The Beauty and the Beast
Vulnerabilities in Red Hat’s Packages

Stephan Neuhaus <Stephan.Neuhaus@disi.unitn.it>
Thomas Zimmermann <tzimmer@microsoft.com>
Vulnerabilities are important because fixing them costs a lot of money (2005 FBI study: 67 Bn $). There are 3241 packages (or were, by August 2008) offered by Red Hat. (There are certainly more being offered for Red Hat!)
Vulnerabilities are important because fixing them costs a lot of money (2005 FBI study: 67 Bn $). There are 3241 packages (or were, by August 2008) offered by Red Hat. (There are certainly more being offered for Red Hat!)
Vulnerabilities are important because fixing them costs a lot of money (2005 FBI study: 67 Bn $). There are 3241 packages (or were, by August 2008) offered by Red Hat. (There are certainly more being offered for Red Hat!)
Explain colours: white = no vulnerabilities, blue -> red: progressively more
Explain colours: white = no vulnerabilities, blue -> red: progressively more
<table>
<thead>
<tr>
<th>Advisory:</th>
<th>RHSA - 2006-0201 - 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>Security Advisory</td>
</tr>
<tr>
<td>Severity:</td>
<td>Important</td>
</tr>
<tr>
<td>Issued on:</td>
<td>2006-02-13</td>
</tr>
<tr>
<td>Last updated on:</td>
<td>2006-02-13</td>
</tr>
<tr>
<td>Affected Products:</td>
<td>Red Hat Desktop (v. 4)</td>
</tr>
<tr>
<td></td>
<td>Red Hat Enterprise Linux AS (v. 4)</td>
</tr>
<tr>
<td></td>
<td>Red Hat Enterprise Linux ES (v. 4)</td>
</tr>
<tr>
<td></td>
<td>Red Hat Enterprise Linux WS (v. 4)</td>
</tr>
<tr>
<td>OVAL:</td>
<td>N/A</td>
</tr>
<tr>
<td>CVEs (cve.mitre.org):</td>
<td>CVE-2006-0301</td>
</tr>
</tbody>
</table>

An updated xpdf package that fixes a buffer overflow security issue is now available.

This update has been rated as having important security impact by the Red Hat Security Response Team.

Explain colours: white = no vulnerabilities, blue -> red: progressively more
Explain colours: white = no vulnerabilities, blue -> red: progressively more
Explain colours: white = no vulnerabilities, blue -> red: progressively more
Explain colours: white = no vulnerabilities, blue -> red: progressively more
Distribution of RHSAs

2/3 of packages

Note logarithmic y-axis. 3241 packages in total, about 2/3 with no known vulnerabilities.
Properties of packages, not properties of the software in the package
Are there properties that correlate with vulnerabilities?

Properties of packages, not properties of the software in the package
Are there properties that correlate with vulnerabilities?

Are there properties that increase or decrease the risk?

Properties of packages, not properties of the software in the package
Are there properties that correlate with vulnerabilities?

Are there properties that increase or decrease the risk?

Can we predict whether a package contains unknown vulnerabilities?

Properties of packages, not properties of the software in the package
Are there properties that correlate with vulnerabilities?

Are there properties that increase or decrease the risk?

Can we predict whether a package contains unknown vulnerabilities?

Properties of packages, not properties of the software in the package
Are there properties that correlate with vulnerabilities?

Are there properties that increase or decrease the risk?

Can we predict whether a package contains unknown vulnerabilities?

- ✔ Dependencies
- ✔ Beauties and Beasts

Properties of packages, not properties of the software in the package
Are there properties that correlate with vulnerabilities?

Are there properties that increase or decrease the risk?

Can we predict whether a package contains unknown vulnerabilities?

