Security at scale:
Web application security in a continuous deployment environment

OWASP AppSec DC – 4/4/2012

Zane Lackey

zane@etsy.com

@zanelackey

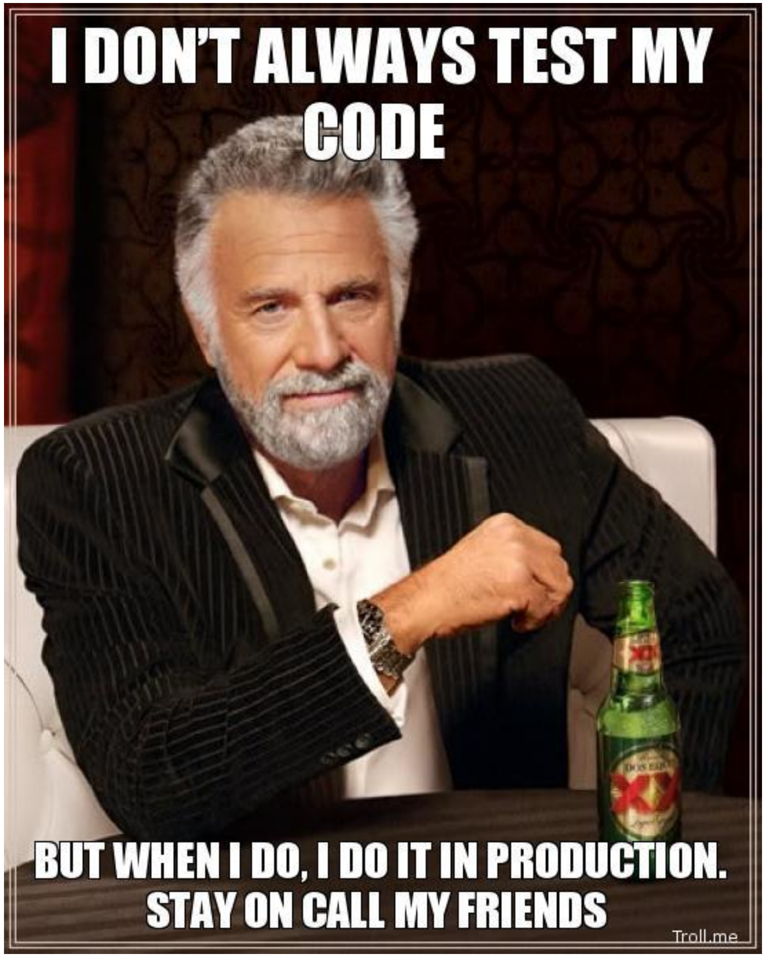
Etsy

About this talk

Web application security techniques that are **simple** and **effective**

Continuous deployment?

Continuous deployment


I DON'T ALWAYS TEST MY CODE
BUT WHEN I DO, I DO IT IN PRODUCTION.
STAY ON CALL MY FRIENDS
Troll.me

<- What it (hopefully) isn't

Continuous deployment

Continuous deployment

Pushing to production **30 times a day** on average

Continuous deployment

(dogs push too)

What it boils down to
(spoiler alert)

- Make things safe by default
- Detect risky functionality / Focus your efforts
- Automate the easy stuff
- Know when the house is burning down

Safe by default

Safe by default

- Traditional defenses for XSS
  - Input validation
  - Output encoding
- Let's illustrate this approach…

Safe by default

Safe by default

- Problems?
  - Often done on a per-input basis
    - Easy to miss an input or output
  - May use defenses in wrong context
    - Input validation pattern may blocks full HTML injection, but not injecting inside JS
  - May put defenses on the client side in JS
  - Etc, …

These problems miss the point

Safe by default

- The real problem is that finding these issues across a codebase is hard

- How can we make it simpler?

