6.858 Lecture 14
SSL/TLS and HTTPS

This lecture is about two related topics:
- How to cryptographically protect network communications, at a larger scale than Kerberos? [Technique: certificates.]
- How to integrate cryptographic protection of network traffic into the web security model? [HTTPS, Secure cookies, etc.]

Recall: two kinds of encryption schemes.
- E is encrypt, D is decrypt
- Symmetric key cryptography means same key is used to encrypt & decrypt
  - ciphertext = $E_k$(plaintext)
  - plaintext = $D_k$(ciphertext)
- Asymmetric key (public-key) cryptography: encrypt & decrypt keys differ
  - ciphertext = $E_{PK}$(plaintext)
  - plaintext = $D_{SK}$(ciphertext)
  - PK and SK are called public and secret (private) key, respectively
- Public-key cryptography is orders of magnitude slower than symmetric

Encryption provides data secrecy, often also want integrity.
- Message authentication code (MAC) with symmetric keys can provide integrity.
  - Look up HMAC if you're interested in more details.
- Can use public-key crypto to sign and verify, almost the opposite:
  - Use secret key to generate signature (compute $D_{SK}$)
  - Use public key to check signature (compute $E_{PK}$)

Recall from last lecture: Kerberos.
- Central KDC knows all principals and their keys.
- When A wants to talk to B, A asks the KDC to issue a ticket.
- Ticket contains a session key for A to talk to B, generated by KDC.

Why is Kerberos not enough? E.g., why isn't the web based on Kerberos?
- Might not have a single KDC trusted to generate session keys.
- Not everyone might have an account on this single KDC.
- KDC might not scale if users contact it every time they went to a web site.
- Unfortunate that KDC knows what service each user is connecting to.
- These limitations are largely inevitable with symmetric encryption.

Alternative plan, using public key encryption.
- Suppose A knows the public key of B.
- Don't want to use public-key encryption all the time (slow).
- Strawman protocol for establishing a secure connection between A and B:
  - A generates a random symmetric session key $S$.
  - A encrypts $S$ for $PK_B$, sends to B.
Now we have secret key S shared between A and B, can encrypt and authenticate messages using symmetric encryption, much like Kerberos.

Good properties of this strawman protocol:
- A's data seen only by B.
  - Only B (with SK_B) can decrypt S.
  - Only B can thus decrypt data encrypted under S.
- No need for a KDC-like central authority to hand out session keys.

What goes wrong with this strawman?
- Adversary can record and later replay A's traffic; B would not notice.
  - Solution: have B send a nonce (random value).
  - Incorporate the nonce into the final master secret S' = f(S, nonce).
  - Often, S is called the pre-master secret, and S' is the master secret.
  - This process to establish S' is called the "handshake".
- Adversary can impersonate A, by sending another symmetric key to B.
  - Many possible solutions, if B cares who A is.
  - E.g., B also chooses and sends a symmetric key to A, encrypted with PK_A.
  - Then both A and B use a hash of the two keys combined.
  - This is roughly how TLS client certificates work.
- Adversary can later obtain SK_B, decrypt symmetric key and all messages.
  - Solution: use a key exchange protocol like Diffie-Hellman, which provides forward secrecy, as discussed in last lecture.

Hard problem: what if neither computer knows each other's public key?
- Common approach: use a trusted third party to generate certificates.
- Certificate is tuple (name, pubkey), signed by certificate authority.
- Meaning: certificate authority claims that name's public key is pubkey.
- B sends A a pubkey along with a certificate.
- If A trusts certificate authority, continue as above.

Why might certificates be better than Kerberos?
- No need to talk to KDC each time client connects to a new server.
- Server can present certificate to client; client can verify signature.
- KDC not involved in generating session keys.
- Can support "anonymous" clients that have no long-lived key / certificate.

Plan for securing web browsers: HTTPS.
- New protocol: https instead of http (e.g., https://www.paypal.com/).
- Need to protect several things:
  - A. Data sent over the network.
  - B. Code/data in user's browser.
  - C. UI seen by the user.

A. How to ensure data is not sniffed or tampered with on the network?
• Use TLS (a cryptographic protocol that uses certificates).
• TLS encrypts and authenticates network traffic.
• Negotiate ciphers (and other features: compression, extensions).
• Negotiation is done in clear.
• Include a MAC of all handshake messages to authenticate.

B. How to protect data and code in the user's browser?
• Goal: connect browser security mechanisms to whatever TLS provides.
• Recall that browser has two main security mechanisms:
  o Same-origin policy.
  o Cookie policy (slightly different).

