libxml2 2.6.32
      2 nano 2.0.7
      2 gzip 1.3.12
      2 denyhosts 2.6
      1 wget 1.11.4
      1 w3m 0.5.2
      1 unzip 5.52
      1 tar 1.20
      1 sudo 1.6.9p17
      1 file 4.26
      1 ed 0.7
      1 cpio 2.9
      1 coreutils 6.10
      1 bsd 8.1.2
      1 bash 3.2
      1 apt 0.7.20.2
      1 acpid 1.0.8
SCORING OF THE VULNERABILITIES PER BINARY/PACKAGE
acpid: Average=5 Highest=5.0 Lowest=5.0
apt: Average=10 Highest=10.0 Lowest=10.0
bash: Average=2.1 Highest=2.1 Lowest=2.1
bsd: Average=7.1 Highest=7.1 Lowest=7.1
coreutils: Average=4.4 Highest=4.4 Lowest=4.4
cpio: Average=6.8 Highest=6.8 Lowest=6.8
denyhosts: Average=5.55 Highest=6.8 Lowest=4.3
ed: Average=9.3 Highest=9.3 Lowest=9.3
file: Average=9.3 Highest=9.3 Lowest=9.3
gzip: Average=6.8 Highest=6.8 Lowest=6.8
kernel: Average=6.07213 Highest=10.0 Lowest=1.9
libxml2: Average=4.3 Highest=4.3 Lowest=4.3
nano: Average=2.8 Highest=3.7 Lowest=1.9
python: Average=7.55556 Highest=10.0 Lowest=5.0
sudo: Average=6.2 Highest=6.2 Lowest=6.2
tar: Average=6.8 Highest=6.8 Lowest=6.8
unzip: Average=1.2 Highest=1.2 Lowest=1.2
w3m: Average=6.8 Highest=6.8 Lowest=6.8
wget: Average=6.8 Highest=6.8 Lowest=6.8
```

Out of a total of 153 vulnerabilities on this system, we see that there are 9 privilege escalations possible. With 9 privilege escalation issues present and the high percentage of high scoring vulnerabilities, we think this system needs to be reviewed by an administrator to check if the vulnerabilities are relevant this system, or that the most severe kernel vulnerabilities are patched in custom kernel release by the developers.

6.1.2 Debian Sid

The kernel of this Debian installation is half a year old (released on December 3 2009). The installed packages are updated to their latest version available in the repository.

Debian Sid results
```
DEBIAN Squeeze/Sid (Linux debian2 2.6.32-trunk-686 #1
SMP Sun Jan 10 06:32:16 UTC 2010 i686 GNU/Linux)
TOTAL AMOUNT OF VULNERABILITIES FOUND: 51
TOTAL ESCALATION VULNERABILITIES FOUND: 0
VULNERABILITIES PER BINARY/PACKAGE
     46 kernel 2.6.32
      2 gzip 1.3.12
      1 w3m 0.5.2
      1 perl 5.10.1
      1 mutt 1.5.20
SCORING OF THE VULNERABILITIES PER BINARY/PACKAGE
gzip: Average=6.8 Highest=6.8 Lowest=6.8
kernel: Average=5.78696 Highest=10.0 Lowest=1.9
mutt: Average=6.8 Highest=6.8 Lowest=6.8
perl: Average=5 Highest=5.0 Lowest=5.0
w3m: Average=6.8 Highest=6.8 Lowest=6.8
```

This tested system results in a total amount of 51 found vulnerabilities. There are no privilege escalations, but there are 5 vulnerable packages that might be interesting to look into. We can assume this system is fairly secure, because no privilege escalations are possible at the time of the test. However, depending on the jobs running on the system, administrators may have to review the gzip, mutt, perl and w3m packages.

6.1.3 Ubuntu 10.04

The base kernel of this Ubuntu version is half a year old (released on December 3 2009). The installed packages are updated to their latest version available in the package manager.

