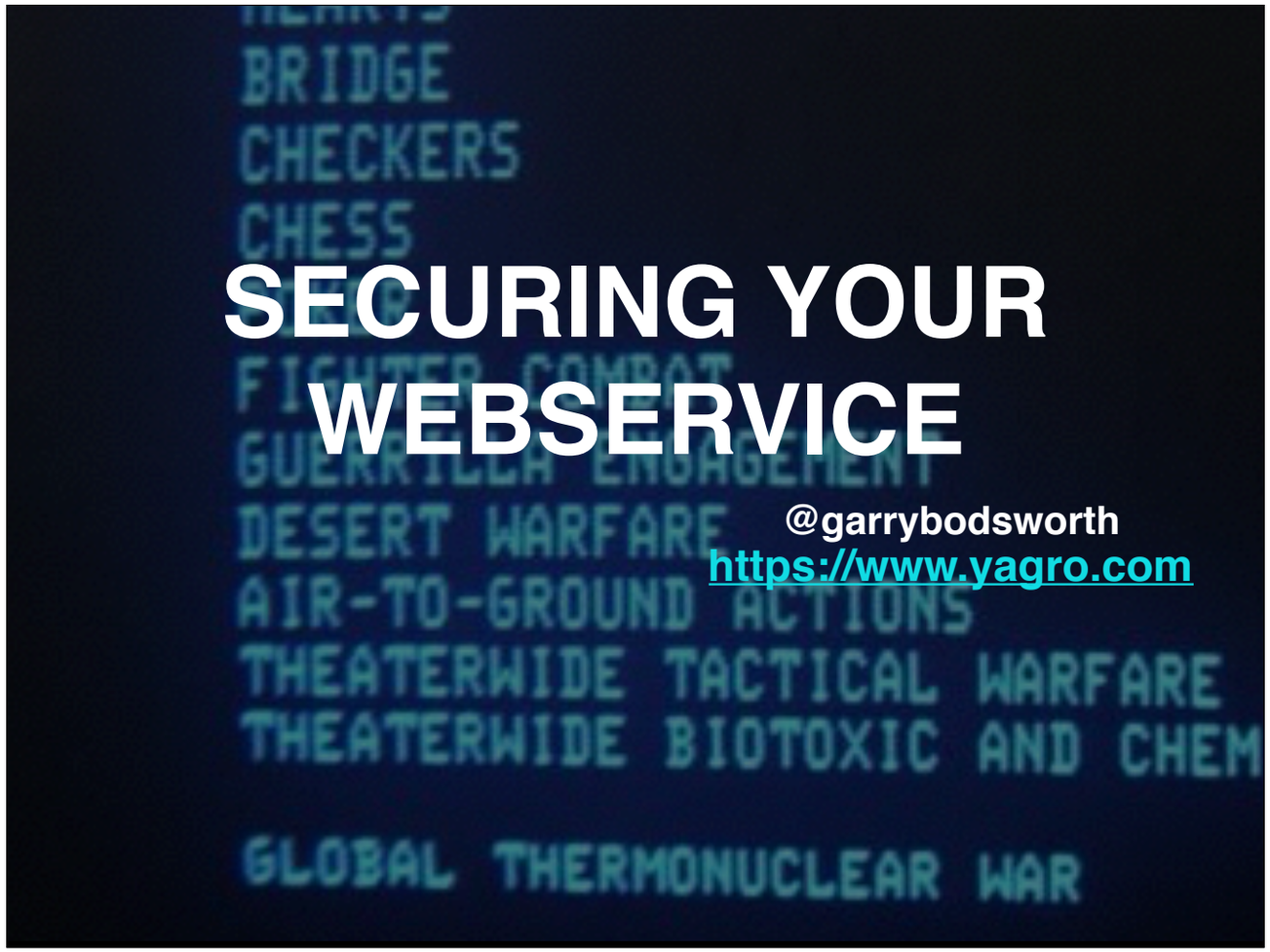
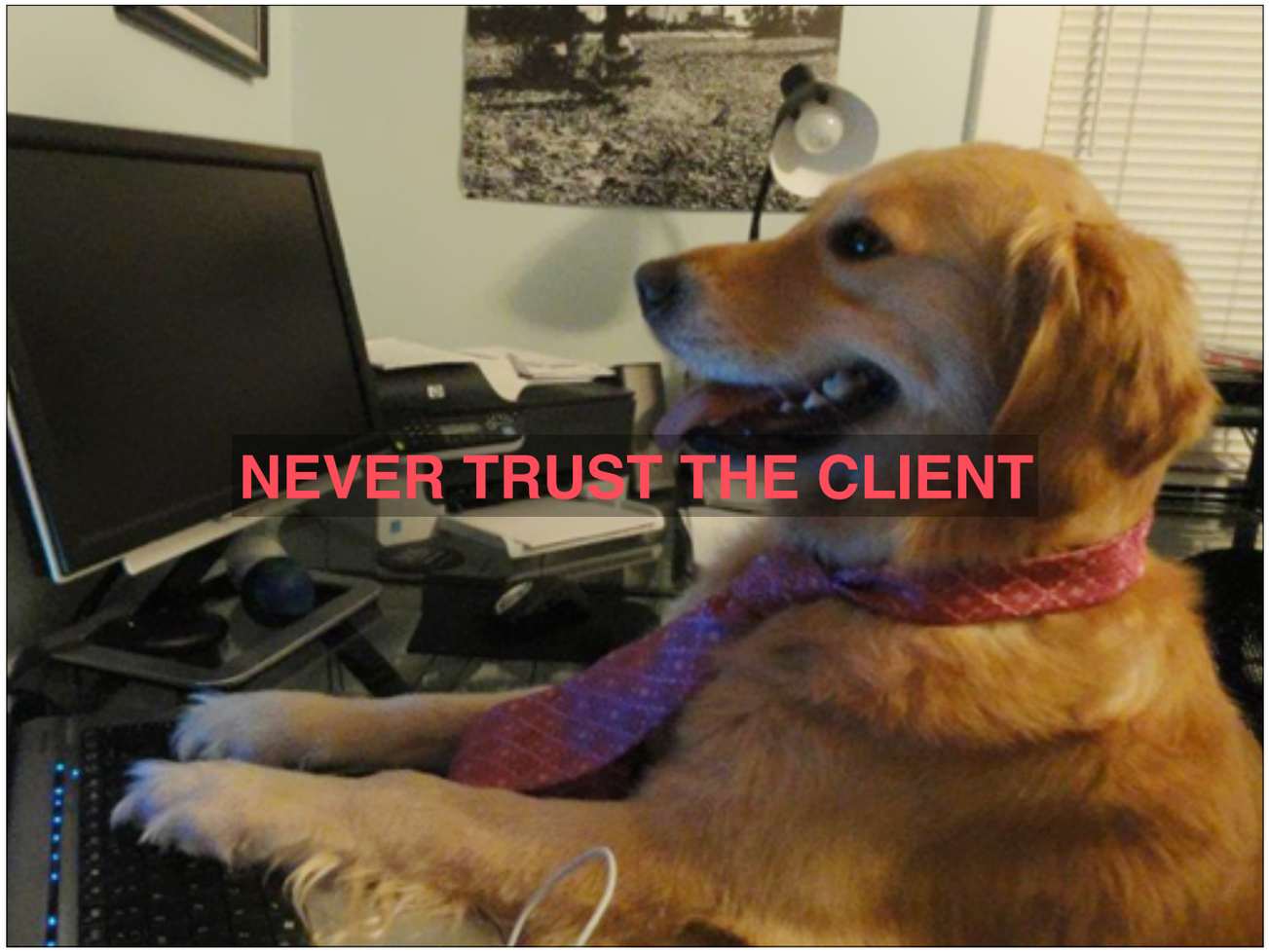



