Detecting the 1%: Growing the Science of Vulnerability Discovery

Laurie Williams
laurie_williams@ncsu.edu

NC STATE UNIVERSITY
SCI
Wolfpack Security and Privacy Research

Real people – Real Projects – Real Impact
Meet the “fishy” vulnerability characters

Adam the Attack-prone

David the Detected

Edwin the Exploitable

Larry the Latent
The goal is to aid **software practitioners** in efficiently detecting exploitable vulnerabilities through empirical study of the characteristics of vulnerabilities and through the development of vulnerability prediction models.
The goal is to aid software practitioners in efficiently detecting exploitable vulnerabilities through empirical study of the characteristics of vulnerabilities and through the development of vulnerability prediction models.
The goal is to aid software practitioners in efficiently detecting exploitable vulnerabilities through empirical study of the characteristics of vulnerabilities and through the development of vulnerability prediction models.
The goal is to aid software practitioners in efficiently detecting exploitable vulnerabilities through empirical study of the characteristics of vulnerabilities and through the development of vulnerability prediction models.
Collaborators

Funded by: NSF, [Institutional Logo]

In cooperation: Microsoft Research
Where are we going?

- Setting the stage
- Complications in vulnerability research
- The real questions …
  - Where shall we look?
  - How shall we look?
  - Which vulnerabilities are likely to be exploited?
- Future directions
Design flaws and implementation bugs
Vulnerabilities are rare events (Firefox 2.0)
Getting, creating, and cleaning the data 😳
Where shall we look?

Larry the Latent → David the Detected

Stage ▶ Complications ▶ Where ▶ How ▶ Exploited ▶ Future
Unfiltered Static Analysis Alerts as Predictor

If a developer has such poor coding practices that he/she causes lots of (unfiltered) static analysis alerts, you should look carefully in that area for other implementation bugs and larger design flaws.
Correlations between static analysis alerts and vulnerability count
(all statistically significant)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Case study 1 (component-level)</th>
<th>Case study 2 (file-level)</th>
<th>Case study 3 (component-level)</th>
</tr>
</thead>
<tbody>
<tr>
<td>All SA alerts</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>Security SA alerts</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
</tbody>
</table>
Complexity as Predictor

Security experts say:

• Bruce Schneier
  • “Complexity is the worst enemy of security.”

• Dan Geer
  • “Complexity provides both opportunity and hiding places for attackers.”

• Gary McGraw
  • “A ... trend impacting software security is unbridled growth in ... complexity ...”
Complexity and Other Metrics

• **14 code complexity metrics**
  • Lines of code, cyclomatic complexity, fan-in/fan-out, coupling, comment density and others

• **3 code churn metrics**
  • Frequency of file changes, lines of code changed, and new lines of code

• **11 developer metrics**
  • Number of developers and other network analysis-inspired metrics (e.g. betweenness, closeness)
Results: Predictability (11 releases Firefox)
Results: Predictability (RHEL)
Developer Metrics as Predictor

“Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone. […] Many eyes make all bugs shallow.”

-Linus’ Law
Eric Raymond
How Many Developers?

- Metric: NumDevs
  The number of distinct developers who changed a given source code file

In all three case studies...

- **Vulnerable files had more developers than neutral files**
  
  $(p<0.001)$

- **Files changed by 6 or more developers were 4 times more likely to have a vulnerability,** $(p<0.001)$

  *(…not quite what Linus’ Law says…)*
Unfocused Contributions

Examined files changed by many developers who were working on many other files at the time (an “unfocused contribution”)

Used contribution network centrality (CNBetweenness)

Vulnerable files had a higher CNBetweenness ($p<0.001$) than neutral files.
Traditional Code Metrics as Predictor

