THESIS

AUTOMATIC ENDPOINT VULNERABILITY DETECTION OF LINUX AND OPEN SOURCE USING THE NATIONAL VULNERABILITY DATABASE

Submitted by

Paul Arthur Whyman

Computer Science Department

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2008

Copyright by Paul Arthur Whyman 2005-2008

All Rights Reserved

COLORADO STATE UNIVERSITY

June 30, 2008

WE HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER OUR SUPERVISION BY PAUL ARTHUR WHYMAN ENTITLED AUTOMATIC ENDPOINT VULNERABILITY DETECTION OF LINUX AND OPEN SOURCE USING THE NATIONAL VULNERABILITY DATABASE BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE.

Committee on Graduate work

Adviser

Department Head/Director

ABSTRACT OF THESIS

AUTOMATED SYSTEM ENDPOINT HEALTH EVALUATION
USING THE NATIONAL VULNERABILITY DATABASE (NVD)

A means to reduce security risks to a network of computers is to manage which computers can participate on a network, and control the participation of systems that do not conform to the security policy. Requiring systems to demonstrate their compliance to the policy can limit the risk of allowing un-compiling systems access to trusted networks.

One aspect of determining the risk a system represents is patch-level, a comparison between the availability of vendor security patches and their application on a system. A fully updated system has all available patches applied. Using patch level as a security policy metric, systems can evaluate as compliant, yet may still contain known vulnerabilities, representing real risks of exploitation.

An alternative approach is a direct comparison of system software to public vulnerability reports contained in the National Vulnerability Database (NVD). This approach may produce a more accurate assessment of system risk for several reasons including removing the delay caused by vendor patch development and by analyzing system risk using vender-independent vulnerability information. This work demonstrates empirically that current, fully patched systems contain numerous software vulnerabilities. This technique can apply to platforms other than those of Open Source origin.

This alternative method, which compares system software components to lists of known software vulnerabilities, must reliably match system components to

those listed as vulnerable. This match requires a precise identification of both the vulnerability and the software that the vulnerability affects.

In the process of this analysis, significant issues arose within the NVD pertaining to the presentation of Open Source vulnerability information. Direct matching is not possible using the current information in the NVD. Furthermore, these issues support the belief that the NVD is not an accurate data source for popular statistical comparisons between closed and open source software.

Paul Arthur Whyman
Computer Science Department
Colorado State University
Fort Collins, CO 80523
Summer 2008

1. Introduction

The evaluation of a computer system's vulnerability state is an important part of protocols that measure a system's "health". These protocols use a health metric to determine the extent that a system can then participate on a trusted network. These protocols abound; and include efforts such as Cisco Network Access Control (CNAC)[0], Open Vulnerability and Assessment Language (OVAL)[1], Information Security Automation Program (ISAP)[2], the Security Content Automation Program (SCAP)[3], and include standards organizations like the Trusted Network Connect (TNC) Work Group[4], and the IETF's Network Endpoint Assessment[5] among others.

The intent of a health evaluation is to determine if systems that attach to a trusted network comply with the networks security policy before a system receives rights to participate on the network. Interrogation of health values can involve queries of the system patch state, system network location or physical location, the state of a system firewall and system virus protection, and may include other aspects depending upon the security policy requirements.

A system's current vulnerability is dependent upon a changing threat environment. To evaluate security policy compliance, up-to-date system health information is necessary. It follows that the security policy should stipulate a check to verify that a system has current security patches applied. The degree to which a system has these security updates and patches applied can form part of a system's "health status". Often a security policy allows "healthy" systems to participate on trusted networks because the system contains all available updates.

Measuring patch level by available vender updates is important; however, there is alternative information available at vulnerability data providers such as the National Vulnerability Database (NVD)[6]. The NVD provides an aggregation source for vulnerabilities, connecting information from various sources and consolidating synonymous security issues to a single identifying Common Vulnerabilities and Exposures (CVE) number.

The NVD represents two factors that are important to this work: It is a source of vulnerability information independent of a single software vender, and provides daily updates in machine-readable format that facilitates automatic analysis. This work illuminates the importance of using vender-independent vulnerability information for health checking, and discovers several critical limitations of the NVD for this type of analysis.

In spite of these limitations, this thesis (this work) will show it is a fallacy to assume a fully up-to-date system is "healthy". This fallacy is apparent by the presence of vulnerabilities (as published in the NVD) within "healthy" systems. Therefore, measuring a system's health status using a vendor's patch information does not produce results as complete as using NVD information.

1.1 Problem Statement

Is it possible to use a vendor-independent vulnerability data source such as the NVD to detect vulnerabilities within currently "up-to-date" systems? Will information obtained from the NVD produce results that are the same as those obtained by using vendor-provided software update appraisals? In other words, if

a vendor's software update utility regards a system patch-level as "up-to-date"; is it possible to demonstrate that there are un-patched vulnerabilities in the system, and therefore prove it is a fallacy to assume an up-to-date patch level is the same as vulnerability-free?

Furthermore, since the vulnerability information at the NVD is stored in machine-readable format, is it possible to automate this process? Will the information contained in the NVD be sufficient to make a complete analysis of a system?

1.2 Expectations

The two different means to evaluate system health, via a vendor's update system, or by a comparison to the NVD should produce different results for several reasons.

First, software vendors prioritize their work on software patches independent from information disclosed in public vulnerability repositories such as the NVD. This is due to development priorities and schedule requirements, which do not necessarily synchronize with the release of a CVE entry by the NVD. Second, software vendors may obtain software vulnerability information by different means than does the NVD.

The discovery of a vulnerability may originate from within the vendor process, or by independent discovery. Vender notification of a discovery may occur by the discreet means of responsible disclosure, may first appear as a bug on the vendor's bug-tracking system, or by the news of an active exploit. These

examples show how the NVD and the vendor may become aware of vulnerabilities at different times.

Software vendors may even disregard the credibility of a vulnerability report, or deem it unnecessary[7][8]. When this occurs, vulnerabilities will never receive a vendor's patch yet will perpetuate within public vulnerability lists.

Yet another cause for differences in the two evaluation means is the latency in the vulnerability lifecycle shown in **Figure 1**. The illustration represents a vulnerability lifecycle, which portrays the risks that a single system faces over time due to a single vulnerability. The period between discovery and patch application allows completely updated systems to contain publicly known vulnerabilities during the time between disclosures and patch application. The representation of "at risk" is intentionally bi-modal; a system either contains, or does not contain a given software flaw.

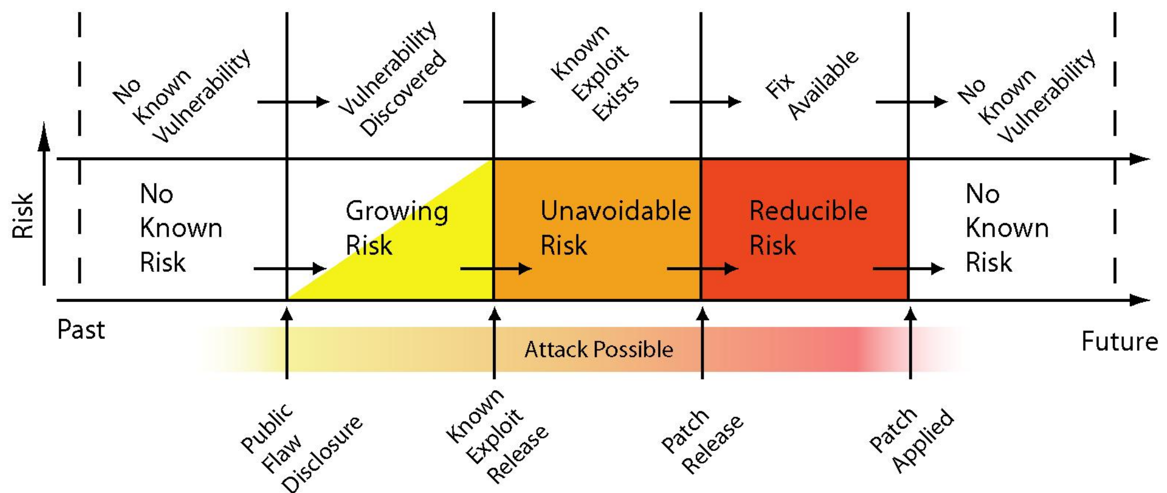

Figure 1 model of a generalized vulnerability risk lifecycle; an alternative means to measure the areas of risk is the purpose of this work.

As a result, we should expect a difference between a vulnerability inventory done by the comparison of system software with the NVD, and that of an inventory done by comparing the system software and vendor update status. It is reasonable to expect that ***if*** software developers and public vulnerability databases had perfect knowledge, the two evaluations would be the same. Yet we would also expect that long-term analysis should produce fewer differences assuming the following: Vendors have the good intention to keep security flaws out of production and to fix those that may appear. In addition, vulnerabilities identified within the NVD are without error and vendors accept them. If these assumptions are true, then eventually vendors will fix all reported security flaws.

Unfortunately, perfect knowledge is unrealistic, and system administrators can only hope these differences are minor and do not represent a significant exposure to un-patched vulnerabilities.

Furthermore, direct comparisons of system vulnerabilities with the NVD eliminate the false sense of security presumed by a vendor update check. The fallacy lies within comparing the system state with information provided by the vendor of the very same system. This check relies upon incestuous data by not including vulnerability data found outside the vendor's development stream. This verification lacks a comparison to publicly known vulnerabilities that represent threats to a fully patched system.

"Up-to-date" system status confuses the true vulnerability status of a system; the difference being between having all available vendor patches applied, and containing vulnerabilities; a system can be ***both***.

2. Background

The Internet is a network of networks, a hierarchy of interconnected computers sharing resources and communication pathways. This interconnectivity has proven to be both the boon and the bane of the Internet: The benefits of the Internet are largely due to the ease of information exchange between systems, the risks of Internet use arises from the ease of vulnerability exploitation across these same interconnected systems.

Certain vulnerabilities are susceptible to remote attack, and connecting systems with such vulnerabilities to a network exposes them to the risk of attack. Given isolation, computer systems are impervious to remote attack; obviously, this solution is not practical for systems providing remote services. Therefore, the securely deploying a systems on a network is complicated due to the ongoing appearance of remote vulnerabilities which represent an ongoing threat to these systems.

The current threat environment is constantly evolving with the discovery of previously unknown threats. Software vulnerabilities are an ongoing issue, and although security efforts attempt to adapt quickly, there are always new threats that are previously unknown.

Consequently, security is a process to manage risk. Understanding the vulnerabilities of a system is core to understanding the risk a system faces. In this manner, understanding the risks of individual systems is core to understanding the risks of a network of systems.

Often a secure perimeter intends to protect systems from these undetermined risks, the goal being to separate the systems which comply with a security policy from those that do not.

Recently, traditional security boundaries have begun to dissolve. Systems can no longer depend upon the protection of a firewall. Simply shielding a single gateway to the Internet is no longer effective due the increase in mobile computing and wireless access. The location of a computer may change from being inside to being outside of the protected perimeter. The systems residing within the firewall perimeter can no longer rely upon the safety of a sanitized Intranet. This is due to the risk of systems that bypass the perimeter walls such as systems returning from the 'wild' and visiting systems.

Network perimeters have the role of filtering what is safe and what is not. However, a firewall cannot reduce risk when an attack originates from a compromised system within the trusted perimeter. Because a secure perimeter is a less reliable means to determine system risk, we must look elsewhere for this determination. Systems containing known vulnerabilities represent risk to other systems because they are susceptible to exploitation; if they succumb to their vulnerability, they can then provide a platform to attack other systems. All potential methods to mitigate this risk begin with the identification of vulnerable systems.

2.1 Scope of this work

The beginning of a vulnerability lifecycle begins with the discovering of the vulnerability. The discovery may or may not appear publicly, however this thesis is concerned ***only*** with known vulnerabilities; managing risk posed by publically unknown vulnerabilities (hidden by responsible disclosure) or ***zero-day*** (previously unknown) attacks are outside of the scope of this thesis.

The validity of a vulnerability is also external to this investigation; that is, whether the vulnerability is verified or even has basis as a security concern. This thesis relies upon the NVD process to determine vulnerabilities regardless of a vendor's acceptance of this determination. In short, if a software packages exists within a NVD CVE, it ***is*** vulnerable within the scope of this thesis.

The examples within this thesis are only relevant to a particular time. The rapidly changing vulnerability landscape does not allow all examples to undergo post-experimental verification. New vulnerability information appears, patches are developed, and the system state continuously changes. Nevertheless, the general findings of this work are verifiable within this changing environment.

The analysis used Linux and Open Source systems, which rely upon the Debian packaging system (.deb), using Advanced Package Manager (apt); in practice this is the Ubuntu and Debian Linux distributions. Although this method can be used on other systems such as .rpm based systems (Red Hat Linux, SUSE Linux), or even Windows based systems, this was not done within the scope of this work.

2.2 The need for an ongoing vulnerability analysis

The vulnerability state of a system is an ongoing process; this relates to the nature of software development. Vulnerabilities are simply a specific form of software flaws. Vulnerabilities affect both Open and Closed source software. Open source software, can have slightly more than one software flaw for every 10,000 lines of code; five in every 100 software flaws is also a security vulnerability[9]. Security flaws are concurrent with software development.

Furthermore, as general software flaws can remain undetected, so can security flaws. Software components undergo a cyclic return to insecurity due to the repeated discovery of new software vulnerabilities; followed by a patch to return the system to a secure state. This pattern repeats throughout the life of software (**Figure 2**).

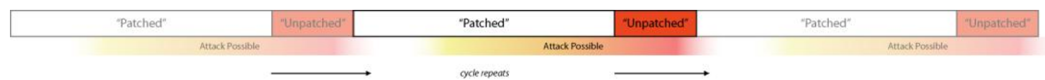

Figure 2 software cycles between patched and un-patched states

2.3 patch management vs. vulnerability management

Given two systems: in the first, a patch management system indicates risk exposure based upon the patch-level; and in the second, a comparison of system components to known vulnerabilities determines the vulnerability exposure. Which method describes the vulnerability exposure of a system with better accuracy?

The first method relies upon software vendors to provide notifications when new patches are available. Surprisingly, the majority of systems that have succumbed to intruders do so because of a known vulnerability for which a patch is readily available[10].Therefore, keeping a system up to date with the most-recent security patches is important to reduce exposure to known vulnerabilities, and can reducing the largest factor of intrusion exposure[10]. What if a publically known vulnerabilities exist, for which there are no patches? In this case, a system can still face security risks hidden by the patch-level.

How can risks measured by patch-level be different from those measured by the vulnerability level? This will occur when there is a period between a vulnerability announcement and the availability of the patch. The vulnerability lifecycle model describes this period.

This interesting period exists because of latency between the head of the software development stream, and patches applied to systems. Patch management reduces the risk of exposure *after* a vendor has produced a patch (**Figure 3)** but does so by relying upon the vendor to produce the patch. In addition, the system can appear vulnerability-free until the vendor indicates that there is something wrong by issuing a patch.

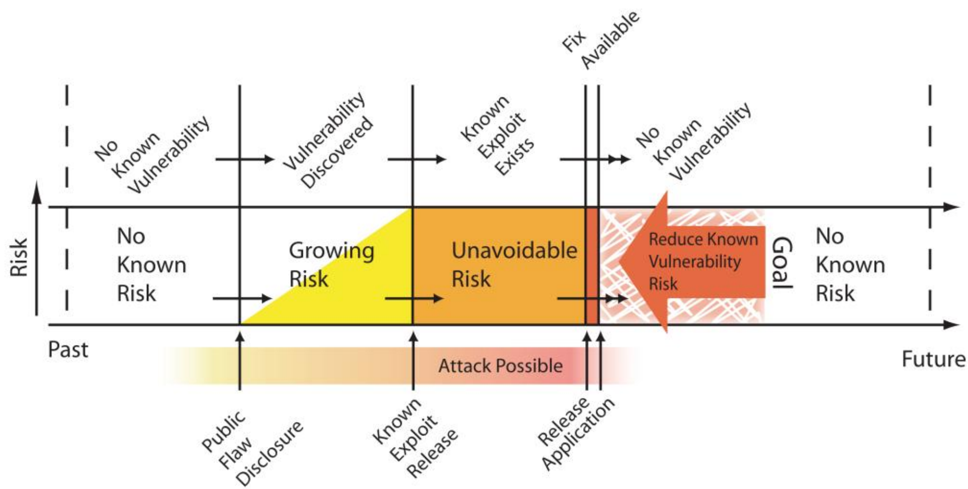

Figure 3 patch applications reduces vulnerability risk, but patches depend upon vendors production

Often there are delays between the public announcement of a vulnerability, and the availability of a patch. These delays occur for various reasons.

The delay begins with the time needed to understand, confirm, fix, test and deploy a solution. Within the Open Source community, this occurs at the head of the stream, by those working on the project itself. After this solution becomes part of the project, the version number is incremented, and a new release created.

Linux distributions managing their own packages, thus another set of delays occur from the downstream package maintainer's work. The solution may already exist for the head-of-stream version, however the process to understand, confirm, fix, test and deploy the fix repeats downstream. Maintainers first need to confirmation flaw because Linux distributions only contain periodic snapshots of upstream development. Consequently, the vulnerability does not always exist in

all snapshots. The fix then requires extraction from the upstream release, and often will need some refactoring to work with the version that the distribution is maintaining. The distribution then applies this patch to their version and makes both the source and binary versions available for their distribution releases and for supported architecture. This work may repeat itself several times by different distributions before it the solution reaches the client system, e.g. upstream-release, to Red Hat Linux, to Red Flag Linux; or upstream-release, to Debian Linux to Ubuntu Linux.