# SECURING YOUR WEBSERVICE

@garrybodsworth
https://www.yagro.com

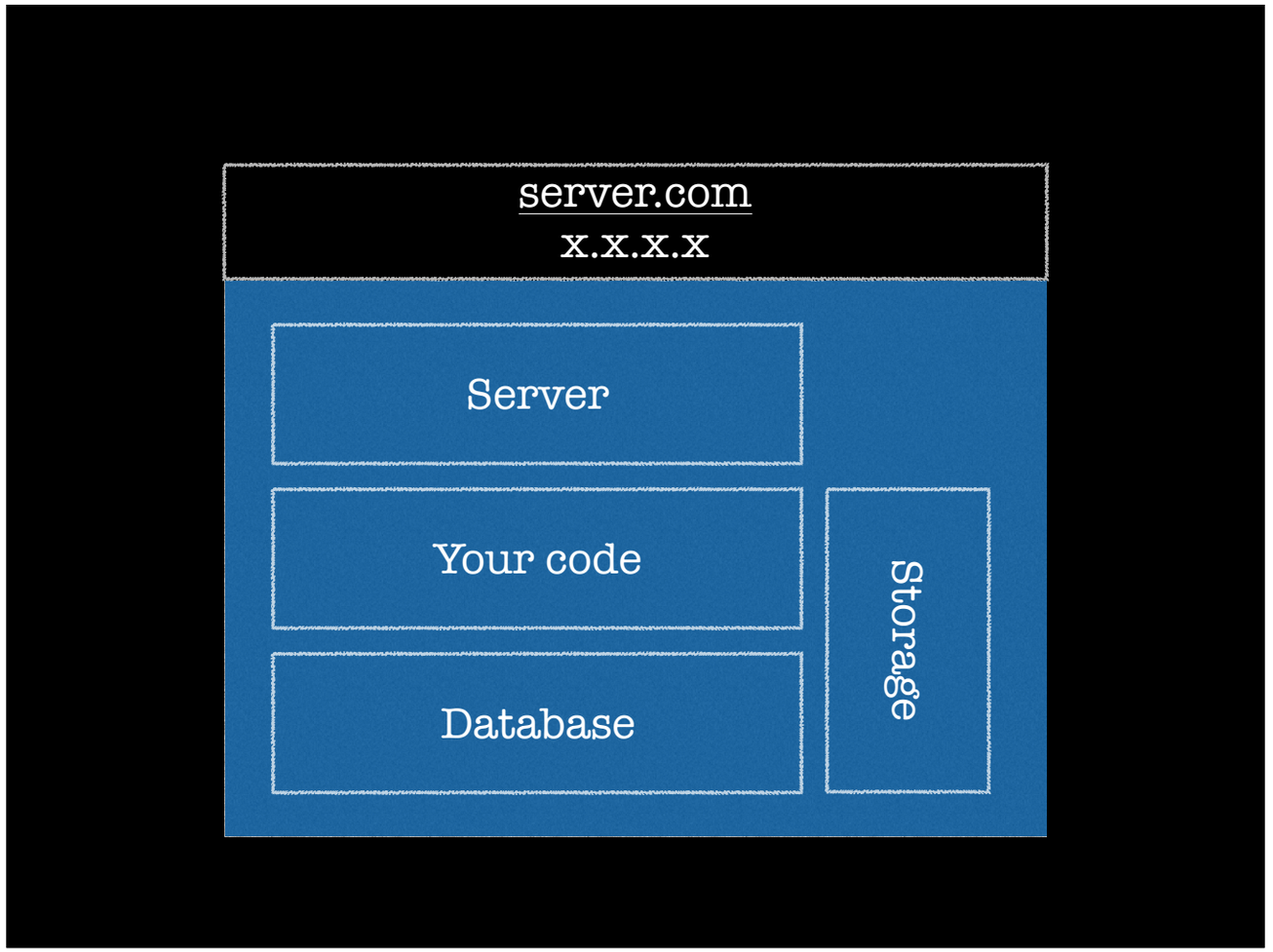
About me:

