Securing a Django (Wait, any!) Website

Nick Thompson

August 2, 2015

Goals:

- The information provided here should allow you to pass a rigorous security audit for cloud services, assuming your server has not been compromised.

Goals:

- The information provided here should allow you to pass a rigorous security audit for cloud services, assuming your server has not been compromised.
- This will not provide information about securing your server, only information about keeping data secure when it is on the wire.

Goals:

- The information provided here should allow you to pass a rigorous security audit for cloud services, assuming your server has not been compromised.
- This will not provide information about securing your server, only information about keeping data secure when it is on the wire.
- More information about securing your server can be found elsewhere.

Goals:

- The information provided here should allow you to pass a rigorous security audit for cloud services, assuming your server has not been compromised.
- This will not provide information about securing your server, only information about keeping data secure when it is on the wire.
- More information about securing your server can be found elsewhere.
- Securing the wire is the easy part; harder is keeping authenticated users from grabbing other users data *because that's code you write yourself*.

Getting started:

```
$ git clone https://github.com/NAThompson/django_https.git
$ pyvenv django_https
$ cd django_https
$ pip3 install -r requirements.txt
$ sudo /bin/bash
$$ . bin/activate
```

Getting started:

```
$ git clone https://github.com/NAThompson/django_https.git
$ pyvenv django_https
$ cd django_https
$ pip3 install -r requirements.txt
$ sudo /bin/bash
$$ . bin/activate
```

We'll need root to open up priviledged ports. Sourcing after acquisition of root shell is necessary.

Server Stack

- We're going to use Django+gunicorn+nginx to serve this website

Server Stack

- We're going to use Django+gunicorn+nginx to serve this website
- gunicorn is a web server gateway interface

Server Stack

- We're going to use Django+gunicorn+nginx to serve this website
- gunicorn is a web server gateway interface
- We need nginx to reverse proxy because gunicorn is trivially vunerable to DOS attacks

Server Stack

- We're going to use Django+gunicorn+nginx to serve this website
- gunicorn is a web server gateway interface
- We need nginx to reverse proxy because gunicorn is trivially vunerable to DOS attacks

Configuring Secure Settings

- Many security settings can be configured either through nginx or through django-secure.

Configuring Secure Settings

- Many security settings can be configured either through nginx or through django-secure.
- In my experience, nginx seems to be faster, so we'll focus on configuring nginx.

Configuring Secure Settings

- Many security settings can be configured either through nginx or through django-secure.
- In my experience, nginx seems to be faster, so we'll focus on configuring nginx.
- django-secure settings will tend to override nginx settings, or they will be set twice in the http headers. So use nginx, or use django-secure, not both.

Configuring django-secure

- Add the following to your INSTALLED_APPS in settings.py:

```
INSTALLED_APPS = (
    'sslserver',
    ...,
    'djangosecure',
)
```

Configuring django-secure

- Add the following to your INSTALLED_APPS in settings.py:

```
INSTALLED_APPS = (
    'sslserver',
    ...,
    'djangosecure',
)
```

- Then run the test server via:

```
$ ./manage.py checksecure
$ ./manage.py runsslserver --addrport 127.0.0.1:443
```

This is nice for development because it uses self-signed certificates with relative paths.

Install nginx

```
$ sudo apt-get install nginx # Ubuntu
$ sudo brew install nginx # Mac
```

Turn SSL on and proxy-pass to gunicorn

```
server {
  listen          443;
  ssl             on;
  server_name     example.com;
  ssl_certificate  /somedir/bundle.crt;
  ssl_certificate_key /somedir/mykey.key;

  location / {
     proxy_pass https://127.0.0.1:8000;
     proxy_set_header Host $host;
     proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  }
}
```

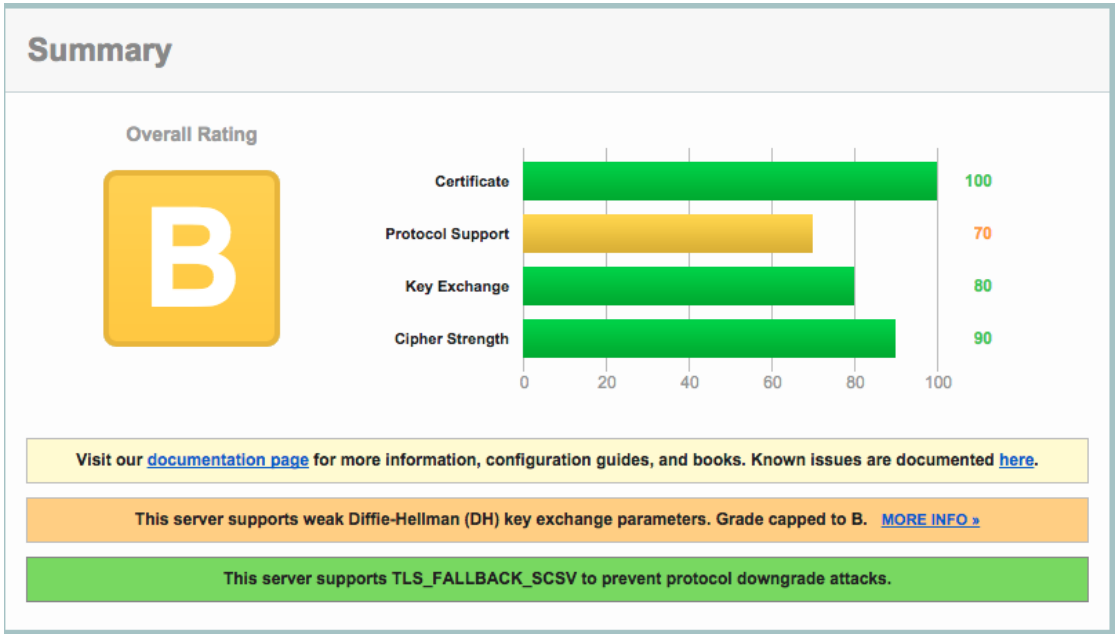
How to serve the website

```
$$ cd django_https/src
$$ gunicorn -c gunicorn_config.py https.wsgi &
$$ nginx -c ~/django_https/nginx.conf
```

(Again, there are some hard-coded paths in this . . .)