Ubuntu 10.04 results

```
Ubuntu 10.04 (Linux rpubuntu 2.6.32-22-generic-pae #36-Ubuntu
SMP Thu Jun 3 23:14:23 UTC 2010 i686 GNU/Linux)
TOTAL AMOUNT OF VULNERABILITIES FOUND: 69
TOTAL ESCALATION VULNERABILITIES FOUND: 2
VULNERABILITIES PER BINARY/PACKAGE
     46 kernel 2.6.32
      9 openssl 0.9.8k
      3 sudo 1.7.2p1
      2 screen 4.0.3
      2 nano 2.2.2
      2 gzip 1.3.12
      2 denyhosts 2.6
      1 w3m 0.5.2
      1 cpio 2.10
      1 coreutils 7.4
SCORING OF THE VULNERABILITIES PER BINARY/PACKAGE
coreutils: Average=4.4 Highest=4.4 Lowest=4.4
cpio: Average=6.8 Highest=6.8 Lowest=6.8
denyhosts: Average=5.55 Highest=6.8 Lowest=4.3
gzip: Average=6.8 Highest=6.8 Lowest=6.8
kernel: Average=5.78696 Highest=10.0 Lowest=1.9
nano: Average=2.8 Highest=3.7 Lowest=1.9
openssl: Average=5.91111 Highest=10.0 Lowest=4.3
screen: Average=6.05 Highest=7.2 Lowest=4.9
sudo: Average=6.66667 Highest=6.9 Lowest=6.2
w3m: Average=6.8 Highest=6.8 Lowest=6.8
```

This scan results in a total of 69 found vulnerabilities, with “only” 2 privilege escalations possible. 10 vulnerable packages are on the system, which may or may not be of interest. For this system, we advice to update the “openssl” package, because there are 9 known vulnerabilities for the version installed on this system.

6.1.4 CentOS 5.5

The base kernel of this version of CentOS is about four years old (released on September 20, 2006). The installed packages are updated to their latest version available in the package manager.

CentOS 5.5 results

```
CentOS 5.5 (Linux localhost.localdomain 2.6.18-194.3.1.el5 #1
SMP Thu May 13 13:09:10 EDT 2010 i686 i686 i386 GNU/Linux)
TOTAL AMOUNT OF VULNERABILITIES FOUND: 193
TOTAL ESCALATION VULNERABILITIES FOUND: 25
VULNERABILITIES PER BINARY/PACKAGE
    159 kernel 2.6.18
      8 python 2.4.3
      7 gzip 1.3.5
      5 tar 1.15.1
      3 acpid 1.0.4
      2 rsync 2.6.8
      2 nano 1.3.12
      2 dnsmasq 2.45
      1 wget 1.11.4
      1 unzip 5.52
      1 mtr 0.71
      1 m4 1.4.5
      1 bzip2 1.0.3
SCORING OF THE VULNERABILITIES PER BINARY/PACKAGE
acpid: Average=6.26667 Highest=6.9 Lowest=5.0
bzip2: Average=4.3 Highest=4.3 Lowest=4.3
dnsmasq: Average=5.55 Highest=6.8 Lowest=4.3
gzip: Average=6.58571 Highest=7.5 Lowest=5.0
kernel: Average=5.91572 Highest=10.0 Lowest=1.2
m4: Average=7.5 Highest=7.5 Lowest=7.5
mtr: Average=6.8 Highest=6.8 Lowest=6.8
nano: Average=2.8 Highest=3.7 Lowest=1.9
python: Average=7.125 Highest=9.3 Lowest=5.0
rsync: Average=9.65 Highest=10.0 Lowest=9.3
tar: Average=6.54 Highest=10.0 Lowest=4.0
unzip: Average=1.2 Highest=1.2 Lowest=1.2
wget: Average=6.8 Highest=6.8 Lowest=6.8
```

Based on these results, we would say this CentOS system would be insecure. There are a total of 193 vulnerabilities, and 25 privilege escalation vulnerabilities. There are 13 packages with vulnerabilities. However, we know Red Hat is patching most of the binaries and kernels them self. This isn't taken into account, because the custom kernel release are not included in the NVD vulnerability databases

6.1.5 Debian Sid manually updated

The kernel of this Debian installation is manually installed by us. The kernel is a few days old (released June 12 2010). We also manually updated the packages to their latest version possible.

Debian Sid Manually updated results

```
DEBIAN Squeeze/Sid (Linux debian2 2.6.35-3 #1
SMP Sun Jun 12 02:26:38 UTC 2010 i686 GNU/Linux)
TOTAL AMOUNT OF VULNERABILITIES FOUND: 4
TOTAL ESCALATION VULNERABILITIES FOUND: 0
VULNERABILITIES PER BINARY/PACKAGE
      2 gzip 1.3.12
      1 w3m 0.5.2
      1 mutt 1.5.20
SCORING OF THE VULNERABILITIES PER BINARY/PACKAGE
gzip: Average=6.8 Highest=6.8 Lowest=6.8
mutt: Average=6.8 Highest=6.8 Lowest=6.8
w3m: Average=6.8 Highest=6.8 Lowest=6.8
```

The results of this system shows 4 found vulnerabilities, with no known privilege escalations. Similar to the previous results, administrators might have to review the packages, depending on the jobs that have to run. It is interesting to see that the same vulnerabilities are still present in this patched system, most likely because of the developer that waits till the next mayor release of the binary to fix the issue.

