Effective approaches to web application security

zane@etsy.com

@zanelackey

Who am I?

- Security Engineering Manager @ Etsy
  - Lead AppSec/NetSec/SecEng teams
- Formerly @ iSEC Partners
- Books/presentations primarily focused on application and mobile security

What is Etsy?

Online marketplace for creative independent businesses

Scale at Etsy

1.5B pageviews/mo

40M uniques/mo

#51 by US traffic*

* April 2012, Alexa site ranking

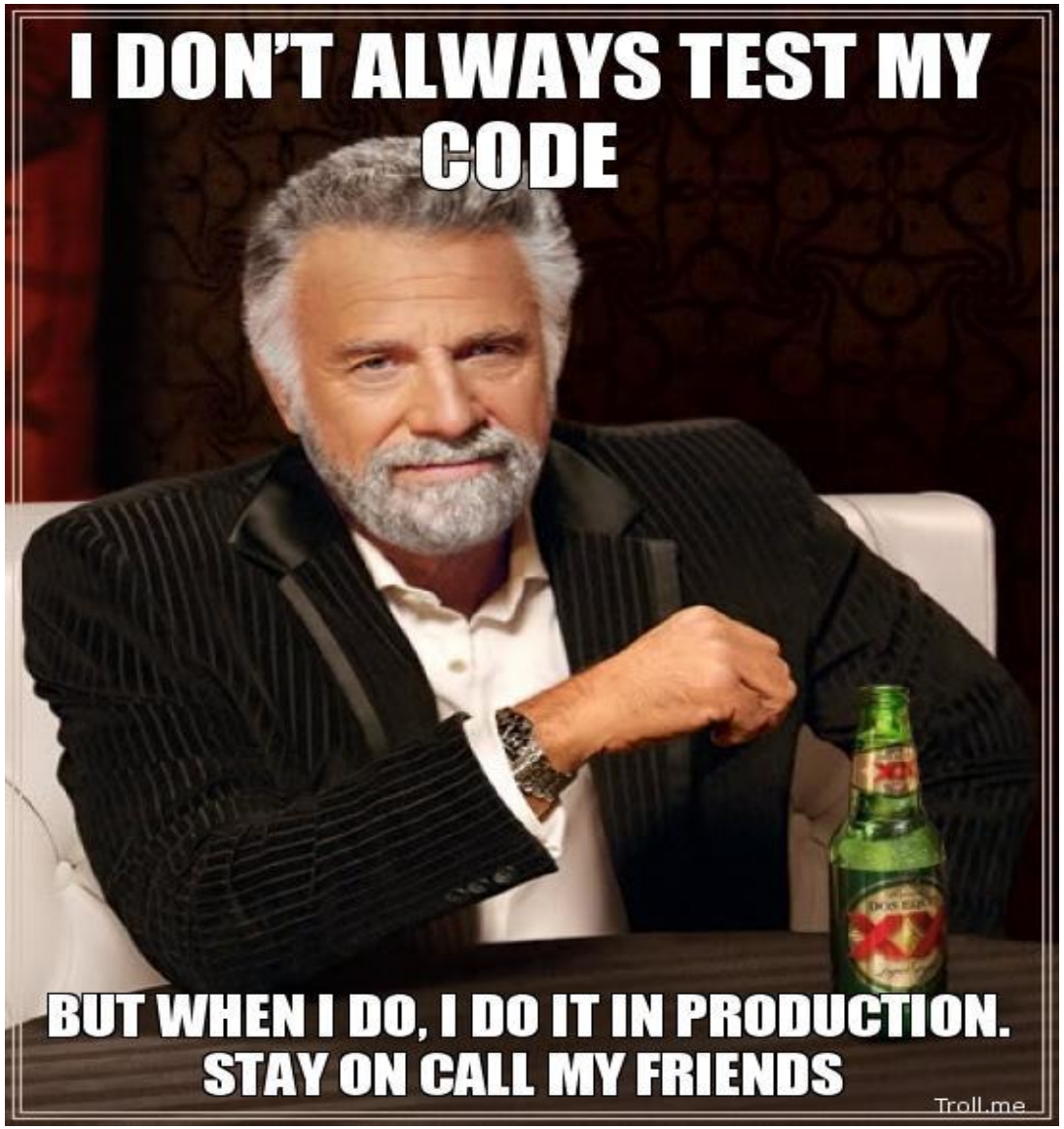
About this talk

Real world approaches to web application security challenges

About this talk

Specifically, techniques that are **simple** and **effective**

Continuous deployment?