Safe by default

Input validation

Output encoding

Etsy

Safe by default

Input validation

Output **encoding**

Safe by default

- Input encoding? Input encoding.
- Encode dangerous HTML characters to HTML entities at the very start of your framework
- Before input reaches main application code

Safe by default

On the surface this doesn’t seem like much of a change

Safe by default

Except, we’ve just made lots of XSS problems **grep-able**

Safe by default

Safe by default

- Now we look for two things:
  - Code that opts out of platform protections
  - HTML entity decoding functions or string replacements on certain characters

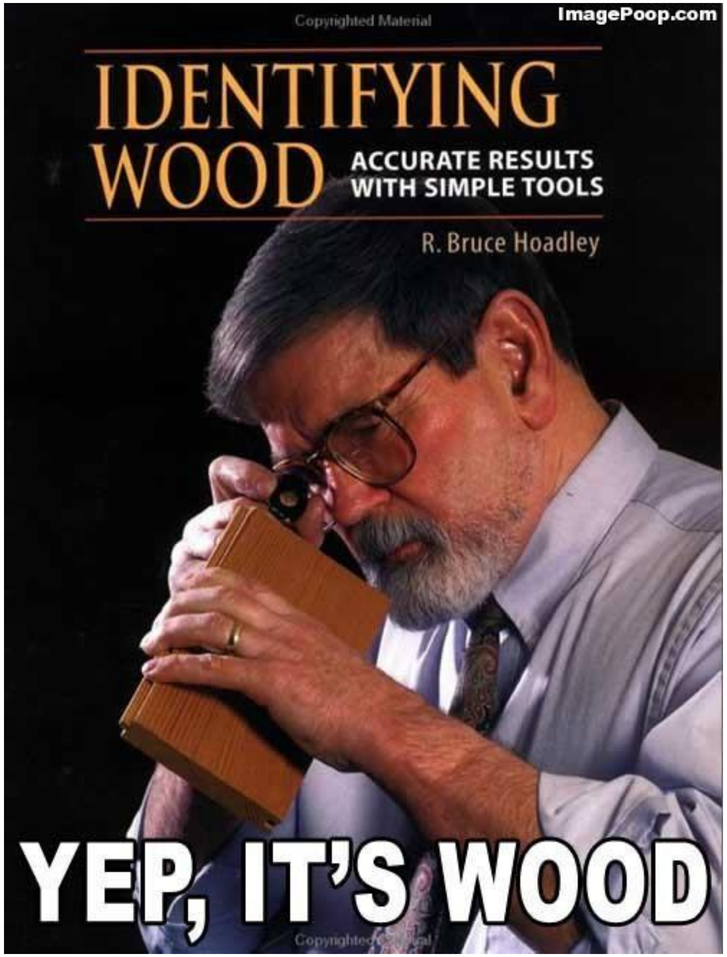
Safe by default

- Obviously not a panacea
  - Javascript: URLs
  - DOM based XSS
  - Is a pain during internationalization efforts

Focus your efforts

Etsy

Focus your efforts



Focus your efforts

- Continuous deployment means code ships fast
- Things will go out the door before security team knows about them
- How can we detect high risk functionality?

Detect risky functionality

- Know when sensitive portions of the codebase have been modified

- Build automatic change alerting on the codebase
  - Identify sensitive portions of the codebase
  - Create automatic alerting on modifications

Detect risky functionality

- Doesn’t have to be complex to be effective

- Approach:
  - sha1sum sensitive platform level files
  - Hourly/daily unit tests alert if hash of the file changes
  - Notifies security team on changes, drives code review

Detect risky functionality

- Watched items typically entire files at the platform level, specific methods at the feature level

- Identifying sensitive methods is part of initial code review/pen test of new features

Detect risky functionality

- Watch for dangerous functions

- Usual candidates:
  - File system operations
  - Process execution/control
  - HTML decoding (if you're input encoding)

Detect risky functionality

- Grep codebase for dangerous functions as hourly/daily unit tests
    - Split into separate high risk/low risk lists
- Alerts are emailed to the appsec team, drive code reviews