• Same-origin policy with HTTPS/TLS.
  o TLS certificate name must match hostname in the URL
  o In our example, certificate name must be www.paypal.com.
  o One level of wildcard is also allowed (*.paypal.com)
  o Browsers trust a number of certificate authorities.

• Origin (from the same-origin policy) includes the protocol.
  o http://www.paypal.com/ is different from https://www.paypal.com/
  o Here, we care about integrity of data (e.g., Javascript code).
  o Result: non-HTTPS pages cannot tamper with HTTPS pages.
  o Rationale: non-HTTPS pages could have been modified by adversary.

• Cookies with HTTPS/TLS.
  o Server certificates help clients differentiate between servers.
  o Cookies (common form of user credentials) have a "Secure" flag.
  o Secure cookies can only be sent with HTTPS requests.
  o Non-Secure cookies can be sent with HTTP and HTTPS requests.

• What happens if adversary tampers with DNS records?
  o Good news: security doesn't depend on DNS.
  o We already assumed adversary can tamper with network packets.
  o Wrong server will not know correct private key matching certificate.

C. Finally, users can enter credentials directly. How to secure that?
• Lock icon in the browser tells user they're interacting with HTTPS site.
• Browser should indicate to the user the name in the site's certificate.
• User should verify site name they intend to give credentials to.

How can this plan go wrong?
• As you might expect, every step above can go wrong.
• Not an exhaustive list, but gets at problems that ForceHTTPS wants to solve.

There have been some attacks on the cryptographic parts of SSL/TLS.

- Attack by Rizzo and Duong can allow adversary to learn some plaintext by issuing many carefully-chosen requests over a single connection.
- Recent attack by same people using compression, mentioned in iSEC lecture.
  - Ref: http://en.wikipedia.org/wiki/CRIME
- Most recently, more padding oracle attacks.
  - Ref: https://www.openssl.org/~bodo/ssl-poodle.pdf
- Some servers/CAs use weak crypto, e.g. certificates using MD5.
- Some clients choose weak crypto (e.g., SSL/TLS on Android).
  - Ref: http://op-co.de/blog/posts/android_ssl downgrade/
- But, cryptography is rarely the weakest part of a system.

2 (B). Authenticating the server.

Adversary may be able to obtain a certificate for someone else's name.

- Used to require a faxed request on company letterhead (but how to check?)
- Now often requires receiving secret token at root@domain.com or similar.
- Security depends on the policy of least secure certificate authority.
- There are 100’s of trusted certificate authorities in most browsers.
- Several CA compromises in 2011 (certs for gmail, facebook, ..)
  - Ref: http://dankaminsky.com/2011/08/31/notnotar/
- Servers may be compromised and the corresponding private key stolen.

How to deal with compromised certificate (e.g., invalid cert or stolen key)?

- Certificates have expiration dates.
- Checking certificate status with CA on every request is hard to scale.
- Certificate Revocation List (CRL) published by some CA's, but relatively few certificates in them (spot-checking: most have zero revoked certs).
- CRL must be periodically downloaded by client.
  - Could be slow, if many certs are revoked.
  - Not a problem if few or zero certs are revoked, but not too useful.
- OCSP: online certificate status protocol.
  - Query whether a certificate is valid or not.
  - One issue: OCSP protocol didn't require signing "try later" messages.
- Various heuristics for guessing whether certificate is OK or not.
  - CertPatrol, EFF's SSL Observatory, ..
  - Not as easy as "did the cert change?". Websites sometimes test new CAs.
- Problem: online revocation checks are soft-fail.
  - An active network attacker can just make the checks unavailable.
  - Browsers don't like blocking on a side channel.
    - Performance, single point of failure, captive portals, etc.
In practice browsers push updates with blacklist after major breaches.
  o Ref: https://www.imperialviolet.org/2011/03/18/revocation.html
  o Ref: https://www.imperialviolet.org/2012/02/05/crlsets.html

Users ignore certificate mismatch errors.
• Despite certificates being easy to obtain, many sites misconfigure them.
• Some don’t want to deal with (non-zero) cost of getting certificates.
• Others forget to renew them (certificates have expiration dates).
• End result: browsers allow users to override mismatched certificates.
  o Problematic: human is now part of the process in deciding if cert is valid.
  o Hard for developers to exactly know what certs will be accepted by browsers.
• Empirically, about 60% of bypass buttons shown by Chrome are clicked through.
  o (Though this data might be stale at this point..)

What’s the risk of a user accepting an invalid certificate?
• Might be benign (expired cert, server operator forgot to renew).
• Might be a man-in-the-middle attack, connecting to adversary’s server.
• Why is this bad?
  o User’s browser will send user’s cookies to the adversary.
  o User might enter sensitive data into adversary’s website.
  o User might assume data on the page is coming from the right site.

