Securing Django Websites

Nick Thompson

July 26, 2015

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
- gunicorn is a web server gateway interface
- We need nginx to reverse proxy because gunicorn is trivially vunerable to DOS attacks

Install nginx

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

Symlink our nginx.conf into the global

```
$ cd django_https
$ rm -f /etc/nginx/nginx.conf
$ rm -f /etc/nginx/sites-available/default
$ ln -s `pwd`/nginx.conf /etc/nginx/nginx.conf
$ ln -s `pwd`/default /etc/nginx/sites-available/default
```

(Sorry peeps I didn't bother to relativize all the paths; you'll need to edit!)

How to serve the website

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

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

Redirects

If we don't listen on port 80, then users will need to type the protocol into the browser, which no one does.
Hence we need nginx to redirect:

```
server {
    listen          80;
    listen          [::]:80;
    server_name     www.example.com example.com;
    return          301 https://$server_name$request_uri;
}
```

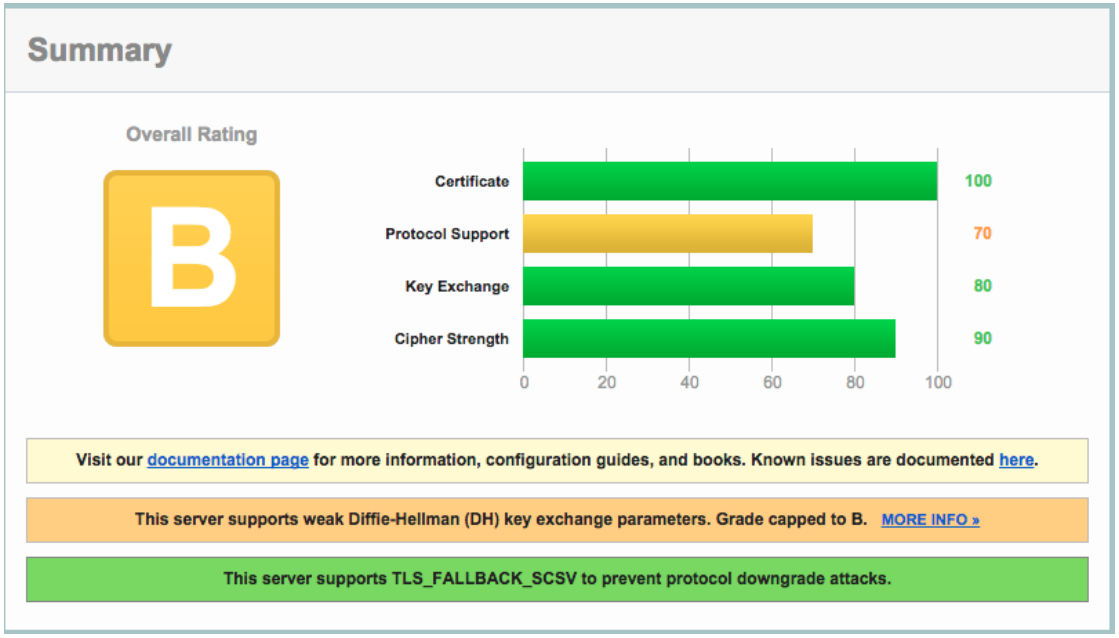
Turn SSL on and proxy-pass to gunicorn

```
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 secure is the default nginx configuration?

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



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 on their report.

Some justification . . .

- The 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 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. It suffers CBC Chaining attacks and Padding Oracle attacks. 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, so we need to restrict the supported ciphers by adding this to the http section of nginx.conf:

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

This is a mess, what does it mean?

This is called ciphersuite negotiation

- The browser tells the server what cipher suites 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.

Interpreting ssl_ciphers

The order matters: Adding “ssl_prefer_server_ciphers on;“ to the nginx.conf tells nginx to ignore the preferences of the browser.
The order of entries to ssl_ciphers tells nginx what cipher suite it should prefer.

Components of a ciphersuite

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

Understanding available ciphersuites

To see what ciphers your nginx supports, run

```
$ openssl ciphers
```

(nginx links against openssl)

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)
```

So ECDHE-RSA-AES256-GCM-SHA384 used elliptic curve Diffie-Helman for key exchange, RSA for authentication, 256 bit Advanced Encryption Standard in Galois Counter mode for encryption, and "authentication encryption with associated data" for message authentication.

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.

SSLLabs thinks 256 bit bulk encryption with SHA-256 or greater is neat:

So let's try this:

```
ssl_ciphers = 'ECDHE-ECDSA-AES128-GCM-SHA256:
                    ECDHE-RSA-AES128-GCM-SHA256:
                    DHE-RSA-AES128-GCM-SHA256:
                    ECDHE-ECDSA-AES256-SHA:
                    ECDHE-RSA-AES256-SHA:
                    DHE-RSA-AES256-SHA:
                    ECDHE-ECDSA-AES128-SHA:
                    ECDHE-RSA-AES128-SHA:
                    DHE-RSA-AES128-SHA:
                    RSA-AES128-GCM-SHA256:
                    RSA-AES256-SHA';
```

Some advice:

- Prefer 128-bit AES to 256-bit AES
- Prefer Galois counter modes as they consume less resources
- Prefer SHA256 over SHA1 as SHA1 will be deprecated.
- Prefer elliptic curves, unless you think the NSA has a quantum computer.

