Scale Your Security
by Embracing Secure Defaults and Eliminating Bug Classes

Grayson Hardaway | r2c.dev

Slides are posted at semgrep.dev
whois

me:  
Grayson Hardaway, sr. security  
engineer @ r2c  
Formerly: U.S. Department of Defense

r2c:  
We’re an SF based static analysis startup on a mission to profoundly improve software security and reliability.
Outline

1. Why Bug-Finding Isn't The Answer
2. How to Eradicate Vulnerability Classes
3. Tools & Techniques To Make It Real
1. Why Bug-Finding Isn’t The Answer
Software Development has Changed
...thus Security Teams must too

In many companies:
● Security teams can hard block engineering rarely, if ever
● Security testing must be continuous, not point in time
● Focus on building, not just breaking
● Embedded or partnered closely with dev teams

We need to re-visit our prior assumptions
Massive Shifts in Tech and Security

Before

Waterfall development
Dev, Ops
On prem

After

Agile development
DevOps
Cloud
Massive Shifts in Tech and Security

Before

Waterfall development
Dev, Ops
On prem
Finding vulnerabilities

After

Agile development
DevOps
Cloud
Secure defaults and invariants
Invariant

A property that must either always or never be true

Key Insight

No context needed to make a decision

==

No operational time for the security team
Quiz: Does this app have XSS?
Quiz: Does this app have XSS?

Context?
- HTML
- HTML attribute
- JavaScript
- ...

What does user control?
- Structure of data?

Input filtered?

Data processed before sent to user?

How is it stored?
- (field types, constraints)

DB type?
Quiz: Does this app have XSS?

**Invariant: Frontend is React, banned dangerouslySetInnerHTML**

What does user control?
- Structure of data?

Input filtered?

How is it stored?
- (field types, constraints)

Data processed before sent to user?

Context?
- HTML
- HTML attribute
- JavaScript
- ...

DB type?
Quiz: Does this app have XSS?

Invariant: Frontend is React, banned dangerouslySetInnerHTML

- What does user control?
- Structure of data?
- Input filtered?
- Data processed before sent to user?
- How is it stored?
  - (field types, constraints)
- DB type?
- Context?
  - HTML
  - HTML attribute
  - JavaScript
  - ...

Icons by Icons8
Quiz: Does this app have RCE?
Quiz: Does this app have RCE?

What does user control?
- Structure of data?

Input filtered?

How is it stored?
- (field types, constraints)

Does the app?
- Deserialize data
- Run shell commands
- Mix data and code
  - eval(), exec()
  - Metaprogramming

Icons by Icons8
Quiz: Does this app have RCE?

*Ban: exec(), eval(), shell exec, deserialization (objects, YAML, XML, JSON)*

What does user control?
Structure of data?

Input filtered?

How is it stored?
(field types, constraints)

Does the app?

- Deserialize data
- Run shell commands
- Mix data and code
  - eval(), exec()
  - Metaprogramming

Input filtered?

Does the app?

- Deserialize data
- Run shell commands
- Mix data and code
  - eval(), exec()
Quiz: Does this app have RCE?

Ban: `exec()`, `eval()`, `shell exec`, deserialization (objects, YAML, XML, JSON)

What does user control?
Structure of data?

Does the app?
● Deserialize data
● Run shell commands
● Mix data and code
  ○ `eval()`, `exec()`
  ○ Metaprogramming

Input filtered?

How is it stored?
(field types, constraints)
Write proof of concept exploit

Task vs Effort Required

- Task: Find potential bug
  - Effort Required: 17
- Task: Confirm it's a real bug
  - Effort Required: 17
- Task: Detect use of (in)secure library
  - Effort Required: 17
Detecting (lack of) use of secure defaults is much easier than finding bugs.
#Broke
Finding every vulnerability

#Woke
Preventing *classes* of vulnerabilities
Your Internal Dialogue?