<table>
<thead>
<tr>
<th>Metric</th>
<th>$\rho$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit Frequency (EF)</td>
<td>0.292</td>
</tr>
<tr>
<td>Total Lines of Code</td>
<td>0.281</td>
</tr>
<tr>
<td>Frequency</td>
<td>0.279</td>
</tr>
<tr>
<td>Total Complexity</td>
<td>0.276</td>
</tr>
<tr>
<td>Repeat Frequency</td>
<td>0.273</td>
</tr>
<tr>
<td>Number of Ex-Engineers (NOEE)</td>
<td>0.270</td>
</tr>
<tr>
<td>TotalFanIn</td>
<td>0.263</td>
</tr>
<tr>
<td>TotalFanOut</td>
<td>0.262</td>
</tr>
<tr>
<td>Number of Engineers (NOE)</td>
<td>0.261</td>
</tr>
<tr>
<td>Total Global Variables</td>
<td>0.255</td>
</tr>
<tr>
<td>Total Churn</td>
<td>0.254</td>
</tr>
<tr>
<td>Max FanIn</td>
<td>0.224</td>
</tr>
<tr>
<td>Max Complexity</td>
<td>0.207</td>
</tr>
<tr>
<td>Max FanOut</td>
<td>0.196</td>
</tr>
<tr>
<td>Max Lines of Code</td>
<td>0.194</td>
</tr>
<tr>
<td>Outgoing direct</td>
<td>0.168</td>
</tr>
<tr>
<td>Total ClassMethods</td>
<td>0.167</td>
</tr>
<tr>
<td>Max ClassMethods</td>
<td>0.164</td>
</tr>
<tr>
<td>Total InheritanceDepth</td>
<td>0.161</td>
</tr>
<tr>
<td>Total BlockCoverage</td>
<td>0.157</td>
</tr>
<tr>
<td>Incoming direct</td>
<td>0.156</td>
</tr>
<tr>
<td>Total ClassCoupling</td>
<td>0.154</td>
</tr>
<tr>
<td>Total ArcCoverage</td>
<td>0.152</td>
</tr>
<tr>
<td>Incoming closure</td>
<td>0.148</td>
</tr>
<tr>
<td>Total SubClasses</td>
<td>0.141</td>
</tr>
<tr>
<td>Max InheritanceDepth</td>
<td>0.137</td>
</tr>
<tr>
<td>Max ClassCoupling</td>
<td>0.137</td>
</tr>
<tr>
<td>Max SubClasses</td>
<td>0.124</td>
</tr>
<tr>
<td>Level of Org. Code Ownership (OCO)</td>
<td>0.123</td>
</tr>
<tr>
<td>Depth of Master Ownership (DMO)</td>
<td>0.101</td>
</tr>
</tbody>
</table>

All correlations values are significant at $p<0.0001$. 
Windows Vista

What you look at will likely be a vulnerability …

… But many vulnerabilities will be missing.
Vulnerability prediction modeling by others

- Without much better results when tested with similar vulnerability scarcity:
  - Dependency structure
  - Text mining
  - Design churn
  - More code metrics
  - Neural networks and deep learners
## Infrastructure as Code Security Smells

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Security Smell</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{power_username} = 'admin'$</td>
<td>Admin by default</td>
</tr>
<tr>
<td>password =&gt; ''</td>
<td>Empty password</td>
</tr>
<tr>
<td>$\text{power_password} = 'admin'$</td>
<td>Hard-coded secret</td>
</tr>
<tr>
<td>$\text{bind_host} = '0.0.0.0'$</td>
<td>Invalid IP address binding</td>
</tr>
<tr>
<td>$\text{quantum_auth_url} = \text{<a href="http://127.0.0.1:35357/v2.0%7D$">http://127.0.0.1:35357/v2.0}$</a></td>
<td>Suspicious comment</td>
</tr>
<tr>
<td>password =&gt; md5($\text{power_password}$)</td>
<td>Use of HTTP without TLS</td>
</tr>
<tr>
<td></td>
<td>Use of weak cryptography algorithm</td>
</tr>
</tbody>
</table>

**Stage**

- Complications
- Where
- How
- Exploited
- Future
Frequency of Security Smells

![Bar chart showing frequency of security smells across different platforms.](chart.png)

- GitHub
- Mozilla
- Openstack
- Wikimedia

Proportion of Script (%)

Stage → Complications → Where → How → Exploited → Future
Actionable and/or Predictive Heuristics

• Static Analysis Alerts
  • Predictive: Static analysis alerts are indicative of all security vulnerabilities.
  • No pre-processing to determine true positive necessary.

• Code complexity
  • Actionable and predictive: Complex code is less secure

```
case 2:
CID 1442508 (#1 of 1): Unintentional integer overflow (OVERFLOW BEFORE WIDEN)
overflow_before_widen: Potentially overflowing expression get_unaligned_be32(&power-
>update_tag) * occ->powr_sample_time_us with type unsigned int (32 bits, unsigned) is
evaluated using 32-bit arithmetic, and then used in a context that expects an expression of type u64
(64 bits, unsigned).

To avoid overflow, cast either get_unaligned_be32(&power->update_tag) or occ-
>powr_sample_time_us to type u64.

val = get_unaligned_be32(&power->update_tag) *
```
Actionable and/or Predictive Heuristics - 2

• **Developer activity metrics**
  • Actionable and predictive
    • Don’t allow too many people to change same (critical) file
    • Watch for the “hummingbirds” that change many files.