Consequently, software patches do not immediately propagate to the various downstream consumers. The fix, submitted to the upstream source repository may take some time for distribution maintainers to pick up, test, and produce a patch. This can result in a gap between a public announcement and the availability of a patch. This process also depends upon relatively easy fixes. If the software flaw is highly coupled within the package, a fix may take some time to produce.

Relying upon the arrival of a vendor patch can leave a system vulnerable for an unnecessary period. The knowledge of a vulnerability before a patch is available can enable other countermeasures to reduce the risk of a system. Various hardening techniques can reduce risks to system that contains vulnerabilities that do not currently have patches available (**Figure 4**). Examples include confinement, resource limitation, and other techniques can protect systems from these vulnerabilities. The process of securely configuring a system can reduce risks in systems that host vulnerable software. This process begins

with the knowledge that a system contains vulnerable software, the knowledge of the vulnerabilities nature, and then proceeds to specific techniques depending upon the specific issues.

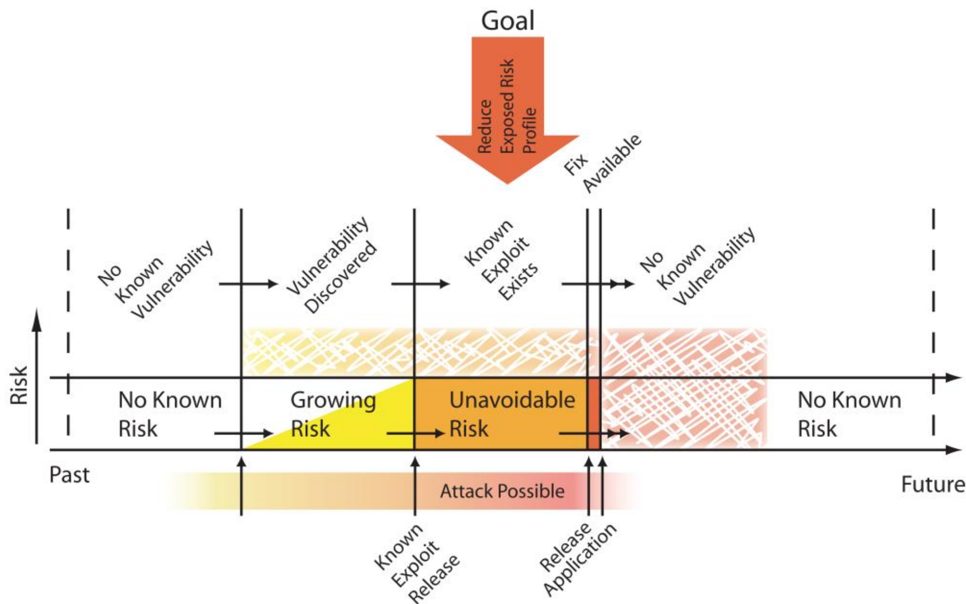

Figure 4 preventative measures can reduce risk of un-patched vulnerabilities; however, the knowledge that a system is vulnerable is required first.

The illustrations of the various periods within the vulnerability lifecycle (**Figures 1, 3, and 4)** describe the fallacy of determining system health based upon "patched" or "un-patched" (**Figure 2**). This is because the "patched" or "un-patched" metric fails to capture the complete period of system vulnerability between public announcements and patch availability.

This thesis focuses on obtaining information to manage risks during this period. The goal is to illuminate the nature of a system's vulnerability state during

this period, and thereby allow risk-mitigation techniques other than vendor-patch management.

2.3.1 Tracking Vulnerabilities in Open Source

The proprietary software development process differs from the Open Source software development process. Generally, a single controlling entity manages the proprietary development process; while cooperating, autonomous entities manage the Open Source development process. The Open Source development process has several tradeoffs. For example, it allows the Open Source community to be agile during the development process as each developer within the community can work independently. However, there is no omnipotent overseer (human or practice) ensuring the management of a given processes as it spans across various domains such as developers, projects, maintainers, distributions, and finally to individual users. This allows aspects of Open Source software to diverge.

2.3.1.1 Not-so unique identifiers

Knowing whether a particular system, component, or library is vulnerable is critical for determining the current risks a system faces. The concise identification of software vulnerabilities has two requirements. Both the software and the vulnerability must have unambiguous identification. One downside of the Open Source infrastructure is that as distributions assimilate software packages downstream, the package names diverge. The result is a difficulty identifying

vulnerable software. One example is the name given to the Apache HTTP Server. On Red Hat Linux systems, it is `httpd`, and on Debian and Ubuntu Linux systems, it will be `apache` or `apache2`.

The same problem exists with naming vulnerabilities, and affects proprietary software as well. Different agencies, such as the Debian security team, the Red Hat Bugzilla, Secunia, Security Focus, and other efforts track the same software vulnerabilities. Therefore, it can be difficult to determine if an individual system may contain two different vulnerabilities, or if there are two names for the same vulnerability. For example, a single vulnerability for the Apache HTTP Server will have a many different identifiers assigned.

The National Vulnerability Database resolves vulnerability naming conflicts by assigning each a unique identifier (a CVE number) and then linking the synonymous information from other agencies to that identifier. The CVE number essentially becomes the canonical name for each vulnerability and thus enables mapping between the various vulnerability reporting agencies.

> *NVD is a comprehensive cyber security vulnerability database that integrates all publicly available U.S. Government vulnerability resources and provides references to industry resources. It is based on and synchronized with the Common Vulnerabilities and Exposures (CVE®) vulnerability naming standard.*[6].

There is no such identification for software package names. Therefore, vulnerability detection efforts become ambiguous if one cannot discern which software a vulnerability affects.

2.3.2 Backporting obscures the upstream version

The process of Open Source development is also ‘open’. One can monitor the developer bulletin boards for critical system components and track vulnerabilities as they flow through the layers of Open Source organizations. Typically, vulnerabilities begin with an initial bug report submitted to the package maintainer, who confirms the submission, produces a security vulnerability announcement, fixes the issue, and adds it to the current stable stream. Linux distributions then produce a patch for the fix, apply it to the vulnerable packages in their distribution, make their own announcement, and provide the new package binaries.

Distributions take a “snapshot” of the ongoing development stream for a given distribution release version. This is to limit new development in the distribution release, and increase stability. Unfortunately, a fix made at the beginning of the development stream might not be compatible with the downstream versions of the vulnerable package. The fixes may need to be “back-ported” for earlier release versions. Different members of the community, from the upstream package maintainer, distribution package maintainer, or even members of the open source community at large may perform backporting resulting in release patches and patched binaries.

This process adds confusion when identifying the patches applied to a given binary version, or when determining a version's current vulnerability status. One cannot determine whether a particular package is vulnerable by comparing its

version to the vulnerable versions at the head of the development stream. One must also account for the applied back-ports.

2.4 Differences between Single-Path and Multi-Path development

The Open Source Software (OSS) development process is different from proprietary, closed source software development. This difference allows a user to procure the "same" software in various different ways. Moreover, although these different distribution paths result in similar naming and versioning, the resulting software can have profoundly different security aspects.

Unlike the management of proprietary software development that exclusively controls the release of software (**Figure 5)**, Open Source development is a composition of developers; software package development may follow multiple paths from the maintainer(s) of the source to a specific package residing in a particular system (**Figure 6)**.

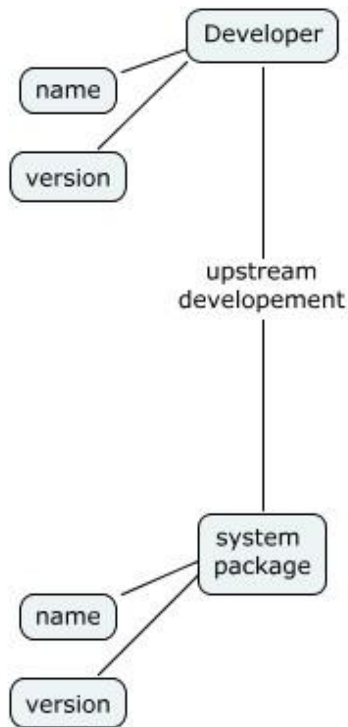

Figure 5 Closed source software has a single path between developer and users

The arbitrary path of OSS, from the head of the development stream to the actual compiled binaries that run on a users system, produces certain difficulties to the identification of software vulnerabilities. The compiling, and inclusion of different portions of the source, is due to the openness of the Open Source process that enables the compiling to take place in multiple locations. Binaries are compiled at the source head, by a project fork, in the processes of various distributions or distribution re-branding, by individual package re-branding, and last, and perhaps most importantly, the subsequent package backporting which may occur by most any of these entities.

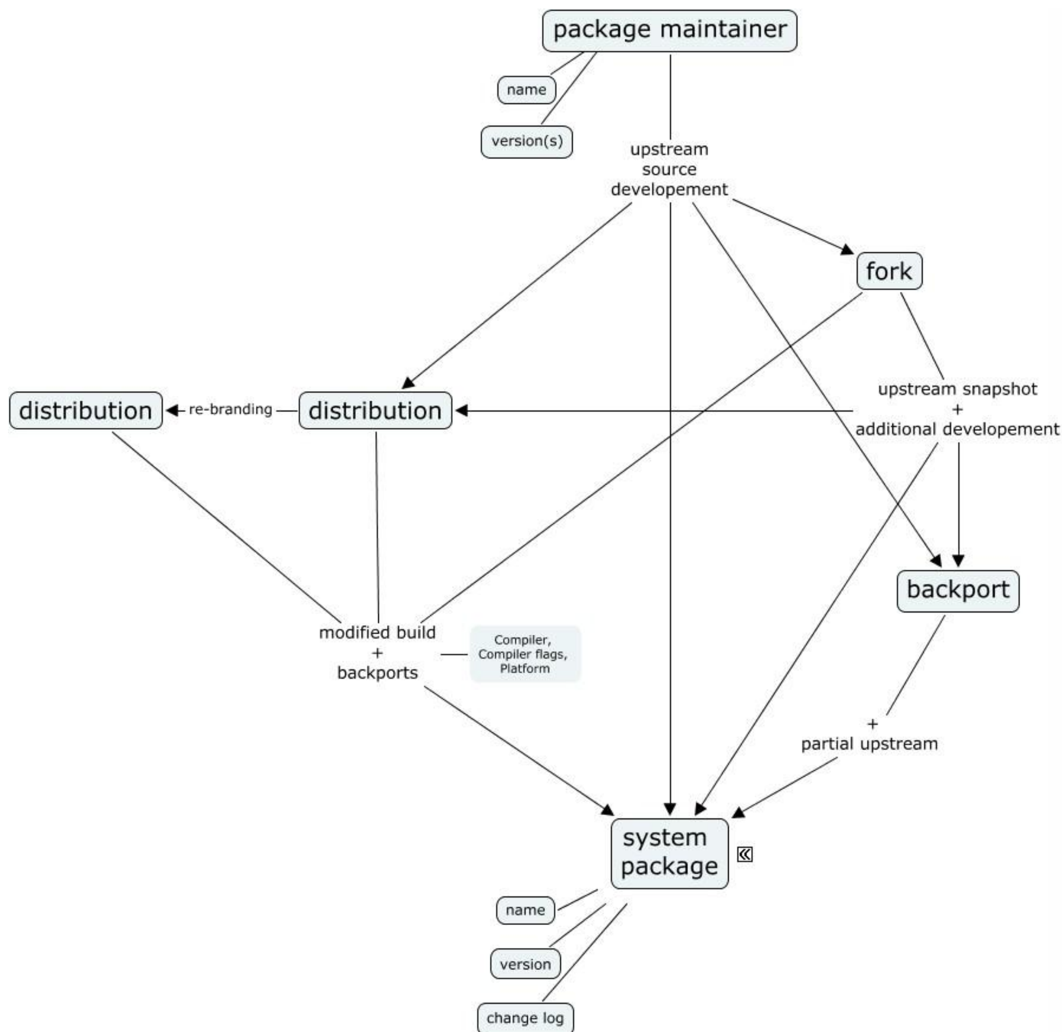

Figure 6 Open Source has multiple paths between the developer and the system. Each path varies the compilation of the same upstream source code

Because of the multiple origins of software binaries, a simple model, which fits commercial software, does not apply to Open Source. In the simple model, a vulnerability identified in a particular software binary applies to all binaries; it is not possible to have a different binary; one which was not compiled by the original developer. For example, Adobe has multiple versions of its popular Acrobat reader; however, Adobe compiles all of the binaries. Therefore,

if a vulnerability is detected in a binary, then it can be tracked by its official name, version, and even by a hash of the binary taken by the vendor at compilation time.

Contrary to this model, two Open Source packages based upon the same upstream project do not indicate the vulnerability will be contained in each. Conversely, a package which does not contain any known vulnerabilities in the upstream source repository, but that is changed and recompiled downstream may have vulnerabilities introduced[11]. In practice, the process of backporting often removes vulnerabilities downstream.

2.5 Related Work

Work can relate to this thesis in two main areas: One, that of detecting vulnerability on systems, Two, that of matching software components to those in the NVD.

2.5.1 Vulnerability vs. update assessment

The detection of vulnerabilities is a common practice, but generally stops where the work in this thesis begins, Vis-à-vis, a system vulnerability analysis simply checks if there are updates available to a given system, and relies solely upon vendor-supplied patch information, not that of independent vulnerability databases. The majority of software exploits occur to systems with patches available but not installed[10][11]. Therefore, the immediate updating of systems with the most recent patches supplied by the vendor is critical.

2.5.1.1 Update management tools

A tool that provides update information for an OSS system is the Advanced Packaging Tool (`apt`), which can compare the version of components installed on a Debian-based system, to those currently available and can also install required updates. Another tool, `apt-show-versions` provides a list of installed package names and their update status in the same manner, but does not install updates. Similar tools perform these functions for rpm-based systems such as the Red Hat Update Agent; also know as up2date, which is similarly limited to vendor-specified updates, not current vulnerabilities.

Many proprietary software vendors provide an update checking service that periodically checks for available software updates; however, these agents only check for updates within the specific vendor's updates and do not report when a software package is vulnerable if there is not an update available. Adobe, Apple, Microsoft, and Sun are among the companies that provide this type of update agents. Adobe provides a menu control for Acrobat Reader, which will even check for updates from a Linux system.

One agnostic update agent is the Secunia PSI[13]. This tool scans the majority of software on a Microsoft Windows system and determines which have outstanding security updates. The PSI agent function is an extension of the typical update check as it checks software originating from multiple vendors for security updates, and even ignores updates that are not security related. The PSI does not however indicate packages, which contain vulnerabilities present on the system,

but do not have available updates; nor does it report vulnerabilities outside of the vendor's own available patches.

2.5.1.2 The Debian vulnerability tool Debsecan

The tool `debsecan` does report vulnerable packages that do not yet have available updates. However, the tool still relies upon vendor-based information. Because the `debsecan` tool relies upon information produced by the Debian security team, the report experiences the latency of the Debian Security Team process. In some cases, vulnerabilities contained in the NVD, and present in the list of Debian Security Team "`TODO`" items are not part of the `debsecan` report. For example, debsecan did not report a current `gpg` vulnerability CVE-2008-1530, which had yet to receive attention from the Debian Security Team (as of 04/20/08).

Vulnerability information from `debsecan` only pertains to packages maintained by the Debian distribution[14]. The Debian Security Team determines, by hand, if vulnerabilities apply to packages within the Debian distribution. In some cases, a vulnerability does not apply to the package maintained by Debian, e.g. CVE-2007-4723 lists the "`Apache HTTP Server`" as vulnerable; however, the Debian security team does not agree, rather Ragnarok Online, a web application using the Apache Web Server, is vulnerable. In this case, the Debian Security Team labels the CVE as "`NOT-FOR-US`". Interestingly, "`NOT-FOR-US`" does not always mean a miss-match, sometimes it means the data does not

exist e.g. `"NOT-FOR-US: Data pre-dating the Security Tracker"`

Another instance when a vulnerability will not be reported by `debsecan` is when the Security Team does not agree that the issue is security related, e.g. CVE-2005-2541[8]:

```
severity="High"
CVSS_score="10.0"

desc= "Tar 1.15.1 does not properly warn the
user when extracting setuid or setgid files,
which may allow local users or remote
attackers to gain privileges."
```

…dismissed by the Debian security team:

```
CAN-2005-2541 (Tar 1.15.1 does not properly warn the user
when extracting setuid or ...)

NOTE: This is intended behaviour, after all tar is an
archiving tool and you need to give -p as a command line
flag
- tar (unfixed; bug #328228; unimportant)
```

Because `debsecan` uses data generated because of Debian Security Team evaluations, the datasets represent a "filtered" subset of the NVD. The data consists only of the NVD entries considered relevant by the Debian Security Team, and contain fewer false-positives. The `debsecan` tool also has a more straightforward means to detecting vulnerable system versions and packages as the security team has converted the NVD data into a Debian format. As a result, `debsecan` does not face matching problems discussed in **Section 3**, and the resulting possibility of injecting errors.

In addition, the Debian Security Team tracks issues that do not have an assigned CVE number[15]. It follows that more information is available to the `debsecan` analysis since the data includes information sources from the Open Source community, however one can speculate that eventually this security information will eventually appear in the NVD.

The work within this thesis explores whether vulnerabilities are still present on a fully patched system, by comparing system files to publicly known vulnerabilities within the NVD. There is little work done in this area outside of `debsecan` however, this tool (for better or for worse) uses domain-specific data and will not detect vulnerabilities outside the domain of the Debian Security Team.