- Dependencies
- Beauties and Beasts
- Machine Learning

Properties of packages, not properties of the software in the package.
Dependencies
Dependencies

amanda-server
Dependencies

amanda-server

↓

glibc
Dependencies

perl depends on readline

readline depends on amanda

amanda depends on glibc

glibc depends on amanda-server

amanda-server depends on grep

grep depends on gnuplot

Perl also depends on coreutils

Libtermcap depends on coreutils
Dependencies and Vulnerabilities

- Dependency A → B exists because A wants to use the services offered by B

- Vulnerability exists in A if
  - A is in an insecure domain (domains are characterised by dependencies)
  - B is insecure and fix in B spills over to A; or
  - B is difficult to use securely

Packages in same domain will tend to have same dependencies. Domain examples are: compilers, games, office applications,
Red Hat Dependencies
Distribution is apparently logarithmic with a long tail. This is not transitive closure. kdebase has 14 RHSAs (but 96 dependencies), kernel has 129 (but 0 dependencies), so number of dependencies is not a good predictor of number of RHSAs.
Are there properties that correlate with vulnerabilities? ✔

Are there properties that increase or decrease the risk? ✔

Can we predict whether a package contains unknown vulnerabilities? ✔
Where does the addition of dependencies significantly increase/decrease the risk?
Where does the addition of dependencies significantly increase/decrease the risk?

1. Data structure: concept lattice
Where does the addition of dependencies significantly increase/decrease the risk?

1. Data structure: concept lattice
2. Compute change in risk
Where does the addition of dependencies significantly increase/decrease the risk?

1. Data structure: concept lattice
2. Compute change in risk
3. Include only statistically significant changes
Step 1: Data Structure

Start with no knowledge about dependencies (top node contains all packages). Add knowledge of glibc (node contains all packages depending on glibc), then qt (node contains all packages depending on qt and glibc), then xorg-x11-libs (node contains all packages
Step 1: Data Structure

Start with no knowledge about dependencies (top node contains all packages). Add knowledge of glibc (node contains all packages depending on glibc), then qt (node contains all packages depending on qt and glibc), then xorg-x11-libs (node contains all packages depending on xorg-x11-libs and qt and glibc). Since we know the packages contained in each node, we can compute the probability of a package in this node being vulnerable.
Step 1: Data Structure

Block 1: All packages depending on glibc

Start with no knowledge about dependencies (top node contains all packages). Add knowledge of glibc (node contains all packages depending on glibc), then qt (node contains all packages depending on qt and glibc), then xorg-x11-libs (node contains all packages depending on xorg-x11-libs and qt and glibc). Since we know the packages contained in each node, we can compute the probability of a package in this node being vulnerable.
Step 1: Data Structure

```
∅   ighting structure (top node contains all packages). Add knowledge of glibc (node contains all packages depending on glibc), then qt (node contains all packages depending on qt and glibc), then xorg-x11-libs (node contains all packages depending on xorg-x11-libs and qt and glibc). Since we know the packages contained in each node, we can compute the probability of a package in this node being vulnerable.
```

Block 1: All packages depending on glibc
Step 1: Data Structure

Start with no knowledge about dependencies (top node contains all packages). Add knowledge of glibc (node contains all packages depending on glibc), then qt (node contains all packages depending on qt and glibc), then xorg-x11-libs (node contains all packages depending on xorg-x11-libs and qt and glibc). Since we know the packages contained in each node, we can compute the probability of a package in this node being vulnerable.
Step 1: Data Structure

Block 1: All packages depending on `glibc`
Block 2: All packages depending on `glibc`, `qt`
Block 3: All packages depending on `glibc`, `qt`, `xorg-x11-libs`

Start with no knowledge about dependencies (top node contains all packages). Add knowledge of glibc (node contains all packages depending on glibc), then qt (node contains all packages depending on qt and glibc), then xorg-x11-libs (node contains all packages depending on xorg-x11-libs and qt and glibc). Since we know the packages contained in each node, we can compute the probability of a package in this node being vulnerable.
Question: Is the rise of 43.9% when going from \{\text{glibc}\} to \{\text{glibc, qt}\} just some random fluctuation? We test this using statistical tests (Chi^2 or Fischer exact) and discard the “random fluctuation” hypothesis when the probability of such a increase happening by chance is 1% or less. So we expect that we wrongly attribute an increase to an actual effect 1% of the time.
Step 2: Compute Risk Change