- devices
- web services
- Microsoft
- Bromium + browsers



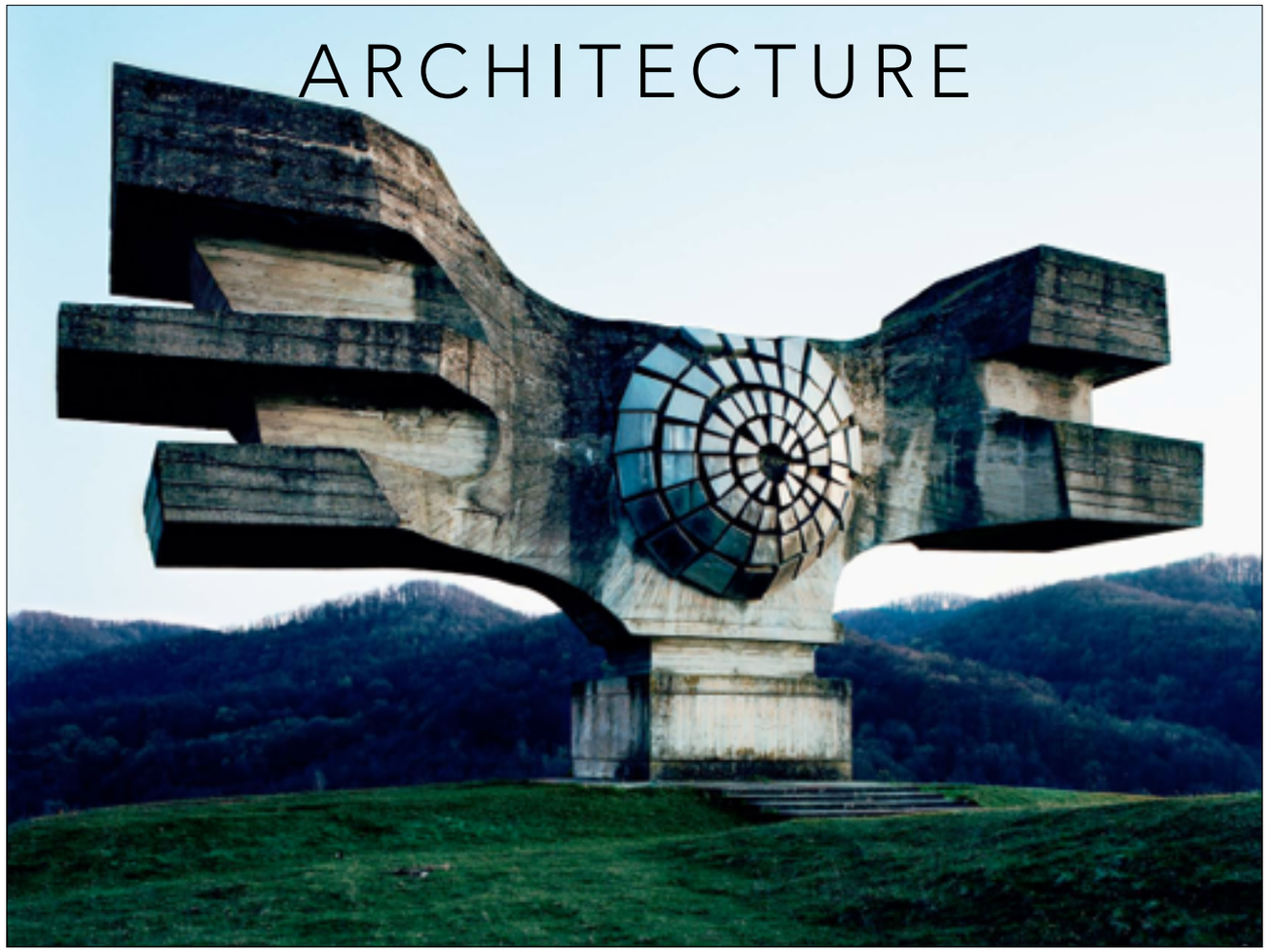

I’ll come back to this.

Critical thing to learn when building all webservices

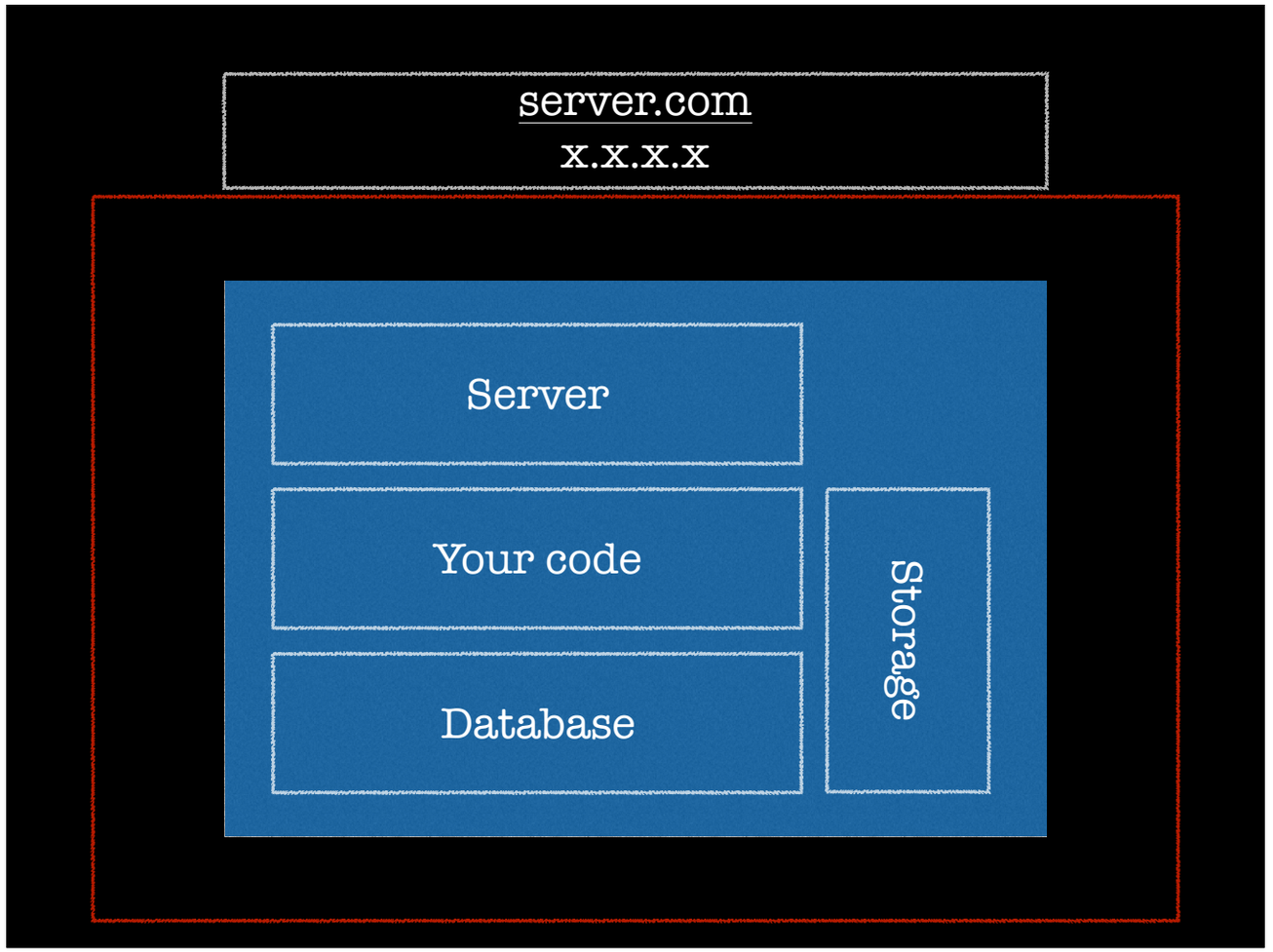



Standard server layout
Exposed single IP address
DNS maps to only one
Server fully exposed on Internet
Must always be kept patched and fully up to date
Compromise this machine and get everything