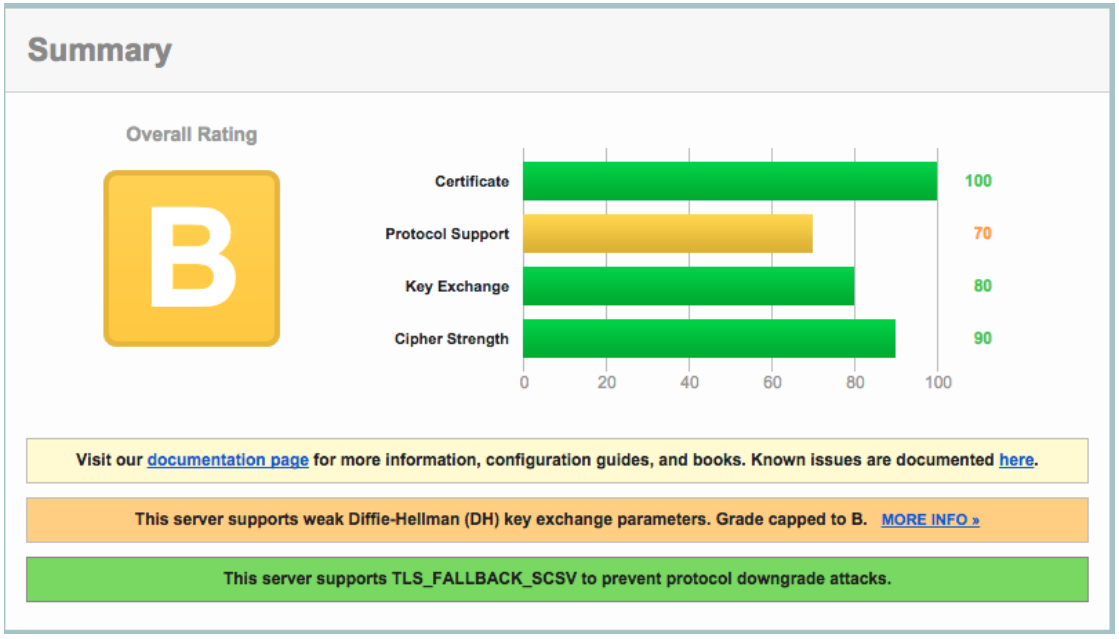
How secure is the default nginx configuration?

- SSL Labs doesn't think it's all that great:



How secure is the default nginx configuration?

- SSL Labs doesn't think it's all that great:



- If your clients have an IT policy, they *will ask about this*.

Improving SSLLabs grade

- In the nginx.conf, change

```
ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
```

  to

```
ssl_protocols TLSv1.2;
```

Improving SSLLabs grade

- In the nginx.conf, change

  ```
  ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
  ```

  to

  ```
  ssl_protocols TLSv1.2;
  ```

- Note: This will lose you some old IE browsers. SSLLabs will tell you which ones in the handshake simulation section of their report.

Some justification for only supporting TLSv1.2

- The Payment Card Industry (PCI) Security Standards Council says you must remove support for TLSv1.0 to be PCI compliant. “SSL and early TLS are not considered strong cryptography and cannot be used as a security control after 30th June, 2016.“

Some justification for only supporting TLSv1.2

- The Payment Card Industry (PCI) Security Standards Council says you must remove support for TLSv1.0 to be PCI compliant. "SSL and early TLS are not considered strong cryptography and cannot be used as a security control after 30th June, 2016."
- OWASP (Open Web Application Security Project) claims "TLS 1.0 is still widely used as 'best' protocol by a lot of browsers, that are not patched to the very latest version. . . TLSv1.0 should only be used only after risk analysis and acceptance."

Some justification for only supporting TLSv1.2

- The Payment Card Industry (PCI) Security Standards Council says you must remove support for TLSv1.0 to be PCI compliant. "SSL and early TLS are not considered strong cryptography and cannot be used as a security control after 30th June, 2016."
- OWASP (Open Web Application Security Project) claims "TLS 1.0 is still widely used as 'best' protocol by a lot of browsers, that are not patched to the very latest version. . . TLSv1.0 should only be used only after risk analysis and acceptance."
- Almost no browsers support TLSv1.1 and *not* TLSv1.2. So make your life easier and just use 1.2.

Improving SSLLabs grade

- SSLLabs thinks that 256 bits symmetric protocols are better than 128 bit protocols, although not everyone agrees.

Improving SSLLabs grade

- SSLLabs thinks that 256 bits symmetric protocols are better than 128 bit protocols, although not everyone agrees.
- But you can still improve your grade by restricting the supported ciphersuite by adding this to the http section of nginx.conf:

```
ssl_ciphers ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA3
-AES256-SHA384:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:ECDHE
-RSA-AES256-GCM-SHA384:DHE-RSA-AES256-GCM-SHA:DHE-RSA-AES256-SHA256
aNULL:!eNULL:!LOW:!3DES:!MD5:!EXP:!PSK:!SRP:!DSS;
```

Improving SSLLabs grade

- SSLLabs thinks that 256 bits symmetric protocols are better than 128 bit protocols, although not everyone agrees.
- But you can still improve your grade by restricting the supported ciphersuite by adding this to the http section of nginx.conf:

```
ssl_ciphers ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA3
-AES256-SHA384:ECDHE-ECDSA-AES256-SHA384:ECDHE-RSA-AES256-SHA:ECDHE
-RSA-AES256-GCM-SHA384:DHE-RSA-AES256-GCM-SHA:DHE-RSA-AES256-SHA256
aNULL:!eNULL:!LOW:!3DES:!MD5:!EXP:!PSK:!SRP:!DSS;
```

- This is a mess, what does it mean?

This configures *ciphersuite negotiation*

- The browser tells the server what ciphersuites it supports

This configures *ciphersuite negotiation*

- The browser tells the server what ciphersuites it supports
- The server selects one that is in the ssl_ciphers list,

This configures *ciphersuite negotiation*

- The browser tells the server what ciphersuites it supports
- The server selects one that is in the ssl_ciphers list,
- The server tells the browser what ciphers they are using, or rejects the connection if they can't agree on a cipher suite.

What is a ciphersuite?

What is a ciphersuite?

- A key exchange algorithm (generally an asymmetric cipher, e.g. RSA, Diffie-Hellman)

What is a ciphersuite?

- A key exchange algorithm (generally an asymmetric cipher, e.g. RSA, Diffie-Hellman)
- An bulk encryption algorithm (generally a symmetric cipher, e.g. AES)

What is a ciphersuite?

- A key exchange algorithm (generally an asymmetric cipher, e.g. RSA, Diffie-Hellman)
- An bulk encryption algorithm (generally a symmetric cipher, e.g. AES)
- A message authentication code algorithm (hash function, SHA256)

What is a ciphersuite?