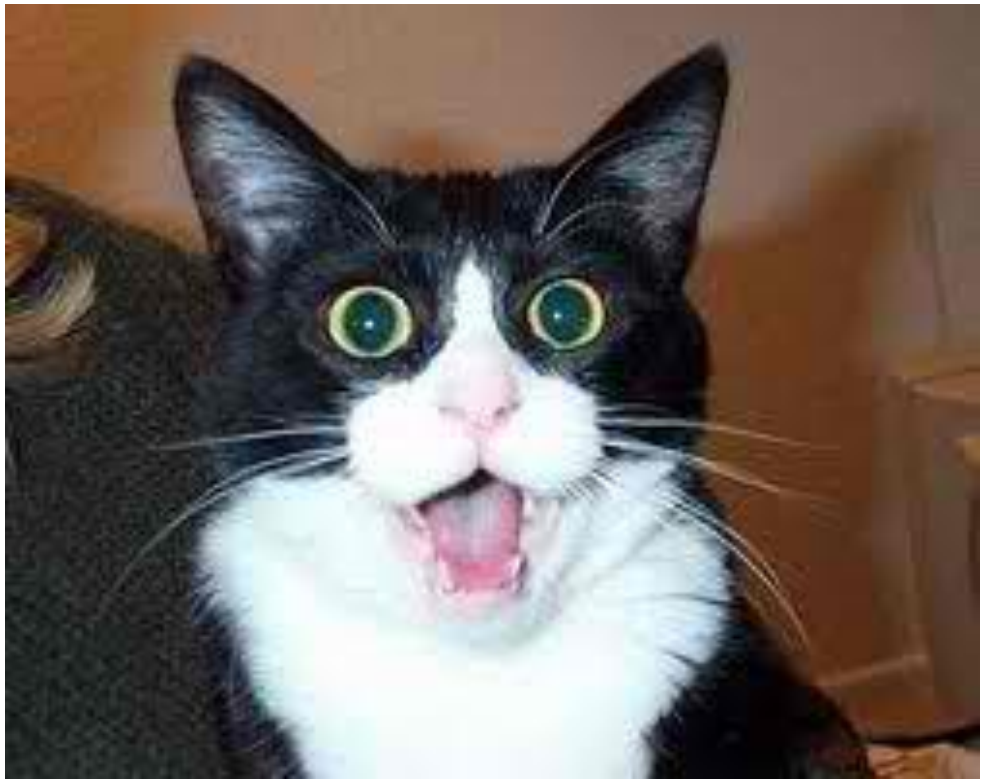
<- What it (hopefully) isn't

Three words: iterate, iterate, iterate


SHH
SHH

Etsy pushes to production **30 times a day** on average

(dogs push too)



But doesn’t the rapid rate of change mean things are less secure?!

Actually, the opposite is true

Being able to deploy quick is our **#1** security feature

Compared to

We'll rush that security fix. It will go out in the next release in about 6 weeks.

- **Former** vendor at Etsy

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

How have the traditional defenses for XSS worked out?

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 block full HTML injection, but not injecting inside JS
  - May put defenses on the client side in JS
  - Etc …

<u>These problems miss the point</u>

Safe by default

- The real problem is that it’s hard to find where protections have been missed

- How can we change our approach to make it simpler?

Safe by default

Input validation

Output encoding

Safe by default

Input validation

Output **encoding**

Safe by default

Encode dangerous HTML characters to HTML entities at the **very start** of your framework

To repeat… **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

Oh yeah!

Safe by default

Now we look for a small number of patterns:

- HTML entity decoding functions or explicit string replacements
- Data in formats that won't be sanitized
  - Ex: Base64 encoded, double URL encoded, etc
- Code that opts out of platform protections

Safe by default

Fundamentally shifts us:

From: “Where is my app missing protections?”

(hard)

To: “Where is it made deliberately unsafe?”

(easy)

Safe by default

Obviously not a panacea

- DOM based XSS
- Javascript: URLs
- Can be a pain during internationalization efforts

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
  - Unit tests alert if hash of the file changes
  - Notifies security team on changes, drives code review

Detect risky functionality

- At the platform level, watching for changes to site-wide sensitive functionality
  - CSRF defenses
  - Session management
  - Encryption wrappers
  - Login/Authentication
  - Etc

Detect risky functionality

- At the feature level, watching for changes to specific sensitive methods
- Identifying these 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
  - Encryption / Hashing
  - Etc

Detect risky functionality

- Unit tests watch codebase for dangerous functions
  - 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 HTML entering the application