● “All you’ve shown me is some hand-wavy diagrams”
● The security industry has focused on bug finding for decades
  ○ SAST, DAST, pen tests, bug bounty
We Come Bearing Gifts: Enabling Prod Security w/ Culture & Cloud

AppSec Cali '18, Patrick Thomas, Astha Singhal

---

**De-emphasized***
- Manual Testing
- Manual Code Review
- Per-App Threat Modeling
- Traditional Vuln Scanning

**Used With Reservations***
- Generic Static/Dynamic Scans
- 3rd Party Pentesting
- Training

**Heavily Emphasized***
- Automated Visibility & Action
- Org-level Partnerships
- AuthN & AuthZ Everywhere
- Paved Road
- Self-Service
- Killing Bug Classes

* This is the current mix. Wasn’t always this way.
A Pragmatic Approach for Internal Security Partnerships
AppSec Cali '19, Scott Behrens, Esha Kanekar

How is the future shaping up for us?

**Today**
- Secure By Default
- Self Service
- Security Partnership

**Mid term**
- Secure By Default
- Self Service
- Security Partnership

**Long term**
- Secure By Default
- Self Service
- Security Partnership
41% of vulnerability reduction from XP → Vista from banning strcpy and friends

Analysis of 63 buffer-related security bugs that affect Windows XP, Windows Server 2003 or Windows 2000 but not Windows Vista: 82% removed through SDL process

"Security Improvements in Windows Vista", Michael Howard

- 27 (43%) found through use of SAL (Annotations)
- 26 (41%) removed through banned API removal
Tools and Training Help, but are Not Enough

We need a safer systems programming language

Security Research & Defense / By MSRC Team / July 18, 2019 / Memory Safety, Rust, Safe Systems Programming Languages, Secure Development

From the Microsoft Security Response Center blog:

- “Tools and guidance are demonstrably not preventing this class of vulnerabilities; memory safety issues have represented almost the same proportion of vulnerabilities assigned a CVE for over a decade.”
Google:

- “It’s unreasonable to expect any developer to be an expert in all these subjects, or to constantly maintain vigilance when writing or reviewing code.

- A better approach is to handle security and reliability in common frameworks, languages, and libraries. Ideally, libraries only expose an interface that makes writing code with common classes of security vulnerabilities impossible.”

Google: Building Secure and Reliable Systems, by Google
"We invest heavily in building frameworks that help engineers prevent and remove entire classes of bugs when writing code."

*Designing Security For Billions* by Facebook
“But I’m not Google”

Framework / tech choices matter

- Mitigate classes of vulnerabilities

Examples:

- Using modern web frameworks
- DOMPurify - output encoding
- re2 - regexes
- tink - crypto
2. How to Eradicate Vulnerability Classes
Compounding Effects of Killing Bug Classes

- Threat Modeling
- Running security tools
- XSS
- SQL Injection
- Triaging bug bounty
- Security training
How to Eradicate Vulnerability Classes

1. Select a vulnerability class
2. Find/prevent it at scale
3. Select a safe pattern and make it the default
4. Train developers to use the safe pattern
5. Use tools to enforce the safe pattern
1. Select a vulnerability class

**Common selection criteria**

Bug classes that are:

1. The most prevalent
2. The highest impact / risk
3. Easiest to tackle (organizationally, technically)
4. Organizational priorities
5. Weighted: \( f(\text{prevalent}, \text{severe}, \text{feasible}, \text{org}) \)
1. Select a vulnerability class

Vulnerability Management  more

Know your current state and if your future efforts actually work
1. Select a vulnerability class

Vulnerability Management (more)

Know your current state and if your future efforts actually work

Track:
- Risk, Severity, Impact
- Vuln class - access controls, XSS, SQLi, open redirect, ...
  - Create a taxonomy (e.g. OWASP Top 10, Bugcrowd's VRT)
  - Aim for 20-40 categories (should have different root cause/fix)
- PR introducing / fixing the issue
- Relevant code base (and team/org)
- Root cause
- What source found this? (DAST, SAST, pen test, bug bounty, ...)
- Mitigating factors
1. Select a vulnerability class

Building the List of Prior Vulnerabilities to Review

When your vuln tracking has been inconsistent

Common Sources

- JIRA/GitHub issues tagged “security”
- Create a list of security-relevant keywords
  - Search pull/merge requests, issue tracker, git commit history
  - `git log --grep "xss"`
- Security tool reports (SAST, DAST, ...)
- Pen test reports, bug bounty submissions
- Ask development, ops, and security teams for examples
1. Select a vulnerability class

Building the List of Prior Vulnerabilities to Review

When your vuln tracking has been inconsistent

Common Sources

● JIRA/GitHub issues tagged “security”
● Create a list of security-relevant keywords
  ○ Search pull/merge requests, issue tracker, git commit history
  ○ `git log --grep "xss"`
● Security tool reports (SAST, DAST, …)
● Pen test reports, bug bounty submissions
● Ask development, ops, and security teams
● Use Google! Use framework documentation!