6.2 Proof of concept statistics

The information in the previous section can be useful. But if we try to visualize the results from the paragraph above in chronological order, you can see that the amount of vulnerabilities differ a lot depending on the age of the base kernel. The base kernel may have custom patches for known vulnerabilities. However, these custom patch levels from the different distributions are not included in the vulnerability database, so it still may have vulnerabilities left from the base kernel that isn't patched in the custom patch level by the vendor.

6.2.1 Vulnerabilities over time

We found that the most vulnerabilities that are detected during the tests are found in the kernel. The chart shows that the vulnerabilities differ a lot depending on how old a kernel is. In the first column you can see the latest stable version of CentOS, which has a base kernel which is 4 years old. It has around 190 vulnerabilities in total, which almost all belong to the kernel (159).

Now take the bleeding edge Debian distribution, with all the latest patches installed and kernel 2.6.35-rc3 which is 15 days old, which has only 4 package vulnerabilities, and no known kernel issues.

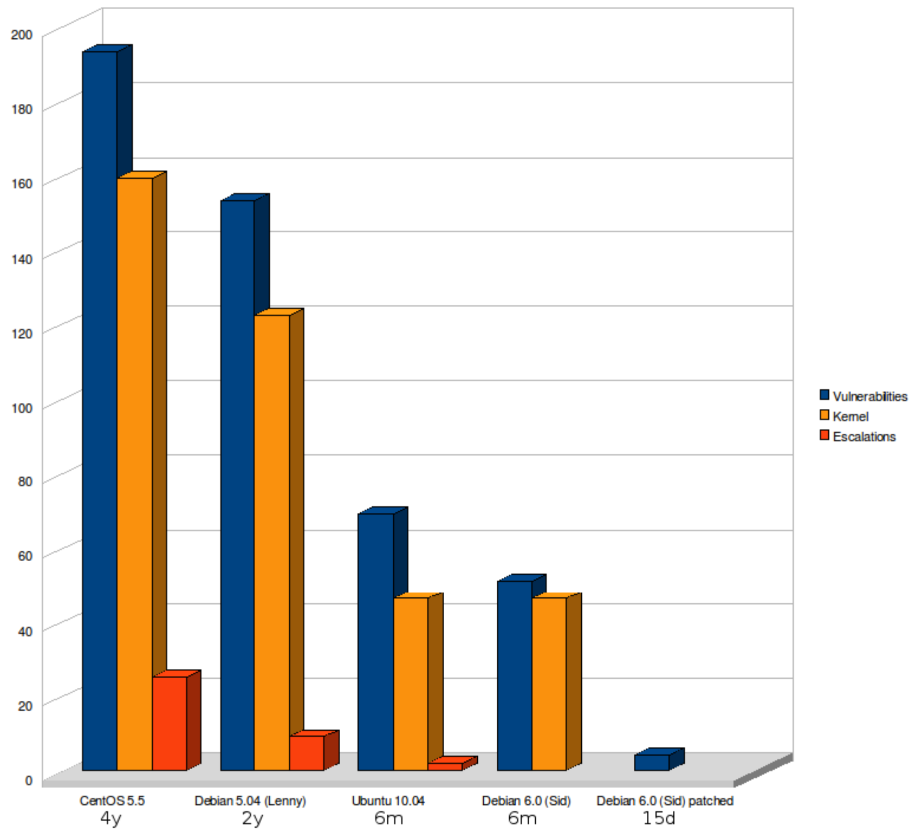

Figure 4: Vulnerabilities over time.

The bleeding edge Debian distribution in our lab setup has all of the latest available patches installed, including kernel 2.6.35-rc3, which was only 15 days old at that time. Our proof of concept shows that there are 4 vulnerable packages, but no known vulnerabilities in the kernel as of yet.