# ARCHITECTURE

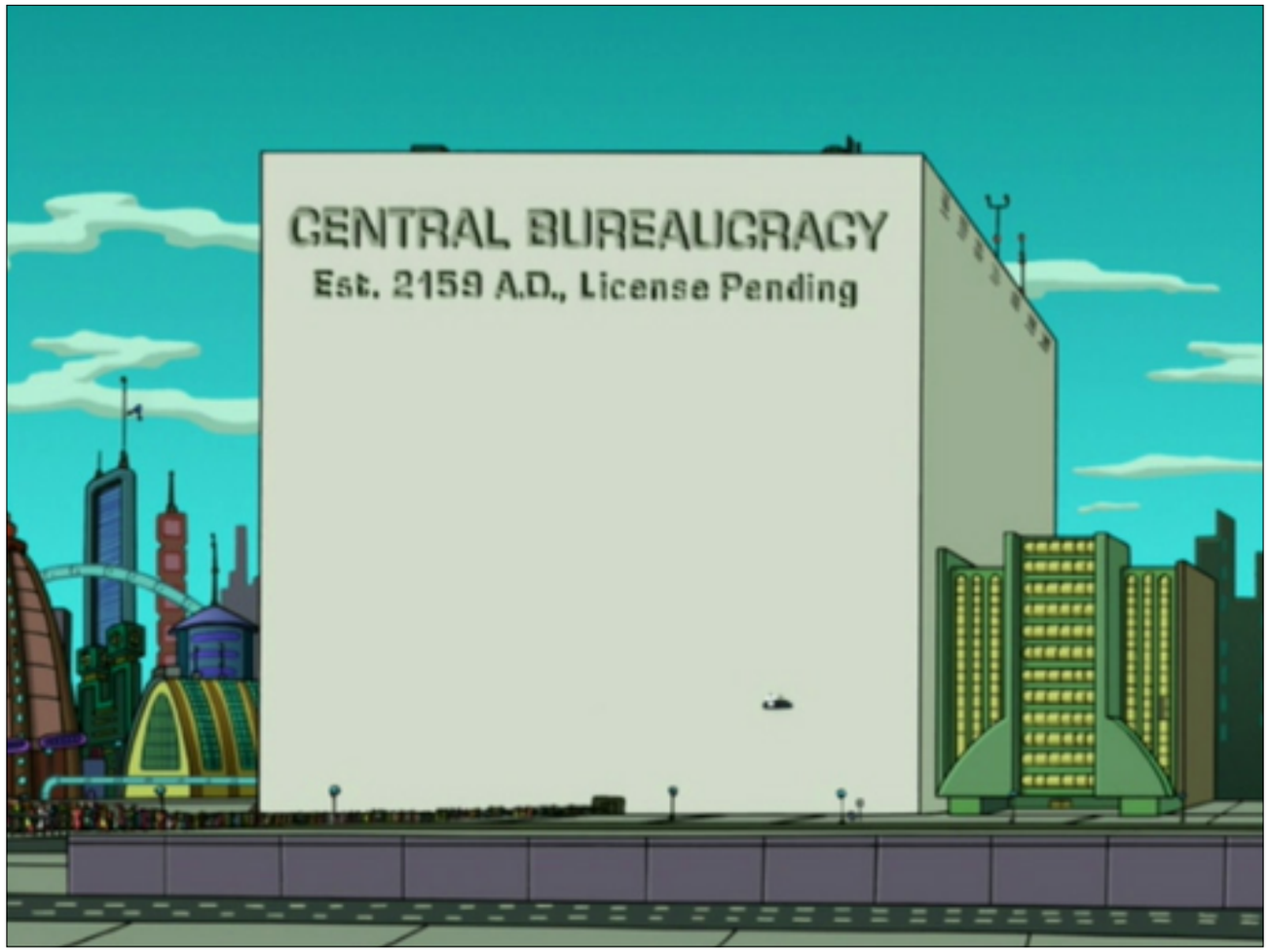


Architecture can solve some of the basic issues surrounding access to servers




Who recognises this?
First of all the most basic thing we can do is protect the server from the general internet
IP tables/firewalls are depressing and confusing.
Put in its own private network which is NATed away from the world.


Load balancing + reverse proxy
Decouple DNS name from target
HTTPS termination
Only way for public internet to get into your network
Can target multiple machines

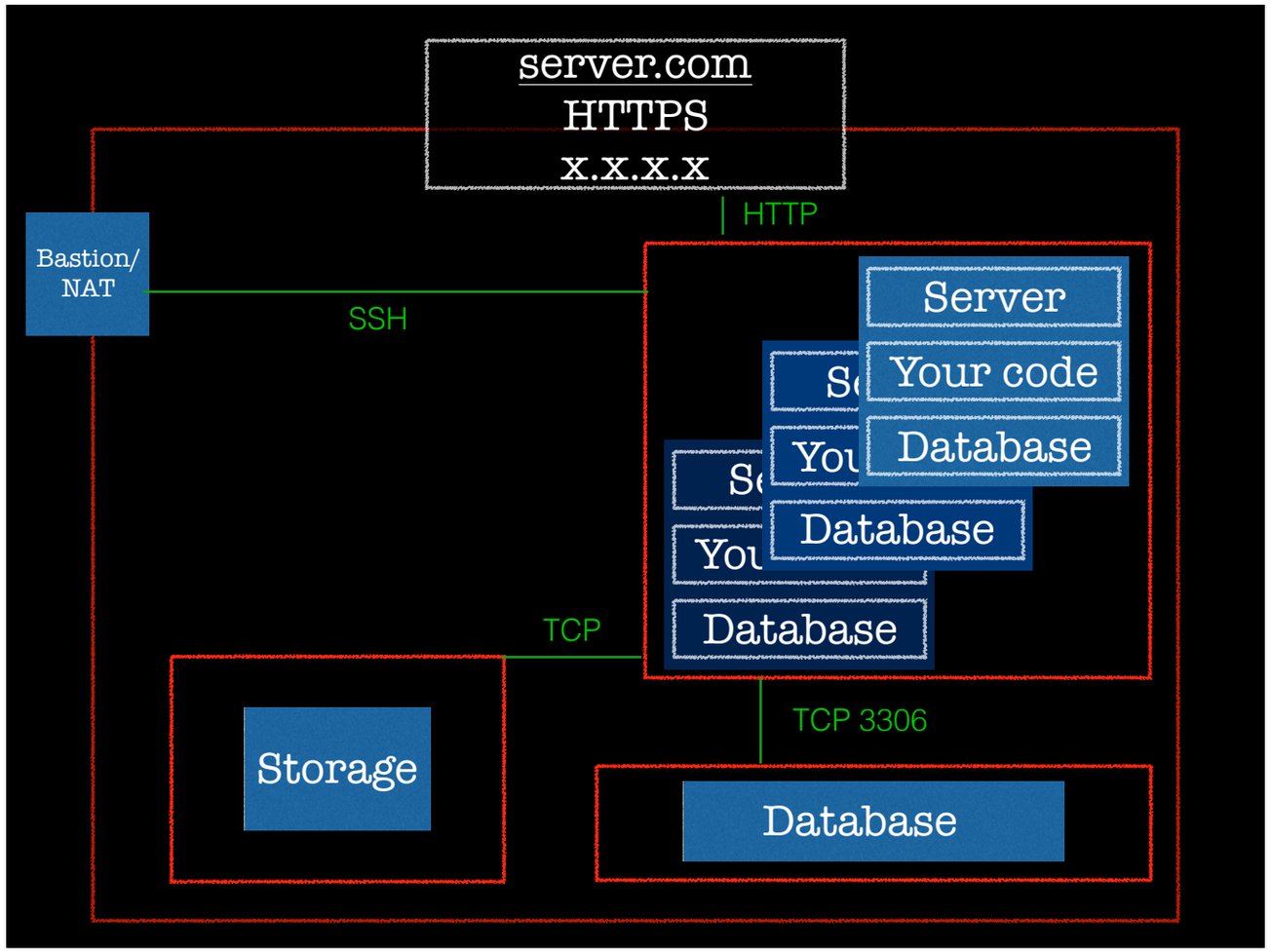



Separate the database
Completely remove any public access from the internet
Only allow clients that should access it (through firewall rules)

Also - make sure you only need the root user for things like backups.
Create a user per database and also have the ability to have readonly users

Advantage of single machine model is unix domain sockets and configuring so no external access.

How do you get into the servers?
Bastions or NATs
Depending on the deployment model of push vs pull (ie, Ansible vs Puppet)
Only should allow ssh for bastion and also only from known IPs
For NAT requires setting up the server instances iptables and networking so they can access the internet




We start to look a little like this.