Detect risky functionality

- Monitor application traffic

- Purpose is twofold:
  - Detecting risky functionality that was missed by earlier processes
  - Groundwork for attack detection and verification

Detect risky functionality

- Regex incoming requests at the framework
  - Sounds like performance nightmare, shockingly isn’t

- Look for HTML/JS in request
  - This creates a huge number of false positives
    - That’s by design, we refine the search later

Detect risky functionality

- We deliberately want to cast a wide net to see where HTML is entering the application

- From there, build a baseline of
  - The amount of traffic containing HTML
  - The features in the application that receive HTML

Detect risky functionality

- What to watch for:
  - Did a new endpoint suddenly show up?
    - A new risky feature might’ve just shipped
  - Did the amount of traffic containing HTML just significantly go up?
    - Something worth looking at is likely happening

Automate the easy stuff

Automate the easy stuff

Automate the easy stuff

- Automate finding simple issues to free up resources for more complex tasks

- Use attacker traffic to automatically drive testing

- We call it *Attacker Driven Testing*

Automate the easy stuff

- Some cases where this is useful:
  - Application faults
  - Reflected XSS
  - SQLi

Automate the easy stuff

- Application faults (HTTP 5xx errors)
- As a pentester, these are one of the first signs of weakness in an app
  - As a defender, pay attention to them!

Automate the easy stuff

- Just watching for 5xx errors results in a lot of ephemeral issues that don’t reproduce

- Instead:
  - Grab last X hours worth of 5xx errors from access logs
  - Replay the original request
  - Alert on any requests which still return a 5xx

Automate the easy stuff

- Cron this script to run every few hours
- If a request still triggers an application fault hours later, it’s worth investigating

Automate the easy stuff

- Similar methodology for reflected XSS

- For reflected XSS we:
  - Identify requests containing basic XSS payloads
  - Replay the request
  - Alert if the XSS payload executed

Automate the easy stuff

- Basic payloads commonly used in testing for XSS:
  - alert()
  - document.write()
  - unescape()
  - eval()
  - etc

Automate the easy stuff

- We created a tool to use NodeJS as a headless browser with full JavaScript

- Methodology:
  - Replay the request (but don’t interpret it yet)
  - Prepend instrumented JS that flags if a method has been executed
  - Interpret response with our instrumented JS
  - Check if execution flags have been set
  - Alert