Question: Is the rise of 43.9% when going from \{glibc\} to \{glibc, qt\} just some random fluctuation? We test this using statistical tests (Chi^2 or Fischer exact) and discard the “random fluctuation” hypothesis when the probability of such an increase happening by chance is 1% or less. So we expect that we wrongly attribute an increase to an actual effect 1% of the time.
Step 2: Compute Risk Change

Risk change by adding qt only when already dependent on glibc! (glibc is the context)

Question: Is the rise of 43.9% when going from \{glibc\} to \{glibc, qt\} just some random fluctuation? We test this using statistical tests (Chi^2 or Fischer exact) and discard the “random fluctuation” hypothesis when the probability of such an increase happening by chance is 1% or less. So we expect that we wrongly attribute an increase to an actual effect 1% of the time.
Step 3: Include Only Significant Changes

- Risk changes with significance $p < 0.01$
- No significant and more general context exists for this dependency
- Risk goes up: “beast”
- Risk goes down: “beauty”
## Selected Beasts

The complete list can be found in the paper

<table>
<thead>
<tr>
<th>Context</th>
<th>Dependency</th>
<th>Risk before</th>
<th>Risk after</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td>openoffice.org-core</td>
<td>0.329</td>
<td>1.000</td>
<td>0.671</td>
</tr>
<tr>
<td>∅</td>
<td>kdelibs</td>
<td>0.329</td>
<td>0.856</td>
<td>0.527</td>
</tr>
<tr>
<td>∅</td>
<td>cups-libs</td>
<td>0.329</td>
<td>0.774</td>
<td>0.445</td>
</tr>
<tr>
<td>∅</td>
<td>libmng</td>
<td>0.329</td>
<td>0.769</td>
<td>0.440</td>
</tr>
<tr>
<td>glibc</td>
<td>qt</td>
<td>0.335</td>
<td>0.774</td>
<td>0.439</td>
</tr>
<tr>
<td>glibc</td>
<td>krb5-libs</td>
<td>0.335</td>
<td>0.769</td>
<td>0.434</td>
</tr>
</tbody>
</table>

Explain packages, don’t just list names
## Selected Beauties

The complete list can be found in the paper

<table>
<thead>
<tr>
<th>Context</th>
<th>Dependency</th>
<th>Risk before</th>
<th>Risk after</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>glibc</td>
<td>xorg-x11-server-Xorg</td>
<td>0.335</td>
<td>0.015</td>
<td>-0.320</td>
</tr>
<tr>
<td>compat-glibc, glibc, zlib</td>
<td>audiofile</td>
<td>0.613</td>
<td>0.359</td>
<td>-0.254</td>
</tr>
<tr>
<td>glibc, glibc-debug, zlib</td>
<td>audiofile</td>
<td>0.590</td>
<td>0.351</td>
<td>-0.239</td>
</tr>
<tr>
<td>∅</td>
<td>gnome-keyring</td>
<td>0.329</td>
<td>0.101</td>
<td>-0.228</td>
</tr>
<tr>
<td>glibc, zlib</td>
<td>gnome-libs</td>
<td>0.456</td>
<td>0.281</td>
<td>-0.175</td>
</tr>
<tr>
<td>∅</td>
<td>python</td>
<td>0.329</td>
<td>0.132</td>
<td>-0.197</td>
</tr>
</tbody>
</table>

Explain possible consequences: new applications: choose less risky dependencies
Are there properties that correlate with vulnerabilities?

Are there properties that increase or decrease the risk?

Can we predict whether a package contains unknown vulnerabilities?

- Dependencies
- Beauties and Beasts
- Machine Learning
Is it possible to predict...

• from the dependencies *which packages are vulnerable* (classification)?

• *which packages* will have the *most vulnerabilities* (ranking)?
Experiment

Repeat 50x
This “self-testing” is a standard evaluation technique for machine learning methods
Repeat 50x
This “self-testing” is a standard evaluation technique for machine learning methods
Repeat 50x
This “self-testing” is a standard evaluation technique for machine learning methods
Repeat 50x
This “self-testing” is a standard evaluation technique for machine learning methods
This “self-testing” is a standard evaluation technique for machine learning methods.