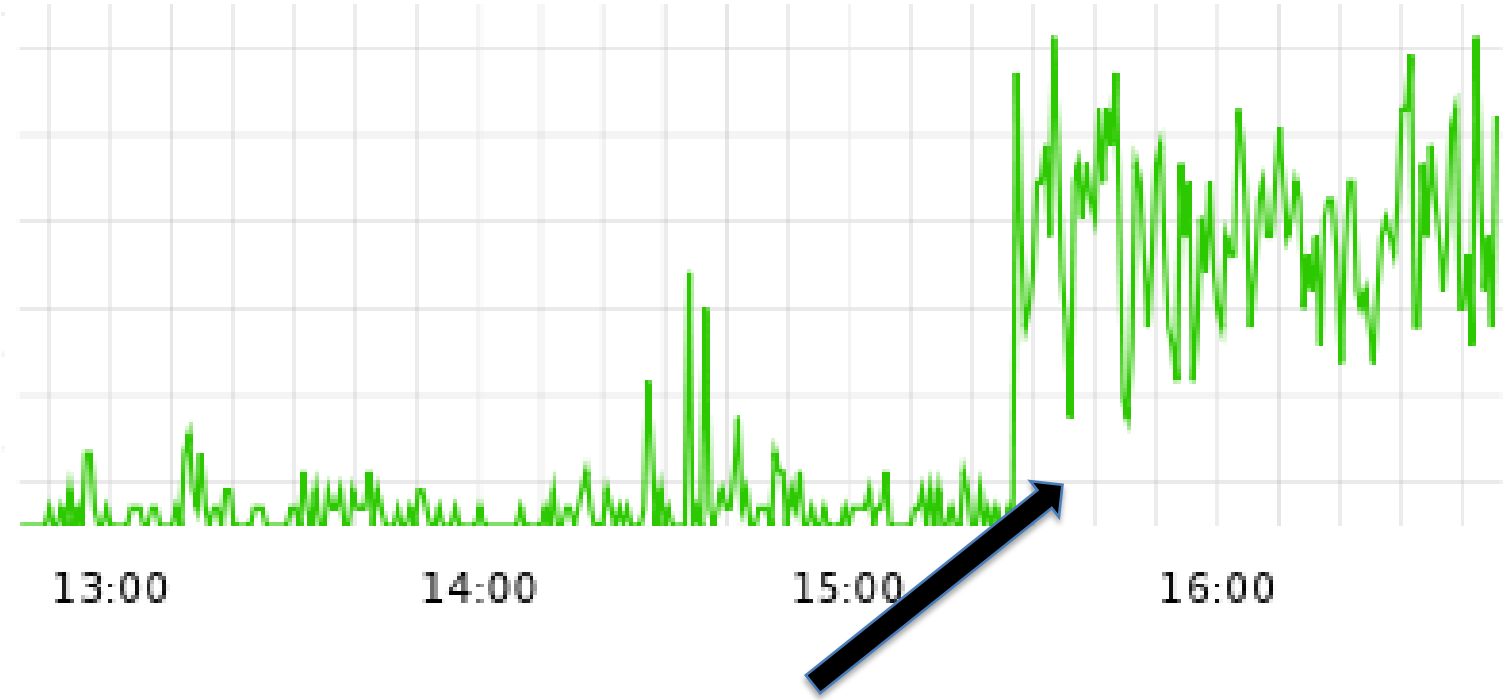
- From there, build a baseline of HTML
  - Entering the application in aggregate
  - Received by specific endpoints

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?
  - Worth investigating

Detect risky functionality



Aggregate increased, time to investigate

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 *Attack 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 an attacker, 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 verifying 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()
  - String.fromCharCode()
  - etc