• **Traditional code metrics**
  • Predictive: Traditional code metrics can be used to find vulnerabilities
  • Support that vulnerabilities have the same characteristics as faults

• **Infrastructure as code smells**
  • Actionable: Identify and mitigate code smells
Vulnerability prediction models are not yet practical ... but patterns of what to watch for have been identified.
How shall we look?
## Comparison of Vulnerability Discovery Techniques

<table>
<thead>
<tr>
<th>Discovery Technique</th>
<th>Vulnerabilities Per Hour</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tolven eCHR</td>
</tr>
<tr>
<td>Exploratory Manual Penetration Testing</td>
<td>0.00</td>
</tr>
<tr>
<td>Systematic Manual Penetration Testing</td>
<td>0.94</td>
</tr>
<tr>
<td>Automated Penetration Testing</td>
<td>22.00</td>
</tr>
<tr>
<td>Static Analysis</td>
<td>2.78</td>
</tr>
</tbody>
</table>
Other observations

No single technique discovered every type of vulnerability.

Very few individual vulnerabilities discovered with multiple discovery techniques.
Which technique?

Systematic manual and exploratory penetration testing

Automated penetration testing and static analysis

Design flaw

Implementation bug
Takeaway

One technique is not enough.
What will be exploited?

Edwin the Exploitable

Adam the Attack-prone

Stage ▶ Complications ▶ Where ▶ How ▶ Exploited ▶ Future
Risk-based Attack Surface Approximation

Code artifacts that appear in crash dump stack traces from a software system are more likely to have exploitable vulnerabilities than code artifacts that do not appear in crash dump stack traces.
Stage  Complications  Where  How  Exploited  Future
Stage
Complications
Where
How
Exploited
Future
### Where the Exploitable Vulnerabilities Lie

<table>
<thead>
<tr>
<th></th>
<th>Code Coverage</th>
<th>Vulnerability Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows (Binaries)</td>
<td>48.4%</td>
<td>94.8%</td>
</tr>
<tr>
<td>Firefox (Source Code Files)</td>
<td>14.8%</td>
<td>85.6%</td>
</tr>
<tr>
<td>Fedora (Packages)</td>
<td>8.9%</td>
<td>63.3%</td>
</tr>
</tbody>
</table>
Clustering on the Boundary?

**Boundary Code (BC):** percentage of code that appears on the boundary of a software system

**Boundary Vulnerabilities (BV):** percentage of vulnerabilities on Boundary Code (BC)

<table>
<thead>
<tr>
<th></th>
<th>2014</th>
<th>2015</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Windows 8</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>4.5%</td>
<td>17.2%</td>
</tr>
<tr>
<td>2015</td>
<td>4.6%</td>
<td>18.6%</td>
</tr>
<tr>
<td><strong>Windows 8.1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>4.6%</td>
<td>16.5%</td>
</tr>
<tr>
<td>2015</td>
<td>6.9%</td>
<td>23.7%</td>
</tr>
<tr>
<td><strong>Windows 10</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>3.4%</td>
<td>10.5%</td>
</tr>
<tr>
<td>2015</td>
<td>3.9%</td>
<td>25.1%</td>
</tr>
</tbody>
</table>
Vulnerabilities found on the attack surface are exploitable. More work need to characterize exploitable and attack-prone vulnerabilities.
The goal is to aid software practitioners in efficiently detecting exploitable vulnerabilities through empirical study of the characteristics of vulnerabilities and through the development of vulnerability prediction models.
Building Vulnerability Datasets
Understanding the 1%

- Vulnerabilities versus non-security defects?
  - What technique was used to detect?
  - What was the role of the detector?
  - What is the complexity of the patch?
  - How much time elapsed from injection until detection?
  - How much time elapsed from the detection until the patch?
  - What patterns exist in the longitudinal arrival rate?
  - Can fault prediction models be used for vulnerabilities?
Where shall we look?

Larry the Latent

David the Discovered
Training learners to recognize rare target

- SMOTE (Synthetic Minority Over-sampling)
- Fiddle the training data (but not the test data)
- Ignore the non-vulnerable files
- Synthesize more examples of the vulnerable files
How shall we look?

Stage > Complications > Where > How > Exploited > Future

NEED FOR SPEED™
# Comparison of Vulnerability Discovery Techniques

<table>
<thead>
<tr>
<th>Discovery Technique</th>
<th>Vulnerabilities Per Hour</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OpenMRS</td>
</tr>
<tr>
<td>Exploratory Manual Penetration Testing</td>
<td></td>
</tr>
<tr>
<td>Systematic Manual Penetration Testing</td>
<td></td>
</tr>
<tr>
<td>Automated Penetration Testing</td>
<td></td>
</tr>
<tr>
<td>Static Analysis</td>
<td></td>
</tr>
</tbody>
</table>
What will be exploited?