2.5.2 Matching OSS packages with different vulnerability data sources

The second area of related work pertains to the matching of software contained within a software system with those listed in a vulnerability database such as the NVD. This issue is central to the reliable automation of system health evaluation, and currently limits the effectiveness of detection. Obviously, a precise mapping must exist between the system software and that listed in a vulnerability database. This currently does not exist; consequently, this thesis uses a heuristic approach to matching. Creating a plethora of matching rules to generate matches will not withstand changes to the naming practices of Ubuntu, Debian, the upstream package maintainer, nor the NVD itself.

The current naming practices between these entities are ambiguous and do not withstand the rigors of an automated system; therefore, an automated system can only relieve some of the work required by human evaluation. Only a robust naming schema will enable automated tools reliable and accurate matching.

2.5.2.1 Matching with the National Vulnerability Database

The NVD relies upon a single-path development model to depict OSS and thus fails to recognize the unique relationship between packages that are derived works of an upstream source, which do not have a superset-subset relation. The derived work of an Open Source project is a new software entity that cannot have superset-subset rules successfully applied, e.g. a vulnerability in a package may simply not exist within its derived work because that portion of the code was never included or compiled downstream. (**Section 3** describes many other NVD matching issues). Conversely, downstream modifications can create original vulnerabilities that are serious and have a widespread effect[16].

A FAQ entry presented on the NVD Website may explain why OSS vulnerabilities are difficult for the NVD:

> ***"How are Linux vulnerabilities handled within NVD?***
>
> *Linux distributions are often made up of a large collections of independently developed software and it is sometimes difficult to determine which software packages should be considered part of the operating system and which should be considered independent but merely included along with the operating system. In addition, some vulnerabilities occur within the Linux kernel and for those vulnerabilities we do not enumerate all of the hundreds of Linux distributions."* [7]

Separating what is part of the Linux Kernel and what is not is indeed difficult when a simpler closed source model is used. Open Source systems use the terms “kernel-space” and “user-space” to distinguish the categories described by the FAQ as “part of the operating system” or “independent of the operating system”. Moreover, it follows that an operational definition of any process is to determine whether it occupies kernel-space memory or user-space memory when executing[17]. Furthermore, the Open Source model uses the term “Kernel Module” to refer to a component similar to that of a “driver”, both of which enable hardware. Forcing the OSS development model to fit into a closed source model causes these issues.

Open Source Software is a significant part of the software world, and is widely incorporated within commercial products and it even runs on closed source platforms and not exclusively on “Linux” systems. The popularity of the Mozilla Firefox Web Browser is an example of this duel nature of OSS[18].

Security risks occur in Open Source Software just as they do in closed-source software and both benefits from the services of the NVD. The NVD has the opportunity to overcome the problems mentioned in the FAQ and thus provide the same support to OSS as closed source. A solution that incorporates ontology into the data model of the NVD, with both the terms and architecture from OSS and the Linux Kernel, will enable the NVD to support the security information needs of OSS systems. This will enable the NVD to provide unambiguous

information for all software regardless of its development process. More discussions of NVD improvements are in **Section 7.1**.

2.5.2.2 Matching with the Common Platform Enumeration

The Common Platform Enumeration (CPE) intends to resolve the issue of different domains using different naming conventions by normalizing the information. The CPE aims to establish a software-naming standard for use by automated security tools. Unfortunately, this effort will not solve an underlying issue that prevents identification: different domains represent essentially the same canonical entity with different names. Even after carefully enumerating each package, these differences will remain.

When the enumeration is complete, it will require approximately 96,821,863 entries to list the current vulnerable software found in Open Source Linux Distributions, the same as the number of hashes required by the NSRL method discussed in **Section 2.5.3.2**. The large number CPE items required to describe Open Source may make the dataset unwieldy.

Another issue arises from the efforts to normalize the information into a standard form: The requirements of the data structure produce a lossy result. Because dashes are not a legal character in XML, this entry does not represent any Debian package:

```
<cpe-item name="cpe:/a:debian:apache:1.3.34.4">
```

The entry for apache replaces the dash from the Debian `version 1.3.34-4` with a dot, which obscures the information the dash had provided. This dash is important; it represents the difference between the upstream version of the package (`1.3.34`) and the Debian update version (4). The dash represents the boundary between the upstream development and the work of the Debian package maintainer. Lost is the indication that this is the fourth package released by Debian, based upon the upstream `1.3.34` version.

Another entry further obscures the Debian package `elvis-tiny`, by replacing the dash with an underscore:

```
<cpe-item name="cpe:/a:debian:elvis_tiny">
```

These examples show that the CPE does not list Open Source packages well because the specification does not correctly enumerate the Open Source development process. Specifically, the CPE does not accommodate the special nuances of multi-path software development (**Figures 6**) annotated within Open Source version strings (**Figure 7**).

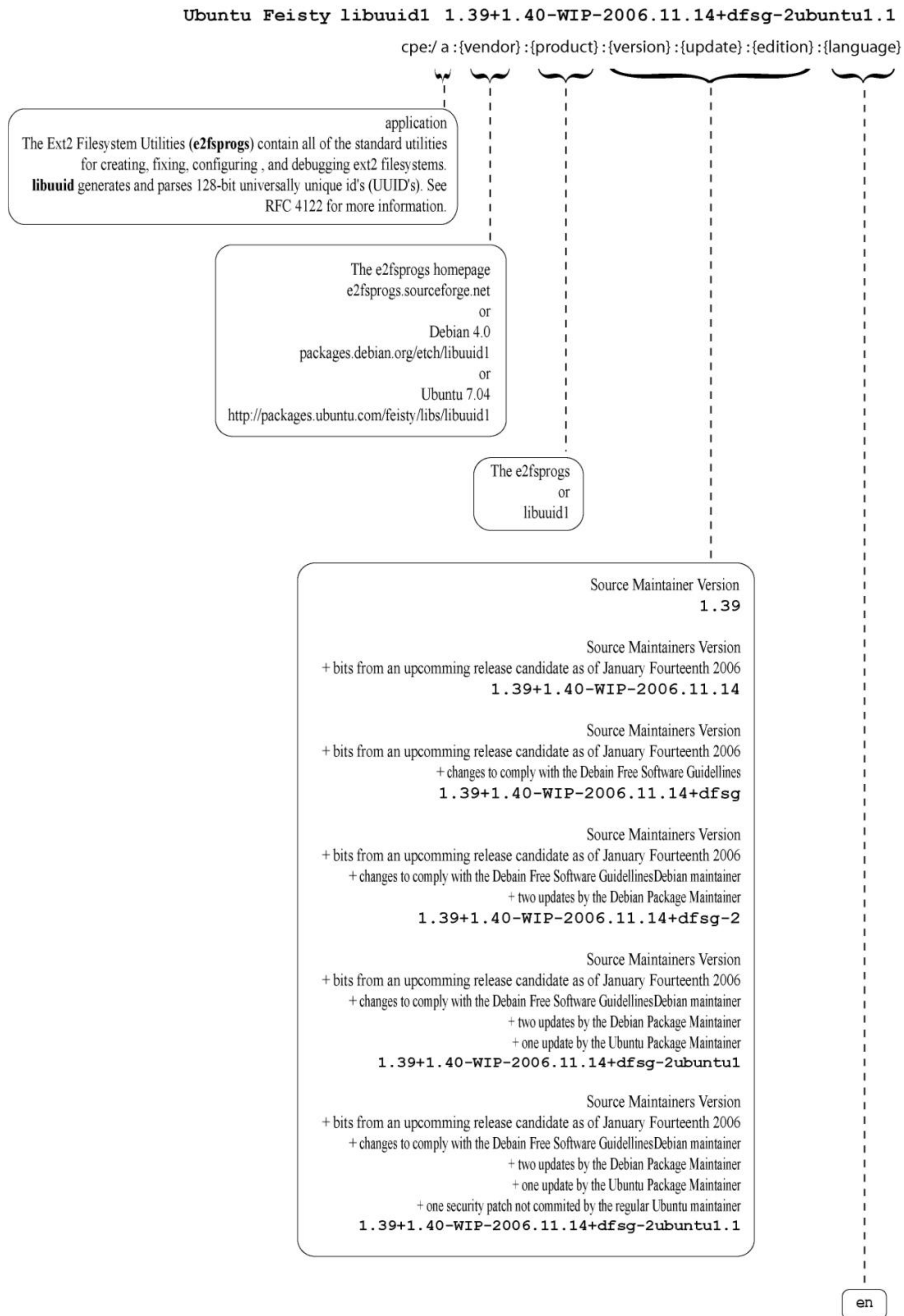

Figure 7 The CPE cannot describe libuuid1

2.5.3.2 Matching with National Software Reference Library techniques

The National Software Reference Library (NSRL) is a set of software signatures used by tools performing forensic evidence analysis of large datasets such as those found on personal computer hard drives. The data sets enable tools to reduce the quantity of files needing further examination by positively identifying files originating from known sources. Comparisons to the reference data can determine the difference between system files to ignore, and user files to examine further (**Figure 8**).

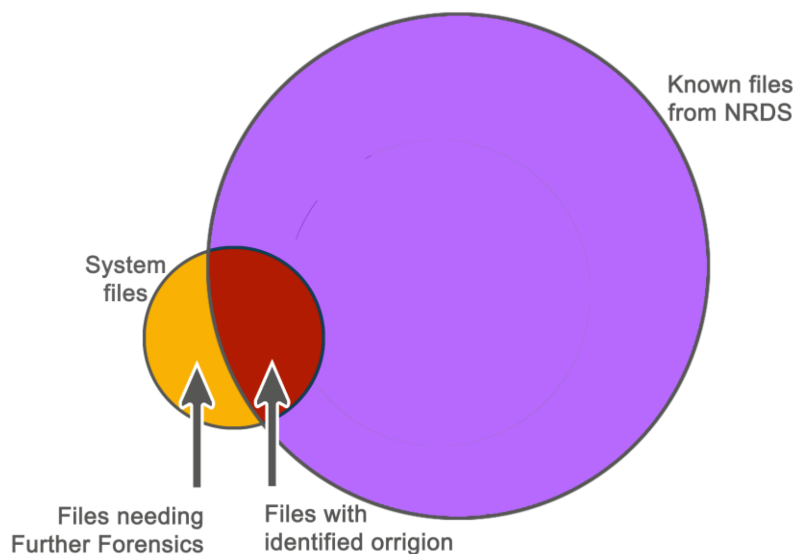

Figure 8 the NSRL identifies computer files of known origins (in red)

Can software signatures positively identify vulnerable files on a system? This depends upon how well this method applies to the problem of identifying OSS. Using signatures eliminates the need to construct ontology to map various naming conventions used by the OSS community to a single identifier.

Signatures also eliminate the need to standardize the various downstream version addendums used by the OSS community. The signature method sidesteps these issues by comparing the set of software on a system to that in a “Vulnerability Reference Library” (VRL).

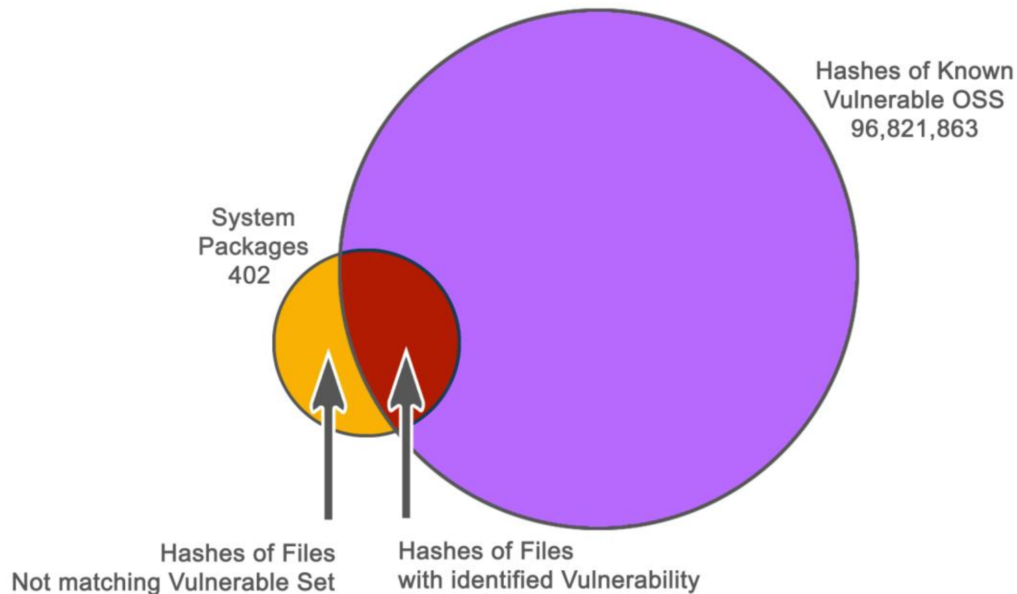

Figure 9 identifying known vulnerabilities (in red) using hashes

A dataset called the “VRL” does not exist at this time, yet the idea is quite simple. This dataset would contain a list of hashes from instances of publicly known vulnerable software, mapped to CVE numbers (**Figure 9**)

The NSRL hash-set does not contain a sufficient number of OSS to enable its use as a tool to detect software vulnerabilities. Furthermore it is unlikely the NSRL will ever do so as the typical means to obtain OSS does not fulfill the requirements to be acquired by the NSRL because software downloads are not accepted, and relatively few OSS is available via “shrink wrap” packages, only major distributions such as Red Hat and SUSE Linux.

The results of a matching comparison between a system package hash, and a set of package hashes can determine the following three outcomes depending upon the extent of the dataset:

Match

1) Matched hash is associated with CVE number
    - package contains a known vulnerability
    - this dataset only need contain hashes of vulnerable software
2) Matched hash is NOT associated with CVE number
    - package does not contain a known vulnerability
    - this dataset must contain ALL software hashes

No Match

Package unknown, dataset will not contain vulnerability information.

Classically, time and space complexities limit computer systems. Likewise, the answer to the question “can a system using techniques like the NSRL be used to identify vulnerable software on Open Source systems” is also bound by these limits. Perhaps this simple analysis can produce a practical answer to our question.

We first assume is that it is possible to create a set of hashes that represent all OSS. This universal set contains hashes representing both vulnerable and non-vulnerable OSS. This universal set allows us to identify with confidence whether any OSS has vulnerabilities. Exploring our complexity limits, the question then becomes “how many hashes are needed determine if a given OSS package contains known vulnerabilities?

We begin by determining the number of hashes needed to represent a single Open Source Linux Distribution. Debian 4.0 Etch has approximately 18,497 packages and 11 architectures. Examining several Debian systems we discover that each packages has an average of 66.59 changes per package (**Appendix I**). This rough estimate indicates that 13,548,867 hashes are needed to represent the current Debian Etch release. We now add a second distribution release, Ubuntu Feisty, which contains 21,183 packages, 7 architectures and approximately 66.24 changes per package equivalent to 9,822,133 hashes. Together, only these two distributions require 23,371,000 hashes. This is roughly equivalent to the number of hashes in the NSRL application file list. However, our hash set only represent two distributions, the current releases in Debian and Ubuntu, not the entire supported release sets from these Distributions, nor does our hash set contain the hash sets from the other 352 distributions. It not feasible to represent all Open Source Software in this way, the number of hashes needed is far too large. Perhaps we can limit the number of hashes by only listing those hashes of vulnerable software.

The set that contains all Open Source Software packages is very large. The diversity of the Apache HTTP Server makes determining the vulnerability of one particular instance of the Apache HTTP Server difficult. Because hashes represent a unique signature, they appear to be an ideal solution to this problem.

The Apache HTTP Server is one of approximately 59 projects maintained by the Apache Software Foundation. The Apache HTTP Server code has

undergone seven major releases, each of which has undergone up to sixty-three minor releases (**Figure 10**)

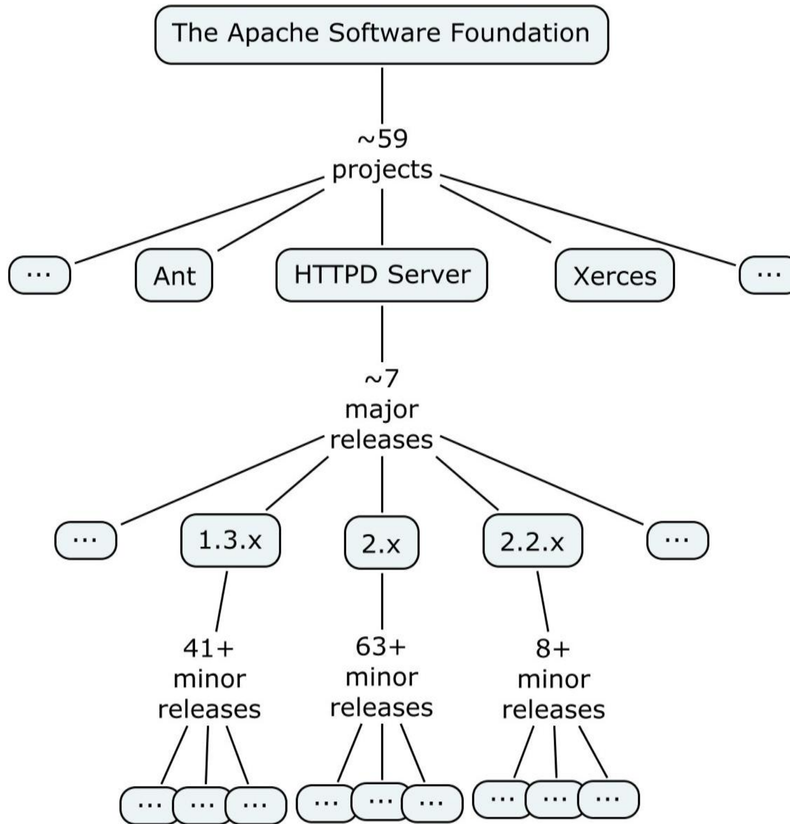

Figure 10 The number of Apache Software Foundation release instances expands by multiplying the number of projects, by the number of major releases and finally by the number of minor releases.