Repeat 50x
Indicators

Don’t mention –1. We want values near 1.
Indicators

Classification

Don’t mention –1. We want values near 1.
**Indicators**

**Classification**

\[
\text{precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}
\]

\[
\text{recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}
\]

Don’t mention –1. We want values near 1.
Indicators

**Classification**

\[
\text{precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}
\]

\[
\text{recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}
\]

Don’t mention –1. We want values near 1.
Indicators

**Classification**

\[ \text{precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}} \]

\[ \text{recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} \]

**Ranking**

Don’t mention –1. We want values near 1.
Indicators

Precision = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}

Recall = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}

Don’t mention –1. We want values near 1.
Indicators

precision = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}

recall = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}

Don’t mention −1. We want values near 1.
Indicators

Precision = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}

Recall = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}

Don’t mention –1. We want values near 1.
Indicators

precision = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}

recall = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}

Don’t mention –1. We want values near 1.
Results of 50 random splits: train with 2/3 of the packages, predict with the rest, record precision and recall.
Results of 50 random splits: train with 2/3 of the packages, predict with the rest, record precision and recall.
Precision versus Recall

Results of 50 random splits: train with 2/3 of the packages, predict with the rest, record precision and recall.
Results of 50 random splits: train with 2/3 of the packages, predict with the rest, record precision and recall.
Results of 50 random splits: train with 2/3 of the packages, predict with the rest, record precision and recall.

Decision Trees worse than SVMs
Precision versus Recall

Results of 50 random splits: train with 2/3 of the packages, predict with the rest, record precision and recall.

Decision Trees worse than SVMs
Results of 50 random splits: train with 2/3 of the packages, predict with the rest, record precision and recall.

- Predictions are correct 83% of the time.
- 65% of all vulnerable packages predicted.
Even though “self-evaluation” is a standard technique, what we really want to know is if the method is able to predict the future... (next slide)
January 1, 2008 to August 31, 2008

Predict
Top 25 out of 2181

Package Name
- mod_php
- php-dbg
- php-dbg-server
- perl-DBD-Pg
- kudzu
- irda-utils
- hpoj
- libbdevid-python
- mrtg
- evolution28-evolution-data-server
- lilo
- ckermit
- dovecot
- kde2-compat
- gq
- vorbis-tools
- k3b
- taskjuggler
- ddd
- tora
- libpurple
- libwvstreams
- pidgin
- linuxwacom
- policycoreutils-newrole
- ...

2156 further packages

Evaluate
73 new vulnerable
Top 25 out of 2181

- mod_php
- php-dbg
- php-dbg-server
- perl-DBD-Pg
- kudzu
- irda-utils
- hpoj
- libbdevid-python
- mrtg
- evolution28-evolution-data-server
- lilo
- ckermit
- dovecot
- kde2-compat
- gq
- vorbis-tools
- k3b
- taskjuggler
- ddd
- tora
- libpurple
- libwvstreams
- pidgin
- linuxwacom
- policycoreutils-newrole

2156 further packages

73 new vulnerable
January 1, 2008

Predict

Top 25 out of 2181

Package Name

- mod_php
- php-dbg
- php-dbg-server
- perl-DBD-Pg
- kudzu
- irda-utils
- hpoj
- libbdevid-python
- mrtg
- evolution28-evolution-data-server
- lilo
- ckermit
- dovecot
- kde2-compat
- gq
- vorbis-tools
- k3b
- taskjuggler
- ddd
- tora
- libpurple
- libwvstreams
- pidgin
- linuxwacom
- policycoreutils-newrole

... 2156 further packages

Evaluate

73 new vulnerable

Patch published 2009-05-12

August 31, 2008
Consequences

• When building new applications, choose less risky dependencies
  – use GNU-SASL instead of cyrus-sasl, Gnome instead of KDE

• When maintaining existing applications, prioritise resources
  – look at krb5-libs, not at gkermit
Conclusions

• Vulnerabilities correlate with dependencies
• Identification of risky dependencies
• Prediction with high precision, recall, correlation

http://research.microsoft.com/projects/esm/
http://www.artdecode.de/

* Have we worked with Red Hat: yes, have received positive feedback
* Usage Data: nonexistent
* Explain Correlation: See previous slide: domains