Automate the easy stuff

We created a tool to use NodeJS as a headless browser for verification

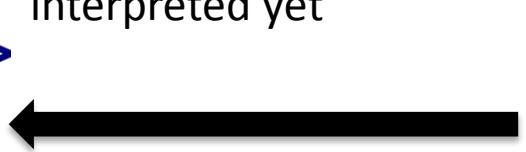
Automate the easy stuff

1. Fetch URL containing potential XSS

Test webserver

Automate the easy stuff

2. Page contents returned to a temp buffer, not interpreted yet



Test webserver

Automate the easy stuff

3. Inject our instrumented JS into page contents

Our JS

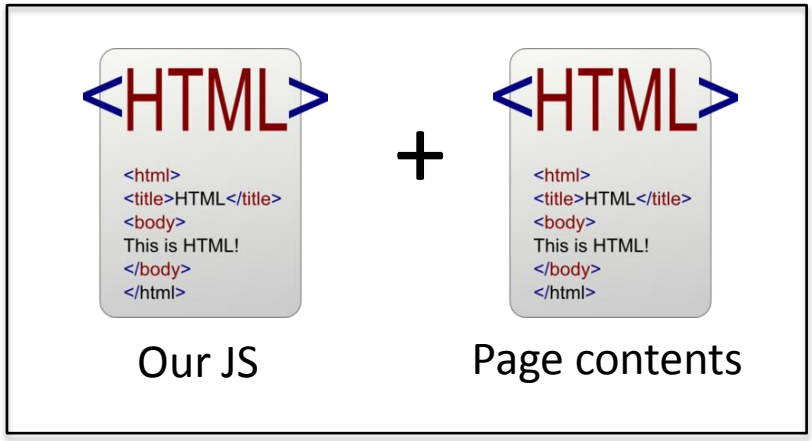
+

Page contents

Test webserver

Automate the easy stuff

4. Combination of instrumented JS + page contents interpreted



Test webserver

Automate the easy stuff

5. If instrumented JS is executed, alert appsec team for review

Test webserver

Automate the easy stuff

- Sample instrumented JS:

```
(function() {
var proxiedAlert = window.alert;
window.alert = function() {
location="XSSDETECTED";
};
})();
```

Automate the easy stuff

- Open sourced NodeJS tool
  - https://github.com/zanelackey/projects
- Combine this approach with driving a browser via Watir/Selenium
  - Make sure to use all major browsers