Automate the easy stuff

- Doesn’t have to be NodeJS
- Can also use a browser driven via Watir/Selenium

Etsy

Know when the house is burning down

Know when the house is burning down

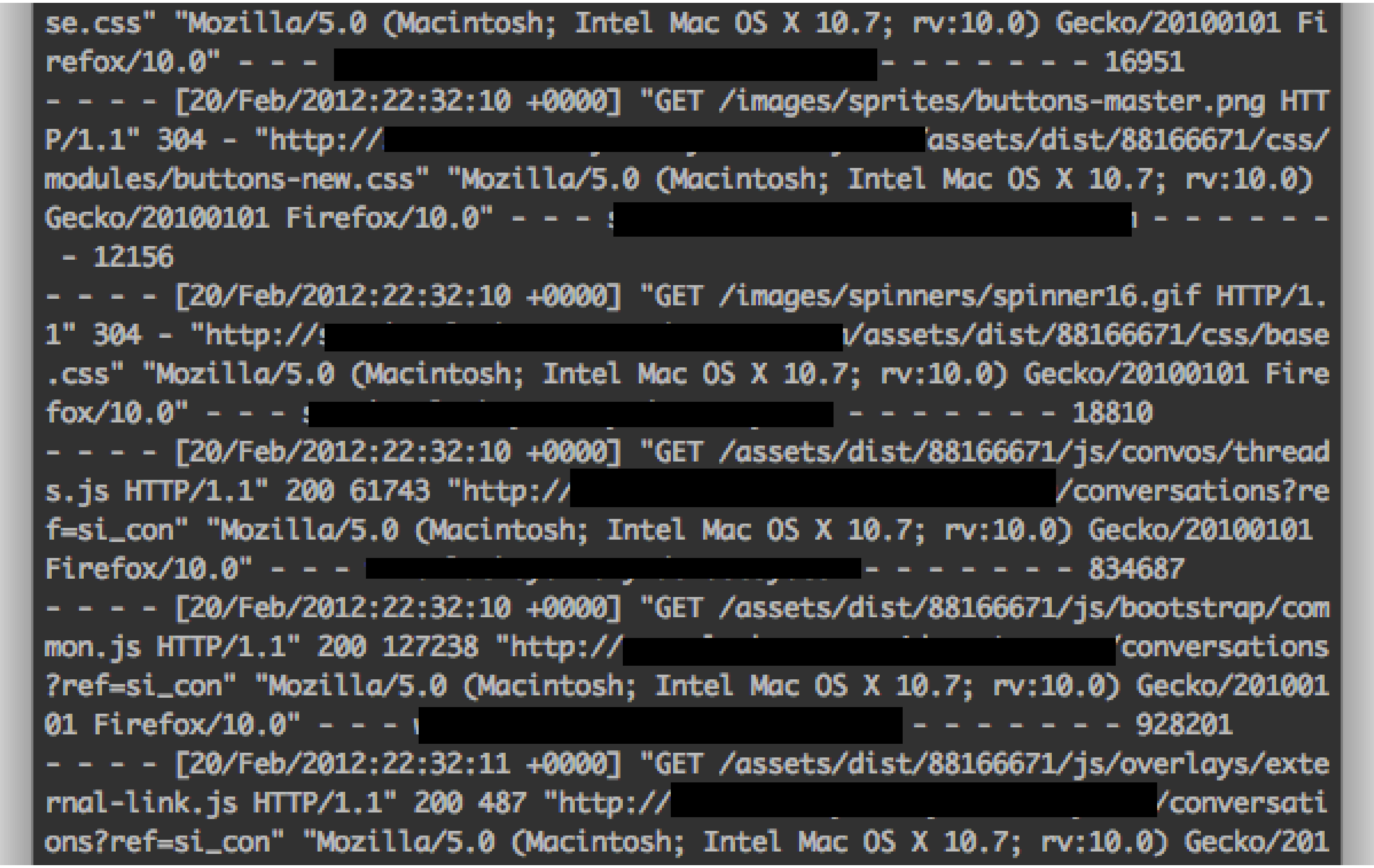
Know when the house is burning down

Graph early, graph often

Know when the house is burning down

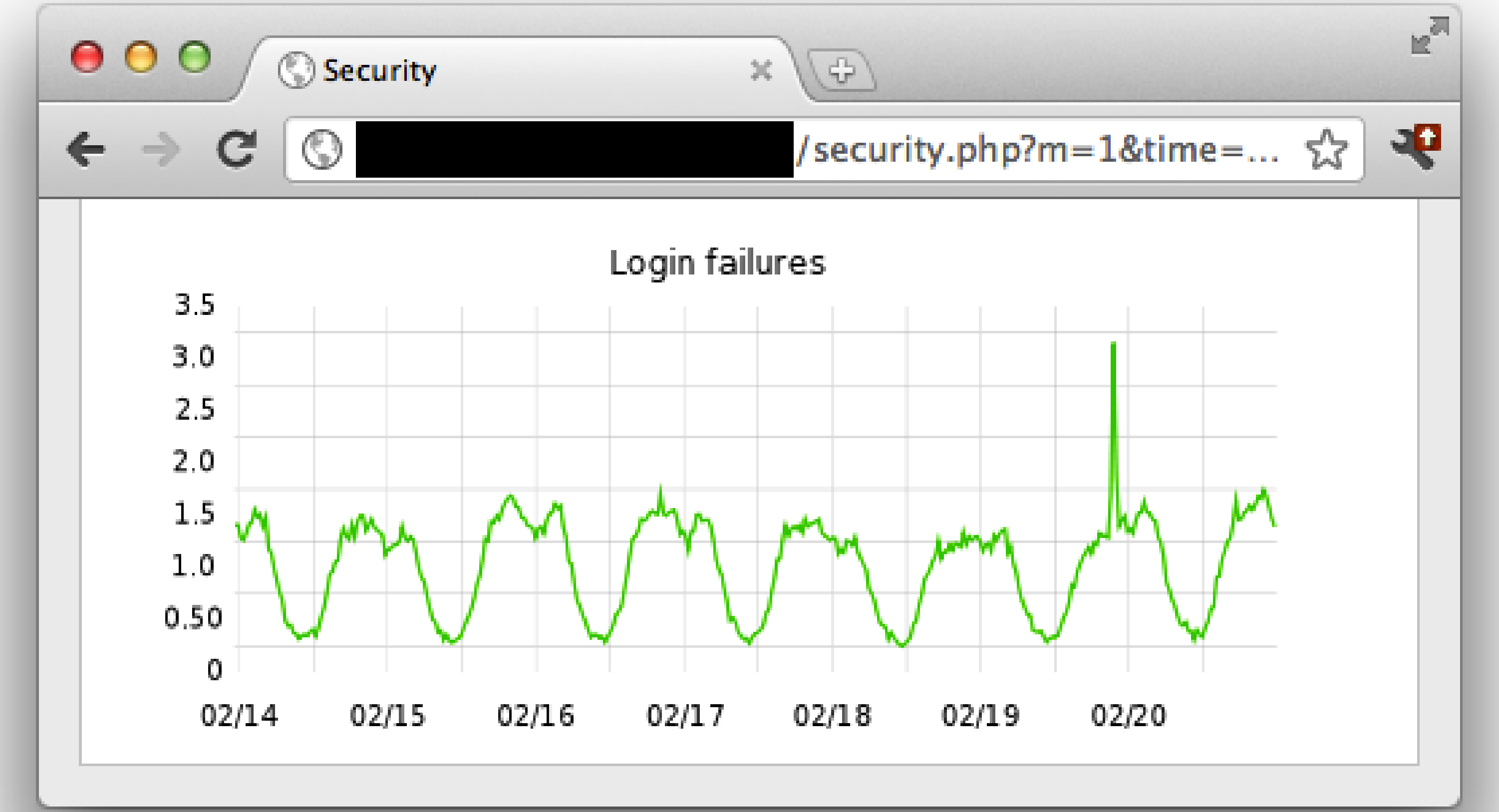
Which of these is a quicker way to spot a problem?

Know when the house is burning down



```
se.css" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:10.0) Gecko/20100101 Fi
refox/10.0" - - - [redacted]- - - - - - - - 16951
- - - - [20/Feb/2012:22:32:10 +0000] "GET /images/sprites/buttons-master.png HTT
P/1.1" 304 - "http://[redacted]assets/dist/88166671/css/
modules/buttons-new.css" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:10.0)
Gecko/20100101 Firefox/10.0" - - - [redacted] - - - - - - -
 - 12156
- - - - [20/Feb/2012:22:32:10 +0000] "GET /images/spinners/spinner16.gif HTTP/1.
1" 304 - "http://[redacted]/assets/dist/88166671/css/base
.css" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:10.0) Gecko/20100101 Fire
fox/10.0" - - - [redacted] - - - - - - - - 18810
- - - - [20/Feb/2012:22:32:10 +0000] "GET /assets/dist/88166671/js/convos/thread
s.js HTTP/1.1" 200 61743 "http://[redacted]/conversations?re
f=si_con" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:10.0) Gecko/20100101
Firefox/10.0" - - - [redacted]- - - - - - - - 834687
- - - - [20/Feb/2012:22:32:10 +0000] "GET /assets/dist/88166671/js/bootstrap/com
mon.js HTTP/1.1" 200 127238 "http://[redacted]conversations
?ref=si_con" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:10.0) Gecko/201001
01 Firefox/10.0" - - - [redacted] - - - - - - - - 928201
- - - - [20/Feb/2012:22:32:11 +0000] "GET /assets/dist/88166671/js/overlays/exte
rnal-link.js HTTP/1.1" 200 487 "http://[redacted]/conversati
ons?ref=si_con" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:10.0) Gecko/201
```