Enabling Perfect-Forward Secrecy

". . . which means a compromise of the server's long term signing key does not compromise the confidentiality of past session"
We can get perfect forward secrecy via use of ephemeral keys . . .

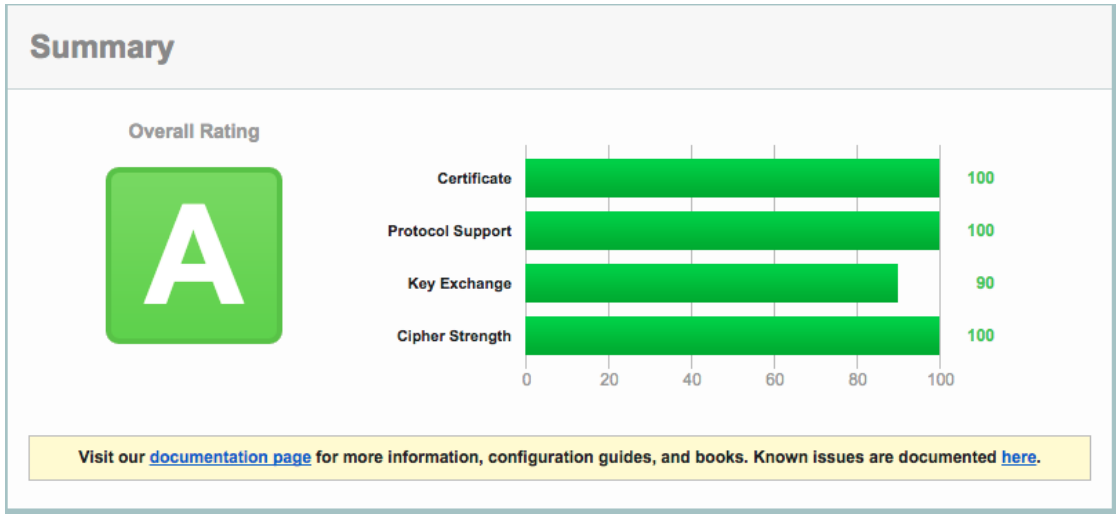
To use ephemeral keys we need to generate big primes:

```
$ 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

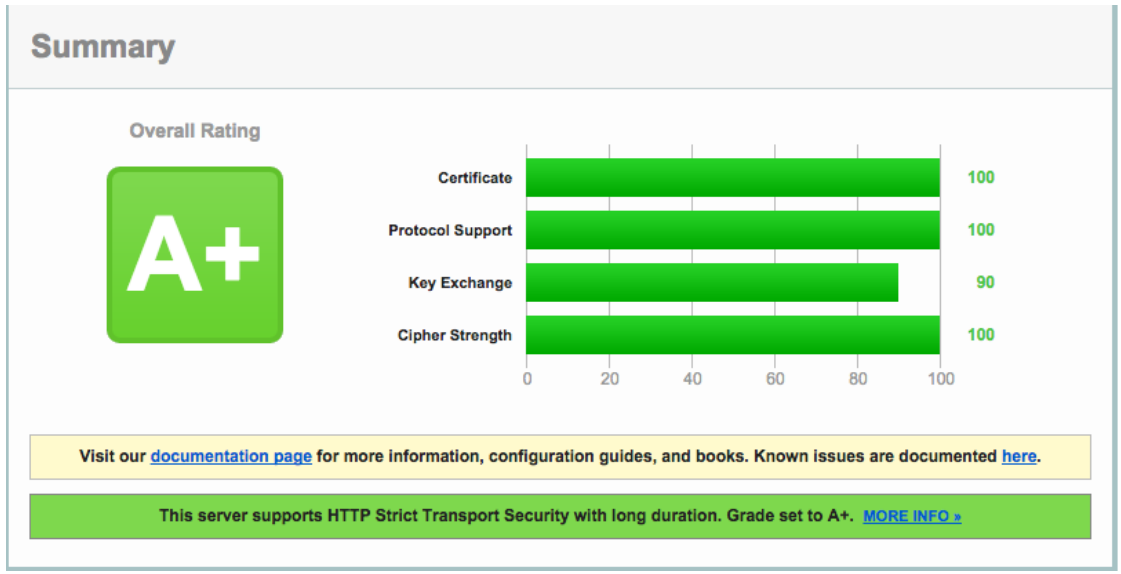


You can see the SSLLabs rating guide here.

But there's still low-hanging fruit!

Add this to the server section in default:

```
add_header Strict-Transport-Security ''max-age=63072000; includeSubdoma
```



Last one!

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

```
$ openssl genrsa -out foo.key 4096
$ 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!