The Apache HTTP Server is a common part of many Linux and Open Source Distributions. There are approximately 352 Distributions, which include the various major and minor releases of the Apache HTTP Server. Distributions have their own releases, architectures and backports, which further multiply the number of Apache HTTP Server instances (**Figure 11**).

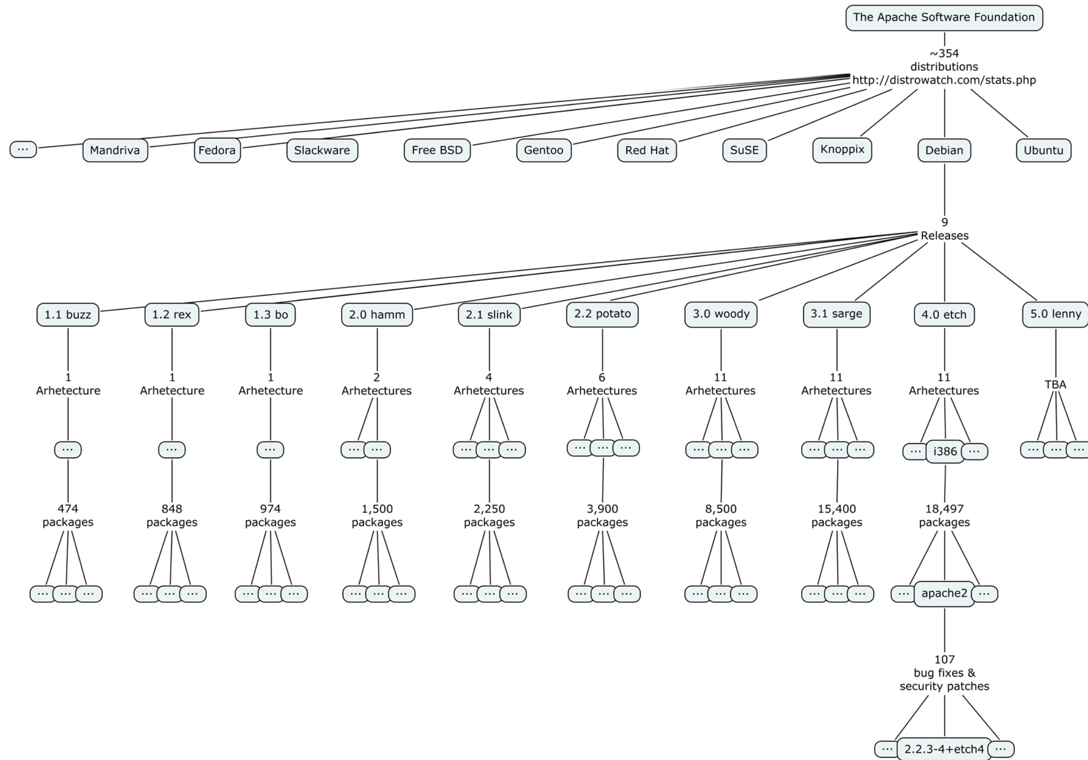

Figure 11 The number of Apache HTTP Server releases expands by multiplying the number of Apache Software Foundation releases by distributions, architectures, and back-port releases.

If we limit our matches to simply indicate that a package contains a vulnerability, (only the first outcome of a match) then our hash set must contain 96,821,863 hashes to represent the current known vulnerabilities in Open Source software (**Table 1**).

The NSRL hash-set RDS_219_C contains some 23,978,697 hashes; it is approximately 2.9 gigabytes. Assuming the dataset needed to represent Open Source vulnerabilities is similar, it would be approximately 11.65 gigabytes in size when uncompressed.

A vulnerability tool needs to compare the system packages to those in the entire dataset on a daily basis to track the daily changes in vulnerability

information. To do so, the tool must download the current dataset and then evaluate each entry and compare it to those on the system. While it is possible, the size of this dataset is prohibitive for processing by vulnerability analysis tools and transmitting over the Internet.

Approximate Number of Distributions	Average Number of Packages per Distribution	Average Number of Architectures per Distribution	Average Number of Releases per Distribution	Average Number of Vulnerabilities per Distribution	Estimated Total Number of hashes needed to represent vulnerable Open Source software
352	8,043	2.306	4.41	3.36	96,821,863

Table 1 estimated number of hashes needed to represent existing vulnerable Open Source software. This table generated from the data shown in Appendix I.

2.6 Future Work

One issue with using the NVD as a data source is the latency between the first report of a vulnerability and the listing of the issue within the NVD. This latency can increase the length of exposure to new exploits, if one solely relies upon the information provided by the NVD. The process to assign a CVE to a given vulnerability takes time; often the documentation of a vulnerability begins with a bug report to the package maintainer or upstream source. Retrieving information from such a source would bring awareness sooner, and could further reduce the time of exposure from known vulnerabilities.

Fine-tuning the match function's result vetting can speed up the analysis of accuracy. The current method is time intensive due to processes requiring human review.

The approach discussed in this work can be applied to other Open Source Distributions e.g. red hat and SUSE. These different domains do require that some methods be adapted to fit different technical requirements such as for systems that use .rpm packages. This does not prevent their analysis; the `rpm` format has a comparable tool-set that allows similar queries as `apt`. Windows-based systems are also conducive to this approach, there exists an API which enables software interrogation, enabling the comparison of vulnerability data with that of the system software.

3. Method

A comparison between a list of vulnerable software from the NVD and a list of software from an Open Source system determines a test system's vulnerability. The system can either be active and used for other work, or a test system, expressly intended for these analyses. The test does not require special system preparation, aside from loading the test scripts and obtaining the current NVD data. The analysis is self-contained; the system can perform the investigation without external interactions.

3.1 System used

Several Open Source Software systems are the test beds for vulnerability analysis. The security patch process, like most Open Source development, is open to allow an insider's perspective of this normally hidden commercial activity. This aids the verification of results that closed development processes would not.

The selection of Ubuntu and Debian from the many possible choices of Linux distributions enables the robust Advanced Packaging Tool (apt), to provide package management information and metadata for Debian (.deb) packages. Furthermore, Ubuntu has a larger and more diverse repository of packages than other popular distributions such as Red Hat, SUSE, or even Debian. The repositories contain otherwise unavailable packages such as proprietary drivers and other commercial software making for a more well rounded test of system vulnerabilities.

3.2 Heuristics for vulnerability detection

Two heuristics determine if a particular system package is vulnerable as defined by the NVD:

1. The system package appears in the NVD.
2. The version of the system package appears in the NVD.

The first heuristic determines if the NVD contains an entry for a software package. This indicates the package has contained a publicly disclosed vulnerability. The second heuristic refines the first. It determines if the software version on the test system still contains the vulnerability. If the package version from the system is greater than *any* of those listed in the NVD the assumption is that the software contains a fix (**Figure 12**).

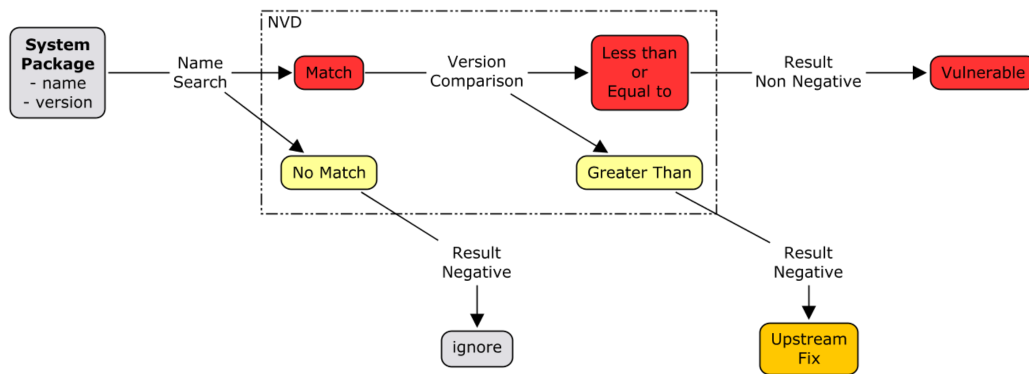

Figure 12 ideal matching

All heuristics intend to maximize vulnerability detection, and to err on the side of 'safety'. This is to reduce the chances of a false negative rather than producing false-positives; It is safer to misdetect packages which are not vulnerable than it is to miss actual vulnerable packages.

3.2.1 Determining if specific software appears in the NVD — Matching

These two heuristics are comparable to a matching exercise: Match system software with the software listed in the NVD, and then match the system version with those listed in the NVD. Conformation of both matches indicates a vulnerability is present on the testing system.

Again, note that in the context of this work this is the definition of a 'vulnerability'. Whether "proof" a vulnerability exists, has a feasible exploit, or is within the current system configuration is outside scope.

3.2.1.1 Name matching

The matching system must find comparable information in both the NVD and the system; on a system, software names identify packages; name collisions would not allow the operating system to determine which component to invoke. NVD documentation indicates the NVD also intends for software names to determine vulnerability matches:

> *National Vulnerability Database Version 2.1*
> *NVD is the U.S. government repository of standards based vulnerability management data represented using the Security Content Automation Protocol (SCAP). This data enables automation of vulnerability management, security measurement, and compliance. NVD includes databases of security checklists, security related software flaws, misconfigurations, **product names**, and impact metrics. NVD supports the Information Security Automation Program (ISAP)[6].*

The NVD documentation contains the following information for the element "prod":

> *Product wrapper tag.*
> *Versions of this product that are affected by this vulnerability are listed within this tag.*
>
> *Attributes:*
> *"name" => Product name*
> *"vendor" => Vendor of this product*

If a package name matches a NVD name, the package is 'vulnerable' unless further test heuristics can change this result to negative (**Figure 12**).

3.2.1.2 Version matching

The NVD documentation contains the following information for the element “vers”:

> *Represents a version of this product that is affected by this vulnerability.*
>
> *Attributes:*
> *"num" => This version number*
> *"prev" => Indicates that versions previous to this version Number are also affected by this vulnerability*

The NVD presents information about vulnerable versions in two ways. By either enumerating every vulnerable version or listing a single version with a flag to indicate that all previous versions are also vulnerable.

Because the NVD fails to recognize the presence of major release versions (**Section 3.3.2**), and the enumeration process is fallible, packages evaluate as vulnerable if their version is less-than-or-equal to the maximum version (**Figure 12**). The goal of the heuristic design is to fail on the side of safety; therefore, even though the NVD may contain more expressive version information, comparisons only use the maximum listed vulnerability.

3.2.2 Issues with matching: The simple two-fold heuristic does not work

Unfortunately, many issues diminish the effectiveness of automated vulnerability detection in Open Source Systems using the NVD. That is not to say that in some cases these issues are unsolvable by human intervention; however, by doing so one cannot take advantage of a computer-automated process.

When ambiguities are present in the CVE listings, assumptions in the matching function heuristics intentionally produce a false positive. The intention is that potential, yet ambiguously determined vulnerabilities will appear, and by making these possible security issues visible, they may undergo further examination. These heuristics also fail on the side of safety

.

3.2.2.1 Names

The identification of vulnerable software is a critical component of an accurate analysis. An ideal positive match between an entry in the NVD and a software package on a system must ensure the software on the system *is the same* as that listed as vulnerable within the NVD. This is to ensure the results do not contain either false positives or false negatives. Software packages must (and do) have unique identification within systems to prevent name collisions. Names are the de-facto identifier on a system. Two packages with the same name cannot exist in the same system location. Path information resolves name collisions present in different locations.

In addition to the ontology issue described in **Section 2.3.1.1**, a software name can vary depending upon its location. One is the name of a file as it resides on a particular system, another is the name of the package as delivered to the system, and yet another is the name as given by the upstream project. Often the names are the same; however, the package name can contain different information depending upon the packaging rules for the various distributions of Linux and Open Source.

Names within NVD entries often do not match the names found on actual systems, preventing name-matching (**Table 2**). Heuristics help resolve this matching issue.

Debian & Ubuntu Name	NVD Name
apache2	Apache
apache2.2-common	
apache2-mpm-prefork	
apache2-utils	
mysql-client-5.0	MySQL
mysql-server-5.0	
mysql-common	
libdns22	BIND

Table 2 System names and their NVD counterparts

3.2.2.2 Versions

Closed source development processes are less stream-like and exhibit deliberate and punctuated public releases. A single entity controls software versions, and the number of versions are less numerous. Open Source Software development represents a continuous stream of development[19]; new features appear at the head and are refined through testing and bug fixes as the stream progresses.

As the package undergoes change downstream, the community adds small descriptive terms after to the version number to represent the changes. Removing this additional information allows a comparison between the package versions and those in the NVD (**Table 3**).

Ubuntu	NVD
2.0.52-38.ent	2.0.52
2.2.3-4+etch1	2.2.3
0.9.8f-1	0.9.8
1:9.3.4-2ubuntu2.1	9.3.4

Table 3 Examples of System versions and their NVD counterparts

A simple comparison between the truncated system version and the set of versions within a CVE is still not possible, the version format which typically contains multiple decimals e.g., xxx.xxx.xxx. One additional step is required before comparing the package and CVE versions; converting both into decimal format (**Table 4**).

Package Name	Version String	Decimal used for comparison
Perl	5.8.8	5.008008
Apache2	2.2.3	2.002003
Bind9	9.4.0	9.004000

Table 4 System package version to decimal conversions

3.3 Developing match heuristics

To automate vulnerability detection, a tool simply implements the heuristics contained in this work. This tool represents a matching function, where the input is a package and NVD data; the output is a determination of vulnerability.

The first heuristic matches a package name to software names in the NVD. Although matching names is a simple string comparison, this simple match function failed to produce a significant number of positive results:

```
$CVE_Name eq $systemPackage
```

Searching the NVD for vulnerabilities published in the year 2007 through September 2007, the evaluation produced only the following nine matches:

```
irssi, tar, gimp, screen, slocate, findutils,
lftp, w3m, xterm
```

If accurate, these results indicate only these few packages have contained vulnerabilities.

3.3.1 Problems with case matching

Widely publicized vulnerabilities in The Mozilla Foundation's Firefox Web Browser are missing from the initial result set. Why does the match function fail to match the Mozilla Firefox Web Browser?

The reason is due to case sensitive Linux systems i.e. names that differ in case but are the same in all other aspects are not equivalent. In contrast, the NVD is case-insensitive and contains a mix of upper and lower case names. Therefore, the comparison function must also ignore case:

```
lc $CVE_Name eq lc $systemPackage
```

After this modification, the matching function reveals twenty additional matches, including Firefox:

```
VLC, VIM, Fetchmail, Samba, ImageMagick, GnuPG,
Firefox, Sudo, Xscreensaver, phpMyAdmin, Python,
GIMP, Snort, TCPDump, Subversion, PostgreSQL,
Evolution, OpenSSL, Ekiga, Rsync
```

This first heuristic of the matching function demonstrates that NVD does not contain consistent capitalization in listing vulnerable software names. Some entries are in lower-case and some in mixed case. The typical practice on a Linux system is to use all lower-case letters for package names, yet the NVD contains CVE records having the product name ambiguously represented with a combination of upper and lower (**Table 5**).

Package	Listings	CVE Number
GNU Image Manipulation Program	'gimp'	CVE-2007-3741
	'GIMP'	CVE-2007-2356
The Open Source toolkit for SSL/TLS	'openssl'	CVE-2004-0079
	'OpenSSL'	CVE-2007-4995

Table 5 Case-based ambiguities within the NVD

This may not appear to be important. However, this practice prevents an automatic health evaluation tool from differentiating between vulnerabilities in different packages such as 'Ant' and 'ANT'. On Linux systems, the letter-case of a package name prevents name collisions; i.e. the package Ant (automated software build tool) is different from ANT (desktop ISDN telephony application) and yet the system can determine the difference by the case.

3.3.2 Problems with major release *name* matching

The Apache HTTP Server is an Open Source Web Server developed by the Apache foundation. It is the most commonly used Web server in the world[20]. Historically, the Apache HTTP Server has contained vulnerabilities. The Apache HTTP Server version 2 is present on the testing system; yet "Apache" fails to appear in the matched list. Why does the match function fail to match the Apache HTTP Server?

The reason is due to the NVD not differentiating between the major-releases of Open Source Software. Currently, the Apache Foundation produces three major-releases of the Apache HTTP Server. The test systems contain the apache2 release; however a search for either a case-insensitive string "apache2" or a case-sensitive search for "Apache2" produce no matches within the entire NVD.

This is because the NVD lists the various Apache Server major-releases under a single product name. This is analogous to listing "Windows 95", "Windows 98", and "Windows 2000" as simply "Windows" – or simply calling all Windows systems, including desktop applications such as Microsoft Word 2003 by the name "Microsoft". The Apache Open Source foundation maintains numerous software projects in addition to the popular Apache HTTP Server. Listing the Apache HTTP Server as '`apache`' also does not differentiate between these projects.