6.2.2 Zooming in

The graph below shows the scoring of the 46 vulnerabilities that were detected by our proof of concept on the test with Debian 6.0 with a kernel of 6 months old. The blue bars are vulnerabilities related to issues with the used file system. The red bars are driver related vulnerabilities, and the yellow ones are related to other various issues. As you can see, the vulnerability scores for the found vulnerabilities are almost equally divided. There are 24 vulnerabilities that have a CVSS scoring that is below 6, and 22 above that are above 6. For now, we think the vulnerabilities with a CVSS scoring higher than 6 are the ones that are the most interesting to look into, because they often are vulnerabilities that can be critical to the availability of the system. However, it is important to also look at the other vulnerabilities.

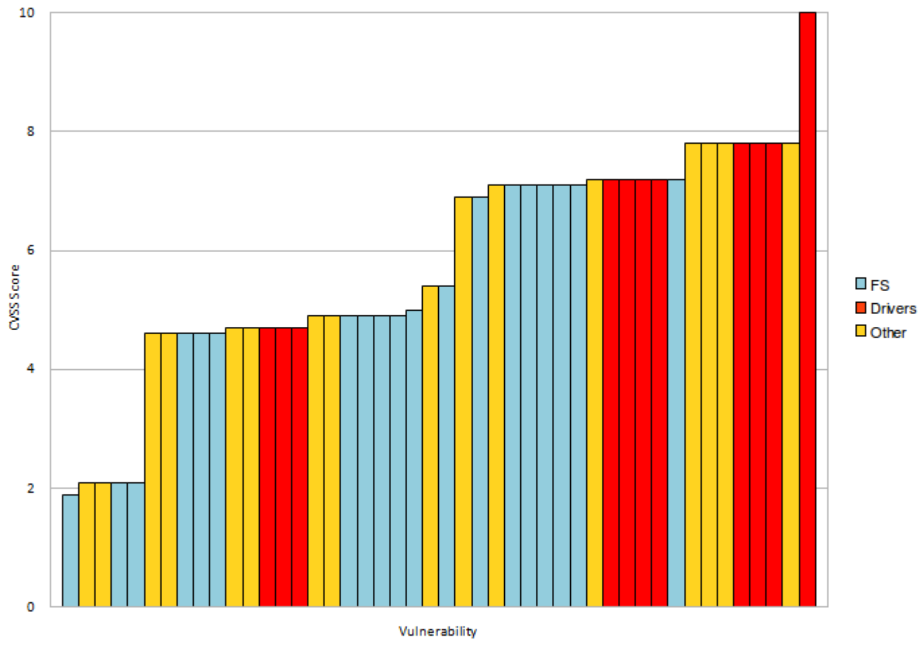

Figure 5: Kernel vulnerability scoring

Some examples of the detected kernel vulnerabilities are:

CVE-2009-3547
```
kernel,2.6.32,CVE-2009-3547,2009-11-04T10:30:00.640-05:00,6.9,LOCAL,NONE,Multiple race
conditions in fs/pipe.c in the Linux kernel before 2.6.32-rc6 allow local users to cause
a denial of service (NULL pointer dereference and system crash) or gain privileges by
attempting to open an anonymous pipe via a /proc/1093/fd/ /proc/10/fd/ /proc/1112/fd/
/proc/1113/fd/ /proc/1114/fd/ /proc/1115/fd/ /proc/1116/fd/ /proc/1117/fd/ /proc/1118/fd/
/proc/11983/fd/ /proc/11/fd/ /proc/12/fd/ /proc/137/fd/ /proc/138/fd/ /proc/13/fd/
/proc/14/fd/ /proc/15244/fd/ /proc/15/fd/ /proc/16257/fd/ /proc/16/fd/ /proc/175/fd/
/proc/176/fd/ /proc/17/fd/ /proc/18559/fd/ /proc/18571/fd/ /proc/18653/fd/ /proc/18/fd/
/proc/19/fd/ /proc/1/fd/ /proc/20106/fd/ /proc/21/fd/ /proc/225/fd/ /proc/22693/fd/
/proc/22/fd/ /proc/23/fd/ /proc/24243/fd/ /proc/24/fd/ /proc/25/fd/ /proc/26/fd/
/proc/2/fd/ /proc/31704/fd/ /proc/320/fd/ /proc/321/fd/ /proc/353/fd/ /proc/3/fd/
/proc/4793/fd/ /proc/4794/fd/ /proc/4795/fd/ /proc/4796/fd/ /proc/4799/fd/ /proc/4800/fd/
/proc/4801/fd/ /proc/4952/fd/ /proc/4966/fd/ /proc/4968/fd/ /proc/4969/fd/ /proc/4997/fd/
/proc/4998/fd/ /proc/4/fd/ /proc/5786/fd/ /proc/5787/fd/ /proc/5893/fd/ /proc/5894/fd/
/proc/5895/fd/ /proc/590/fd/ /proc/5/fd/ /proc/602/fd/ /proc/696/fd/ /proc/6/fd/
/proc/767/fd/ /proc/7/fd/ /proc/801/fd/ /proc/818/fd/ /proc/875/fd/ /proc/8/fd/
/proc/9007/fd/ /proc/9009/fd/ /proc/9010/fd/ /proc/9021/fd/ /proc/9022/fd/ /proc/9031/fd/
/proc/967/fd/ /proc/9942/fd/ /proc/9945/fd/ /proc/9946/fd/ /proc/9971/fd/ /proc/9972/fd/
/proc/9/fd/ /proc/self/fd/ pathname.