Going Forward

Fully analyzing ad hoc historical data may not be worth the time

Now: create and document a lightweight, standardized process
● Make your life easier next time
1. Select a vulnerability class

Slice and Dice

- Group by vulnerability class
- Group by source (DAST, SAST, BB...)
- Weight by severity/risk/impact

Data Driven Bug Bounty by @arkadiyt
1. Select a vulnerability class

Slice and Dice

- Group by vulnerability class
- Group by source (DAST, SAST, BB...)
- Weight by severity/risk/impact

Choose a bug class and review the fixes

- What did the vulnerable code look like?
- What did the fix look like?

What trends do you see?

- **Good**: vulnerable code looks similar
- **Bad**: all buggy code looks different
1. Select a vulnerability class

**Ideal World**

Choose a vulnerability class that is:

- Widespread across teams/repos
- High Risk
- Feasible to get devs to fix
- Aligns with company priorities
- Always broken in the same way
1. Select a vulnerability class

Pick one and eliminate it!
2. Find/prevent at Scale

<table>
<thead>
<tr>
<th>Problem</th>
<th>Security Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big picture, architectural flaws</td>
<td>Threat Modeling</td>
</tr>
<tr>
<td>Cloud misconfigurations</td>
<td>IaaC scanning, Cartography, BB</td>
</tr>
<tr>
<td>Complex business logic bugs</td>
<td>Pen tests, bug bounty</td>
</tr>
<tr>
<td>Protect vulns until they’re patched</td>
<td>WAF, RASP</td>
</tr>
<tr>
<td>Known good/known bad code</td>
<td>Lightweight static analysis</td>
</tr>
</tbody>
</table>
3. Select a Safe Pattern and Make it the Default

- Based on internal coding guidelines, standards, your expertise, ...
3. Select a Safe Pattern and Make it the Default

Update all internal coding guidelines (security & dev)
● READMEs, developer documentation, wiki pages, FAQs
  ○ Training slides, onboarding presentations, ...
● Explain why these patterns exist and how to use them

Work with developer productivity team
● Secure version should have an even better dev UX than the old way
  ○ Potentially: build a secure library. Make the insecure pattern hard to use while still letting devs go fast
  ○ How can we increase dev productivity and security?
● Integrate security at the right points (e.g. new project starter templates) to get automatic, widespread adoption
● “Hitch your security wagon to dev productivity.” - Astha Singhal
4. Train Developers to Use the Safe Pattern

Making Communications Successful

- **What and why** something is insecure should be clear
  - Use terms developers understand, no security jargon
- **Convey impact** in terms devs care about
  - Risk to the business, damaging user trust, reliability, up time
- **How to fix** it should be **concise and clear**
  - Link to additional docs and resources with more info
  - Have a dedicated #AppSec chat channel for further questions
Don’t Run with Scissors: How to standardize how developers use dangerous aspects of your framework by Morgan Roman

How to write training/documentation

If you use *<THE BAD WAY>*; *<BAD THING WILL HAPPEN>*. *Instead use the* *<THE GOOD WAY>* since *<IT STOPS THE BAD THING>*.

**DO NOT DO THIS:**

*<EXAMPLE OF THE BAD WAY>*

**DO THIS INSTEAD:**

*<EXAMPLE OF THE GOOD WAY>*

Explain it simply and clearly.
You do not need to use security lingo like XXE/ReDoS/XSS/RCE etc.

First show an example on how the developer intends to do it if you have one
Then show how they can do it correctly.
Make sure it is simple to do.
4. Train Developers to Use the Safe Pattern

How to Engage: Some Options

- During developer onboarding
- Lead educational brown bag sessions over lunch
- Internal CTFs
- Security champions
- When in-person interaction is feasible again
  - Grab lunch with dev teams and/or schedule a happy hour
  - Have candy on desks by the security team
5. Use Tools to Enforce the Safe Pattern

Use lightweight static analysis (grep, linting) to ensure the safe patterns are used.
3. Tools & Techniques To Make It Real
How to Eradicate Vulnerability Classes

1. Evaluate which vulnerability class to focus on
2. Determine the best approach to find/prevent it at scale
   → How to set up continuous code scanning