The Open Source development credo is "Release-Early, Release-Often"[19], which is often contrary to many commercial practices. Open Source

encourages prototyping new ideas and immediately releasing them into the community for evaluation. Open Source software development releases are an almost-continuous stream of iterative versions, with both flaw-fixes and new feature development occurring at the same time, and appearing at the head of the stream. An unwanted repercussion of early-release development is that the software may never completely finish the development process; this practice, if unmanaged, can lack rigorous testing, bug fixes, and the like before initial release. This practice is by design; yet, the rapid release model may not fit the need of enterprise users requiring software stability. To address this need, the Open Source community will often "freeze" a development branch by stopping the inclusion of new features and concentrate on software stabilization.

This stabilizing technique 'forks' the software, creating two branches, one that continues with the addition of new features, and the other that no longer receives new features and the potential for instability they bring. When this happens, the project community will begin work on the new features in a new major version of the software, the version number assigned to this development branch of the fork being "significantly" different (**Table 6**).

Forking the project, by intent, creates two different bodies of code. As a result, modules that work on one fork may not work in the other, calls to the API of one may not be the same as to the other. Moreover, and significant to this discussion, security vulnerabilities affecting one branch of the fork may not affect the other. Using PHP as an example, CVE-2007-3294 only affects PHP 5, and CVE-2007-1286 only affects PHP 4.

Package	Major Releases
Apache HTTP Server	1.3.x
	2.0.x
	2.2.x
The Perl Programming Language	4.x.x
	5.x.x
	6.x.x
Linux Kernel	2.0.x.x
	2.2.x.x
	2.4.x.x
	2.6.x.x
PHP Scripting Language	4.x.x
	5.x.x

Table 6 Examples of OSS not differentiated within the NVD. The major-releases represent significant changes between Open Source Software, and do not represent a continuous stream.

Returning to the Apache HTTP Server example, vulnerabilities affecting Apache 2 may not affect Apache 1.3. Using the same name for both of these major-releases affects the accurate determination of their current vulnerability status. If Apache 1.3 is present on a system, searching for the string 'apache' will produce false-positives from vulnerabilities in Apache 2. Conversely, searching for the string '`apache2`' will not match any entry in the NVD and therefore implies it is not vulnerable. One must know that searching for vulnerabilities in packages with major-releases is a special case for the NVD.

Because the NVD does not list major versions, the match function must first drop any trailing number from a package name; these numbers represent the major release version (**Table 7**). From the perspective of the system, this combines otherwise unique software units. Nevertheless, this match function heuristic adheres to the policy to error in favor of false positives.

To illustrate the combining heuristic; the system package name '`apache2`' becomes the search string '`apache`', packages '`perl4`', '`perl5`', and '`perl6`' become the search string '`perl`', the packages '`php4`' and '`php5`' become '`php`' etc.

System Package	NVD match	NVD Vulnerability
php5	php	CVE-2007-1286
libgtop2	libgtop	CVE-2007-0235
libpng12	libpng	CVE-2007-5269

Table 7 Examples of vulnerability matches discovered ONLY after removing major release numbers from system package names

3.3.3 Problems with major release *version* matching

Combining major-releases of OSS discussed in **Section 3.3.2** also increases the difficulty to compare system versions with a NVD entry. The match function must ignore the major release found on the system, and then treat the versions found in an NVD entry as continues. This is required because the NVD treats major-releases as separate versions and not as separate entities.

Major release versions of OSS confound the notion of the "normal" commercial software model of the NVD, which typically assigns a single CVE product name for each vulnerability. As an example, Windows 95 and Windows 98 have a similar code base yet appear as separate entities in the NVD. This makes sense as each represent a separate body of code.

This is not to say that a flaw's effect cannot span between major-releases—it can. Software forks contain a common ancestral body of code and so they can share common vulnerabilities introduced into their common ancestor.

Vulnerabilities begin when a security flaws enters the development stream, and will persist until detected. Consequently, the initial vulnerable version, and all subsequent releases of the software are vulnerable. Also possible is the vulnerability resides within a single major-release only, and that the previous and later major-releases are unaffected.

A safe inference is to assume all prior releases contain this vulnerability up until and including the version where the vulnerability first appeared. The same logic holds when searching the NVD. When a software package has several current major release versions (e.g. `Apache`, `Perl`, `PHP`, etc), the safe inference is that all prior releases of the software contain this vulnerability up until and including the highest version listed.

Yet another problem arises when matching vulnerable software versions due to the absence of major release versions within the NVD. The NVD data structure has a flag that indicates if prior versions of the named software are also vulnerable. The CVE-2007-3996 `<vers>` entry indicates a specific version of `PHP` is vulnerable, and that all prior versions are vulnerable:

```
<vers num="5.2.3" prev="1" />
```

Because the highest affected version is within `PHP 5,` any version of `PHP 4` will also evaluate as vulnerable. Yet `PHP 4` is part of a separate major-release, body of code, patch and revision process. The Open Source community will produce individual patches for each major-release and will increment their

versions individually. This cannot be discerned in the CVE entry; when `PHP 4` is patched it will still appear vulnerable compared to this NVD entry.

The `<prod>` element of CVE-2007-3799 generalizes the major-releases of `PHP 4` and `PHP 5` therefore the match function must also by dropping the release number. However, the `<vers>` element specifies versions of `PHP 4` and `PHP 5`. In this example, versions of `PHP 4` will also evaluate as vulnerable because the major release of `PHP 4` is always less than `5.2.3`:

```
<vuln_soft>
  <prod name="PHP" vendor="PHP">
    <vers num="4.4.7" prev="1" />
    <vers num="5.2.3" prev="1" />
  </prod>
</vuln_soft>
```

Two issues exist with another CVE entry for the Linux Kernel, CVE-2008-0001 that indicates vulnerable versions as:

```
<prod name="Kernel" vendor="Linux">
     <vers num="2.6.22.16" prev="1" />
     <vers num="2.6.23.14" prev="1" />
   </prod>
```

The issues are: 1) An unnecessary version number of `2.6.22.16` that is *less than* the `2.6.23.14` 2) Other `Linux Kernel` major version release such as the `2.4` and the `2.2` series will always evaluate as vulnerable by a matching function.

Vulnerability CVE-2008-0455 has a good description of the ranges in the different `Apache` major-releases of `2.2`, `2.0`, and `1.3`:

```
<descript source="cve"> Cross-site scripting
(XSS) vulnerability in the mod_negotiation module
in the Apache HTTP Server 2.2.6 and earlier in
the 2.2.x series, 2.0.61 and earlier in the 2.0.x
series, and 1.3.39 and earlier in the 1.3.x
series allows remote authenticated users to…
</descript>
```

Unfortunately, a free-text description that contains the information necessary to evaluate an Apache HTTP Server version for a vulnerably is not machine-readable.

3.3.4 Problems with *version set* matching

The NVD contains entries that either 1) describe them as a range extending to all previous versions before a known vulnerable version, or 2) enumerates the exact set of vulnerable versions. If an NVD entry contains an incomplete list of enumerated versions, the possibility exists of injecting a 'clerical' error.

Again, CVE-2008-0455 is an example of this kind of error. The machine-readable version information for CVE-2008-0455 is not complete, therefore does not allow an accurate evaluation. Among the missing versions are `v2.2.1`, `v2.0.11` –through– `v2.0.27`, `v1.3.10`, and `v1.3.11`. Overall, approximately 42 versions indicated as vulnerable in the description are excluded in the machine-readable element `<vers=" "/>`

Because of this uncertainty, the match function must test for the maximum vulnerability listed within a given CVE, and then compare it to the version found on the system. Again, this heuristic produces errors in favor of false positives. This method does not miss a vulnerable version missing from a list of vulnerable versions within the CVE-2008-0455 entry.

3.3.5 Problems with consistent granularity of component entries

Complex software systems are often broken down into components. The Apache Software Foundation currently designs the Apache HTTP Server in this manner. Among the intentions to componentized large software projects is to separate concerns, decouple dependencies, and promote software re-use.

Software vulnerabilities may appear in any component of a system, and it is important to consider how to disclose the vulnerability with respect to the de-composed system. This issue is one of "granularity": should the individual components appear in a vulnerability disclosure, or should the entire composed system? In practical terms: it better for the NVD to list individual components, or the package of which the component is part? These are important questions to consider, and although their answers are outside of the scope of this thesis, the matching function requires information contained in the NVD CVE entries to follow a well-documented and internally consistent practice.

The NVD does not present consistent component naming between entries. As an example, Apache Module vulnerabilities may appear different ways:

Named as part of an Apache HTTP Server entry or named in a separate entry (**Table 8**).

A typical Apache installation contains a standard set of common modules, and optional modules are added as needed. Both the standard set and the option set of modules can be enabled or disabled individually within a specific installation, both types of modules are present in the NVD.

CVE	<prod name=' '>	Affected component	Vendor	Module Set
2007-3303	"Apache HTTP Server"	Prefork MPM module	"Apache Software Foundation"	"Core"
2007-1862	"Apache HTTP Server"	"mod_mem_cache"	"Apache Software Foundation"	"Core Other"
2007-6258	"mod_jk"	"mod_jk"	"Apache Software Foundation"	“Optional”
2007-1349	"mod_perl", "Apache"	"mod_perl"	"Apache Software Foundation"	“Optional”
2007-3847	"Apache HTTP Server"	"mod_proxy"	"Apache Software Foundation"	"Core Other"
2007-4465	"Apache HTTP Server"	"mod_autoindex"	"Apache Software Foundation"	"Core Other"
2007-5000	"Apache HTTP Server"	"mod_imap", "mod_imagemap"	"Apache Software Foundation"	"Core Other"
2007-6421	"Apache HTTP Server"	"mod_proxy_balancer"	"Apache Software Foundation"	"Core Other"
2007-0450	"Apache HTTP Server", "Tomcat"	"mod_proxy", "mod_rewrite", "mod_jk"	"Apache Software Foundation"	"Core Other", "Core Other", Optional
2006-1095	mod_python	"mod_python"	"Apache Software Foundation"	“Optional”
2005-0088	mod_python	"mod_python"	"Apache Software Foundation"	“Optional”
2005-0108	mod_auth_radius	"mod_auth_radius"	"Apache Software Foundation"	Not – Apache “freeradius”
2005-1268	"Apache"	"mod_ssl"	"Apache Software Foundation"	"Core Other"
2005-2700	“Enterprise Linux AS” “Enterprise Linux WS” "Apache", “Mod_ssl” "Desktop"	"mod_ssl"	“Red Hat” “Red Hat” "Apache Software Foundation", "mod_ssl", “Red Hat”	"Core Other"

Table 8 various examples of NVD Apache modules that have an ambiguous “granularity” associated with their names.

Because of this ambiguity, the matching function needs to check for the name of the actual module *and* the parent package when searching the NVD. **Section 3.3.6** discusses this issue in detail.

3.3.6 Problems with the canonical form of packages

Matching names of system packages with those found in the NVD is difficult. Many system names are similar, but not the same as those present within entries in the NVD. The policy: "false positives are better than false negatives" intends to limit the number of undetected vulnerable packages present on a system. Heuristics intend to produce false positives enable the matching function to discover more name matches, and expose hidden and potentially vulnerable packages to further scrutiny.

Employing these heuristics, matching becomes an exercise of examining the system package to discover the canonical name. The difficult part of matching is the successfully mapping of system packages and CVE entries to their canonical form. Once this is accomplished, matching is trivial.

The first example is that of inconsistent names for basic entries. The Apache HTTP Server appears as either "`Apache`" or as the "`Apache HTTP Server`". Since there is no "canonical form" for the Apache HTTP Server, the matching function must search two times, one for "`Apache`" and one for "`Apache HTTP Server`" (**Table 9**).

CVE	<prod name=' '>	Affected component	Vendor	Number of Entries
2007-6203	"Apache"	"Apache HTTP Server"	"Apache Software Foundation"	90
2007-6388	"Apache HTTP Server"	"Apache HTTP Server"	"Apache Software Foundation"	38

Table 9 the Apache HTTPD Server appears in the NVD under more than one way

3.3.6.1 Problems with adjectives

The second example of canonical naming issues relates to the way that package names appear on OSS systems. Many system package names are composed of a "base" name, often this is similar to a "canonical name", and various descriptive words or adjectives added to this base name, often separated by dashes. Examples of these names are `apache-common`, `libapache`, `apache-utils`. On occasion, system package names do not contain dashes but the adjectives are embedded within the package name, e.g. `libapache`, `libssl, and libmpeg3`. System package names that contain adjectives cannot match with NVD entries as CVE product names seldom contain adjectives. An examination of the package names contained on a Debian-based system reveals many common adjectives (**Table 10**).

Name	Occurrences	Name	Occurrences
"lib"	7,693	"bin"	203
"dev"	2,759	"app"	196
"perl"	1,277	"base"	179
"plugin"	303	"all"	163
"mod"	553	"conf"	163
"data"	313	"core"	97
"php"	253	"driver"	21

Table 10 A Debian packages contain repeating adjectives

A match between a system package name and a NVD entry may be determined by removing various descriptive terms found within the package name. The match function attempts to do so and then tries to match against CVE entries (**Tables 11, 12**).

System Name	Successful Match	Vulnerability
gnome-terminal-data	gnome-terminal	CVE-2003-0070
gedit-common	gedit	CVE-2005-1686
irssi-text	irssi	CVE-2003-1020

Table 11 vulnerabilities matched by alternative searches using adjective-removal productions separated by dashes

System Name	Successful Match	Vulnerability
libtar	tar	CVE-2005-2541

Table 12 Productions that remove adjectives imbedded in package names

3.3.6.2 Problems with significantly different names

Another issue with matching package names with NVD entries occurs when the two names are significantly different and do not contain similar base names. An approach to discover the canonical name of a package involves examining the package itself. Like other package files, Debian package files (.deb), contain metadata describing the binary package. The package tool apt-cache presents the metadata contained in the Apache 2.0 HTTP Server as:

```
$ apt-cache show apache2
Package: apache2
Priority: optional
Section: web
Installed-Size: 84
Maintainer: Ubuntu Core Developers <ubuntu-devel-
   discuss@lists.ubuntu.com>
Original-Maintainer: Debian Apache Maintainers
   <debian-apache@lists.debian.org>
```

```
Architecture: all
Version: 2.2.3-3.2ubuntu2
Depends: apache2-mpm-worker (>= 2.2.3-3.2ubuntu2) |
   apache2-mpm-prefork (>= 2.2.3-3.2ubuntu2) |
   apache2-mpm-event (>= 2.2.3-3.2ubuntu2)
Filename: pool/main/a/apache2/apache2_2.2.3-
3.2ubuntu2_all.deb
Size: 38764
MD5sum: 94bc013993063da5830e8a57ddc99694
SHA1: ef3e480e5bc1cb7e71708b7ac15ef8ae878307da
SHA256:
   ec711521f176b091dd8ac5f003269a00c5c491b722cb8608be
   9cc82ce3bbd9fd
Description: Next generation, scalable, extendable
   web server. Apache v2 is the next generation of
   the omnipresent Apache web server. This version -
   a total rewrite - introduces many new
   improvements, such as threading, a new API, IPv6
   support, request/response filtering, and more.
Bugs: mailto:ubuntu-users@lists.ubuntu.com
Origin: Ubuntu
Task: lamp-server
```

Within the metadata is an optional field “Source” which indicates, if present, that the package is a subcomponent part of a parent package. This field is not present within apache2 metadata therefore the apache2 package is not provided by another source other than itself.

If present, the Source field contains the name of the parent package that supplies the sub-package. Often, such as in the apache example, the source packages require several other components to supply the entire functionality. The components start with the binary itself, libraries needed for various functions, modules for different services, etc.

An example of a package that contains the “Source” metadata field is the package `avifile-xvid-plugin`:

```
$ apt-cache show avifile-xvid-plugin
Package: avifile-xvid-plugin
Priority: optional
```

```
Section: contrib/libs
Installed-Size: 28
Maintainer: Zdenek Kabelac <kabi@debian.org>
Architecture: i386
Source: avifile (1:0.7.44.20051021-2.2)
Version: 1:0.7.44.20051021-2.2+b1
Depends: libavifile-0.7c2 (>= 1:0.7.43.20050224-1),
   libc6
Filename:pool/contrib/a/avifile/avifile-xvid-
   plugin_0.7.44.20051021-2.2+b1_i386.deb
Size: 930
MD5sum: 55db4dbea6277dba90a8d0145855b257
SHA1: 01277b2426be9b58afa41c0c093a27d6aa5c1a51
SHA256:
be5e6d1766458e0a4d785e95978710cea2ec294cf61f7dd9e8bda
150ebeb821c
Description: XviD video encoding plugin for
   libavifile
 Plugin for encoding DivX4 video.
 NOTICE: This plugin requires separate installation
   of ibxvidcore 1.0 library which is not a part of
   this package nor official Debian itself. See
   documentation for more details.
   In general you do not need this plugin.
```

After the adjective production removes the adjective "plugin", the name string becomes "`avifile-xvid`" which is still not the canonical name. The adjective "`xvid`" is not a common adjective, therefore not on the adjective removal list, therefore no further decomposition will occur by the adjective – removing productions.

However, a substitution of the name "`avifile`" found in the "Source" metadata field of the package "`avifile-xvid-plugin`" yields the canonical package "`avifile`". The package name "`avifile`" would not otherwise receive any attention from the matching function and thus not compared to NVD entries because it is not loaded on the system:

```
$ apt-cache show avifile
W: Unable to locate package avifile
E: No packages found
```

Nevertheless, the name "`avifile`" is likely to appear in a CVE entry when describing vulnerabilities and not the name "`avifile-xvid-plugin`"**.** By adding the source name, when found in the package metadata, to the search of the NVD we increase the accuracy of the evaluation. There exist examples when the "Source" metadata field produces redundant results, which are the same as the adjective decomposition, however in other cases, metadata production produce surprising and unexpected names (**Table 13**).