```

The vulnerability with CVE-ID “CVE-2009-3547”, has a CVSS score of 6.9. This is one of the vulnerabilities from Figure 4. We see that it is possible to cause a denial of service or even gain privileges by exploiting this vulnerability. It is interesting that it is possible to gain privileges, but we found that this isn’t noted in the “security-protection” field in the NVD database. Because of the possible privilege escalation however, we think it is still an interesting vulnerability to look into.

CVE-2010-1086
```
kernel,2.6.32,CVE-2010-1086,2010-04-06T18:30:00.610-04:00,7.8,NETWORK,NONE,The ULE
decapsulation functionality in drivers/media/dvb/dvb-core/dvb_net.c in dvb-core in
Linux kernel 2.6.33 and earlier allows attackers to cause a denial of service
(infinite loop) via a crafted MPEG2-TS frame, related to an invalid Payload Pointer ULE.
```

The vulnerability with CVE-ID “CVE-2010-1086”, concerns a driver for TV functionality (DVB standard). This particular vulnerability has a CVSS score of 7.8 assigned. The driver might be present on regular systems, but most likely not present on a grid system, as these kind of systems don’t use, or at least, don’t load such drivers. We think that this vulnerability is not that interesting, but it might still be important to look into because of the relative high scoring.

CVE-2009-4004
```
kernel,2.6.32,CVE-2009-4004,2009-11-19T21:30:00.670-05:00,7.2,LOCAL,NONE,Buffer overflow
in the kvm_vcpu_ioctl_x86_setup_mce function in arch/x86/kvm/x86.c in the KVM subsystem
in the Linux kernel before 2.6.32-rc7 allows local users to cause a denial of service
(memory corruption) or possibly gain privileges via a KVM_X86_SETUP_MCE IOCTL request
that specifies a large number of Machine Check Exception (MCE) banks.
```

The vulnerability CVE-ID “CVE-2009-4004”, which has a score of 7.2, and can only be executed when there is local access, we see that there is a buffer overflow in the kvm_vcpu_ioctl_x86_setup_mce function. This can cause a denial of service, but possibly a privilege escalation too. This isn't in the "security-protection" field also, just like with "CVE-2009-3547". We think it isn't noted in this field, because they are not sure if you really can get a privilege escalation. We also see that this vulnerability is fixed after version 2.6.32-rc7.

To also give an example of a kernel vulnerability below 6:

CVE-2010-0003
```
kernel,2.6.32,CVE-2010-0003,2010-01-26T13:30:01.010-05:00,5.4,LOCAL,NONE,The
print_fatal_signal function in kernel/signal.c in the Linux kernel before 2.6.32.4 on the
i386 platform, when print-fatal-signals is enabled, allows local users to discover the
contents of arbitrary memory locations by jumping to an address and then reading a log
file, and might allow local users to cause a denial of service (system slowdown or crash)
by jumping to an address.
```

This vulnerability with CVE-ID "CVE-2010-0003" has a CVSS score of 5.4. There is a kernel function that can allow a user to read the contents of arbitrary memory locations, which can result in users reading log files or cause a denial of service that can cause the system to slow down or crash. The assigned CVSS score is 5.4, which is just below 6. This does not mean the vulnerability is not interesting, but should be looked into because of the users that can access arbitrary memory locations.