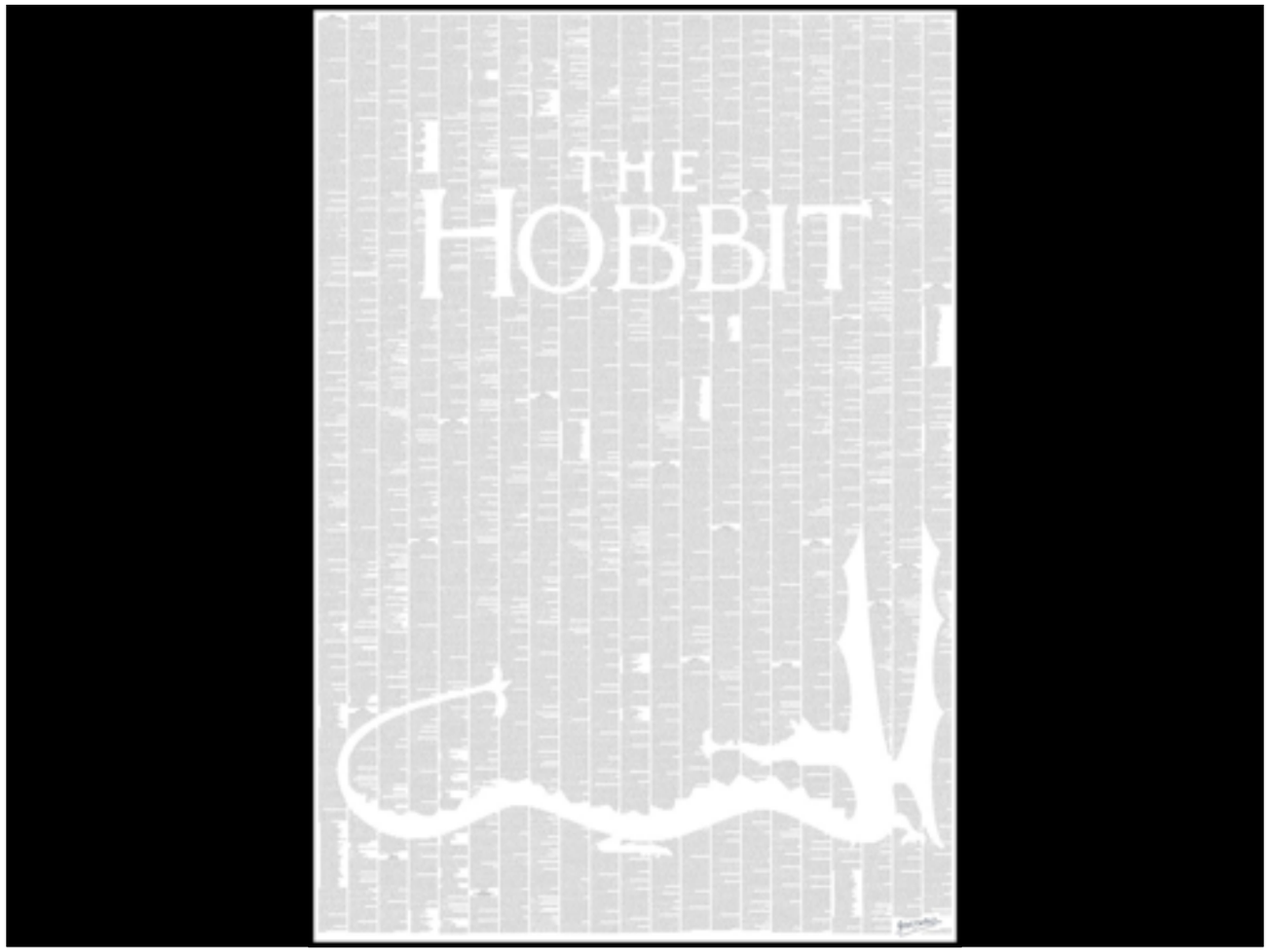

Out of the box your choice of framework is insecure
What can you do?
Learn stuff
Some have best security practices - e.g. django cookie cutter


Right, we've got problems, now we need to fix them.


Lets talk a little about cookies
What are they?
How do they work?
What properties do they have?
domain, path, secure, httponly


Tokens + Oauth2

- Unwittingly exposing internal implementation and data relationships - I get quite a lot of push back on this - especially from REST purists.
- Your API should abstract into higher level semantics, because if it looks like the building blocks you can't protect how those interact easily.
- You give the mobile client the user ID when it registers then it can use that to identify to the API.
- Can you think of a way to exploit this?
- Change the number!
- OK, it's a GUID - can you think of ways to get the numbers?  Get your friends numbers!  Go mad!
- Does the client need to know the user ID?
- Usually login session and user ID should be decoupled.
- For instance, you can use a social login that maps to a particular user, this can be managed completely server side.
- Is a mobile client the user?
- TOKENS!
- You need something to represent the user allowing a piece of software to do something on their behalf.
- The token should encapsulate
  - a set of permissions




SPA + JWT
JWT supposed to be stateless
Bad for sessions
Can be reused in other contexts
Javascript can access it

Combine with sessions?
Then you have CSRF problems again.  There is no clear example.


- BANHAMMER!
- You get granular revocation all the way up to nuking a user from orbit.
- Doesn’t need to compromise the whole account.
- Makes more complex ownership, sharing and workflows possible
- The matchmaking example - list of users or a list of matches identified uniquely.

# SSL CERTIFICATES


- Who has bought one?
- Who has generated one?
- Who has invalidated one?
- Who can deploy one?
- Ordering one - certificate signing request
- Know how to get one into production?  Know about IP/vHost limitations and SNI?
- Have you made your own certificate authority before?
- Managed CRLs before?
- Recent revocations
- Recent applications - Amazon, LetsEncrypt
- Configuring servers, protocols and ciphers

# ENCRYPTION

- Transport
- Data at rest


- Cryptographic hash (one-way)
- Reversible (two-way)