System Name	Source Name	Vulnerability
extract	libextractor	CVE-2006-1244
libgadu3	ekg	CVE-2005-1850
bsdutils	util-linux	CVE-2005-2876

Table 13 Vulnerabilities detected via source names that other heuristic productions will not discover

3.3.3.5 Recursive lookups

After attempting to find a match between the system name and a NVD name, the matching function attempts to find matches by recursively applying productions. Matching some system packages is not possible without several applied productions (**Table 14**).

System Name	Intermediate Production	Successful Match	Vulnerability
libcurl3	libcurl	curl	CVE-2006-1061
php4-cgi	php4	php	CVE-2007-1286

Table 14 Recursive productions of major version, imbedded-adjective, or word-adjectives.

Such is the case of BIND, an important component found on many Linux Systems. The Berkeley Internet Name Domain (BIND) is an implementation of the Domain Name System (DNS) protocols. In addition to Linux systems, the vast majority of name serving machines on the Internet[21] uses it. BIND also has experienced a number of vulnerabilities; the NVD contains 42 CVE entries that explicitly name BIND vulnerabilities between 1999 and 2008. The most recent CVE-2008-0122 published January 15th 2008.

Figure 13 illustrates several issues when matching the BIND package from Linux systems. 1) Matching BIND system names cannot occur directly; normalization of the names must occur first. Both the major release numbers "`9`" and "`9-0`"and the adjectives "`lib`" and "`host`" need removal before "`BIND`" will match. Once normalized, three system packages are vulnerable. 2) Six additional packages are vulnerable by discovering their source is from BIND. 3) The issue of "granularity" exists within the BIND packages; The NVD contains six current BIND vulnerabilities; but which of the nine components contain these flaws?

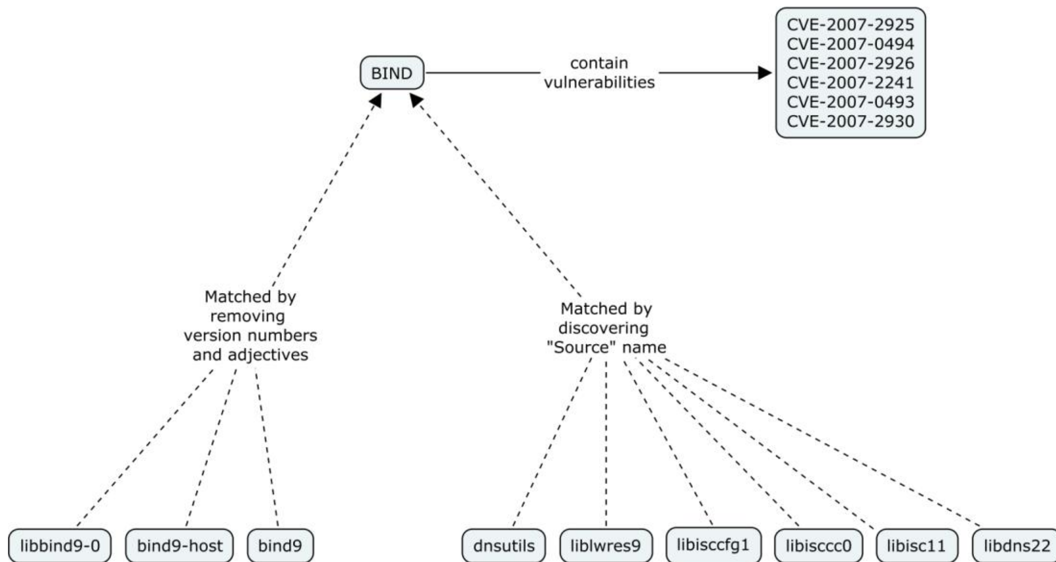

Figure 13 the relationships between BIND vulnerabilities and matched packages on a Linux system.

3.3.3.5 Two additional matching heuristics

Often a CVE <prod name=" "> entry contains several words, or even a description separated by spaces. In such cases, the package name, which never contains spaces, cannot ever match. Nevertheless, it may be possible to attempt to match the single word within the description. E.g. the word "Linux" cannot match <prod name="Enterprise Linux"; however the single word “Linux” from the CVE can be matched back to the system name. Similarly, the system name "libwpd" with <prod name="libwpd library">. The match function attempts to match each word found in the CVE <prod name=" "> with a package name from the system. This heuristic name is a “reverse” match.

Similarly, any single word from within the CVE <prod name=" "> entry may be matched with any word derived from the Match Function

productions, e.g. the system name "`libmagic1`" produces the word "`magic`" which in turn is contained in `<prod name="Fx Magic Music">` This match function heuristic name is a “any” match.

Although these two matching heuristics *do* produce additional matches, the quality of these matches is very poor. They produce a tremendous quantity of false positives, and the correct results they produce are generally duplicates of other heuristic productions. Package names that contain the adjectives ‘`file`’, ‘`Linux`’, ‘`ftp`’, or ‘`telnet`’, are particularly problematic, as they are quite common in the NVD. The string ‘`file`’ even occurs in both the system and the NVD.

3.3.3.5 Summary of match heuristics

Of the three general match heuristics tried, “Reverse”, “Alternate”, and “Any” only the “Alternate” method is successful. Discovering alternate names by using case-insensitive matching, removing numbers and adjectives or by replacing the package name with the source name produced the best results (**Figure 14 and Table 15**).

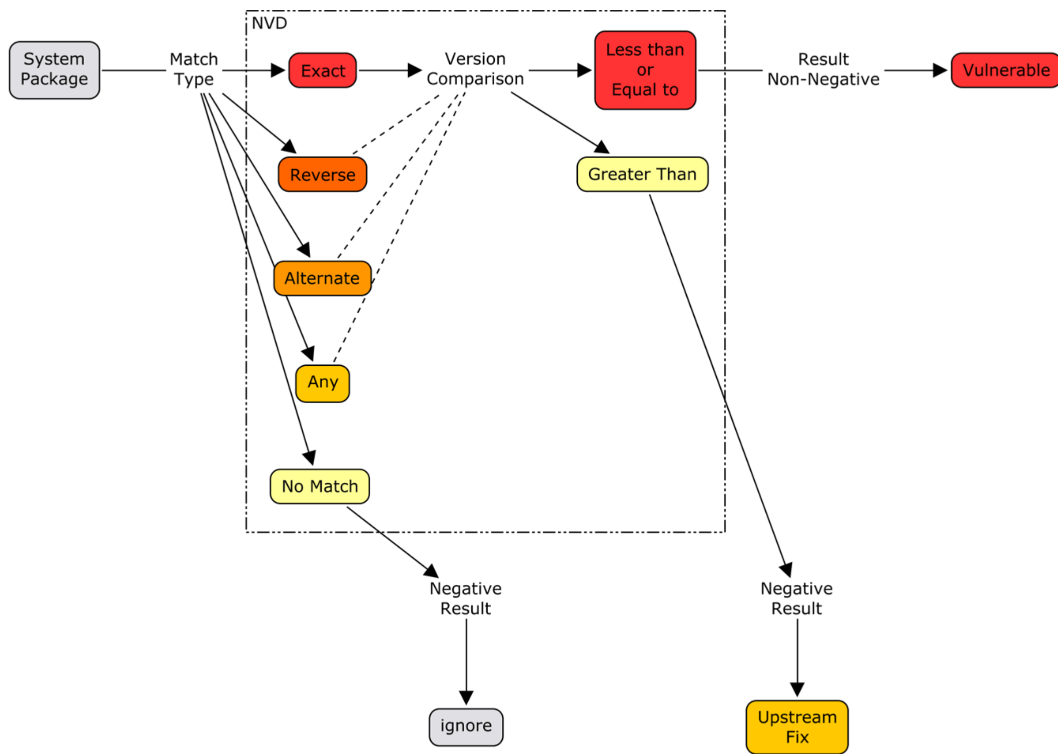

Figure 14 Summary of name matching heuristics

Heuristic	Added Matches	Match Function	Description
Precise	7	`$CVE_Name eq $systemPackage`	String match, including case e.g 'gimp' will not match 'Gimp' Note: does not produce a significant matches -not used
Exact	30	`lc $CVE_Name eq lc $sysPackage`	Case insensitive string match, creates case collisions e.g. Ant and ANT
Reverse	21	`$systemPackage ~= $CVE_Name`	CVE name matches a substring within the system name e.g. matches system name 'file' with NVD entry 'Text File Search'. Note: produces many false positives-not used

Alternate	335	`foreach $alt (@productions) {`
		`   $CVE_Name eq $alt`
		`}`
			Productions remove adjective and numbers from system names, result matched to NVD entries along with the source package name if it exists e.g. matches CVE name "Bind" by mapping system name "liblwres9" with source name "bind", also system name "apache2.2-common" with CVE name "Apache"

Any	263	`lc $CVE_Name ~= lc systemPackage`	The system name matches a substring of the CVE name e.g. system name "libusb-0.1-4" with NVD entry "Secustick USB flash drive" Note: produces many false positives-not used

Table 15 the results of matching between a system containing roughly ~2000 packages and the NVD_CVE_2007.xml file (as of September) containing 5164 vulnerability entries

3.3.4 Match Accuracy

Confidence in matches between CVE names and alternative names found by productions cannot be the same as the confidence in the exact matches. Production heuristics contain assumptions; and these can be false. The matching

function compensates by labeling each successful match with the heuristic name that produced the match. In this way, the labels provide information for further scrutiny of the match confidence.

The policy of the match function is to discover matches between a system's package names and NVD entries, and when there are uncertain results, to fail by generating false-positives. The various productions, multiplied by the intentional production of uncertain results as positives, do create duplicate results.

The matching function often identifies several files corresponding to one vulnerability. This is intentional as the exact location of the vulnerability is unknown and therefore all potential vulnerable files need recognition. Often, as in the case of a recent `e2fs` vulnerability, CVE-2007-5497, many related packages required security patches: `libcomerr2`, `libuuid1`, `libss2`, `libblkid1`, `e2fslibs`, `e2fsprogs`.

3.3.5 The problem of detecting downstream alteration

Open Source Software, by definition, has permissive software licenses that permit the modification of packages. The license, along with the availability of the source code, allows changes to occur at any time in the software lifecycle. This is very different from the closed source process. Significant changes to Open Source Software do not necessarily occur at the head of the development stream; software experiences changes all along the development stream including changes by the end-user. This poses additional problems to determine if vulnerable software exists on a system using the NVD. The software name and version found in the NVD is likely from the head of the development stream and does not reflect

the myriad of changes the downstream community has made. This issue is the same as the name and version matching problems described in **Section 3.2.2**

This downstream alteration effect continues after the matching function has accurately matched a system name with a CVE entry, and has determined that the system version is less-than or equal-to the greatest vulnerable version in the CVE. Surprisingly it is very likely this matched package is still not vulnerable. This has caused much confusion for IT personnel attempting to detect vulnerable OSS software.

The root of the confusion is that CVE entries contain versions as listed at the head of the development stream. The subsequent downstream alternations are not apparent in the CVE, and without accounting for these alterations, comparing version information is worthless.

Section 3.2.2.2 describes the truncation of system version information after the upstream version. This must occur to enable a comparison with the NVD. Unfortunately, the information added after the upstream system version represents important alterations that affected the package including the backporting of security patches. **Table 3** shows versions of several Ubuntu packages and their truncated counterparts. These annotations signify profound changes from the upstream version.

At the head of a development stream (**Figure 6)**, the project maintainer, produces release versions, and downstream, various entities make changes to this version. These changes fundamentally alter the software to such a degree that it may no longer behave the same. Alterations may be for many purposes, not only

for security. The changes modify the package in many ways. They may even introduce new vulnerabilities e.g. CVE-2007-3379, is a vulnerability introduced into the red hat Linux Kernel or, as in the case of a security back-port, a patch from the head of the development stream fixes a downstream version. The backporting process eliminates the vulnerability within the older "version", but the older version will still appears to be "vulnerable" by comparison to the NVD version regardless that it no longer contains the faulty code.

The practice of "backporting" a security fix will reduce the latency of the patch to precede down the development stream, anyone can do the work; however, it is often the work of distribution package maintainers. The work done to produce the security patch begins with a patch submission to the head of the development stream. Thereafter, this fix becomes part of all future versions. Unfortunately, this resolves the security vulnerability only at the head of the development stream; the older, stable portions of the stream are still subject to the exploitation of the security flaw. Therefore, the patch must also apply to the "stable" versions of the software. This takes additional work, as the stable versions may not be the same as the code at the head of development in lieu of the freeze for stability. The process of refactoring the patch to apply it to the stable software is one form of "backporting" the security patch.

3.3.6 Detecting downstream alteration with the changelog

We cannot directly compare a CVE version with a System version. Left unaltered, the system package version often contains extra annotations added by downstream package maintainer (**Figure 15**). These annotations indicate if the system version is vulnerable or if a back-port has been applied. Removing annotations such as show in **Table 3** enables numerical comparison to those in the NVD; however, the results now contain false positives. The package versions detected as vulnerable by this comparison may actually contain applied back-ported security patches.

Fortunately, .deb package binaries include a changelog file that lists a history of the package maintainer's work, including applied security patches[22]. Each security patch, which is back-ported by a package maintainer to fix a bug or security fault, will reference the original issue by its unique CVE (or candidate) id.

Because CVE numbers mark when patches have fixed security flaws, the matching function can equate the presence of a CVE number to a back-ported fix within the package. The changelog can provide information that the package is not vulnerable even when the comparison of the system and CVE version indicates the package is vulnerable. By parsing the changelog for a specific CVE, the match function can automatically determine if a back-port fix for CVE is present in the package binary. **Figure 16** shows this heuristic added to the matching process.

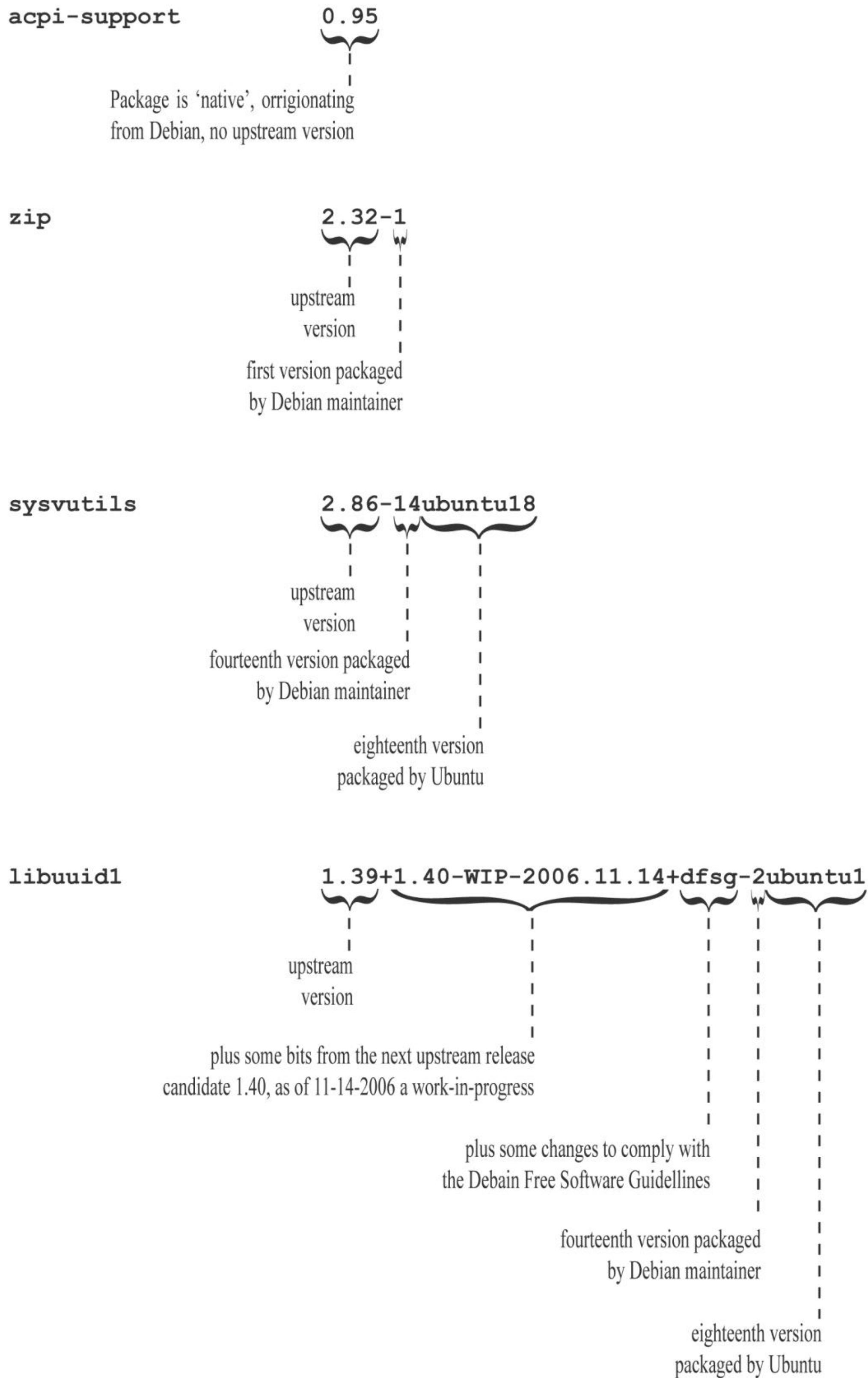

Figure 15 examples of package versions present on an Ubuntu 7.10 system, annotations by downstream package maintainers describe downstream changes

If a package is listed in the NVD, and has an upstream version less than or equal to any version listed in the NVD, it is vulnerable only if the CVE number is not found in its changelog. Note that there is an implicit decision to trust the work of the package maintainer. In addition, determining whether the presence of the CVE number is accurate in the package changelog is beyond the scope of this work. Moreover, all Debian packagers do not follow the Debian packager's manual, e.g. `gcc`, does not list CVE numbers when back-port security patches are applied.