6.3 Analysis

The results in Figure 4 show that the amount of vulnerabilities found during the scans are related to the age of the base kernel. The older the base kernel of the system is, the more vulnerabilities are known, and thus detected during our scans. For example, for our CentOS 5.5 system, our proof of concept detected a total of 193 vulnerabilities, most of which are kernel vulnerabilities (159). Our proof of concept looks for vulnerabilities that are present in the standard base kernels, which are released at kernel.org [20]. However, the developers of each distribution maintain their own patches for the kernels. This isn't taken into account in the proof of concept, but also not by the CVE specification. This means that it might be possible that the known vulnerabilities for the base kernel can be fixed in the kernel released for a specific distribution. We see this is done by using custom patch levels and descriptions.

The other thing we can see in the results is that the binaries and packages in the stable versions of the distributions are not always totally up-to-date. This is because the developers of Debian and Red Hat-based distributions keep the newest packages in a separate repository for testing before they are added to the main repository. This is done because they want to ensure the stability of the system. Only after a long testing period, which ensures there are no issues which can lead to a system crash or worse, the packages are placed into the main repository. As you can also see in the bleeding edge distributions, the newest versions are installed. This leads to less vulnerabilities, but can result in an unstable system.

If we analyze the second graph in Figure 5, the 46 kernel vulnerability CVSS scores, we see that the scores are almost equally divided. 24 vulnerabilities are under the score of 6, and 22 vulnerabilities above the score of 6. To mitigate the situation if there are vulnerabilities found in a kernel, we think it is wise to check all of the found vulnerabilities.

By analysing this further, we see that vulnerabilities with a score higher than 6, 8 are related to driver issues. Because the batch systems in a Grid are different systems then regular systems, these vulnerabilities are much less relevant. This is because batch systems are a different type of system then a regular system. For instance, batch systems have a far more simplified configuration. This goes for both hardware and software used by the system. Peripherals like bluetooth dongles and webcams are not attached to batch systems causing them to use less drivers that are needed for such input- and output devices. Kernel issues related to unloaded drivers are not relevant. This doesn't mean that these vulnerabilities shouldn't be checked. It is possible that in those vulnerabilities, there is a loaded driver and therefore relevant.

For example, if we analyse the most outstanding vulnerability in this kernel, we see it is related to driver for a network card. This vulnerability has an unknown impact, so NIST assigns "Complete" to the three "Impact" variables which defines the CVSS score (see section 4.1.2). This causes a higher score. Somebody has to see if this network card is used to rule out the vulnerability.

The Scarfone and Mell paper [8] discussed in Section 2.3.1 goes deeper into the matter of vulnerabilities introduced by faulty configurations. It is interesting to know what drivers are installed and what daemons are running, for example. Faulty configurations are often the reason why systems are vulnerable to exploits because of human mistakes. To get a more detailed and complete view of everything running on a system, it is easier to compile a more complete list, although configurations are out of the scope of this research.

One of the ideas we had to list the loaded drivers and running daemons, is by listing the loaded drivers by using the well-known command:

```
$ lsmod
```

The state of present daemons can be requested by using init scripts like in the following command:

```
$ /etc/init.d/daemon status
```

7 Conclusion

Applications or jobs running on a Grid infrastructure might need the assurance that the systems they are running on have a certain degree of security. For example, health care applications, where it is important that the patient records remain private. It is difficult for the administrators that are managing such Grids to assure this degree of security to the party that wants to run an application or job on the infrastructure.

The CVSS scoring that we used to see what the severity of a vulnerability is for a specific system, can be somewhat useful. However, the CVSS specification uses the CVE specifation for vulnerabilities, which is based on regular systems. We think this scoring specification may be optimized specifically for Grid systems. Just like Scarfone and Mell [8] adjusted the CVSS specification with information for configurations, instead of vulnerabilities, we think adjustments to the formula's will optimize the resulting score to be more in line with Grid systems.

During the testing of our prototype, we scanned the test systems from our lab setup (see Section 2.2). The different operating systems that we have tested with our proof of concept show results that are very different from each other. Our test machines with a fully updated standard installation of Debian or CentOS showed interesting results after running our proof of concept. Even though the systems are fully updated, there are still vulnerabilities to be found. Most of these vulnerabilities are located in the kernel. The results show that the kernels that are more recent are the most secure, but kernels that are older have the most issues. The base kernels have the most issues, but as discussed in Section 6.3, the developers might apply fixes themselves, although the vulnerability database does not include these separate custom kernel patches. The remaining vulnerabilities that are detected by the proof of concept are present because the maintainers of the distribution's repository have to test new releases of the packages that have issues before they are added to the repository. For example, in a case like CentOS this can take a while, because the maintainers want to guarantee a stable system because of the Enterprise status of the distribution. In other cases, developers choose to wait till the next featured release to fix known vulnerabilities.