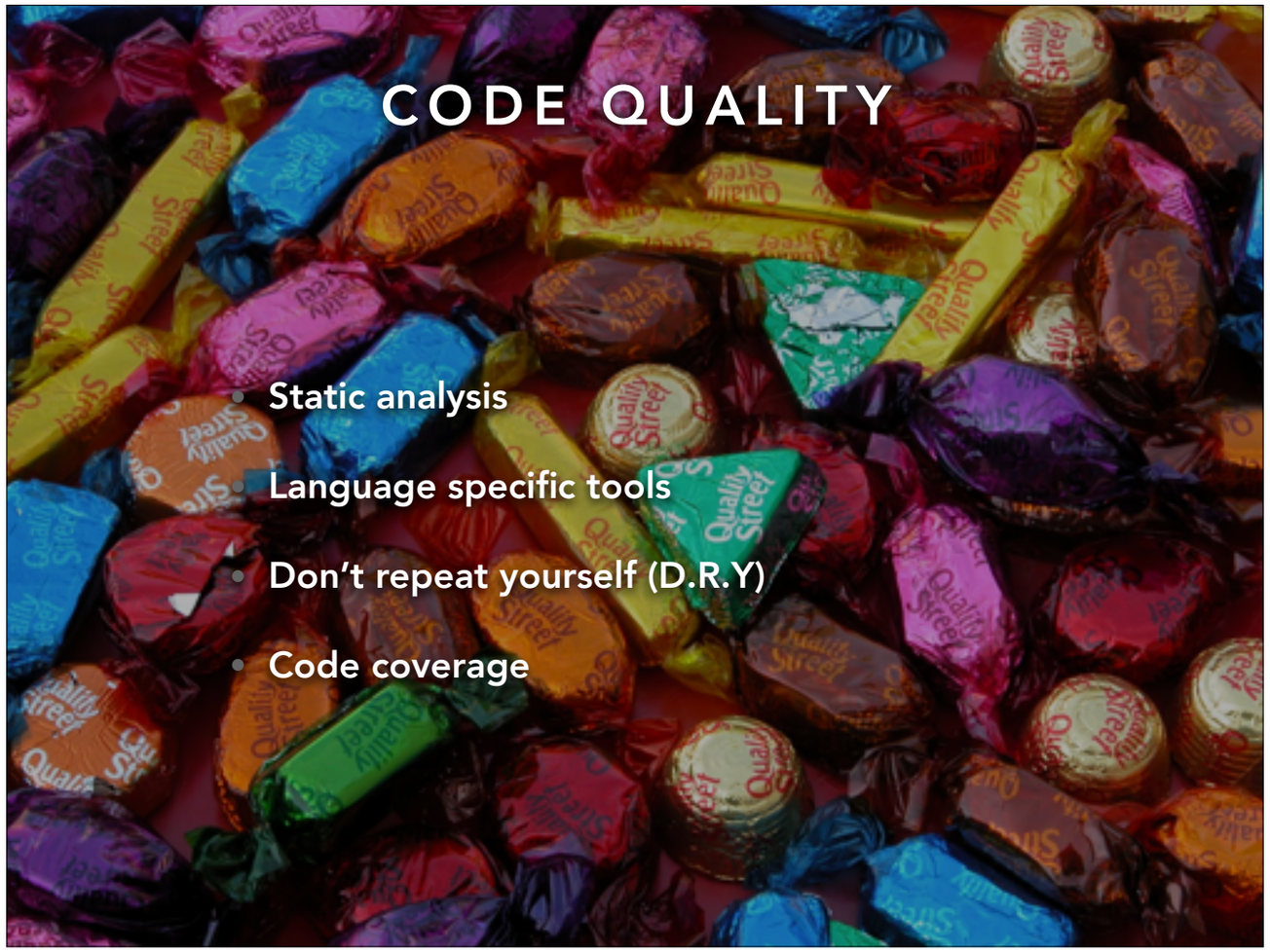

- Symmetric
- Asymmetric

- Transport - SSL, TLS, HTTPS
- Data at rest - your storage - database, filesystem, nosql
- Cryptographic hash - for storing passwords
- Reversible - you eventually need the original data like files on your hard drive
- Symmetric - Both know the same secret - less computationally intensive typically
- Assymetric - Public key encryption - big primes


- We know all your secrets
- Where do you store
  - certificates
  - passwords
  - critical information
- Source control?  VERY BAD
- Secret server/service?  Allows you to decouple secrets from the service.
- Encrypted file? Not bad.
- Rolling keys - have a plan for keys being changed or rotated.  Reasons: compromise, age, management….
- USE PASSWORDS ON YOUR DATABASE

# SQL INJECTION

- SQL Injection
- Always use the ORM!
- Escaping!
- Django has no known SQL injection attacks open




- Start with the most secure settings.
- All warnings on, all warnings as errors
- Static analysis - stops the dumbest of errors
- Language specific tools - Python: e.g. PEP8, PyLint, PyFlakes
- Don’t Repeat Yourself - fix a bug once - easier to test
- Code coverage - if the code is never executed except in production how do you know it is good?


- Continuous deployment
- 3rd party code drops and updates
- Updating certificates and third party service dependencies
- Use short(ish) expiries to get used to updating
- Be ready to switch providers, the decision may be out of your hands for money, buddies, or plain security disasters (I have an example there…)

# TESTING

- Load
- Unit
- API/Client
- Browser
- Integration
- Tests should run easily on developer’s machines
- Developers should understand and edit the test code and framework
- Developers should run local SSL server
- Should know how to regenerate snake oil


Exploits your trust of another site to craft an attack - forms with side-effects
Referer can be spoofed even if you check

CSRF “synchroniser” tokens in forms
Cookie to header token - sets in JS - relies on same origin security
Uniqueness of CSRF tokens

Related - single use tokens

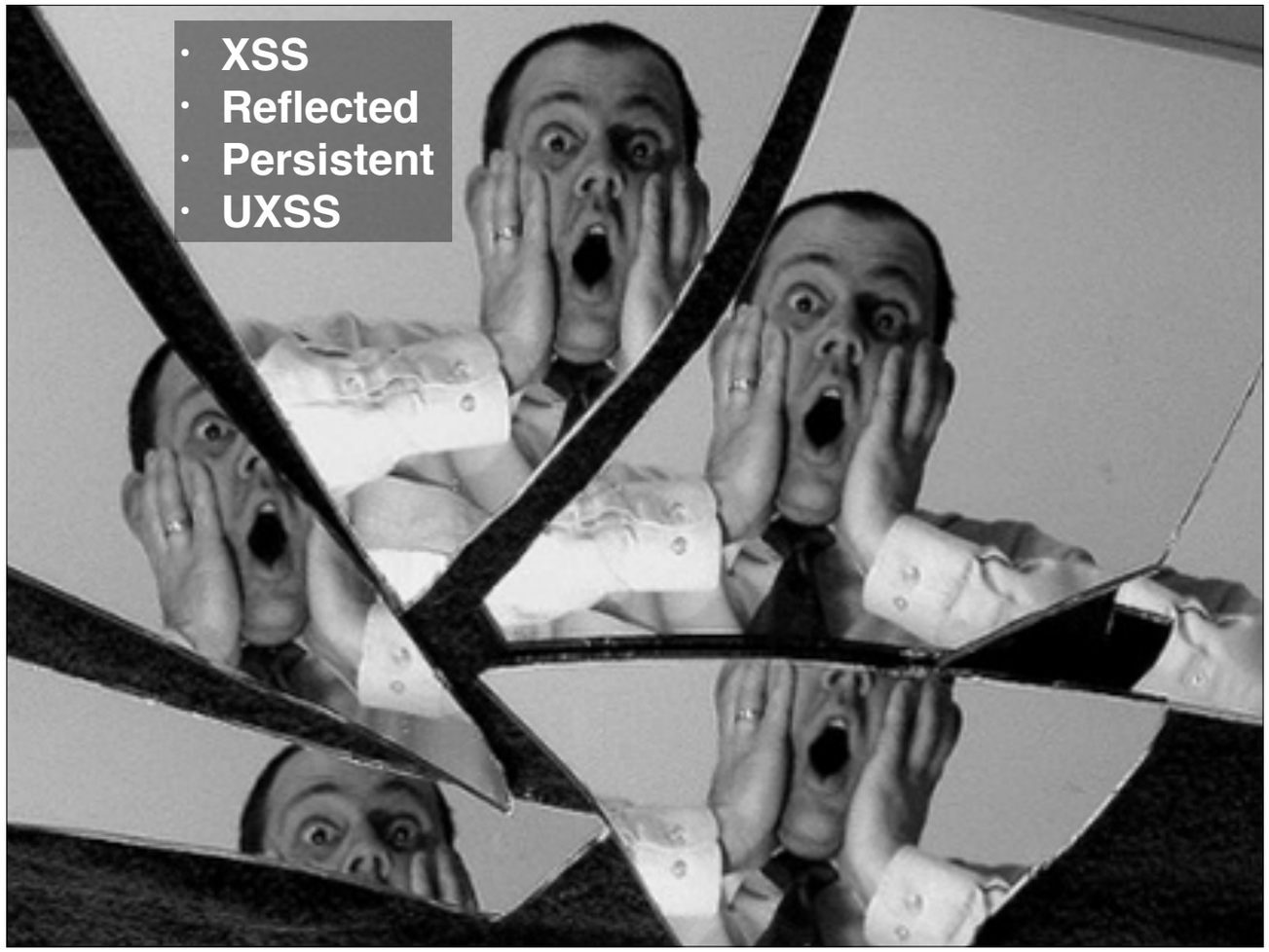
# MIXED CONTENT


- Mixed content
- HTTP can extract HTTPS content on the page - they share the DOM
- Understand your cookies, secure, httponly