1. Select a safe pattern and make it the default
2. Train developers to use the safe pattern
3. Use tools to enforce the safe pattern
   → Checking for escape hatches in secure
Continuous Scanning: Related Work

AppSec USA:

- Put Your Robots to Work: Security Automation at Twitter | '12
- Providence: rapid vuln prevention (blog, code) | '15
- Cleaning Your Applications’ Dirty Laundry with Scumblr (code) | '16
- Scaling Security Assessment at the Speed of DevOps | '16
- SCORE Bot: Shift Left, at Scale! | ‘18
Continuous Scanning: Related Work

Salus: How Coinbase Sales Security Automation  (blog, code)
DevSecCon London ’18

Orchestrating Security Tools with AWS Step Functions  (slides)
DeepSec ’18

A Case Study of our Journey in Continuous Security  (code)
DevSecCon London ’19

Dracon- Knative Security Pipelines  (code)
Global AppSec Amsterdam ‘19
Continuous Scanning: Best Practices

Scan **Pull Requests**
every commit is too noisy, e.g. WIP commits

Scan **Fast** (<5min)
feedback while context is fresh
can do longer / more in depth scans daily or weekly

Two Scanning Workflows
*audit* (sec team, visibility), *blocking* (devs, pls fix)

Make **Adjustment Easy**
Make it cheap to add/remove tools and new rules
Scan Fast

Don't come in last!

Security checks should not be the slowest check blocking developer from merging.

<table>
<thead>
<tr>
<th>Check Description</th>
<th>Status</th>
<th>Time</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build Tests / Build and Test semgrep-core (pull_request)</td>
<td>Successful in 6m</td>
<td></td>
<td>Details</td>
</tr>
<tr>
<td>Lint / pre-commit (pull_request)</td>
<td>Successful in 12s</td>
<td></td>
<td>Details</td>
</tr>
<tr>
<td>Push Semgrep Docker Image / docker-build (pull_request)</td>
<td>Successful in 22m</td>
<td></td>
<td>Details</td>
</tr>
<tr>
<td>Security checks</td>
<td>Successful in 6m</td>
<td></td>
<td>Details</td>
</tr>
<tr>
<td>Build Tests / Check builds for macOS (pull_request)</td>
<td>Successful in 26m</td>
<td></td>
<td>Details</td>
</tr>
<tr>
<td>Lint / semgrep with r2c registry (pull_request)</td>
<td>Successful in 16s</td>
<td></td>
<td>Details</td>
</tr>
</tbody>
</table>
Scan Early

Tell me as soon as possible, ideally in the editor.

Also enforce in CI so that it can't be ignored.
Autofix

Make security fixes fast and easy.
Even an imperfect suggestion is better than nothing!

Try:  https://semgrep.dev/ievans:tlsautofix
Continuous Scanning: Best Practices

Show tool findings **within dev systems** (e.g. on PR as a comment)

Clear, actionable, with link to more info

Capture **metrics** about check types, scan runtime, and false positive rates

Track & evict **low signal** checks:
keep only +95% true positives
Otherwise causes ill will with devs + too much security team operational cost
If we use secure frameworks that maintain invariants, all we need to do is detect the functions that let you "escape" from those invariants. For instance:

- dangerouslySetInnerHTML
- exec
- rawSQL(...)
How to find them?

● Grep
  ○ Pro: easy to use, interactive, fast
  ○ Con: line-oriented, mismatch with program structure (ASTs)

● Code-Aware Linter
  ○ Pro: robust, precise (handles whitespace, comments, ...)
  ○ Con: Each parser represents ASTs differently; have to learn each syntax
What we do

Semgrep

Static analysis at ludicrous speed
Find bugs and enforce code standards

Open source, works on 17+ languages
Not proprietary and not only for legacy languages

Scan with 1,000+ community rules
Not vendor controlled

Write rules that look like your code
No painful and complex DSL

Quickly get results in the terminal, editor, or CI/CD
Don't wait hours or days for results

Flag issues moving forward, get results in pull requests, Slack, + more
Don't be forced to fix all existing issues just to get started