Know when the house is burning down

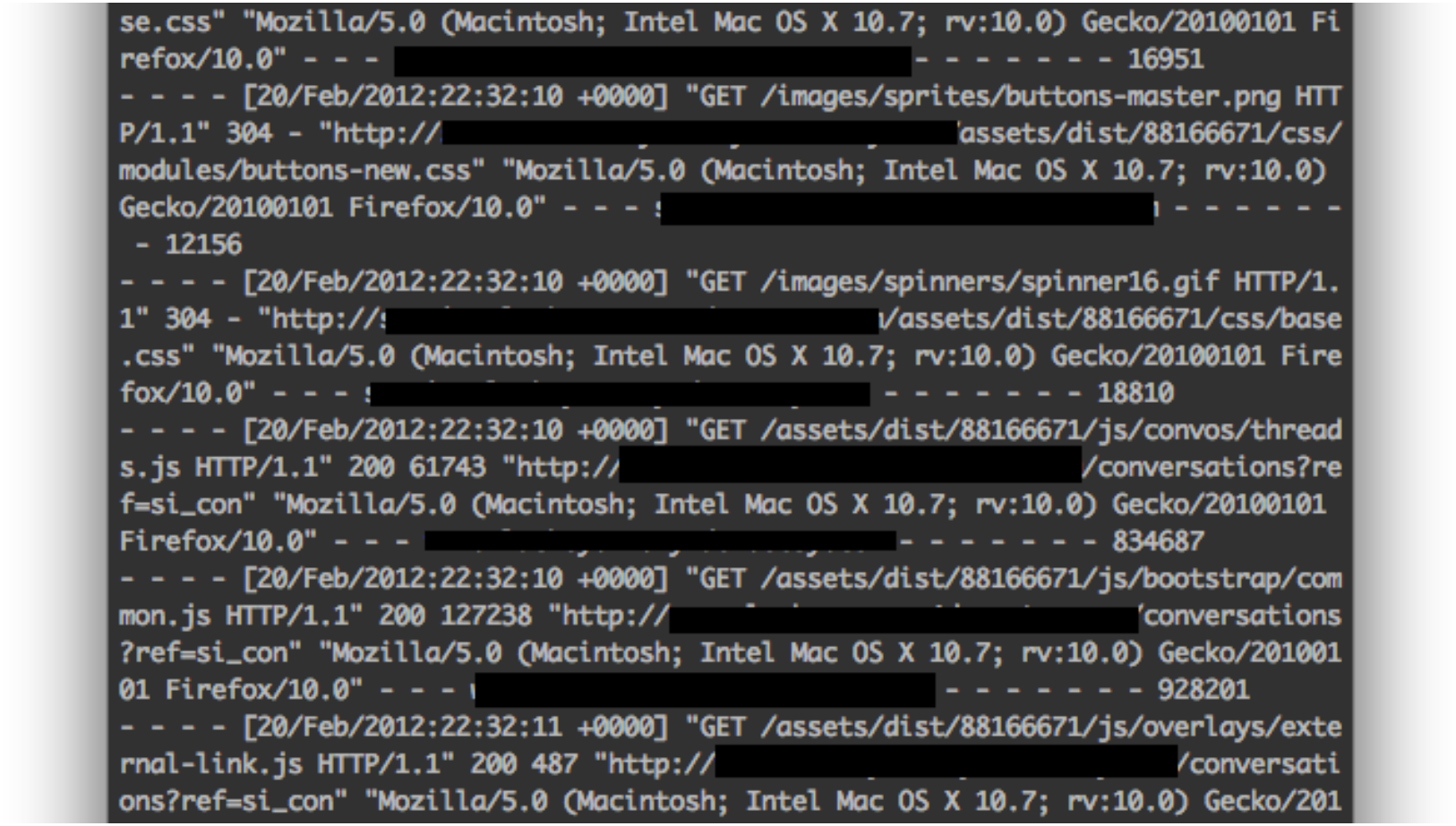
Know when the house is burning down

Graph early, graph often

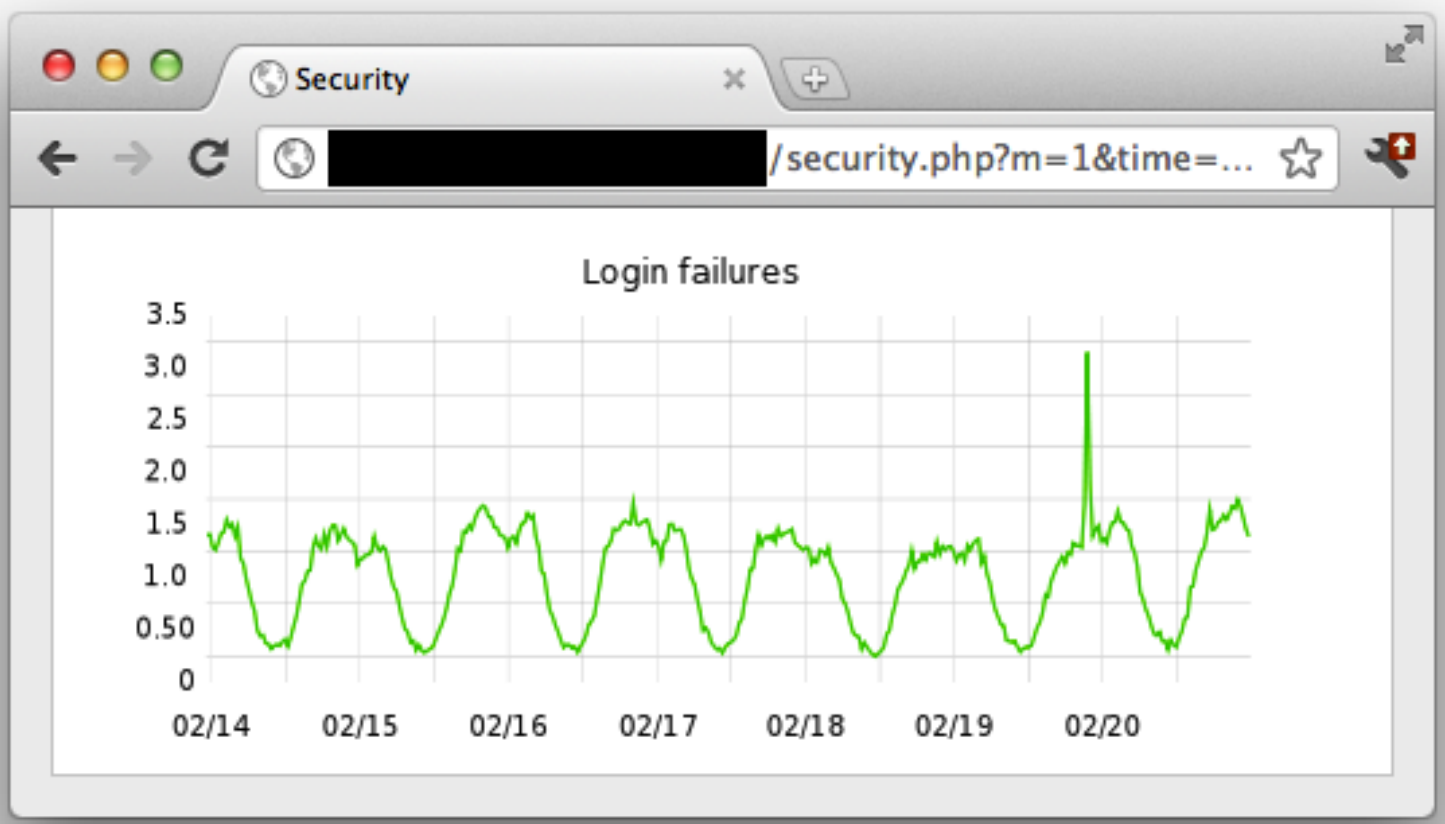
Know when the house is burning down

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

Know when the house is burning down



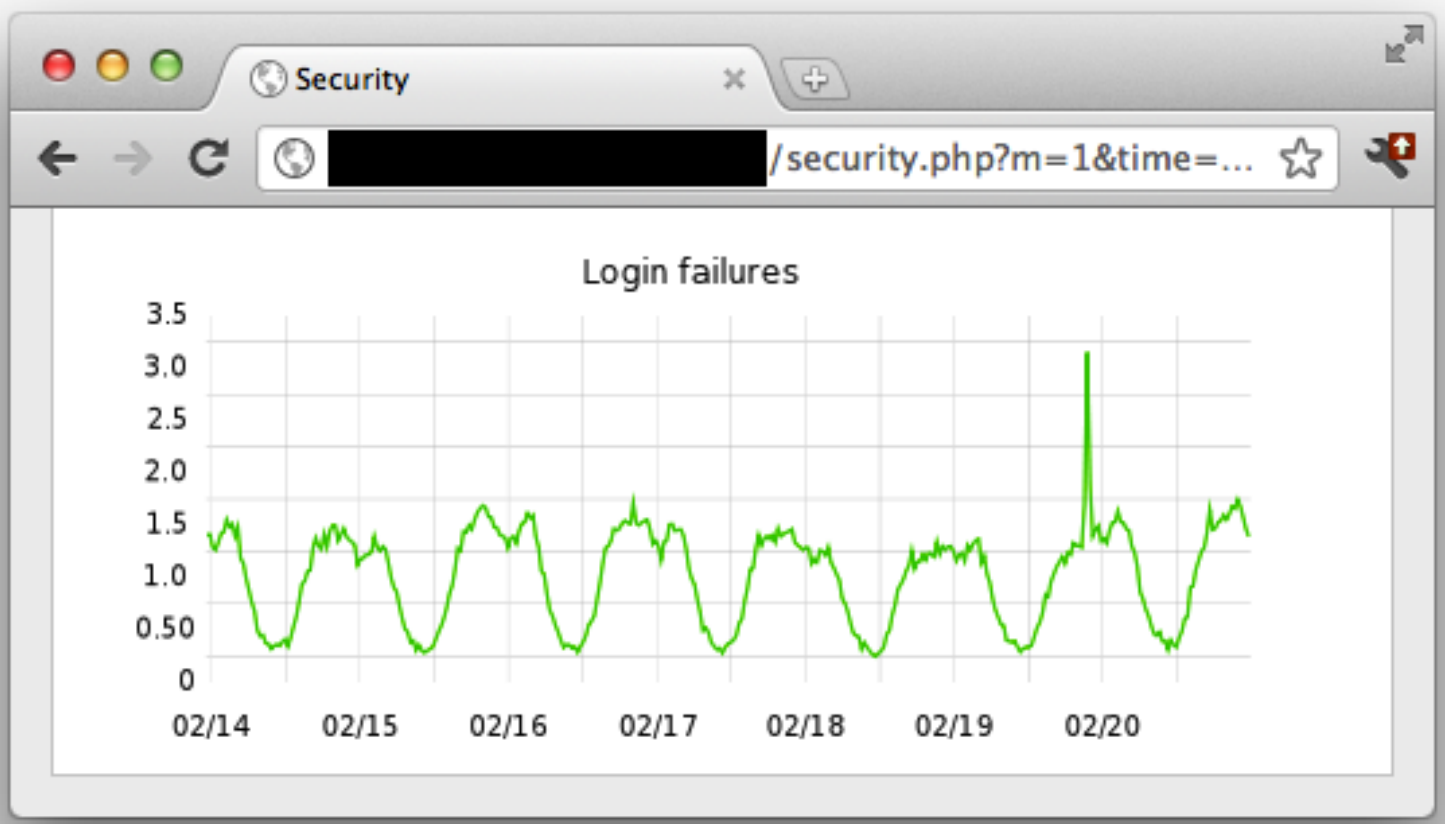
Know when the house is burning down



Know when the house is burning down

- Methodology:
  - Instrument application to collect data points
  - Fire them off to an aggregation backend
  - Build individual graphs
  - Combine groups of graphs into dashboards
- We've open sourced our instrumentation library
  - https://github.com/etsy/statsd