- A key exchange algorithm (generally an asymmetric cipher, e.g. RSA, Diffie-Hellman)
- An bulk encryption algorithm (generally a symmetric cipher, e.g. AES)
- A message authentication code algorithm (hash function, SHA256)
- An authentication protocol (e.g., RSA)

Understanding available ciphersuites

To see what ciphers your nginx supports, run

```
$ openssl ciphers | tr ':' '\n'
```

(nginx links against openssl's libraries)

Understanding available ciphersuites

- To understand a given cipher, we use

```
$ openssl ciphers -v ECDHE-RSA-AES256-GCM-SHA384
ECDHE-RSA-AES256-GCM-SHA384
TLSv1.2 Kx=ECDH     Au=RSA  Enc=AESGCM(256) Mac=AEAD
```

Understanding available ciphersuites

- To understand a given cipher, we use

```
$ openssl ciphers -v ECDHE-RSA-AES256-GCM-SHA384
ECDHE-RSA-AES256-GCM-SHA384
TLSv1.2 Kx=ECDH     Au=RSA  Enc=AESGCM(256) Mac=AEAD
```

- Kx = Key exchange; ECDH=elliptic curve Diffie-Hellman

Understanding available ciphersuites

- To understand a given cipher, we use

```
$ openssl ciphers -v ECDHE-RSA-AES256-GCM-SHA384
ECDHE-RSA-AES256-GCM-SHA384
TLSv1.2 Kx=ECDH     Au=RSA  Enc=AESGCM(256) Mac=AEAD
```

- Kx = Key exchange; ECDH=elliptic curve Diffie-Hellman
- Au=Authentication; uses RSA

Understanding available ciphersuites

- To understand a given cipher, we use

```
$ openssl ciphers -v ECDHE-RSA-AES256-GCM-SHA384
ECDHE-RSA-AES256-GCM-SHA384
TLSv1.2 Kx=ECDH     Au=RSA  Enc=AESGCM(256) Mac=AEAD
```

- Kx = Key exchange; ECDH=elliptic curve Diffie-Hellman
- Au=Authentication; uses RSA
- Enc= Encryption; AESGCM(256) is 256 bit Advanced Encryption Standard in Galois Counter mode for encryption (please don't ask about GCM!)

Understanding available ciphersuites

- To understand a given cipher, we use

```
$ openssl ciphers -v ECDHE-RSA-AES256-GCM-SHA384
ECDHE-RSA-AES256-GCM-SHA384
TLSv1.2 Kx=ECDH     Au=RSA  Enc=AESGCM(256) Mac=AEAD
```

- Kx = Key exchange; ECDH=elliptic curve Diffie-Hellman
- Au=Authentication; uses RSA
- Enc= Encryption; AESGCM(256) is 256 bit Advanced Encryption Standard in Galois Counter mode for encryption (please don't ask about GCM!)
- Mac= Message Authentication; AEAD ="authentication encryption with associated data" for message authentication (inherited from the "Galois counter mode" of AES)

Interpreting ssl_ciphers

Suppose this was our configuration:

```
ssl_ciphers ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384;
```

- The order matters: Adding “ssl_prefer_server_ciphers on;“ to the nginx.conf tells nginx to ignore the preferences of the browser.

Interpreting ssl_ciphers

Suppose this was our configuration:

```
ssl_ciphers ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384;
```

- The order matters: Adding "ssl_prefer_server_ciphers on;" to the nginx.conf tells nginx to ignore the preferences of the browser.
- So first we would use ECDHE-RSA-AES256-GCM-SHA384, but if the browser doesn't support it, then we use ECDHE-ECDSA-AES256-GCM-SHA384.

Interpreting ssl_ciphers

Suppose this was our configuration:

```
ssl_ciphers ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES256-GCM-SHA384;
```

- The order matters: Adding "ssl_prefer_server_ciphers on;" to the nginx.conf tells nginx to ignore the preferences of the browser.
- So first we would use ECDHE-RSA-AES256-GCM-SHA384, but if the browser doesn't support it, then we use ECDHE-ECDSA-AES256-GCM-SHA384.
- Otherwise, we don't make the connection to the browser.

What ciphersuites does my browser support?

There are some websites who will tell you:

Cipher Suites Supported by Your Browser (ordered by preference):

Spec	Cipher Suite Name	Key Size	Description
(cc,14)	ECDHE-ECDSA-CHACHA20-POLY1305-SHA256	128 Bit	Key exchange: ECDH, encryption: ChaCha20 Poly1305, MAC: SHA256.
(cc,13)	ECDHE-RSA-CHACHA20-POLY1305-SHA256	128 Bit	Key exchange: ECDH, encryption: ChaCha20 Poly1305, MAC: SHA256.
(cc,15)	DHE-RSA-CHACHA20-POLY1305-SHA256	128 Bit	Key exchange: DH, encryption: ChaCha20 Poly1305, MAC: SHA256.
(c0,2b)	ECDHE-ECDSA-AES128-GCM-SHA256	128 Bit	Key exchange: ECDH, encryption: AES, MAC: SHA256.
(c0,2f)	ECDHE-RSA-AES128-GCM-SHA256	128 Bit	Key exchange: ECDH, encryption: AES, MAC: SHA256.
(00,9e)	DHE-RSA-AES128-GCM-SHA256	128 Bit	Key exchange: DH, encryption: AES, MAC: SHA256.
(c0,0a)	ECDHE-ECDSA-AES256-SHA	256 Bit	Key exchange: ECDH, encryption: AES, MAC: SHA1.
(c0,14)	ECDHE-RSA-AES256-SHA	256 Bit	Key exchange: ECDH, encryption: AES, MAC: SHA1.
(00,39)	DHE-RSA-AES256-SHA	256 Bit	Key exchange: DH, encryption: AES, MAC: SHA1.
(c0,09)	ECDHE-ECDSA-AES128-SHA	128 Bit	Key exchange: ECDH, encryption: AES, MAC: SHA1.
(c0,13)	ECDHE-RSA-AES128-SHA	128 Bit	Key exchange: ECDH, encryption: AES, MAC: SHA1.
(00,33)	DHE-RSA-AES128-SHA	128 Bit	Key exchange: DH, encryption: AES, MAC: SHA1.
(00,9c)	RSA-AES128-GCM-SHA256	128 Bit	Key exchange: RSA, encryption: AES, MAC: SHA256.
(00,35)	RSA-AES256-SHA	256 Bit	Key exchange: RSA, encryption: AES, MAC: SHA1.
(00,2f)	RSA-AES128-SHA	128 Bit	Key exchange: RSA, encryption: AES, MAC: SHA1.
(00,0a)	RSA-3DES-EDE-SHA	168 Bit	Key exchange: RSA, encryption: 3DES, MAC: SHA1.
(00,ff)	EMPTY-RENEGOTIATION-INFO-SCSV	0 Bit	Used for secure renegotation.

Some advice:

- Prefer Galois counter modes (AES-GCM) as they consume less resources

Some advice:

- Prefer Galois counter modes (AES-GCM) as they consume less resources
- Prefer SHA256 over SHA1 as SHA1 will be deprecated.

Some advice:

- Prefer Galois counter modes (AES-GCM) as they consume less resources
- Prefer SHA256 over SHA1 as SHA1 will be deprecated.
- Make sure to sign your certificates with SHA256 over SHA1, or else it will not be trusted by Google Chrome.

Some advice:

- Prefer Galois counter modes (AES-GCM) as they consume less resources
- Prefer SHA256 over SHA1 as SHA1 will be deprecated.
- Make sure to sign your certificates with SHA256 over SHA1, or else it will not be trusted by Google Chrome.
- Prefer elliptic curves, unless you think the NSA has a quantum computer.

Defense Against the Logjam attack

- The logjam attack was discovered relatively recently, and is an attack against Diffie-Hellman key exchange.

Defense Against the Logjam attack

- The logjam attack was discovered relatively recently, and is an attack against Diffie-Hellman key exchange.
- It's thought that it was only used by state-level adversaries, but it seems that this will change soon.