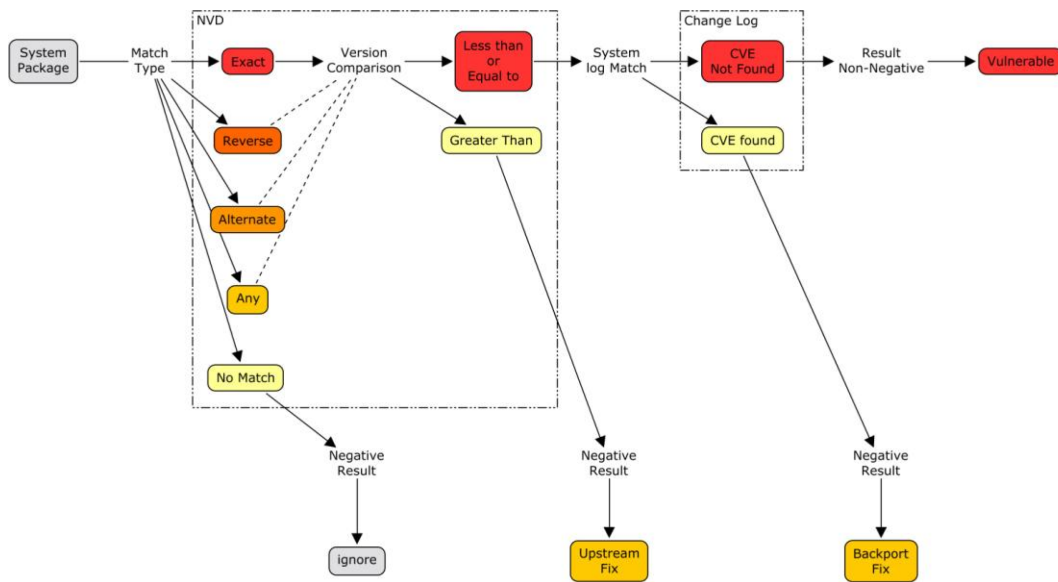

Figure 16 adding the back-port check to the end of the analysis

3.3.7 Verification of matching function accuracy

The aim of this work is to determine if it is possible to detect vulnerabilities in a fully "patched" system by using the information contained in the NVD. A secondary concern is to do this comparison by exclusively using information available on the system and the NVD. This secondary exercise is to

determine if an analysis of this kind is possible, independent of the determination by the vendor of the system, i.e. the Linux distribution that provides the system.

External verification is required to determine accuracy of such an independent analysis. One method for verification is to compare each non-negative result produced by the matching function against the list of packages the system itself determines are vulnerable. Of course, this verification is impractical if done "by hand" through the systematic examination of each vulnerability and each NVD entry.

To avoid this impracticality, the Debian system tools `apt-get`, `apt-show-versions`, and the security tool `debsecan` partially automate the verification. The system tools simply check if there are updates available on the update mirror, the tool `debsecan` compares the installed system packages to the vulnerability notes of the Debian security team.

4. Results of testing the matching function

Using the heuristics described in this work, an automated matching function was able to detect publically disclosed vulnerabilities within fully patched Linux systems. The results produced by the matching function consistently indicated 5%-12% more vulnerabilities than the tool *`debsecan`* (**Table 16**).

		Matching Function				Debsecan				
Year	Packages	Matched		Vulnerable		Matched		Vulnerable		Δ
2008*	402	55	12%	47	10%	6%	n/a %	16	4%	6%
2007	402	90	22%	68	17%	7%	n/a %	39	10%	7%
2006	402	73	18%	52	13%	12%	n/a %	4	1%	12%
2005	402	85	21%	49	12%	12%	n/a %	1	0%	12%
2004	402	93	23%	36	9%	9%	n/a %	0	0%	9%
2003	402	59	15%	22	5%	5%	n/a %	0	0%	5%
2002	402	108	27%	32	8%	n/a	n/a %	0	0%	8%

Table 16 current system vulnerabilities reported by the matching function and `debsecan`, by year, analysis as of April 14 2008. *detail results for 2008 in Figure 13

To compare these tools further, results for the year 2008 are show in **Figure 17**. Note the matching function detects ***all*** the vulnerable packages detected by debsecan, several *false positives*, and several that debsecan *misses*. **Sections 4.1.1** and **4.1.2** describe each numerated portion of this figure in detail.

Each numbered portion of **Figure 17** has text description consisting of:

a) Synopsis of match accuracy

b) Packages names involved

c) CVE numbers involved

d) Analysis

Debsecan	Match Function	Callout
apache2	apache2	
apache2.2-common	apache2.2-common	
apache2-mpm-prefork	apache2-mpm-prefork	
apache2-utils	apache2-utils	
apache-common	apache-common	
bzip2	bzip2	
gallery2		1
	cpp-4.1	2
	gcc	2
	gcc-3.3-base	2
	gcc-3.4-base	2
	gcc-4.1	2
	gcc-4.1-base	2
	gnupg	3
	gpgv	3
imagemagick	imagemagick	
	libapache2-mod-php4	4
	libapache2-svn	5
libapache-mod-perl	libapache-mod-perl	
	libapache-mod-php4	4
libbz2-1.0	libbz2-1.0	
	libg2c0	2
	libgcc1	2
	libgfortran1	2
	libldap2	6
libmagick9	libmagick9	
	libnetpbm10	7
	libssp0	2
	libstdc++6	2
	libtorrent9	8
linux-image-2.6.18-6-686	linux-image-2.6.18-6-686	
	linux-image-2.6-686	9
	netpbm	7
openssh-client	openssh-client	
openssh-server	openssh-server	
	php4	4
	php4-cgi	4
	php4-cli	4
	php4-common	4
	php4-mysql	4
phpmyadmin	phpmyadmin	
	python	10
	python2.4	10
	python2.4-minimal	10
	python-apt	10
	python-support	10
ssh	ssh	

Figure 17 results diff between debsecan and the match function. No highlighting indicates both tools detected the same vulnerability, gray indicates name not detected, yellow indicates an additional match. Enumerated callouts discussed in the text.

4.1 Items only reported by debsecan

1. <u>debsecan false-positive</u>

   Package:

   ```
   Gallery2
   ```

   - CVE-2008-1066 (The modifier.regex_replace.php plugin in Smarty before 2.6.19, as used by Serendipity (S9Y) and other products,...)

   Analysis:

   The comparison shows the matching function does not detect the vulnerable package "gallery2" which is assigned the vulnerability entry CVE-2008-1066. However, this CVE is for the package "Smarty", not "Gallery2" according to the NVD and confirmed by the Debian security advisory DSA-1520. This vulnerability is an example of a composition issue similar to the one described in **Section 3.3.5** but in contrast, the composition not present in the NVD. Otherwise, components affected by the smarty vulnerability, such as the package gallery2, would be part of CVE-2008-1066. Due to the stipulation that the contents of the NVD determine vulnerability status, not the vendor, this result is a false positive. Regardless, the tool debsecan, which relies upon vendor information, has correctly indicated that the package Gallery2 is vulnerable.

4.2 Items only reported by the Match Function

2. Match Function false-positive

Packages provided by the GNU Compiler Collection:

```
cpp-4.1          4.1.1
gcc
gcc-3.3-base     3.3.6
gcc-3.4-base
gcc-4.1
gcc-4.1-base
libg2c0
libgcc1
libgfortran1
libssp0
libstdc++6
```

Two CVEs are reported by the Match Function:

- CVE-2008-1685 (gcc 4.2.0 through 4.3.0 in GNU Compiler Collection, when casts are not ...)
- CVE-2008-1367 (gcc 4.3.x does not generate a cld instruction while compiling ...)

Analysis:

Both of these vulnerabilities are for specific versions. The matching function determines vulnerabilities by the highest version found in the NVD (**Section 3.2.1.2**); therefore, by these heuristics the GCC packages receive a vulnerable label, even when the NVD does not indicate they are.

3. Match Function false-positive

Packages provided by gpg:

```
gnupg            1.4.6
gpgv             1.4.6
```

- CVE-2008-1530 (GnuPG (gpg) 1.4.8 and 2.0.8 allows remote attackers to cause a denial ...)

Analysis:

This vulnerability is for a specific version. The matching function determines vulnerabilities by the highest version found in the NVD (**Section 3.2.1.2)**; therefore, by these heuristics GnuPG and gpgv receive a vulnerable label, when debsecan does not indicate they are.

4. Vulnerabilities not reported by debsecan

   -also-

   Match Function false-positive

   Packages provided by php4:

```
libapache2-mod-php  4.4.4
libapache-mod-php4  4.4.4
php4                4.4.4
php4-cgi            4.4.4
php4-cli            4.4.4
php4-common         4.4.4
php4-mysql          4.4.4
```

- CVE-2008-1384 (Integer overflow in PHP 5.2.5 and earlier allows context-dependent ...)
- CVE-2008-0145 (Unspecified vulnerability in glob in PHP before 4.4.8, when ...)

Package:

```
mysql         5.0.23
```

- CVE-2008-0226 (Multiple buffer overflows in yaSSL 1.7.5 and earlier, as used in MySQL ...)

Analysis:

The CVE indicates the PHP vulnerability is for all versions preceding a specific version. The upstream version represented by the system is less than both 5.2.5 and 4.4.8. In addition, both CVE numbers are not within the package changelogs. Therefore, by these heuristics the packages provided by php4 are vulnerable.

The detection of the mysql vulnerability is due to the decomposition of the package name. The NVD contains a CVE for a product named “mysql”, however there is no version given. The match function will attempt to discover if the CVE number is present in the changelog for the reported package, in this case mysql which is not found on the system. MySQL has multiple components on a system: mysql-common, mysql-server, and mysql-client. A mapping between the NVD package name and the system package name is needed to resolve this false positive.

5. Match Function false-positive

Package:

libapache2-svn	1.4.2

- CVE-2008-0005 (mod_proxy_ftp in Apache 2.2.x before 2.2.7-dev, 2.0.x before ...)
- CVE-2008-0455 (Cross-site scripting (XSS) vulnerability in the mod_negotiation module ...)
- CVE-2008-0456 (CRLF injection vulnerability in the mod_negotiation module in the ...)

Analysis:

The package libapache2-svn extends the functionality of the Apache HTTP Server, and to three of the Web Servers vulnerabilities.

6. Match Function false-positive

Package:

libldap2 2.1.30

- CVE-2008-0658 (slapd/back-bdb/modrdn.c in the BDB backend for slapd in OpenLDAP ...)

Analysis:

This vulnerability is for a specific version. The matching function determines vulnerabilities by the highest version found in the NVD (**Section 3.2.1.2**); therefore, by these heuristics libldap2 receive a vulnerable label, when debsecan does not indicate so.

7. Vulnerabilities not reported by debsecan

Packages provided by netpbm:

libnetpbm10	10.000
netpbm	10.000

- CVE-2008-0554 (Buffer overflow in the readImageData function in giftopnm.c in netpbm ...)

Analysis:

This vulnerability is for all versions preceding a specific version. The upstream version represented by the system is less than 10.26. In addition, the CVE

number is not within the package changelogs. Therefore, by these heuristics the packages provided by netpbm are vulnerable.

8. Vulnerability not reported by debsecan

Package:

libtorrent9 0.10.4-1

- CVE-2008-0646 (The bdecode_recursive function in include/libtorrent/bencode.hpp in Rasterbar Software libtorrent before 0.12.1, as used in Deluge before 0.5.8.3 and other products…)

Analysis:

This vulnerability is for all versions preceding a specific version. The upstream version represented by the system is less than 0.12.1. In addition, the CVE number is not within the package changelogs. Therefore, by these heuristics the package libtorrent9 is vulnerable. Why debsecan does not detect this vulnerability is unclear, perhaps it is because the package libtorrent9 is sourced by the package libtorrent which is not on the system.

9. Virtual Package

Analysis:

The Match function reports two vulnerable components of the latest kernel 2.6.18 within Debian Etch. The package linux-image-2.6-686 is a virtual package[23], not reported by `debsecan`, and not be reported by the match function.

Vulnerabilities not reported by debsecan

-also-

Match Function false-positive

Packages provided by python:

```
python                2.4.4
python2.4             2.4.4
python2.4-minimal     2.4.4
python-support        0.5.6
python-apt            0.6.19
```

- CVE-2008-1721 (Integer signedness error in the zlib extension module in Python 2.5.2 and earlier allows remote attackers to execute arbitrary…)

Analysis:

This vulnerability is for all versions preceding a specific version. The upstream version represented by the system is less than 2.5.2. In addition, the CVE number is not within the package changelogs. Therefore, by these heuristics the packages `python, python2.4, and python2.4-minimal` are vulnerable.

The packages `python-support` and `python-apt` should not be reported vulnerable. They match because the normalization productions decomposed their names to match with python, although they are not part of the python package.

4.3 Testing the entire system

Using the entire NVD data set, an automated matching function matched publically disclosed vulnerability within updated Linux systems. A summary of the vulnerabilities discovered in a fully patched system matched against NVD files between 2002 and 2008 is in **Table 17**. Additional detailed analysis were not performed, but is expected be similar to that of the details of the 2008 analysis.

Year	Matching CVEs	Unfixed Vulnerabilities		Not for us		Back-port fix		Upstream fix	
2008	105	80	76%	1	1%	9	9%	15	14%
2007	1319	883	67%	62	5%	233	18%	141	11%
2006	749	150	20%	14	2%	71	9%	514	69%
2005	649	61	9%	0	0%	42	6%	546	84%
2004	651	104	16%	4	1%	39	6%	504	77%
2003	390	57	15%	6	2%	2	1%	325	83%
2002	1234	49	4%	159	13%	2	0%	1024	83%

Table 17 System vulnerabilities, as detected by the matching function, categories by year

Several interesting trends are present in **Table 17**. First, there is an inverse-relationship between back-port fixes and upstream fixes. This fits our understanding of the Open Source security model: The maintainers do not wait for the upstream security fix to arrive through the normal distribution update schedule. Instead, security fixes are back-ported in order to secure the package and to preserve the stability of the package. We see this by noting that recent years have many more back-port fixes, earlier years have more upstream fixes. Second, there exists public vulnerabilities in all years analyzed. The increased amount of vulnerabilities that the Debian security team labels as “NOT-FOR-US” in the file nvd-cve-2002.txt is due to the number of CVEs that predate the Debian security tracker.

5. Conclusion

This work investigates the feasibility of using a vendor-independent vulnerability data source such as the NVD to determine whether vulnerabilities exist within fully patched and "up-to-date" Open Source computer systems. This method discovered a set of vulnerable packages on fully patched systems. A comparison of these results examined their content verses two other result sets produced by independent tools. These comparison result sets used vendor-specific data; one set by the Debian tool `debsecan`, and the other set by the package update manager *`apt`* using Debian update mirrors.

The results differ between all three methods. First, a fully up-to-date system, as reported by the update manager `apt`, yields apparent vulnerabilities when analyzed by either the matching function, or the vendor-specific tool `debsecan`. Therefore, it is possible to demonstrate the presence of un-patched vulnerabilities in fully updated system, proving it is a fallacy to assume an up-to-date or fully patched system will also be vulnerability-free.

This work also attempted to automate the matching process. This portion of the thesis was partially successful, limited by inconsistencies and absence of critical information within the NVD.

The tool `debsecan` produced fewer false positives than the method used in this thesis but is not able to perform a complete analysis of a system without domain-specific information. Therefore, although a partially automated analysis is feasible, it is not possible to automate a complete and accurate result by

comparing the information contained in the NVD and that of the packages present on an Open Source computer system.

Conversely, the matching function was able to bypass the latency required to generate the domain-specific analysis. The matching function detected vulnerabilities in the system impossible to detect by `debsecan` due to the Debian Security Team having not finished the analysis of the CVE entry.

The information presented in this thesis can only represent a precise snapshot in time. All of the data sources relied upon to generate the analysis experience ongoing changes, which affect their outcome. The NVD changes daily, new CVE entries appear and existing entries modified. Information produced by the Debian Security Team also changes daily, if not hourly. Mirror sites reflect ongoing updates and releases of Open Source packages with new releases, back-ports, and updates.

Significant issues exist within the NVD regarding the presentation of Open Source vulnerability information. These issues impair the accuracy of conclusions drawn from NVD data, such as the numerous vulnerability comparisons between closed and open source software[24].

5.1 Recommendations for the NVD

One great value of the NVD and the CVE entries therein, is that of aggregation; divergent vulnerability information is associated and assigned a common identifier to resolve synonymous data. This standardization facilitates human communication, and greatly enables the interoperability of automated

tools. Unfortunately, the identification of vulnerabilities requires *two* identified components: the vulnerably *and* the software. Unfortunately, the NVD does not sufficiently support the identification of software, and consequently the value of NVD suffers. Furthermore, if users of the NVD do not accommodate these shortcomings, then too the conclusions drawn from NVD data also suffer.