# IFRAMES

Use XFrame headers - you have to be sure the browsers implement it right

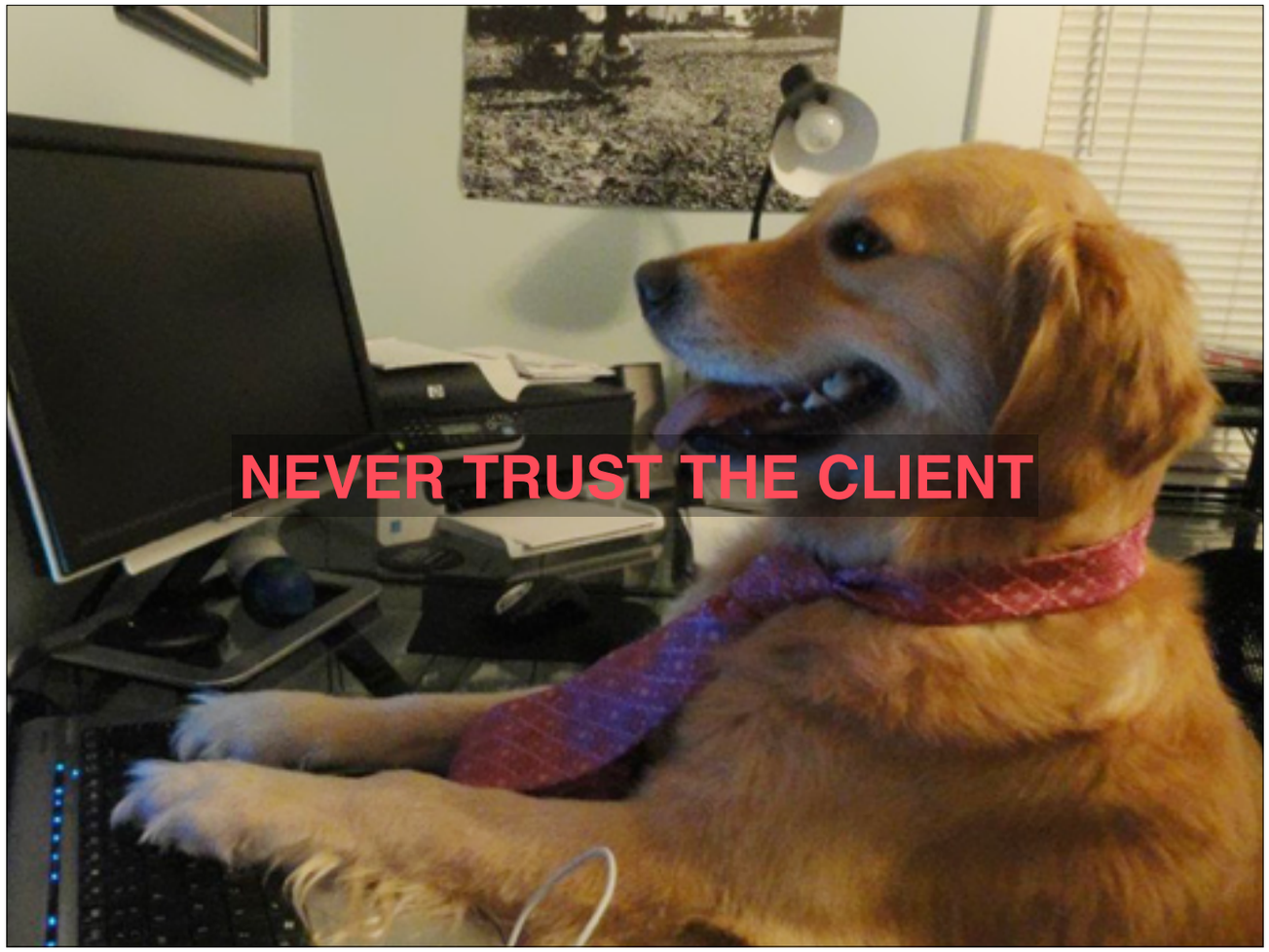
Indicates whether a page should be allows to be framed and a set of rules for that DENY, SAMEORIGIN, ALLOW-FROM




Reflected from form submission
Stored and persistent

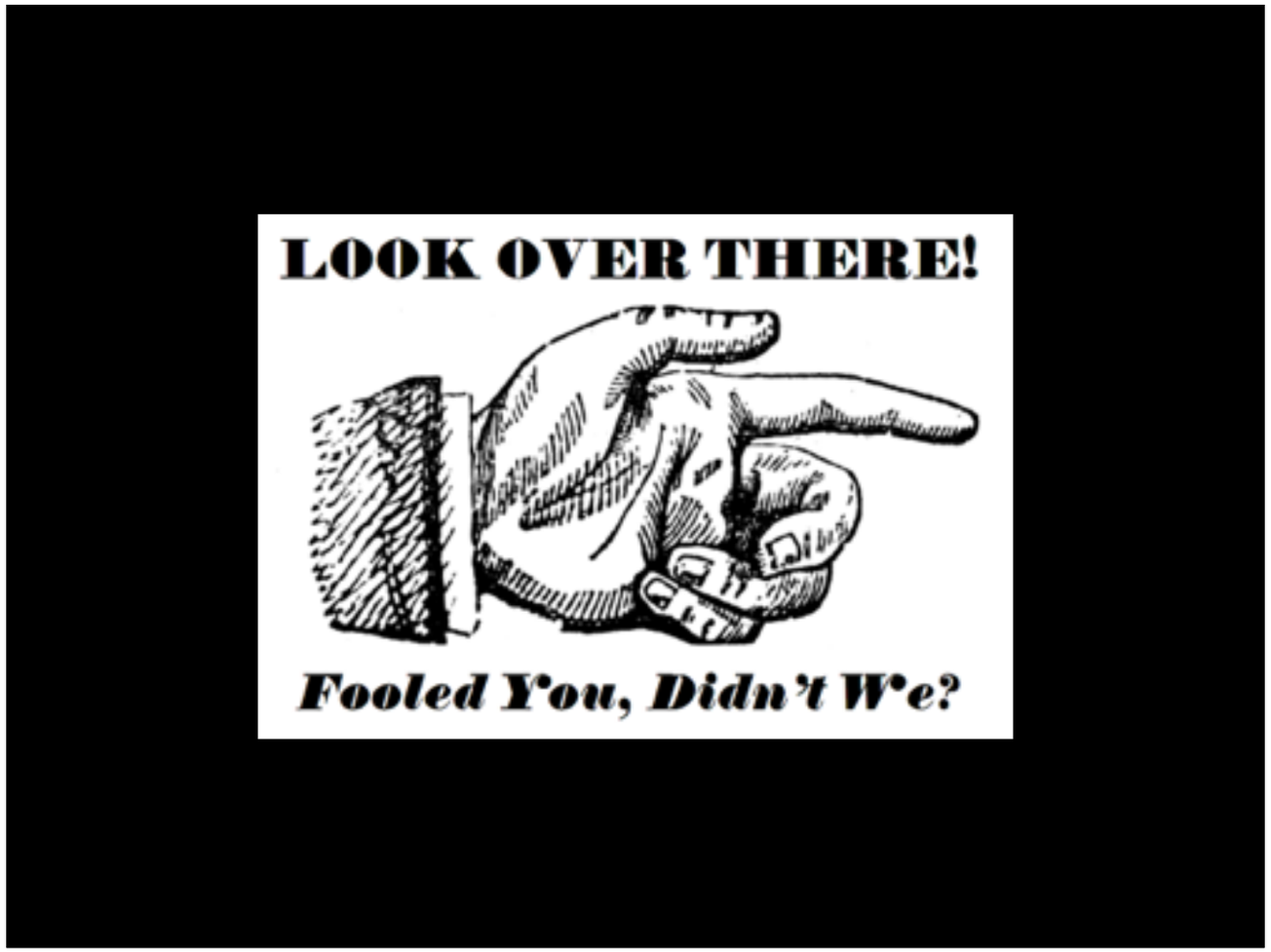
Escape output - very difficult
Tie session cookies to the IP address
httponly cookies to stop the being stolen
Newest mitigation - Content Security Policy (CSP) - complicated