Know when the house is burning down



Security

[redacted]/security.php?m=1&time=...

Know when the house is burning down

- Methodology:
  - Instrument application to collect data points
  - Fire them off to an aggregation backend
  - Build data visualization dashboards

- We’ve open sourced our instrumentation library
  - https://github.com/etsy/statsd

Know when the house is burning down

Now we can visually spot attacks

Know when the house is burning down

But who’s watching at 4AM?

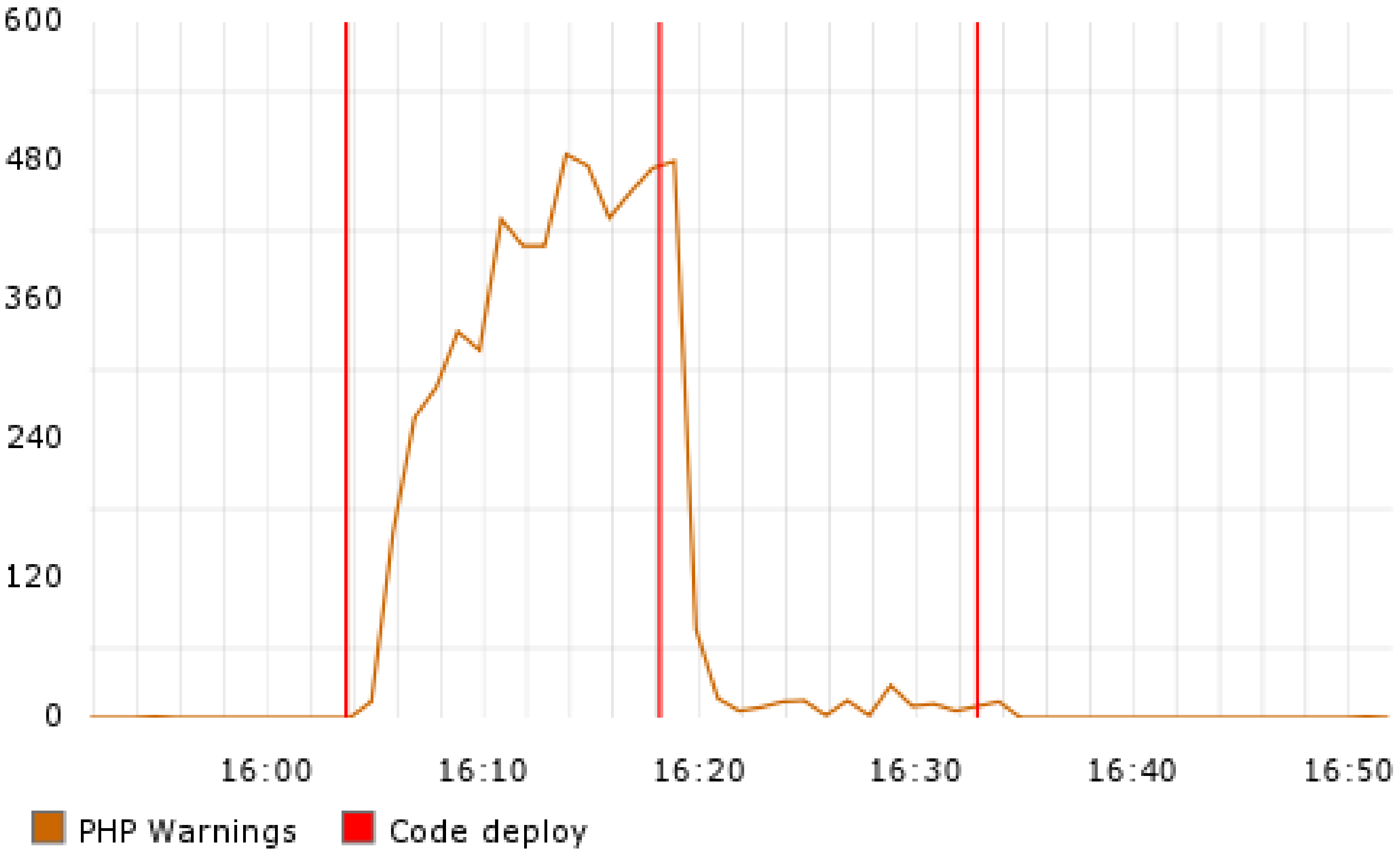
Know when the house is burning down