The results also show that these problems are not occurring with the bleeding edge installations that have the latest version installed manually. But even then, some vulnerabilities are not patched immediately. in these cases, the developer seems to delay a bug fix to the next featured release. A downside of bleeding edge installations is that systems can be very unstable and may not be deployed as production systems.

To answer the research question, we think it is possible to reason about the security of a system. Our research shows that it is possible to facilitate in assuring the robustness of the systems to withstand attacks. By focusing on the state of the systems in the Grid, we can get a view of each individual system. We chose to do this by creating a proof of concept which is able to scan the version numbers of all the binaries on the system and match them to another list that contains all the known vulnerabilities to date. By taking this approach, we enable the Grid administrators to reason over the scanned system. It may also be possible to export the (scripted) host description so that users of a Grid can reason over the security of a remote machine or cluster.

8 Future work

Because of the limited time span of the project, namely 4 to 5 weeks, we were forced to limit the scope of the project. However, it might be interesting to give ideas for the possible work that can be done in the future to obtain even more interesting results. For future work, we think the following points are interesting enough to continue this research.

8.1 Applicability of CVSS scores

Like we showed in Section 6.2, CVSS scores can differ quite a lot from each other. We zoomed in on the CVSS scores and analyzed the scoring given to these vulnerabilities. We found that for vulnerabilities that have an unknown impact, the CVSS score assigned is always 10.0. We think this is an unfair score. The impact can be very severe, or not that interesting, so we think a scoring of between 5.0 and 6.0 would be fairer. Also, we found that most kernel vulnerabilities are related to the drivers and file systems used by the system. If a system is running a kernel with such vulnerabilities, it is not needed to mark the system as vulnerable because a certain type of file system or driver isn't being used by the system.

Grid systems are often used by the organizations that need such infrastructures to do large calculations. Because the CVSS scoring is based on variables that are normally encountered on regular systems, which are using many drivers and can be accessed using the network. While Grid systems are shielded from the Internet and are mainly used for batch processing, meaning only jobs that cause persistent problems are relevant.

Because of the differences between regular systems and Grid systems, a separate scoring method should be defined that is focused specifically on Grid systems.

8.2 Developing the proof of concept

Another improvement to the outcome of our work can be achieved by assigning a score to a system that is part of the Grid infrastructure. If our proof of concept is more developed to calculate a score for the scanned system, the output would be more sophisticated, and thus more usable.

By making use of hashes, it is possible to uniquely identify a file. If our proof of concept can distinguish vulnerable binaries by looking at the hash values, it might speed up the process. However, this would require having these hashes to also be included in the vulnerability database.

9 Appendix

9.1 Virtual Machine specifications

9.1.1 Debian 5.04

Our Debian 5.04 (Lenny) installation has the following specifications.

Kernel Linux debian 2.6.26-2-686 #1 SMP Wed May 12 21:56:10 UTC 2010 i686 GNU/Linux

Repositories The following repositories are used:

- http://ftp.surfnet.nl/os/Linux/distr/debian/ lenny main
- http://security.debian.org/ lenny/updates main
- http://volatile.debian.org/debian-volatile lenny/volatile main

9.1.2 Debian 6.0

Our Debian 6.0 (Sid) installation has the following specifications. Keep in mind that the used repositories aren't all the same. This is because Debian doesn't have a security and volatile repository for Squeeze or Sid yet.

Kernel Linux debian2 2.6.35-3 #1 SMP Sun Jun 12 02:26:38 UTC 2010 i686 GNU/Linux

Repositories The following repositories are used:

- http://ftp.surfnet.nl/os/Linux/distr/debian/ sid main non-free contrib
- http://security.debian.org/ squeeze/updates main non-free contrib
- http://volatile.debian.org/debian-volatile lenny/volatile main

9.1.3 CentOS 5.5

Our CentOS 5.5 installation has the following specifications.

Kernel Linux localhost.localdomain 2.6.18-194.3.1.el5 #1 SMP Thu May 13 13:09:10 EDT 2010 i686 i686 i386 GNU/Linux

Repositories The following repositories are used:

- [base]
- [updates]
- [addons]
- [extras]
- [centosplus]
- [contrib]

9.1.4 Ubuntu 10.04

Our Ubuntu 10.04 installation has the following specifications.