Know when the house is burning down



Know when the house is burning down

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

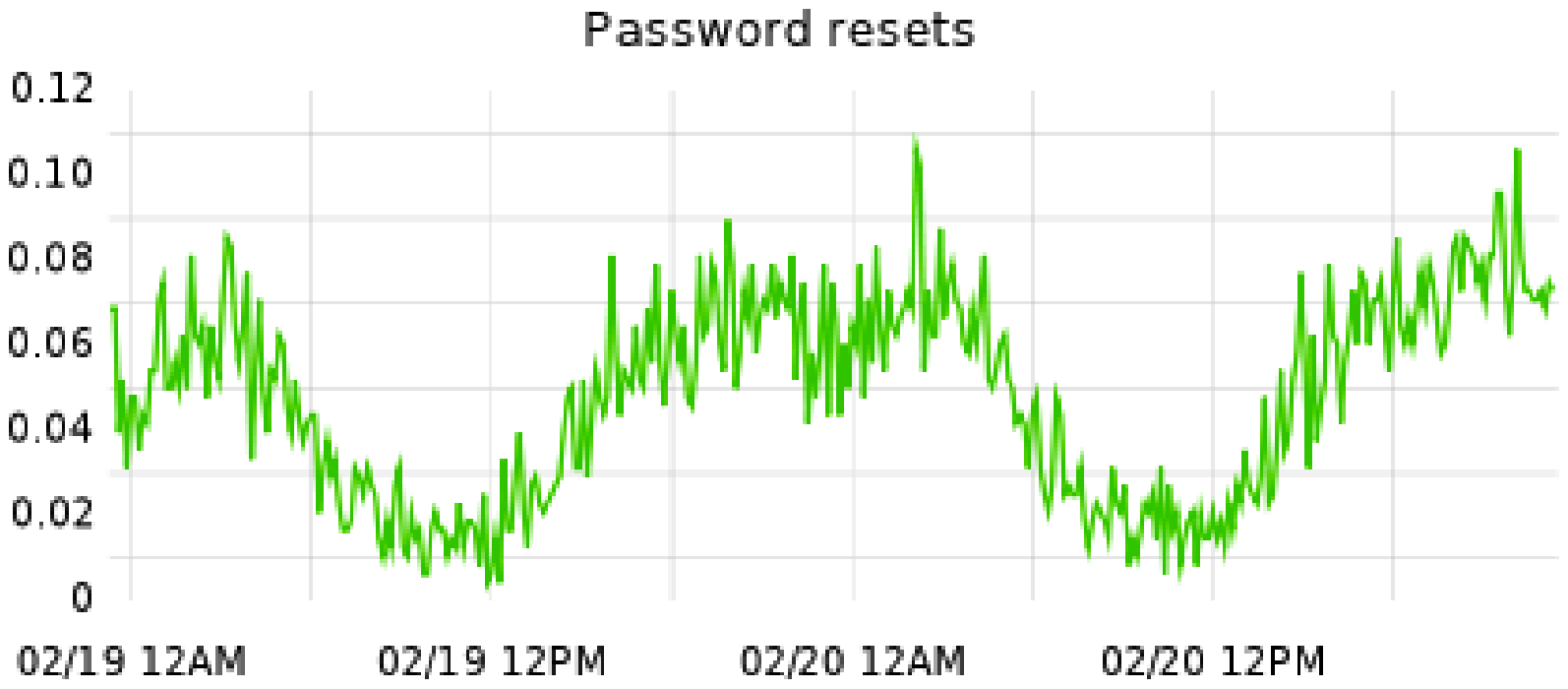
- In addition to data visualizations, we need automatic alerting
- Look at the raw data to see if it exceeds certain thresholds
- Works well for graphs like this…

