
Web Security Basics

This is a quick review of basic web security concepts. Contributions are always welcome :)

Table of Contents

- SSL/TLS
- CORS (Cross-Origin Resource Sharing)
- Cross-site Scripting attack
- CSRF (Cross-Site Request Forgery)
- Access and Refresh Tokens

SSL/TLS

General Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), are cryptographic protocols designed to provide communications security over a computer network.

SSL (and its successor, TLS) is a protocol that operates directly on top of TCP. This way, protocols on higher layers (such as HTTP) can be left unchanged while still providing a secure connection. Underneath the SSL layer, HTTP is identical to HTTPS.

When using SSL/TLS correctly, all an attacker can see on the cable is which IP and port you are connected to, roughly how much data you are sending, and what encryption and compression is used. He can also terminate the connection, but both sides will know that the connection has been interrupted by a third party.

In typical use, the attacker will also be able to figure out which host name you're connecting to (but not the rest of the URL): although HTTPS itself does not expose the host name, your browser will usually need to make a DNS request first to find out what IP address to send the request to.

TLS is based on specifications developed by Netscape Communications' SSL protocol, which is the predecessor of TLS. TLS and SSL are not interoperable, i.e. TLS cannot be implemented as SSL.

High-level description of the protocol After building a TCP connection, the **SSL handshake** is started by the client. The client sends a number of specifications: which version of SSL/TLS it is running, what ciphersuites it wants to use, and what compression methods it wants to use. The server checks what the highest SSL/TLS version is that is supported by them both, picks a cipher suite from one of the client's options (if it supports one), and optionally picks a compression method.

After this the basic setup is done, the **server sends its certificate**. This certificate must be trusted by either the client itself or a party that the client trusts. For example if the client trusts GeoTrust,

then the client can trust the certificate from Google.com, because GeoTrust cryptographically signed Google's certificate.

Having verified the certificate and being certain this server really is who he claims to be (and not a man in the middle), **a key is exchanged**. This can be a public key, a "PreMasterSecret" or simply nothing, depending on the chosen ciphersuite. Both the server and the client can now compute the key for the symmetric encryption. The client tells the server that from now on, all communication will be encrypted, and sends an encrypted and authenticated message to the server.

The **server verifies that the MAC** (Message Authentication Code) used for authentication is correct, and that the message can be correctly decrypted. It then returns a message, which the client verifies as well. The **handshake is now finished**, and the two hosts can communicate securely.

To close the connection, a close_notify 'alert' is used. If an attacker tries to terminate the connection by finishing the TCP connection (injecting a FIN packet), both sides will know the connection was improperly terminated. The connection cannot be compromised by this though, merely interrupted.

An SSL Connection Next, we're going to look into a very basic outline of the process of establishing an SSL connection.

- Browser requests a HTTPS webpage
- Web Server sends public key and certificate
- Browser examines the SSL Certificate
- Browser creates a symmetric key and sends it to server
- Web server decrypts symmetric key with its private key
- Web server sends browser the page with symmetric key
- Browser decrypts the data and displays page



Advantages of Implementing HTTPS

- **Trust** – If you get an EV certificate that shows the green address bar in the browser, you're going to be giving your visitors a sense of trust. And when they know you're taking their security seriously, they're going to be appreciative.
- **Verification** – One of the best things about installing an SSL certificate on your server is that it guarantees your visitors you really are who you say you are. This is important when trying to do business online.
- **Integrity of Data** – Additionally, with SSL, you can guarantee integrity of data. For example, without SSL, it's possible to not only intercept data going to and from the web server, but to change it as well!
- **Google and SEO** – Last but not least, you have to take into consideration the recent announcements by Google that they're going to be using whether or not a server uses SSL as a ranking signal.

Disadvantages of SSL

- **Cost of Certificate** – It is possible to get a free SSL certificate, but this isn't recommended for a lot of reasons. Depending on the type of cert you buy, the price will vary quite a bit. However, when you consider the added level of security, the cost isn't really prohibitive for most websites.
- **Mixed Modes** – If your SSL implementation isn't setup correctly and you still have some files being served via HTTP rather than HTTPS, visitors are going to get a warning message in their

browser letting them know some of the data isn't protected. This can be confusing to some website visitors.