Language support

Go  Java  JavaScript  JSON  Python  Ruby  TypeScript  JSX

TSX  Generic (YAML, ERB, Jinja, etc)  + More languages

Get Started

⭐ 3.3k  ⏰ v0.42.0 (14 hours ago)
1. Select a vulnerability class

- r2c is young
  - Two (2) primary codebases
  - Limited vulnerability history

- Prioritize based on common problems for the type of application:
  - Web application → XSS
  - Command line interface → Code and Command injection
2. Prevent it at scale
3. Select a safe pattern and make it the default

Example 1

```javascript
function TestComponent2() {
  // ruleid:react-dangerouslysetinnerhtml
  return <li className="foobar" dangerouslySetInnerHTML={{createMarkup()}} />;
}
```

Example 2

```javascript
function TestComponent3() {
  // ruleid:react-dangerouslysetinnerhtml
  return ...
}
```

Setting HTML from code is risky because it's easy to inadvertently expose your users to a cross-site scripting (XSS) attack.
<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
<th>Semgrep rule</th>
<th>Recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.A.</td>
<td>Ban render_template_string()</td>
<td>python.flask.security.audit.render-template-string.render-template-string</td>
<td>Use render_template().</td>
</tr>
<tr>
<td>1.B.</td>
<td>Ban unescaped extensions</td>
<td>python.flask.security.unescaped-template-extension.unescaped-template-extension</td>
<td>Only use .html extensions for templates. If no escaping is needed, review each case and exempt with # nosem.</td>
</tr>
<tr>
<td>1.C.</td>
<td>Ban Markup()</td>
<td>python.flask.security.xss.audit.explicit-unescape-with-markup.explicit-unescape-with-markup</td>
<td>If needed, review each usage and exempt with # nosem.</td>
</tr>
<tr>
<td>2.A.</td>
<td>Ban returning values directly from routes</td>
<td>python.flask.security.audit.directly-returned-format-string.directly-returned-format-string</td>
<td>Use render_template() or jsonify().</td>
</tr>
<tr>
<td>2.B.</td>
<td>Ban using Jinja2 directly</td>
<td>python.flask.security.xss.audit.direct-use-of-jinja2.direct-use-of-jinja2</td>
<td>Use render_template().</td>
</tr>
<tr>
<td>3.B.</td>
<td>{$ autoescape false %}</td>
<td>python.flask.security.xss.audit.template-autoescape-off.template-autoescape-off</td>
<td>Use Markup() in Python code if necessary.</td>
</tr>
<tr>
<td>4.B.</td>
<td>Flag template variables in href attributes</td>
<td>python.flask.security.xss.audit.template-href-var.template-href-var</td>
<td>Use url_for to generate links.</td>
</tr>
<tr>
<td>4.C.</td>
<td>Ban template variables in &lt;script&gt; blocks.</td>
<td>N/A</td>
<td>Use the tojson filter inside a data attribute and JSON.parse() in JavaScript.</td>
</tr>
</tbody>
</table>
Making Secure Defaults Easier

https://semgrep.dev/explore

insecure-transport
by Colleen Dai

Ensure your code communicates over encrypted channels instead of plaintext.
Java JavaScript Go

jwt
by Vasili Ermilov

Avoid common JWT security mistakes
Go Ruby Python Java JavaScript TypeScript

XSS
by Grayson Hardaway

Secure defaults for XSS prevention across 5 different languages
Go Ruby Python Java JavaScript

https://semgrep.dev/docs/cheat-sheets/django-xss/
4. Train developers to use the safe pattern +
5. Use tools to enforce the safe pattern

vuln_application.py

```python
4. Train developers to use the safe pattern +
5. Use tools to enforce the safe pattern
```
Semgrep Findings Overview over the last 30 days

Fix Rate: 76% (45 / 59)

13 Open Findings

45 Fixed Findings

1 Muted Findings

Open Findings Over Time
BONUS: Quietly monitor new policies

Secrets - Notify

1 item
Used on:
no repositories

Secrets - Notify

Integrations: email-grayson

Inline PR Comments

Blocking

Search...

Name

Type

secrets

0 disabled rules

+ add a disabled rule
Conclusion

- **Secure defaults** are the best way to scalably raise your security bar
  - *Not* finding bugs (bug whack-a-mole)
- **Killing bug classes** makes your AppSec team more leveraged
- **Define safe pattern → educate / roll out → enforce continuously**
  - *Fast & lightweight* (e.g. [semgrep](https://semgrep.dev)), focus on dev UX

Slides:

Grayson Hardaway
grayson@r2c.dev