- In addition to data visualizations, we need automatic alerting

- Look at the raw data to see if it exceeds certain thresholds

- Works well for graphs like this…

Know when the house is burning down

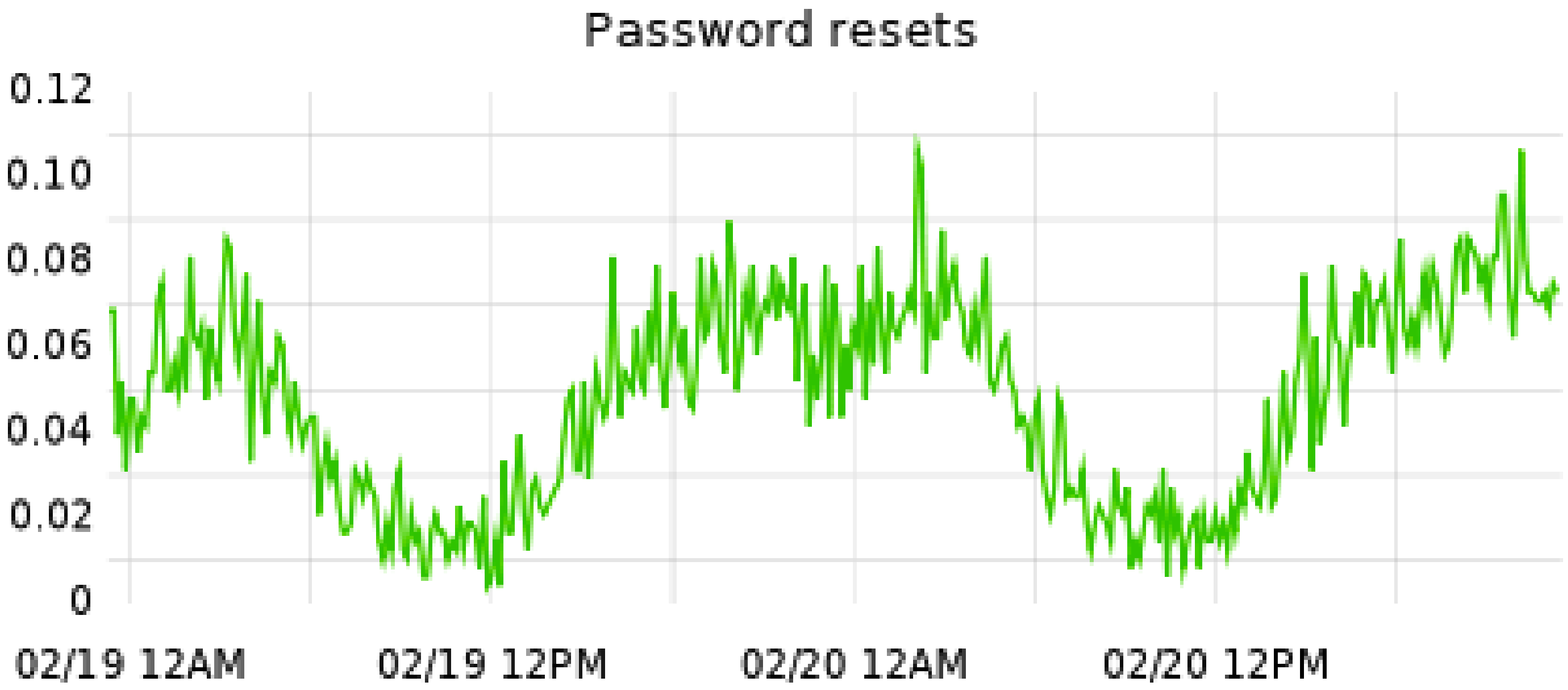


Etsy

Know when the house is burning down

But not like this…

Know when the house is burning down

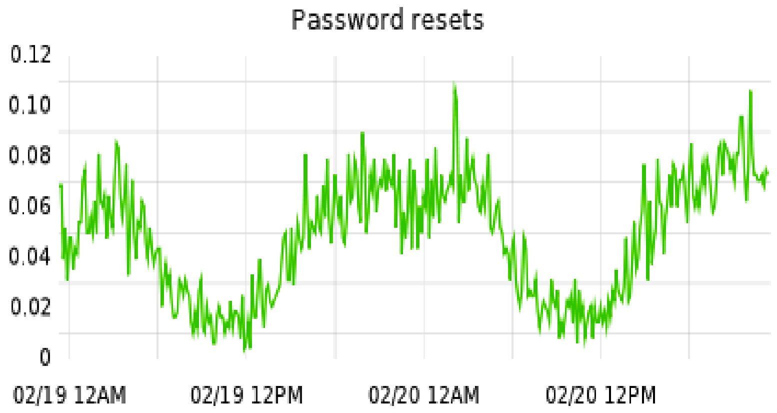


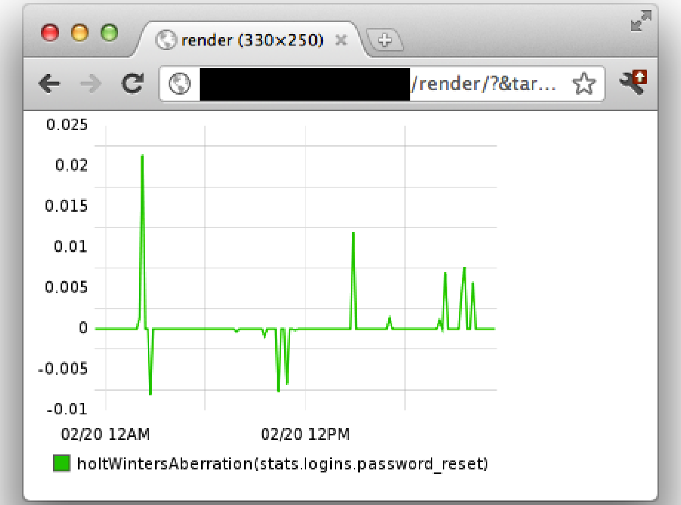
Etsy

Know when the house is burning down

- We need to smooth out graphs that follow usage patterns
- Use exponential smoothing formulas like Holt-Winters
- Math is hard, let’s look at screenshots!

Know when the house is burning down





Know when the house is burning down

- Now that we've smoothed out the graphs…

- Use the same approach as before:
  - Grab the raw data
  - Look for values above/below a set threshold
  - Alert

Conclusions

Conclusions

Conclusions

Don’t turn the Internet switch off

Conclusions

- Make things safe by default
- Focus your efforts / Detect risky functionality
- Automate the easy stuff
- Know when the house is burning down

If you haven’t heckled yet, now is your last chance

Etsy

Thanks!

Etsy

zane@etsy.com @zanelackey