Pretty complex.
Very easy to break your site.
Do you know how your site is set out?
Third-party? Google analytics, advertising, tracking, and so on.



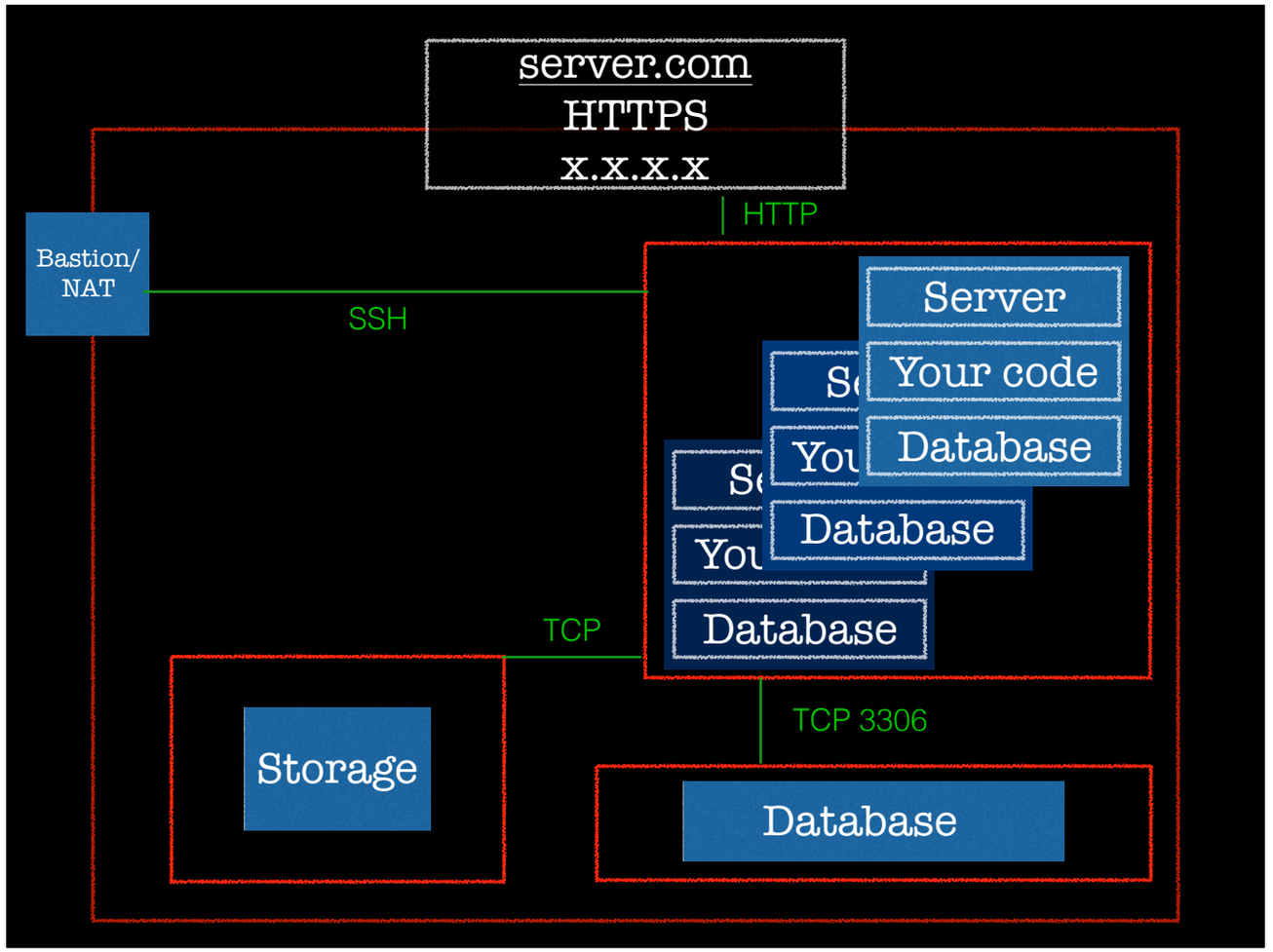

Referers

Trusting headers for constructing things




Open redirects

Do not use user input for constructing forwards and redirects
Allows phishing scams by looking like the link is legitimate




Where to put headers and the advantages/disadvantages
HSTS would make sense at load balancer - but no configuration access
Web frameworks add the headers there.  No good for static content.
Add to the server nginx/Apache

List of things:

- Secure cookie + httponly
- XSS filter
- HSTS
- SSL check
- Same origin policy
- Content Security Policy
- Sessions


https://observatory.mozilla.org

# What have I missed?

- Multiple headers
- DoS - head of line blocking
- Brute force
- Hashing is not encryption!
- SSL ciphers
- Certificate pinning
- PCI compliance
- PII
- CVEs and vulnerability disclosures
- File format parsing
- Pen tests
- Physical pen tests
- Reproducible builds
- Guessable URLs
- Examples of exploits!

- SChannel
- Enterprise networks
- CORS
- MitM
- RSA4
- goto fail;
- Malicious actors
- Physical intrusion
- More passwords stolen
- Incident response
- Physical security solutions
- Fuzz testing
- Session hijacking
- Misconfiguration
- And sooooooooooo much more

So much, so much, my head, this could take HOURS!

**And finally….. get a password manager!**

- **OWASP** https://www.owasp.org/ (see Reference section on left)
- **Mozilla Security Blog** https://blog.mozilla.org/security/
- **Content Security Policy** https://scotthelme.co.uk/content-security-policy-an-introduction/
- **CVEs** https://cve.mitre.org
- **SPA Security** http://www.slideshare.net/carlo.bonamico/angularjs-security-defend-your-single-page-application
- **HTML5 Security Cheatsheet** https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet