SSL for Advertisers

As HTTPS is used more widely by web sites, advertising networks will need to be able to serve adverts to this growing fraction of the market. In order to show adverts on HTTPS sites, the ad-network Javascript and creatives need to be served over HTTPS themselves.

Javascript that isn't served over HTTPS cannot be loaded by HTTPS websites in modern browsers. Creatives (images) served over HTTP *can* be loaded by HTTPS sites, but will trigger unsightly warnings in browsers, which will appear to be a problem with the site showing the ad.

This document aims to guide you in correct configuration of HTTPS with minimal cost.

Glossary

SSL	A protocol for establishing a secure connection between two computers.
TLS	Another name for SSL. SSL was created by Netscape and then standardised at the IETF. When the IETF took over the protocol they renamed and renumbered it. Thus TLS 1.0 came after SSL 3.0. Many places will write SSL/TLS in order to be clear that they mean either.
HTTPS	The combination of HTTP (the web protocol) and SSL - used for secure web sites.

Certificates

In order to serve an HTTPS site, you need a private key and corresponding certificate. A private key is the solution to a randomly generated, mathematical puzzle. The puzzle that a private key can solve is called the *public key*. A certificate asserts that the server that can solve a given puzzle is the true owner of one or more domains.

Your private key should be a 2048-bit, RSA key. Larger private keys are slower and smaller keys are being phased out. ECDSA is an alternative, more efficient scheme, but not enough browsers support it to use it exclusively. It is possible to have both an RSA and ECDSA key in order to use the more efficient, ECDSA key with browsers that support it. But that is a rare and advanced configuration that will not be covered here.

You must keep the private key secure. It must be generated on a trusted computer that you control and securely distributed to the servers that will need to use it. Never allow the private key outside of your organisation.

Once you have a private key, you will need to send the corresponding public key to a Certificate Authority, who will provide a certificate linking that public key and your domain names. When sending the public key to a CA, a format called a Certificate Signing Request (CSR) is often used and this standard format should be supported by any key generation tool. *Never send your private key to the CA.*

The certificate should include all the domains that you use. Don't forget that `www.example.com` and `example.com` are two different domains. If you have many subdomains, it's possible to obtain a wildcard certificate that covers, say, `*.example.com`. The wildcard matches exactly one label, so `*.example.com` covers `foo.example.com`, but not `foo.bar.example.com` and not `example.com` itself.

Certificates are valid for a limited period of time and will need to be periodically replaced. Certificate Authorities will sell certificates that are valid for several years, but forgetting to update certificates and serving with an expired certificate is a common failure. An expired certificate will cause all HTTPS serving to fail.

Because of this, you should get certificates that expire after twelve months and mark your calendar to replace them several weeks prior to that date. By doing this once a year, you ensure that the institutional knowledge needed to replace certificates is maintained in your organisation.

Certificate Authorities themselves also have certificates and your servers must be configured to send browsers both *your* certificate and one or more certificates for the CA that you use. These latter certificates are called *intermediate certificates*. Without them, many clients will still work but some will not and those that do will be slower because they need to fetch the missing, intermediate certificates from the CA. The set of certificates for a given site is called a *certificate chain*. Your site's certificate must be the first certificate in the chain and your CA will provide one or more certificates to add to that.

(Some server software expects your certificate to be in one file and then the intermediate certificates to be in another file called the "chain file". Consult the documentation for your server software.)

Configuration

Almost all web server software will support HTTPS. Notable, open-source servers that support HTTPS include Apache and nginx. In each case, HTTPS support has to be separately configured. Apache has mod_ssl and nginx needs the `ssl` parameter to be set.

SSL is, sadly, a protocol with many versions and options:

1. SSLv2 is absolutely obsolete and must be disabled.
2. SSLv3 is obsolete and should be disabled.
3. TLS 1.0, 1.1 and 1.2 should all be supported. Modern browsers use TLS 1.2 and, if your server does not, you should look into updating it. Apache and nginx with OpenSSL 1.0.1 will support TLS 1.2.