Kernel Linux rpubuntu 2.6.32-22-generic-pae #36-Ubuntu SMP Thu Jun 3 23:14:23 UTC 2010 i686 GNU/Linux

Repositories The following repositories are used:

- http://nl.archive.ubuntu.com/ubuntu/ lucid main restricted universe multiverse
- http://nl.archive.ubuntu.com/ubuntu/ lucid-updates main restricted universe multiverse
- http://nl.archive.ubuntu.com/ubuntu/ lucid-backports main restricted universe multiverse
- http://security.ubuntu.com/ubuntu lucid-security main restricted universe multiverse

9.2 Script download location

`http://dl.dropbox.com/u/91127/script.tar`

9.3 NVD tag descriptions

Tag	**Description**
nvd	Defines the start of the database.
entry	Defines a vulnerability entry.
vuln:vulnerable-configuration	Defines what software packages should be on the system for the system to be vulnerable.
vuln:vulnerable-software-list	A list of software that is affected by the vulnerability.
vuln:product	The specific software with version information.
vuln:cve-id	Defines the CVE-ID.
vuln:published-datetime	Defines on which date the vulnerability is published.
vuln:last-modified-datetime	Defines on which date the vulnerability is modified.
vuln:cvss	Within these tags the CVSS is defined.
cvss:score	Defines the score.
cvss:access-vector	Defines if the vulnerability is exploitable from the network or only local.
cvss:access-complexity	Defines how hard it is to exploit the vulnerability.
cvss:authentication	Defines what kind of authentication is required to exploit.
cvss:confidentiality-impact	Defines if confidential data can be accessed because of this vulnerability.
cvss:integrity-impact	Defines if data can be changed because of this vulnerability.
cvss:availability-impact	Defines if the vulnerability can cause a system crash.
cvss:source	Defines what source calculated the CVSS.
cvss:generated-on-datetime	Defines on what date the CVSS is generated.
vuln:security-protection	Defines the type of privilege escalation (Admin/User/Other).
vuln:references	Defines which sources published the vulnerability.
vuln:summary	Summary of the vulnerability.

References

[1] *A Trusted Data Storage Infrastructure for Grid-based Medical Applications*
`http://www.science.uva.nl/~noordend/publications/ccgrid08.pd`

[2] *Wikipedia: Xen hypervisor*
`http://upload.wikimedia.org/wikipedia/commons/5/5b/XEN-schema.png`
Retrieved on 1st of July 2010

[3] *The National Institute of Standards and Technology (NIST)*
`http://www.nist.gov`

[4] *Common Vulnerabilities and Exposures*
`http://cve.mitre.org`

[5] *Common Platform Enumberation*
`http://cpe.mitre.org`

[6] *Common Vulnerability Scoring System*
`http://www.first.org/cvss`

[7] *An Analysis of CVSS Version 2 Vulnerability Scoring by Karen Scarfone & Peter Mell*
`http://portal.acm.org/citation.cfm?id=1671248.1671289`
Date Published: October 14, 2009

[8] *Vulnerability Scoring for Security Configuration Settings by Karen Scarfone & Peter Mell*
`http://www.nist.gov/manuscript-publication-search.cfm?pub_id=152154`
Date Published: October 27, 2008

[9] *Puppet Labs*
`http://www.puppetlabs.com/`

[10] *Spacewalk*
`http://www.redhat.com/spacewalk/`

[11] *Secunia*
`http://www.secunia.com/`

[12] *Vupen Security*
`http://www.vupen.com`

[13] *The National Vulnerability Database (NVD)*
`http://nvd.nist.gov`

[14] *Wikipedia: National Vulnerability Database*
`http://en.wikipedia.org/wiki/National_Vulnerability_Database`
Retrieved on 7th of June 2010

[15] *Making Security Measurable initiative*
`http://measurablesecurity.mitre.org`

[16] *Wikipedia: Common Vulnerabilities and Exposures*
`http://en.wikipedia.org/wiki/Common_Vulnerabilities_and_Exposures`
Retrieved on 8th of June 2010

[17] *The Open Source Vulnerability Database*
`http://www.osvdb.org`

[18] *Wikipedia: Standard streams*
`http://en.wikipedia.org/wiki/Standard_streams`
Retrieved on 22nd of June 2010

[19] *Python minidom tutorial*
`http://www.faqs.org/docs/diveintopython/kgp_search.html`

[20] *Official Kernel site*
`http://www.kernel.org/`