The question this work answers is whether publically known vulnerabilities exist within fully updated OSS systems. During the process of resolving this question, many issues within the NVD became apparent. These issues were significant enough to limit the effectiveness of a determination of vulnerability on Open Source systems. Many accommodations were then included into the software matching heuristics of this work that increased the precision of the result. However, these accommodations do not represent a complete solution. Matching software techniques have a limited ability to resolve the data issues within the NVD, and certainly not to a sufficient extent as to make the NVD robust enough to support complete vulnerability matching. For complete resolution, the NVD must adopt a data model that includes the means to identify vulnerable software.

This work has noted many shortcomings in the NVD while attempting to identify vulnerable software; the intent of this last section is to provide insight into the resolution of some of these issues.

5.2 Resolutions for the NVD

The solution organization is in two categories:

1) Resolution of consistency in the format and conventions within NVD entries

2) Resolution of ambiguous matching of vulnerable OSS

5.2.1 Resolutions for the NVD consistency problems

In order for the NVD to provide unambiguous name matching, a single name it must reference different vulnerabilities of the same software with the same name. As an example of currently there are 90 CVE entries with a product name of "Apache" and 39 entries with the product name of "Apache HTTP Server". This is a problem of data normalization, and the solution is to choose one name or the other. In this case, the name "Apache HTTP Server" is more precise.

NVD entries provide limited information about the vulnerable software itself. A description of the vulnerable software can help to determine which package the vulnerability affects. Such an description may be from the software vendor itself:

> *Apache HTTP Server*
> *Versatile, high-performance HTTP server The most popular server in the world, Apache features a modular design and supports dynamic selection of extension modules at runtime. Some of its strong points are its range of possible customization, dynamic adjustment of the number of server processes, and a whole range of available modules including many authentication mechanisms, server-parsed HTML, server-side includes, access control, CERN httpd metafiles emulation, proxy caching, etc. Apache also supports multiple virtual homing. . Separate packages are*

available for PHP, mod_perl, Java Servlet support, Apache-SSL, and other common extensions. More information is available at http://www.apache.org/.

Resolving match ambiguities due to treating the names of the same software entities with different font cases can also be resolved through data normalization. The case of the names needs also to be consistent; referring to the Gnu Compiler Collection as both “GCC” and “gcc” raises the question of whether these names refer to the same software entity or not. If the NVD does not use case to identify software, then it should at least consistently refer to software with the same case. However, not using case disallows the identification between software names that depend upon case, such as discerning the differences between Ant (automated software build tool) and ANT (desktop ISDN telephony application) and therefore is not a good idea.

Resolving match ambiguities due to granularity inconsistencies will require the NVD entries to use a consistent naming convention. Many complex software solutions have components. Devising a consistent means to identify these components is important to resolve the ambiguities currently in the NVD. **Table 15** lists some of the “granularity” inconsistencies within the names of such a system, the Apache HTTP Server. This is a difficult problem to solve, as it may be important to provide vulnerability information regarding both the software component, e.g. “`mod_ssl`” and the software composition, “`Apache HTTP Server`”. The resolution for this issue is within **Section 5.2.2**

5.2.2 Resolutions for matching problems with NVD names and versions

After the inconsistencies within the NVD names and conventions are resolved there will still problems matching the names and the versions of vulnerable software. One of these problems is due to the same software entities having different names within different domains. The good example being that the name for the Apache HTTP Server is "`apache`" on Debian Linux systems and "`httpd`" on Red Hat Linux systems.

The problem of mapping various name synonyms, originating from different domains, to a single identifier is the issue that the NVD was created to solve. This same solution, applied to software names, will resolve many of the matching issues this work uncovered.

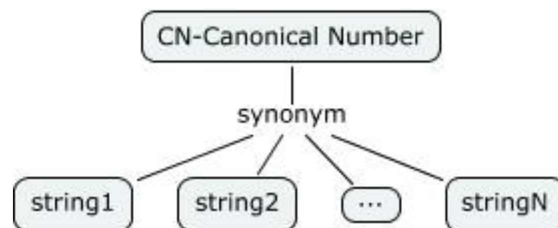

Figure 18 Ontology can help resolve naming problems due to different names

A unique identifier, called a Canonical Number (CN) allows unambiguous matches of case-sensitive names (**Figure 18**). Adding the number CN-1234-5678 to an Ant vulnerability to allows Ant (automated software build tool) to match, while not matching the number CN-0123-4567 for ANT (desktop ISDN telephony application). Moreover, it also resolves ambiguities caused by various Linux Distributions naming the same software by different names.

The number CN-1762-5678 can refer to both "apache", "httpd", and to "Apache HTTP Server" With the same result, names with adjectives added will resolve without the complexity of the adjective removing production upon which this work relies. "libapache-mod-ssl" (Debian name), "mod_ssl" (project name) and "Mod_ssl" (NVD name) can all resolve to CN-2587-4750.

The examples so far are recognizable matches by human review. A canonical number will also allow matching between entities that would otherwise be un-matchable even by a human. The series of BIND vulnerabilities shown in **Figure 13** requires the tool to use package metadata to conclude six other components are part of the BIND package. Ontology with information regarding the composition of packages, their components and dependencies, will allow tools to traverse these mappings to find matches un-readable by human inspection.

Packages are often a component part of a larger system (e.g. Apache and Apache modules). One form of canonicalization involves the understanding that an Apache module is "part of" an Apache installation. This enables the match to discover vulnerabilities where direct name matching will fail.

Resolving match ambiguities due to major release name and version matching still needs to be addressed for a different reason, that of the fact that major releases represent vastly different bodies of code. The simple solution is to delineate the major releases by giving them a product name of their own, just as successive major releases of windows receives different entries. The NVD currently has product entries for Windows 3.51, Windows 95, Windows 98,

Windows NT, Windows 2000, Windows ME, Windows XP, and Windows Vista. Similarly, there should be product names for PHP 4, PHP 5 and the major releases of Perl, and the Apache HTTP Server, etc. This will allow a specific major release to match without results from other major releases confusing the match results and other sets of versions. An example for PHP:

```
<vuln_soft>
  <prod name="PHP4" vendor="PHP">
    <vers num="4.4.7" prev="1" />
  </prod>
  <prod name="PHP5" vendor="PHP">
    <vers num="5.2.3" prev="1" />
  </prod>
</vuln_soft>
```

The attempt to present a set of vulnerable versions in a CVE has produced errors as noted before (**Section 3.3.4**). This resolution follows the method in this work, which is to present only the highest known vulnerable version. In this, way both the data consistency and the comparison by a matching tool will be simpler and less prone to error. Software with major releases can still share the same vulnerabilities, which requires a maximum vulnerable version listing in each.

Some of the responsibility for version ambiguities rests on the shoulders of the Distributions. The resolution of this issue is the one used in this thesis. Debian, for the most part, is a good example of a Linux Distribution whose changelogs contain machine-readable annotations regarding security fixes. Other Linux Distributions can replicate this example and therefore back-ported security patches will be visible to automated tools.

6. References

[0] *Network Admission Control*. Cisco Systems, Inc.. Retrieved April 12, 2008, from http://www.cisco.com/go/nac

[1] *Open Vulnerability Assessment Language*. The MITRE Corporation. Retrieved June 20, 2008, from http://oval.mitre.org/index.html

[2]*The Information Security Automation Program*. NIST, OSD, DHS, NSA, and DISA. Retrieved June 20, 2008, from http://nvd.nist.gov/scap.cfm

[3] *The Security Content Automation Protocol*. NIST, OSD, DHS, NSA, and DISA. Retrieved June 20, 2008, from http://nvd.nist.gov/scap.cfm

[4] *Trusted Network Connect Work Group*. Trusted Computing Group. Retrieved Feb 15, 2008, from https://www.trustedcomputinggroup.org/groups/network/

[5] Sangster P., et al. (2008). *RFC 5209 - Network Endpoint Assessment (NEA): Overview and Requirements*. IETF Network Working Group . Retrieved June, 2008, from http://www.ietf.org/rfc/rfc5209.txt

[6] *National Vulnerability Database*. Version 2.1. (2008). US-CERT. Retrieved June 30, 2008, from http://nvd.nist.gov/

[7] *NVD Frequently Asked Questions*. NIST Computer Security Division. Retrieved June 15, 2008, from http://nvd.nist.gov/faq.cfm

[8] Pitt M., Garbee B., Schilling J. et al. (2005). *Debian Bug report log - #328228 tar should warn when extracting setuid/setgid files*. Debian. Retrieved May 11, 2008, from http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=328228

[9] Alhazmi O. H., Malaiya I. K., Ray I. (2006). Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security 25*(7), 1-10. Retrieved November 24, 2007, from www.elsevier.com/locate/cose

[10] J. Howard, *An Analysis Of Security Incidents On The Internet: 1989–1995*. Carnegie-Mellon University, April 1997.

[11] Pescatore , J. (2008). *OpenSSL Vulnerability Shows Open-Source Process Weaknesses*. Gartner, Inc. Retrieved 22 May, 2008 , from http://www.gartner.com/DisplayDocument?doc_cd=158357

[12] Balle, J. (2007). *28% of all detected applications are insecure*. Secunia . Retrieved May 8, 2008, from http://secunia.com/blog/11/

[13] (2007). *Secunia PSI - Personal Software Inspector v0.9.0.4* [Computer Software]. Secunia .

[14] Testing Security Team. (2008). *Vulnerable source packages in the stable suite*. Debian. Retrieved June 30, 2008, from http://security-tracker.debian.net/tracker/status/release/stable

[15] (2008). *Automatically generated issue names*. Debian. Retrieved June 30, 2008, from http://security-tracker.debian.net/tracker/data/fake-names

[16] (2008). *Ubuntu Security Notice USN-612-1* . Canonical Ltd.. Retrieved May 15, 2008, from http://www.ubuntu.com/usn/usn-612-1

[17] (2005). *Kernel Space Definition*. Bellevue Linux. Retrieved Feb 22, 2008, from http://www.bellevuelinux.org/kernel_space.html

[18] (2008). *Browser Statistics*. W3Schools . Retrieved from http://www.w3schools.com/browsers/browsers_stats.asp

[19] Eric S. Raymond (1999). *The Cathedral & the Bazaar*. O'Reilly. ISBN 1-56592-724-9.

[20] (2008). *June 2008 Web Server Survey*. Netcraft Ltd. Retrieved June 10, 2008, from http://news.netcraft.com/archives/2008/06/22/june_2008_web_server_survey.html

[21] (2008). *Berkeley Internet Name Domain (BIND)*. Internet Systems Consortium, Inc. Retrieved June 30, 2008, from http://www.isc.org/index.pl?/sw/bind/index.php

[22] Barth A., Di Carlo A., Hertzog R., Schwarz C., and Jackson J.. (2007). *Debian Developer's Reference - Chapter 5 - Managing Packages*. Developer's Reference Team. Retrieved June 30, 2008, from http://www.debian.org/doc/developers-reference/ch-pkgs.en.html

[23] (2007). *Debian Developer's Reference - Chapter 3 - Managing Packages*. Developer's Reference Team. Retrieved June 30, 2008, from http://www.debian.org/doc/debian-policy/ch-binary.html

[24] Shaver. (2007). *counting still easy, critical thinking still surprisingly hard*. Retrieved June 17, 2008, from http://shaver.off.net/diary/2007/11/30/counting-still-easy-critical-thinking-still-surprisingly-hard/

Appendix I,

Tables Used for Calculating the Number of Packages

Represented by Open Source Distributions

Name	Releases
64 Studio	2
aLinux	13
ALT Linux	4
Annvix	2
Arch Linux	2
Ark Linux	2
Arudius	2
Asianux	3
Aurox	12
BLAG	7
Caixa Mágica	2
José Guimarães	2
Paulo Trezentos	2
CentOS	5
CRUX	3
Damn Small Linux	5
Debian	6
Desktop Light Linux	2
DeMuDi	2
dyne:bolic	3
Elive	1
EnGarde Secure Linux	2
Fedora	8
Finnix	10
Foresight Linux	2
Fox Linux	1
Frugalware	2
Gentoo Linux	2
gNewSense	2
Gnoppix	3
gnuLinEx	2
GoboLinux	13
Impi Linux	7
Kanotix	2
Knoppix	6
KnoppMyth	2
Kurumin Linux	7
Linspire	6
Freespire	2
Linux Mint	4
Lunar Linux	2
Mandriva Linux	2
MontaVista Linux	5
Musix GNU+Linux	2
Myah OS	3
NimbleX	2
Nitix	5
Open Enterprise Server	2
openSUSE	11
Paipix	8
Pardus	2
Parsix	2
PCLinuxOS	2
PLD Linux Distribution	2
Pie Box Enterprise Linux	5
Puppy Linux	4
QiLinux	2
Red Flag Linux	5
Red Hat Enterprise Linux	6
Red Hat Linux	9
Rxart	2
Sabayon Linux	4
Satux	2
Scientific Linux	6
sidux	2
SimplyMEPIS	7
Slackware	12
SLAX	2
Source Mage GNU/Linux	2
SUSE Linux	10
Symphony OS	2
Trustix Secure Linux	4
Ubuntu	8
Kubuntu	8
Xubuntu	8
Edubuntu	8
Gobuntu	8
Ututo	2
VectorLinux	6
Xandros	4
Yoper	4
Zenwalk Linux	5
Average	4.41

Table 1 numbers of releases for various Linux distributions, and the average of this dataset.

Name	Architectures
64 Studio	2
aLinux	2
ALT Linux	2
Annvix	2
Arch Linux	2
Ark Linux	1
Arudius	1
Asianux	5
Aurox	1
BLAG Linux	1
Caixa Mágica	3
CentOS	3
CRUX	1
Damn Small Linux	1
Debian	12
DeLi Linux	1
DeMuDi	1
dyne:bolic	1
Elive	1
EnGarde Secure	2
Fedora	3
Finnix	4
Foresight Linux	1
Frugalware	2
Gentoo	13
gnuLinEx	2
GoboLinux	2
Impi Linux	2
Kanotix	2
Knoppix	1
Kurumin Linux	1
Linspire	1
Linux Mint	1
Lunar Linux	2
Mandriva Linux	3
MEPIS	2
Musix GNU+Linux	1
Myah OS	1
NimbleX	1
Nitix	1
OES2-Linux	3
openSUSE	7
Paipix	2
Pardus	1
Parsix	2
PCLinuxOS	1
Pie Box Enterprise	1
PLD Linux	4
Puppy Linux	1
QiLinux	1
Red Flag Linux	4
Red Hat	5
Rxart Desktop	2
Sabayon Linux	2
Satux	1
Scientific Linux	3
sidux	2
Slackware	1
Slax	1
Source Mage	3
SUSE Linux	7
Symphony OS	1
Ubuntu	3
UTUTO GNU/Linux	2
VectorLinux	1
Xandros Desktop OS	1
Yellow Dog Linux	2
Yoper	1
Zenwalk Linux	1
Xubuntu	3
Edubuntu	3
Kubuntu	3
Average	2.306

Table 2 Numbers of architectures supported by various Linux distributions, and the average of this dataset.

Name	Packages
aLinux	1200
ALT Linux	7500
Arch Linux	15000
Ark Linux	4000
Aurox	3000
BLAG Linux and GNU	9000
Caixa Mágica	1155
CentOS	1660
CRUX	610
Damn Small Linux	26000
Debian	150
DeLi Linux	875
Elive	20000
EnGarde Secure Linux	500
Fedora	8000
Finnix	350
Foresight Linux	15000
Frugalware	3000
Gentoo	12000
gnuLinEx	200
GoboLinux	636
Kanotix	1200
Knoppix	3600
Linspire	2200
Linux Mint	20000
Lunar Linux	3120
Mandriva Linux	16000
MEPIS	20000
Musix GNU+Linux	1300
NimbleX	500
openSUSE	22000
Paipix	2000
Pardus	1600
PCLinuxOS	5025
Pie Box Enterprise Linux	1500
PLD Linux Distribution	13500
Puppy Linux	300
QiLinux	2500
Red Hat Enterprise Linux	3000
Rxart Desktop	5000
Sabayon Linux	12000
sidux	22950
Slackware	544
Slax	2050
Source Mage GNU/Linux	5514
SUSE Linux	22000
Ubuntu	23000
Xandros Desktop OS	5000
Yoper	2000
Xubuntu	23000
Edubuntu	23000
Kubuntu	23000
Average	8,043

Table 3 Various numbers of packages contained in different Linux Distributions, and the average of this dataset.

System	Distro	Release	Packages	Changes	CVE	Vuln/ Pkg	Chgs/ Pkg	Description
PTS	Ubuntu	Feisty	1357	112351	4379	3.23	82.79	Server, enterprise packages
Mahalo	Ubuntu	Feisty	2033	107514	3868	1.90	52.88	Workstation, user packages
Doc	Ubuntu	Feisty	319	22195	1276	4.00	69.58	Server, minimum install
Sleepy	Ubuntu	Feisty	331	24307	1611	4.87	73.44	Server, minimum install
GNU	Debian	Etch	749	49036	2509	3.35	65.47	Server, medium install
Weeber	Debian	Etch	403	27286	1372	3.40	67.71	Server, minimum install
Legstrong	Ubuntu	Feisty	1199	62949	3298	2.75	52.50	Workstation, user packages

Table 4 Total number of vulnerabilities and changes as reported within package changelogs, and the averages of each dataset.

System	Packages	Changes	CVE	Vulnerabilities / Package	Changes / Package
Ubuntu	1,048	65,863	2,886	3.35	66.24
Debian	576	38,161	1,941	3.38	66.59
Workstation	1,616	85,231	3,583	2.33	52.69
Server	631	47,035	2,229	3.77	71.80
Overall	913	57,948	2,616	3.36	66.34

Table 5 Average numbers of vulnerabilities and changes found on various systems.