Edwin the Exploitable

Adam the Attack-prone
Characteristics of Exploitable Vulnerabilities

- Detected versus Exploitable versus Attack-prone
  - What vulnerability type (CWE)?
  - What severity (CVSS) per CWE type (in the NVD)?
  - Time to discover?
  - Distance from the attack surface edge?
  - Detectable in how many ways?
  - Who detected? Who exploited? What assets involved?
Summary

Where?

How?

David the Detected

Adam the Attack-prone

Edwin the Exploitable
Graduate studies at NCSU

**Degrees**
- PhD
- Master of Science
- Master of Computer Science
  - Track in Data Science
  - Track in Security
  - Track in Software Engineering
- Master of Computer Science (Distance Education)
- Master of Science in Computer Networking
- Master of Science in Computer Networking (Distance Education)

**Certificate**
- Computer Science
- Data Science Foundations
Images

- https://dementiacarebooks.com/how-to-become-a-dementia-behavior-detective/
- http://www.brianbarber.com/illustration/
- https://drawception.com/game/HM8CfM7pHD/sleepy-fish/
- Vectorstock.com/9961574
- https://requestreduce.org/categories/fish-trap-clipart.html#overlayGallery9_post_17509_fish-trap-clipart-17.png
- https://www.zazzle.com/red_star_1st_prize_round_sticker_red-217743138139492519
- https://easydrawingguides.com/how-to-draw-a-whale/
- https://achievingbeautifuldreams.files.wordpress.com/2015/09/50-50.jpg
Images

- https://www.monitis.com/blog/why-your-small-business-needs-penetration-testing/
- https://www.foolishbricks.com/day-276-the-needle-in-the-haystack/
- https://betanews.com/2016/06/30/solve-shortage-data-scientists/
- https://simpleprogrammer.com/get-programming-job-no-experience/
- https://towardsdatascience.com/organizing-your-first-text-analytics-project-ce350dea3a4a
- https://marketeer.kapost.com/programming-for-marketers/
Possible fish


http://www.brianbarber.com/illustration/

https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcQFnTWQGJl6jLxeHmzDNeJCI2Rrgm2Fp5hiwZFBv3XvKOhoG1PC6

https://www.designbyhumans.com/shop/sticker/mean-fish/660022/

https://drawception.com/game/HM8CfM7pHD/sleepy-fish/

https://suzyssitcom.com/2013/08/can-you-do-the-heimlich-on-a-fish.html
Q: How to synthesize examples of vulnerable software?
A: SMOTE (*Synthetic Minority Over-sampling*)

```python
function SMOTE()
    while Majority > m do delete any Majority item
    while Minority < m do add something_like(any Minority item)

function something_like( X0 )
    { X1, X2, ... } = k nearest neighbors of X0
    Z = any of X0
    Y = interpolate( X0, Z)
    return Y

function minkowski_distance(a, b, r)
    return ( Σ abs(a.i - b.i)^r ) ^ (1/r)

Q: How to do this better?
A1: Tune the magic parameters of SMOTE <m,k,r>
```
Case Studies

Three empirical case studies

- RHEL4 Linux kernel, PHP, and Wireshark
- Pre-release version control logs
- Post-release security vulnerabilities
- Viewed files as **vulnerable** (≥0 vulnerabilities) or **neutral** (none found yet)

<table>
<thead>
<tr>
<th></th>
<th>RHEL4 kernel</th>
<th>PHP</th>
<th>Wireshark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of committers</td>
<td>557</td>
<td>84</td>
<td>19</td>
</tr>
<tr>
<td>Source code files</td>
<td>14,454</td>
<td>1,039</td>
<td>2,688</td>
</tr>
<tr>
<td>% files vulnerable</td>
<td>3%</td>
<td>6%</td>
<td>3%</td>
</tr>
<tr>
<td>Pre-release version control log data</td>
<td>16 months</td>
<td>2 years</td>
<td>2 years</td>
</tr>
<tr>
<td>Years of security data</td>
<td>5 years</td>
<td>3 years, 5 months</td>
<td>3 years, 5 months</td>
</tr>
</tbody>
</table>
Preliminary Findings

• 5 projects – Linux, Firefox, Samba, Qt, Kodi
  • Median alert count: 10171
  • Median Triage Rate: 17.5%
  • Median Fix Rate: 51.3%
  • Median Unactionable* Rate: 45.9%
  • Median Bug Rate: 23.6%
  • Median Lifespan: 33 weeks

• Security alerts are *Not* likely to be fixed more often than non-security alerts
• Security alerts are *Not* likely to be fixed quicker than non-security alerts

*marked by developer as false positive or intentional
What we currently do with vulnerabilities (BSIMM8)
Results: Predictability (11 releases Firefox)
Results: Predictability (RHEL)
Vulnerability Resolution

Vulnerabilities are fixed at a faster rate than defects.

In Mozilla, vulnerabilities are resolved 33% more quickly than defects.