- Proxy Caching – Another possible problem is if you have a complex proxy caching system setup on your web server. Encrypted content isn't going to be able to be cached. To get around this, you need to add a server to handle the encryption before it gets to the caching server. This will require additional costs, but it's a good way to make sure you're encrypting your visitors' data when they're accessing your website.
- Mobile – When SSL was first implemented, it was meant for web based applications. While the ability to go beyond HTTPS has come a long way in the last few years, it can sometimes be a pain to setup and might require changes to in-house software or buying additional modules from application vendors. Still, for everything that is on the web or accessible via a web browser, SSL / TLS is definitely the way to go.

Myths About SSL / TLS

- Resource Hog – When setup correctly, SSL is not going to eat up all your server resources. While this may have been true ten years ago or more, this isn't the case these days on most modern servers.
- Latencies – Another problem people worry about needlessly is that setting up SSL on their server will cause all of their web pages to load slower in browsers when people view them. Just like the last point, this just isn't the case.
- Cache Problems – In the past SSL could cause problems if you had a system of caching setup on your web server, but this isn't the case for most servers these days. That said, Internet Explorer 6 may still have problems.
- Scary Warning Messages – Another thing that's not really a problem is the myth that installing an SSL certificate will lead to a lot of scary error and warning messages. This is only true if you don't set it up correctly, and it's really not that difficult to be honest.

How to crack SSL There is no simple and straight-forward way; SSL is secure when done correctly. An attacker can try **if the user ignores certificate warnings** though, which would break the security instantly. When a user does this, the attacker doesn't need a private key from a CA to forge a certificate, he merely has to send a certificate of his own.

Another way would be by a flaw in the application (server- or client-side). An easy example is in websites: **if one of the resources used by the website (such as an image or a script) is loaded over HTTP**, the confidentiality cannot be guaranteed anymore. Even though browsers do not send the

HTTP Referer header when requesting non-secure resources from a secure page (source), it is still possible for someone eavesdropping on traffic to guess where you're visiting from; for example, if they know images X, Y, and Z are used on one page, they can guess you are visiting that page when they see your browser request those three images at once. Additionally, when loading Javascript, the entire page can be compromised. An attacker can execute any script on the page, modifying for example to whom the bank transaction will go. When this happens (a resource being loaded over HTTP), the browser gives a mixed-content warning: Chrome, Firefox, Internet Explorer 9

Another trick for HTTP is when the login page is not secured, and it submits to an https page. "Great," the developer probably thought, "now I save server load and the password is still sent encrypted!" The problem is sslstrip, a tool that modifies the insecure login page so that it submits somewhere so that the attacker can read it.

Last but not least, you can resort to other methods to obtain the info that SSL denies you to obtain. If you can already see and tamper with the user's connection, it might not be that hard to replace one of his/her .exe downloads with a keylogger, or simply to physically attack that person. Cryptography may be rather secure, but humans and human error is still a weak factor.

Reference:

How does SSL/TLS work?

Diffie Helman Exchange

CORS (Cross-Origin Resource Sharing)

CORS, is an HTML5 feature that allows one site to access another site's resources despite being under different domain names. Let me explain that a little more. Prior to CORS, a web browser security restriction, known as the Same-Origin Policy, would prevent my web application from calling an external API. The browser would consider two resources to be of the same-origin only if they used the same protocol (http vs. https), the same port, and the same domain (even different subdomains would fail).

Before CORS, you could get around this security restriction by creating some sort of server-side component to shuttle API requests, which was often unduly complicated and unnecessary. You could also use JSONP (JSON with padding) in APIs that supported it but many did not, and, even if they did, JSONP is limited to GET requests only.