Know when the house is burning down

Know when the house is burning down

But not like this…

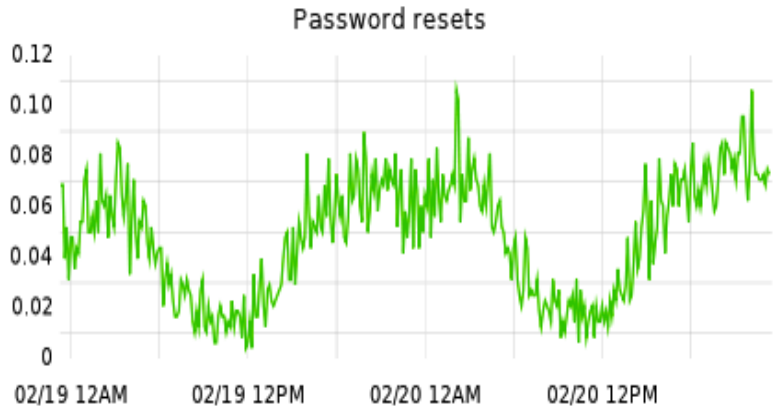
Know when the house is burning down

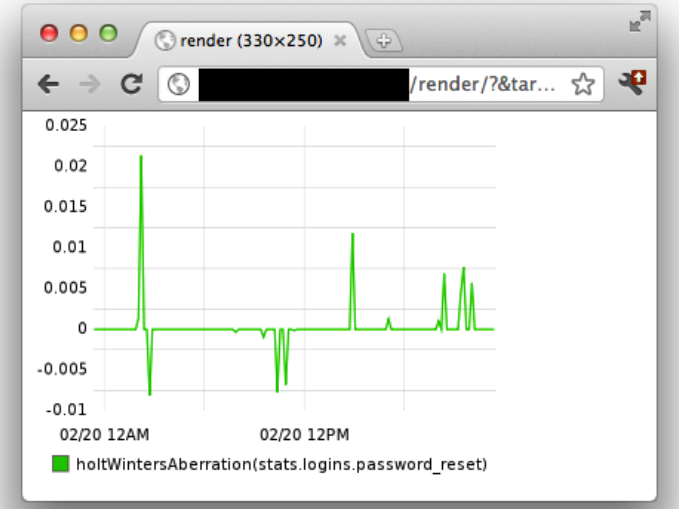


Know when the house is burning down

- 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

Know when the house is burning down

Alert on events that (*should*) never happen

Know when the house is burning down

Successful attacks don’t happen in a vacuum!
They generate **signals**

Know when the house is burning down

- Figure out what the signal of a weakness being identified looks like
- Alert when a signal occurs
- Fix the identified weaknesses

Know when the house is burning down

Two examples: SQLi and code execution

Know when the house is burning down

- The road to exploited SQLi is littered with broken queries

1. Watch the logs for SQL syntax errors
2. Alert when they appear
3. Fix the lack of validation allowing the error

Know when the house is burning down

- Further along the attack process, a SQLi attack looks like… your database
- Sensitive DB table names shouldn’t be showing up in requests
- Alert if they do!

Know when the house is burning down

A funny story about code execution…

Know when the house is burning down

- preg_replace() in PHP has an interesting modifier

"e (*PREG_REPLACE_EVAL*) If this modifier is set, preg_replace() does normal substitution of backreferences in the replacement string, evaluates it as PHP code, and uses the result for replacing the search string. "

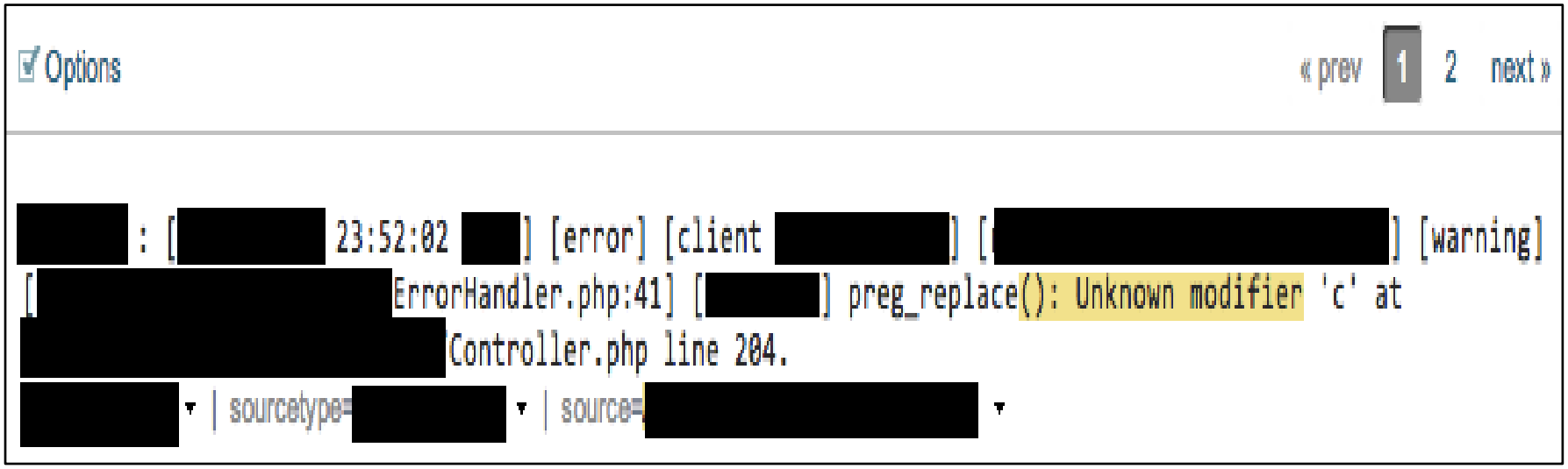
Know when the house is burning down

- preg_replace() in PHP has an interesting modifier

"e (*PREG_REPLACE_EVAL*) If this modifier is set, preg_replace() does normal substitution of backreferences in the replacement string, **evaluates it as PHP code**, and uses the result for replacing the search string."

Know when the house is burning down

- What do the signals for this look like?



Know when the house is burning down

You can’t fix what you’re not alerting on

Conclusions


Safety begins with you.
Metro
237
Metro
DANGER
HIGH VOLTAGE

Have the ability to deploy/respond quickly

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

Thanks!

zane@etsy.com @zanelackey

References / Thanks

- DevOpsSec: http://www.slideshare.net/nickgsuperstar/devopssec-apply-devops-principles-to-security

- Special Thanks:
  - Nick Galbreath, Dan Kaminsky, Marcus Barczak