SSL provides a framework within which many cryptographic primitives can be used. A set of primitives that can be used on a given connection is called a *cipher suite*. You are likely to need to configure the cipher suites that your server will use and give a preference order:

1. We recommend ECDHE-RSA-AES128-GCM (which may be written in full, IANA form as `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`) as your top preference cipher suite. If your server software doesn't support it then look into updating.
2. ECDHE-RSA-AES256-GCM, AES128-GCM, AES256-GCM are all fine, second preference cipher suites.
3. Older browsers won't support AES-GCM. For them, the question of whether AES-CBC or RC4 is better for them is complex, but the industry is trending against RC4 now. For compatibility with old browsers, you should enable RC4-SHA and AES128-SHA, but at a lower preference than the GCM cipher suites.
4. Any cipher suites mentioning DES, 3DES or Triple-DES are obsolete and should be the least preferable if enabled.
5. Any cipher suites mentioning ADH, AECDH, NULL, EXP or Export are inappropriate for HTTPS and must be disabled.

SSL also provides other, optional, features:

1. Renegotiation should be disabled. It is not needed by the *vast* majority of sites and provides an attack surface that attackers can use to overload the server.
2. Compression should be disabled. It is not supported by most browsers because it can be used to extract secrets from encrypted connections.

SSL *resumption* is an important feature that reduces the load on servers. If it is not working correctly then everything will still function but you'll be missing out on an important benefit.

Traditional session caching depends on all frontends sharing the same session cache. This is easy to setup (Apache, nginx) for a single machine, but harder to build if several machines need to share the same cache. Apache supports distcache for this while nginx doesn't appear to have a documented mechanism. Fragmenting the session cache across frontend servers may dramatically reduce resumption rates depending on how load-balancing is configured across them.

Session Tickets keys can be configured (Apache, nginx) in order to support resumption across frontends without a shared cache for clients that support Session Tickets. But be mindful of the fact that not all clients support Session Tickets and they can compromise forward-security unless care is taken. In nginx, Session Tickets can be disabled, although seemingly not in Apache.

The HTTP in HTTPS

An important part of an efficient, HTTPS site is getting the HTTP parts correctly configured. Configuring caching correctly and using persistent connections are standard practices that apply even without HTTPS, but their benefits can be magnified when HTTPS is used.

When serving HTML or Javascript for use on an HTTPS site, the URLs in that HTML and Javascript must be either HTTPS URLs, or must be protocol relative. Protocol relative URLs omit any protocol (e.g. `//www.google.com/image.png`) and inherit the protocol of the parent resource..

Any included subresources must also follow this rule, transitively. So ad-network Javascript loaded by an HTTPS site must be served over HTTPS. If that Javascript adds an iframe to the page then the source of that iframe must be HTTPS. If that iframe source, in turn, includes images then they must also be served over HTTPS.

Cookies, even with HTTPS, are not protected unless the `secure` flag is given when setting them. If you plan to do all of your ad serving over HTTPS, then it is best to add the `secure` attribute to your cookies also. But if you expect to serve ads over both HTTP and HTTPS, then do not mark your cookies as `secure`, since that will prevent them from being sent in the HTTP case.

Checking your config

Qualys provide an excellent, free scanner for HTTPS configurations. You should check your own site's configuration using it and pay attention to any warnings that it gives you.

Advanced Topics

SSL/TLS packages the bytes of the application protocol (HTTP in the case of HTTPS) into records. Each record has a signature and a header and are packed into packets. Each packet has headers. The overhead of a record is typically 21 to 40 bytes (based on common ciphersuites) and the overhead of a packet is around 52 bytes. So it's vitally important not to send lots of small packets with small records in them.

By using Wireshark, a common network tool, one can observe the sizes of the record sent on an HTTPS connection. One, large, HTTPS site sent records of the following sizes in response to an HTTP request: 638 bytes, 1363, 15628, 69, 182, 34, 18, … This is often caused because OpenSSL will build a record from each call to `SSL_write` and the kernel, with Nagle disabled, will send out packets to minimise latency.

This can be fixed with a couple of tricks: buffer in front of OpenSSL and don't make `SSL_write` calls with small amounts of data if you have more coming. Also, if code limitations mean that you are building small records in some cases, then use TCP_CORK to pack several of them into a packet.

However, records that are too large are also a problem. No part of a record can be parsed by the browser until the whole record has been received. As the congestion window opens up at the beginning of a TCP connection, large records tend to span several windows and so cause an extra round trip of delay before the browser can process any of the data. Since the browser is pre-parsing the HTML for subresources, it'll delay discovery and cause more knock-on effects.

There's always going to be some uncertainty about the optimal SSL record size because the TCP header size depends on the OS and the number of SACK blocks that need to be sent. In the ideal case, each packet is full and contains exactly one record. You should start with a value of 1415 bytes for a non-padded ciphersuite (like AES-GCM or RC4), or 1403 bytes for an AES based ciphersuite, and look at the packets that result with a tool like Wireshark.