With CORS, my web app on one domain can freely communicate with your API on another domain, even using the methods POST, PUT, and DELETE, provided that your API's security restrictions specify that this is allowed and that you have established the communication through the CORS specification

as well. This means that you can eliminate the need for a server-side component and do all the API communication on the client-side using JavaScript.

For more complex requests, the browser will “preflight” the request by sending an OPTIONS request to the server first. This request is basically there to ask the server if the full request is permissible. Note that while this requires extra setup on the server side, the browser will do this automatically depending on the characteristics of the request.

Reference:

Using CORS

Cross-site Scripting attack

Cross-site Scripting (XSS) refers to client-side code injection attack wherein an attacker can execute malicious scripts (also commonly referred to as a malicious payload) into a legitimate website or web application. XSS is amongst the most rampant of web application vulnerabilities and occurs when a web application makes use of unvalidated or unencoded user input within the output it generates.

By leveraging XSS, an attacker does not target a victim directly. Instead, an attacker would exploit a vulnerability within a website or web application that the victim would visit, essentially using the vulnerable website as a vehicle to deliver a malicious script to the victim’s browser.

While XSS can be taken advantage of within VBScript, ActiveX and Flash (although now considered legacy or even obsolete), unquestionably, the most widely abused is JavaScript – primarily because JavaScript is fundamental to most browsing experiences.

Reference

Cross-site Scripting (XSS) Attack

CSRF (Cross-Site Request Forgery)

Reference

Understanding CSRF

CSRF Demystified

Video: Cross Site Request Forgery explained

CSRF Attacks

How would you explain CSRF token to a newbie?

Access and Refresh Tokens

Since the HTTP protocol is stateless, this means that if we authenticate a user with a username and password, then on the next request, our application won't know who we are. We would have to authenticate again.

OAuth (Open Authorization) is the open standard for token-based authentication and authorization on the Internet.

Consumer key is essentially the API key associated with the application (Twitter, Facebook, etc.). This key (or 'client ID', as Facebook calls it) is what identifies the client. By the way, a client is a website/service that is trying to access an end-user's resources.

Consumer secret is the client password that is used to authenticate with the authentication server, which is a Twitter/Facebook/etc. server that authenticates the client.

Access token is what is issued to the client once the client successfully authenticates itself (using the consumer key & secret). This access token defines the privileges of the client (what data the client can and cannot access). Now every time the client wants to access the end-user's data, the access token secret is sent with the access token as a password (similar to the consumer secret).

Refresh token is a special kind of JWT that is used to authenticate a user without them needing to re-authenticate.

The main advantage of a refresh token is that it is easier to detect if it is compromised.

Consider these two scenarios:

1. A single long-lasting auth token is used.
2. A short duration auth token is used, and a long-lasting refresh token periodically requests a new auth token once the previous one has expired.

In scenario 1, if the auth token is compromised it would be hard for anyone to detect this, and the unauthorized access could continue indefinitely.

In scenario 2, if only the auth token is compromised (refresh token is not compromised too), it could only continue until the token expires.

In scenario 2, if the refresh token is compromised, once the refresh token is invoked, all other auth tokens that were generated using that refresh token are invalidated, so only 1 party can use the api (per refresh token) at a time. This results in multiple users repeatedly invalidating each other's auth

tokens by generating new ones. The api could detect the breach because refreshes are being made prior to the auth token expiration, and would know to immediately revoke the refresh token. A hacker cannot use the refresh token to create a new access token because the client ID and client secret are needed along with the refresh token in order to generate the new access token.

Reference

Tokens

The Ins and Outs of Token Based Authentication

The Anatomy of a JSON Web Token

Stackoverflow: What is token based authentication?

10 Things You Should Know about Tokens

OAuth

An introduction to OAuth2

Dancing with OAuth: Understanding how Authorization Works

Understanding OAuth2

Access Token

Why do access tokens expire?

Why use an authentication token instead of the username/password per request?

Refresh Token

Why Does OAuth v2 Have Both Access and Refresh Tokens?

Understanding Refresh Tokens

Refresh Tokens

How secure are expiring tokens and refresh tokens?

Other Links

What should every programmer know about security?