Diffie-Hellman Key Exchange Review

- Server picks a prime p and $g < p$ along with random $a < p$. p and g are public, a is secret.

Diffie-Hellman Key Exchange Review

- Server picks a prime p and $g < p$ along with random $a < p$. p and g are public, a is secret.
- Over the network is transferred $A := g^a \mod p$, g, p, cryptographically signed.

Diffie-Hellman Key Exchange Review

- Server picks a prime p and $g < p$ along with random $a < p$. p and g are public, a is secret.
- Over the network is transferred $A := g^a \mod p$, g, p, cryptographically signed.
- Client chooses $b < p$ and sends the server $B = g^b \mod p$.

Diffie-Hellman Key Exchange Review

- Server picks a prime p and $g < p$ along with random $a < p$. p and g are public, a is secret.
- Over the network is transferred $A := g^a \mod p$, g, p, cryptographically signed.
- Client chooses $b < p$ and sends the server $B = g^b \mod p$.
- The shared secret is $s := g^{ab} \mod p$.

Diffie-Hellman Key Exchange Review

- Server picks a prime p and $g < p$ along with random $a < p$. p and g are public, a is secret.
- Over the network is transferred $A := g^a \mod p$, g, p, cryptographically signed.
- Client chooses $b < p$ and sends the server $B = g^b \mod p$.
- The shared secret is $s := g^{ab} \mod p$.

This is a secure protocol as solving $A = g^a \mod p$ for a (called "the discrete logarithm") is hard.

Diversion: Enabling Forward Secrecy

". . . which means a compromise of the server's long term signing key does not compromise the confidentiality of past session"

Diversion: Forward secrecy in Diffie-Hellman

- Question: What does the server do with a (the random secret) after the SSL session ends?

Diversion: Forward secrecy in Diffie-Hellman

- Question: What does the server do with a (the random secret) after the SSL session ends?
- Can the NSA subpoena your Diffie-Helman session keys?

What if it takes 2 minutes to generate a prime p for Diffie-Hellman key exchange?

- Then everyone uses the same damn prime.

What if it takes 2 minutes to generate a prime p for Diffie-Hellman key exchange?

- Then everyone uses the same damn prime.
- "The situation for export Diffie-Hellman is particularly awful, with only two (!) primes used across up 92% of enabled Apache/mod_ssl sites."

What if we have to support 20 year old browsers?

- Then an attacker can do a downgrade attack so that the prime p is only 512 bits.

What if we have to support 20 year old browsers?

- Then an attacker can do a downgrade attack so that the prime p is only 512 bits.
- This allows a MITM to intercept your server message, and send it saying "we only support 512 bit DH"; the browser supports it, so it agrees to use of export-grade crypto.

What if the prime p is only 512 bits? That's still a lot

- Solving a 512 bit discrete logarithm is hard; it takes an academic group about a week.

What if the prime p is only 512 bits? That's still a lot

- Solving a 512 bit discrete logarithm is hard; it takes an academic group about a week.
- But for fixed p, using a *very sophisticated* lookup table, computing the *next* discrete log only takes 90 seconds.

What if the prime p is only 512 bits? That's still a lot

- Solving a 512 bit discrete logarithm is hard; it takes an academic group about a week.
- But for fixed p, using a *very sophisticated* lookup table, computing the *next* discrete log only takes 90 seconds.

The solution is . . .

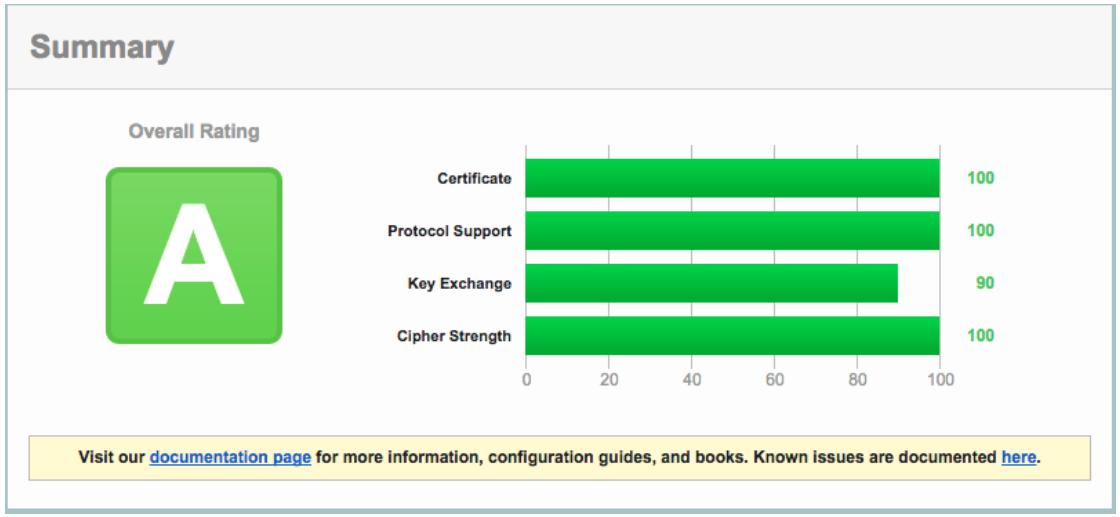
Generating a large, unique Diffie-Hellman Prime

```
$ openssl dhparam -out dhparam.pem 4096
```

and add the following line to the nginx.conf:

```
ssl_dhparam /path_to_pem/dhparam.pem
```

SSLLabs is now Happy



You can see the SSLLabs rating guide here.

What is HTTP Strict Transport Security? (HSTS)

- A protocol by which servers force all traffic to come over https.

What is HTTP Strict Transport Security? (HSTS)

- A protocol by which servers force all traffic to come over https.
- Server informs browser to recognize MITM attack by requests for http traffic

What is HTTP Strict Transport Security? (HSTS)

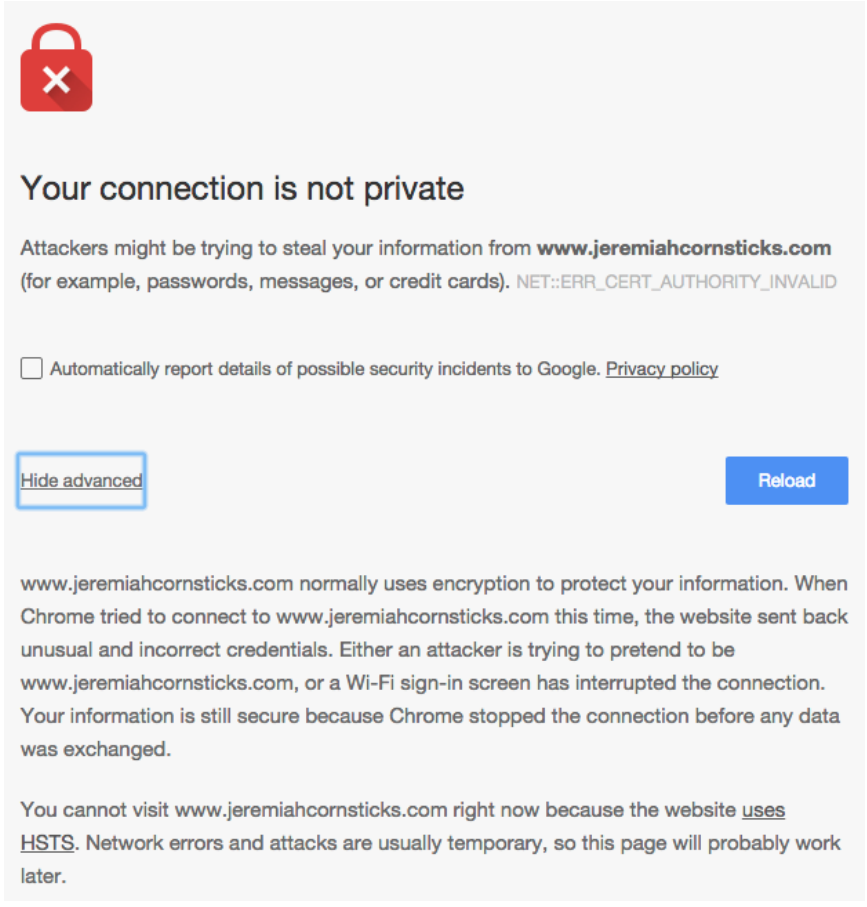
- A protocol by which servers force all traffic to come over https.
- Server informs browser to recognize MITM attack by requests for http traffic
- Stops downgrade attacks and cookie hijacking.

What is HTTP Strict Transport Security? (HSTS)

- A protocol by which servers force all traffic to come over https.
- Server informs browser to recognize MITM attack by requests for http traffic
- Stops downgrade attacks and cookie hijacking.
- Stops SSL stripping!

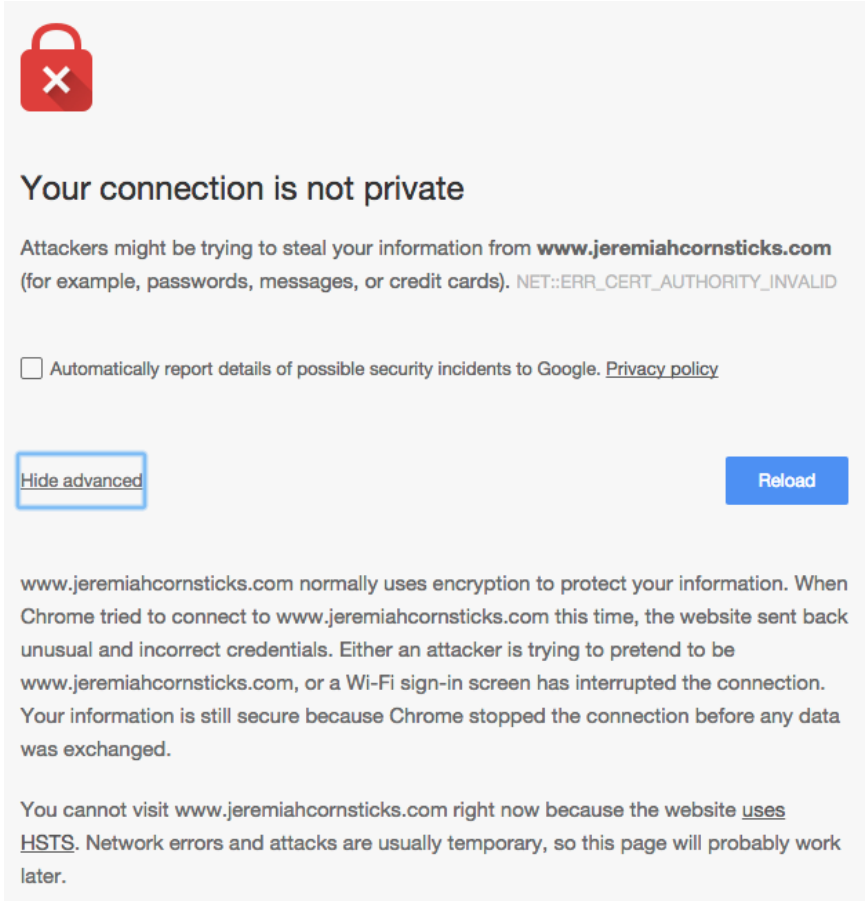
HSTS

A user who has visited your site previously *cannot* proceed past a bad certificate:



HSTS

A user who has visited your site previously *cannot* proceed past a bad certificate:



(Unless they clear their browser cache . . . then the user can ignore the warning and proceed.)

HSTS Redirects http to https

But only after a user visits the first time . . .

HSTS Redirects http to https

But only after a user visits the first time . . .

- Workaround for first-time users:

HSTS Redirects http to https

But only after a user visits the first time . . .

- Workaround for first-time users:
- Register your site in the preload list!

HSTS Redirects http to https

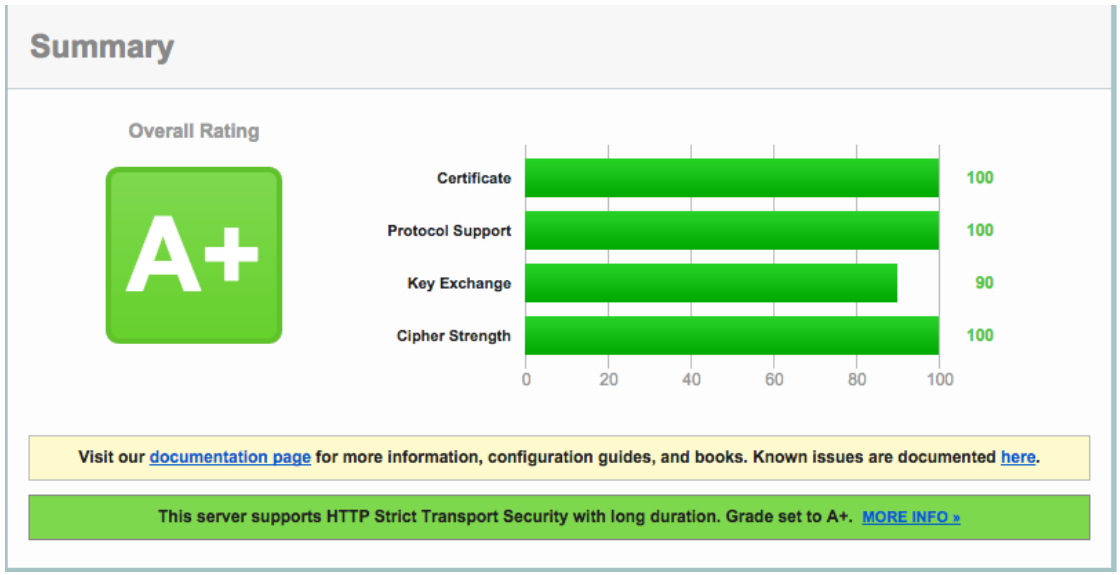
But only after a user visits the first time . . .

- Workaround for first-time users:
- Register your site in the preload list!
- "This form is used to submit domains for inclusion in Chrome's HTTP Strict Transport Security (HSTS) preload list. This is a list of sites that are hardcoded into Chrome as being HTTPS only."

Configuring HSTS

- Add this to the server section in “default”:

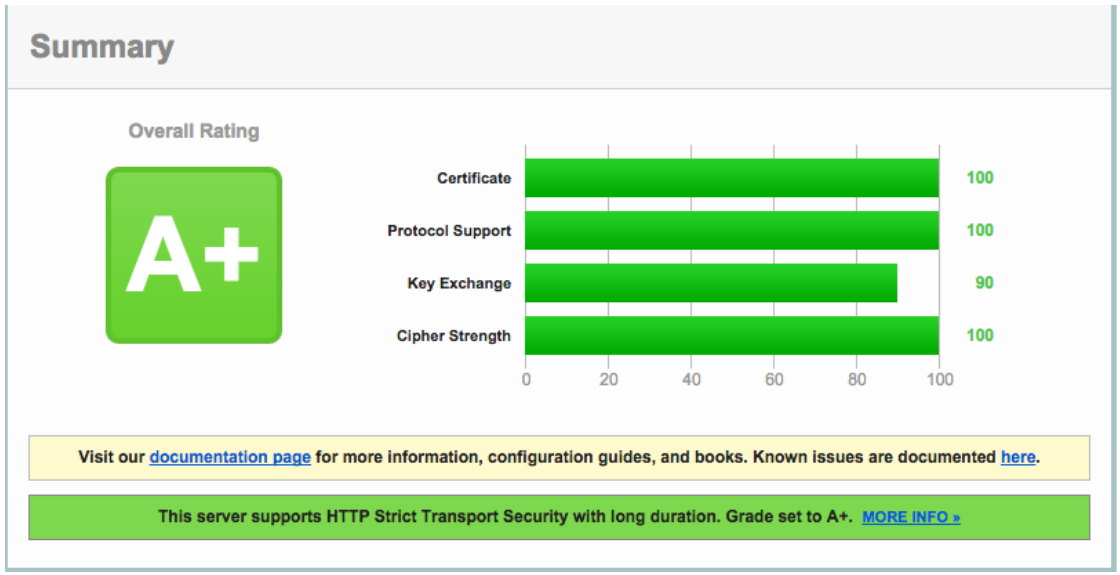
```
add_header Strict-Transport-Security \
 "max-age=63072000; includeSubdomains";
```



Configuring HSTS

- Add this to the server section in "default":

```
add_header Strict-Transport-Security \
 "max-age=63072000; includeSubdomains";
```



- The server now says: "For the next 63072000 seconds, this server and its subdomains will only be using https. Any http traffic is a MITM attack."

HSTS Configured in django-secure

- Add the following to your settings.py:

```
SECURE_HSTS_SECONDS = 63072000
SECURE_HSTS_SUBDOMAINS = True
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
```

HSTS Configured in django-secure

- Add the following to your settings.py:

```
SECURE_HSTS_SECONDS = 63072000
SECURE_HSTS_SUBDOMAINS = True
SECURE_HSTS_INCLUDE_SUBDOMAINS = True
```

- Not that this will override the nginx settings, if you set them both.

Debugging Aid:

- The server's use of HSTS is communicated via http headers, and nginx doesn't do a strong validation of the nginx.conf. In addition, nginx version changes can silently break your nginx.conf.

Debugging Aid:

- The server's use of HSTS is communicated via http headers, and nginx doesn't do a strong validation of the nginx.conf. In addition, nginx version changes can silently break your nginx.conf.
- To validate that you've actually set a http header:

Debugging Aid:

- The server's use of HSTS is communicated via http headers, and nginx doesn't do a strong validation of the nginx.conf. In addition, nginx version changes can silently break your nginx.conf.
- To validate that you've actually set a http header:
- Use "Live HTTP Headers" Chrome browser extension (I like it!)

Debugging Aid:

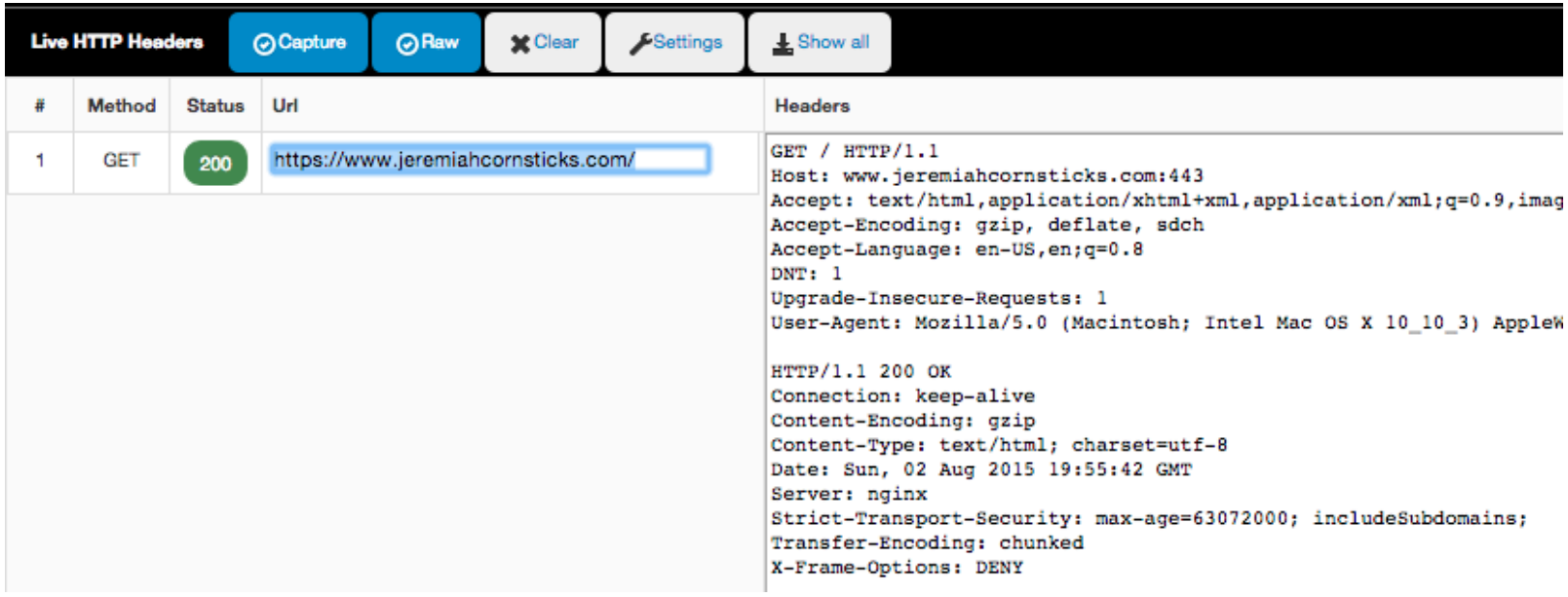
- The server's use of HSTS is communicated via http headers, and nginx doesn't do a strong validation of the nginx.conf. In addition, nginx version changes can silently break your nginx.conf.
- To validate that you've actually set a http header:
- Use "Live HTTP Headers" Chrome browser extension (I like it!)
- Use alt-command-I on Chrome (more painful)

Debugging Aid:

- The server's use of HSTS is communicated via http headers, and nginx doesn't do a strong validation of the nginx.conf. In addition, nginx version changes can silently break your nginx.conf.
- To validate that you've actually set a http header:
- Use "Live HTTP Headers" Chrome browser extension (I like it!)
- Use alt-command-I on Chrome (more painful)
- Use Wireshark

Debugging http headers:

Example using Live HTTP headers Chrome extension:



Strong key-exchange

To get a strong key-exchange, we need at least 4096 bit keys:

Strong key-exchange

To get a strong key-exchange, we need at least 4096 bit keys:

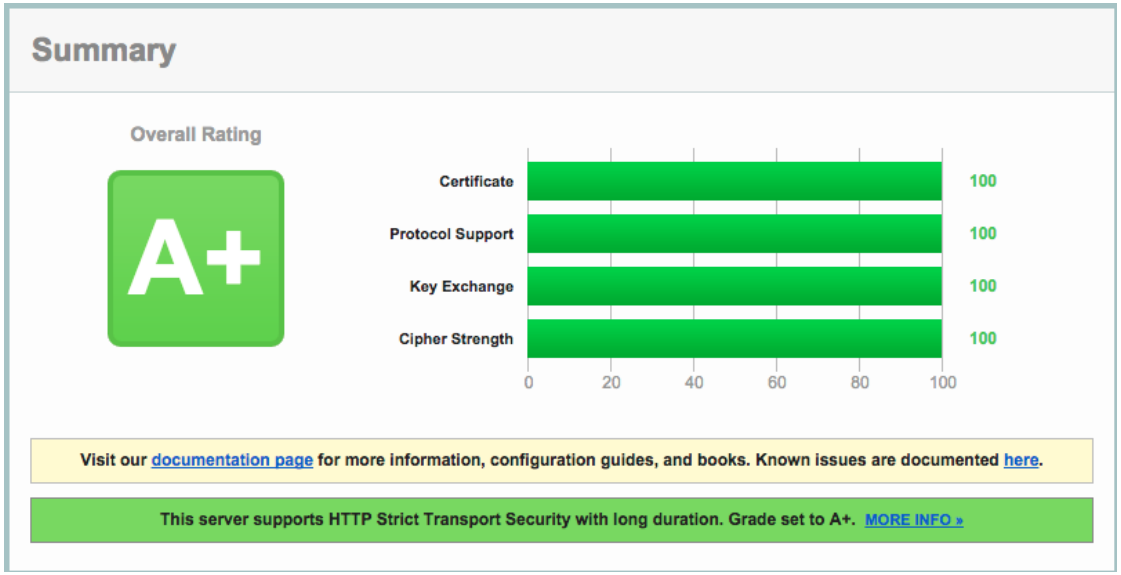
```
$ openssl genrsa -out foo.key 4096; chmod 400 foo.key;
$ openssl req -new -sha256 -key foo.key -out foo.csr
```

Strong key-exchange

To get a strong key-exchange, we need at least 4096 bit keys:

```
$ openssl genrsa -out foo.key 4096; chmod 400 foo.key;
$ openssl req -new -sha256 -key foo.key -out foo.csr
```

Use this certificate signing request to get certs from your provider, and you're done!



OCSP Stapling

- OCSP := Online Certificate Status Protocol

OCSP Stapling

- OCSP := Online Certificate Status Protocol
- Used to determine if a certificate has been revoked

OCSP Stapling

- OCSP := Online Certificate Status Protocol
- Used to determine if a certificate has been revoked
- Browsers must query certificate authority (CA), revealing websites they visit

OCSP Stapling

- OCSP := Online Certificate Status Protocol
- Used to determine if a certificate has been revoked
- Browsers must query certificate authority (CA), revealing websites they visit
- OCSP Stapling: Server caches CA's OCSP digitally signed response, increasing privacy as well as speed

Enabling OCSP Stapling in nginx

- Bundle intermediate and root certifications:

  `cat  foo1.crt foo2.crt > ocsp.crt;`

  Since I bought $5 certificates for this talk, I did:

```
cat COMODORSADomainValidationSecureServerCA.crt \
 COMODORSAAddTrustCA.crt > ocsp.crt
```

Enabling OCSP Stapling in nginx

- Bundle intermediate and root certifications:

  ```
  cat  foo1.crt foo2.crt > ocsp.crt;
  ```

  Since I bought $5 certificates for this talk, I did:

  ```
  cat COMODORSADomainValidationSecureServerCA.crt \
   COMODORSAAddTrustCA.crt > ocsp.crt
  ```

- Then add the following lines to the nginx.conf:

  ```
  resolver 8.8.8.8;
  ssl_stapling on;
  ssl_stapling_verify on;
  ssl_trusted_certificate /pathtocerts/ocsp.crt;
  ```

Diversion: Certificate Authorities

- Free, trusted certificates coming next month from Let's Encrypt

Clickjacking

- Someone renders your page in theirs (example: ebay.com/buynicecar)

Clickjacking

- Someone renders your page in theirs (example: ebay.com/buynicecar)
- Then they make your page invisible, but put "Win free iPad!", and a place to click over the "Buy it now" link of the ebay listing

Clickjacking

- Someone renders your page in theirs (example: ebay.com/buynicecar)
- Then they make your page invisible, but put "Win free iPad!", and a place to click over the "Buy it now" link of the ebay listing
- If you are logged in on ebay, you are the proud owner of a new car!

Clickjacking

"One of the most notorious examples of Clickjacking was an attack against the Adobe Flash plugin settings page. By loading this page into an invisible iframe, an attacker could trick a user into altering the security settings of Flash, giving permission for any Flash animation to utilize the computer's microphone and camera."

Defense against clickjacking

As a web user: You're screwed.

Defense against clickjacking

- As a developer: Add the following to your nginx.conf:
  `add_header X-Frame-Options DENY;`

Defense against clickjacking

- As a developer: Add the following to your nginx.conf:

  ```
  add_header X-Frame-Options DENY;
  ```

- Or to your settings.py:

  ```
  SECURE_FRAME_DENY = True
  ```

Defense against clickjacking

- As a developer: Add the following to your nginx.conf:

  ```
  add_header X-Frame-Options DENY;
  ```

- Or to your settings.py:

  ```
  SECURE_FRAME_DENY = True
  ```

- Note that this is a non-standard extension to html. There is a standardized way (see content security policies), but it's not yet supported by all modern browsers

Defense against clickjacking

- This is such a huge problem that the default django configuration actually sets the http response header "X-Frame-Options: SAMEORIGIN":

```
MIDDLEWARE_CLASSES = (
...
'django.middleware.clickjacking.XFrameOptionsMiddleware',
...
)
```

Defense against clickjacking

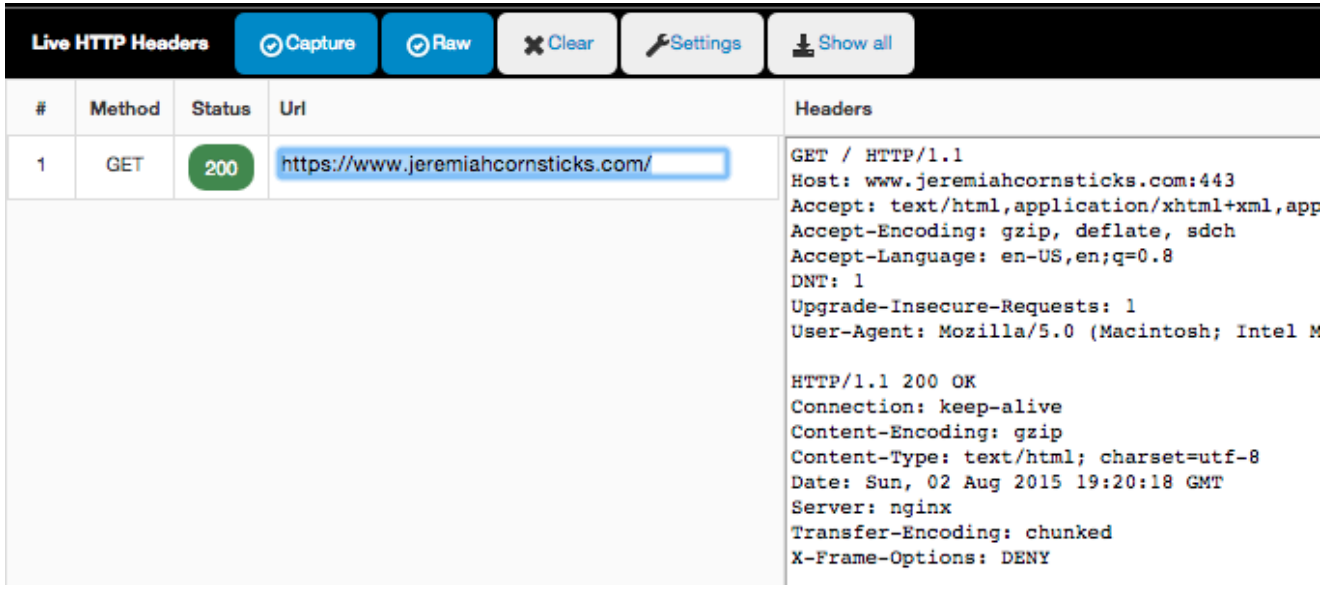
- This is such a huge problem that the default django configuration actually sets the http response header "X-Frame-Options: SAMEORIGIN":

```
MIDDLEWARE_CLASSES = (
...
'django.middleware.clickjacking.XFrameOptionsMiddleware',
...
)
```

- SAMEORIGIN means you can embed your own webpages in html frames, but disallows its embedding in other people's websites.

Verify the http header

Again, verify that the `X-Frame-Options: DENY` header has been sent:



Content Sniffing

- Some html content is not given the proper tags for its interpretation

Content Sniffing

- Some html content is not given the proper tags for its interpretation
- So Microsoft decided to give IE the capacity to guess the interpretation of a byte-stream, called content-sniffing

Content Sniffing

- Some html content is not given the proper tags for its interpretation
- So Microsoft decided to give IE the capacity to guess the interpretation of a byte-stream, called content-sniffing
- And that algorithm sucks.

Content Sniffing

- By using bugs in the IE content sniffer, users can be deceived about what sort of content they are downloading (they thing .jpg, they get a script).

Content Sniffing

- By using bugs in the IE content sniffer, users can be deceived about what sort of content they are downloading (they thing .jpg, they get a script).
- This is mainly a problem on sites where users can both upload and download data.

Preventing Content Sniffing Attacks

- Add the following to the nginx.conf:

```
add_header X-Content-Type-Options nosniff;
```

Preventing Content Sniffing Attacks

- Add the following to the nginx.conf:

  ```
  add_header X-Content-Type-Options nosniff;
  ```

- Or to your settings.py:

  ```
  SECURE_CONTENT_TYPE_NOSNIFF = True
  ```

Certificate Authority Fraud

- What happens if two certificate authorities issue certificates for the same website?

Certificate Authority Fraud

- What happens if two certificate authorities issue certificates for the same website?
- Example: The Iranian gov't hacked the Dutch certificate authority, and issued a certificate for google.com.

Certificate Authority Fraud

- What happens if two certificate authorities issue certificates for the same website?
- Example: The Iranian gov't hacked the Dutch certificate authority, and issued a certificate for google.com.
- An Iranian hacked Comodo and issued numerous certificates for various websites, allowing him to eavesdrop on anyone who resolved to his certs.

Mitigation for Certificate Authority Fraud:

- A way of dealing with CA fraud is called *public key pinning*.

Mitigation for Certificate Authority Fraud:

- A way of dealing with CA fraud is called *public key pinning*.
- This tells your browser to remember what the public keys were for your website; *trusting what it receives the first time*

Public Key Pinning

- In order to generate the hashes of your public keys use

Public Key Pinning

- In order to generate the hashes of your public keys use
- If that key gets compromised, users browsers will detect fraud if you have to generate new keys. So you need to generate a hash of a backup key:

```
$ openssl rsa -in backup.key -outform der -pubout |
    openssl dgst -sha256 -binary |
      openssl enc -base64
```

Public Key Pinning

- In order to generate the hashes of your public keys use
- If that key gets compromised, users browsers will detect fraud if you have to generate new keys. So you need to generate a hash of a backup key:

```
$ openssl rsa -in backup.key -outform der -pubout |
      openssl dgst -sha256 -binary |
        openssl enc -base64
```

- Copy and paste these hashes into the server section of `default`:

```
add_header Public-Key-Pins
      'pin-sha256="N75JcN+pnfzlS9WlZ5MQ5bMrYf8FixevQdnXECdeI8k=";
      pin-sha256="LK8yU6d5hJnXaONIycD2bYNCwu9MVBL3MjM/Fs1a9pg=";
      includeSubDomains; max-age=0';
```

Diminishing Returns

- nginx announces its version to all the world

Diminishing Returns

- nginx announces its version to all the world
- This makes life easy for zero-day hoarders

Diminishing Returns

- nginx announces its version to all the world
- This makes life easy for zero-day hoarders
- Make life *a bit* harder for them by adding this to the http section of nginx.conf:

  ```
  server_tokens off;
  ```

Diminishing Returns

- nginx announces its version to all the world
- This makes life easy for zero-day hoarders
- Make life *a bit* harder for them by adding this to the http section of nginx.conf:

  ```
  server_tokens off;
  ```

- SSLlabs can still determine that nginx is serving the website, but doesn't know the version.