3 (B). Mixing HTTP and HTTPS content.
Web page origin is determined by the URL of the page itself. Page can have many embedded elements:
• Javascript via <SCRIPT> tags
• CSS style sheets via <STYLE> tags
• Flash code via <EMBED> tags
• Images via <IMG> tags
If adversary can tamper with these elements, could control the page. In particular, Javascript and Flash code give control over page.
• CSS: less control, but still abusable, esp w/ complex attribute selectors.
Probably the developer wouldn’t include Javascript from attacker’s site. But, if the URL is non-HTTPS, adversary can tamper with HTTP response.

Alternative approach: explicitly authenticate embedded elements.
• E.g., could include a hash of the Javascript code being loaded.
  o Prevents an adversary from tampering with response.
  o Does not require full HTTPS.
• Might be deployed in browsers in the near future.
  o Ref: http://www.w3.org/TR/SRI/

4 (B). Protecting cookies.
• Web application developer could make a mistake, forgets the Secure flag.

Suppose the user only visits https://bank.com/. Why is this still a problem?
• Adversary can cause another HTTP site to redirect to http://bank.com/.
• Even if user never visits any HTTP site, application code might be buggy.
  o Some sites serve login forms over HTTPS and serve other content over HTTP.
  o Be careful when serving over both HTTP and HTTPS.
    ▪ E.g., Google's login service creates new cookies on request.
    ▪ Login service has its own (Secure) cookie.
    ▪ Can request login to a Google site by loading login's HTTPS URL.
    ▪ Used to be able to also login via cookie that wasn't Secure.
    ▪ ForceHTTPS solves problem by redirecting HTTP URLs to HTTPS.
    ▪ Ref: http://blog.icir.org/2008/02/sidejacking-forced-sidejacking-and.html

Cookie integrity problems.
• No way to determine who set the cookie.

5 (C). Users directly entering credentials.
• Phishing attacks.
• Users don't check for lock icon.
• Users don't carefully check domain name, don't know what to look for.
  o E.g., typo domains (paypa1.com), uncode
• Web developers put login forms on HTTP pages (target login script is HTTPS).
  o Adversary can modify login form to point to another URL.
  o Login form not protected from tampering, user has no way to tell.

How does ForceHTTPS (this paper) address some of these problems?
• Server can set a flag for its own hostname in the user's browser.
  o Makes SSL/TLS certificate misconfigurations into a fatal error.
  o Redirects HTTP requests to HTTPS.
  o Prohibits non-HTTPS embedding (+ performs ForceHTTPS for them).

What problems does ForceHTTPS solve?
• Mostly 2, 3, and to some extent 4.
  o Users accepting invalid certificates.
  o Developer mistakes: embedding insecure content.
  o Developer mistakes: forgetting to flag cookie as Secure.
  o Adversary injecting cookies via HTTP.

Is this really necessary? Can we just only use HTTPS, set Secure cookies, etc?
• Users can still click-through errors, so it still helps for #2.
• Not necessary for #3 assuming the web developer never makes a mistake.
• Still helpful for #4.
  o Marking cookies as Secure gives confidentiality, but not integrity.

Why not just turn on ForceHTTPS for everyone?
• HTTPS site might not exist.
• If it does, might not be the same site (https://web.mit.edu is authenticated, but http://web.mit.edu isn’t).
• HTTPS page may expect users to click through (self-signed certs).

Implementing ForceHTTPS.
• The ForceHTTPS bit is stored in a cookie.
• Interesting issues:
  o State exhaustion (the ForceHTTPS cookie getting evicted).
  o Denial of service (force entire domain; force via JS; force via HTTP).
    ▪ Why does ForceHTTPS only allow specific hosts, instead of entire domain?
    ▪ Why does ForceHTTPS require cookie to be set via headers and not via JS?
    ▪ Why does ForceHTTPS require cookie to be set via HTTPS, not HTTP?
  o Bootstrapping (how to get ForceHTTPS bit; how to avoid privacy leaks).
    ▪ Possible solution 1: DNSSEC.
    ▪ Possible solution 2: embed ForceHTTPS bit in URL name (if possible).
    ▪ If there’s a way to get some authenticated bits from server owner (DNSSEC, URL name, etc), should we just get the public key directly?
    ▪ Difficulties: users have unreliable networks. Browsers are unwilling to block the handshake on a side-channel request.

Current status of ForceHTTPS.
• Some ideas from ForceHTTPS have been adopted into standards.
• HTTP Strict-Transport-Security header is similar to a ForceHTTPS cookie.
  o Ref: http://tools.ietf.org/html/rfc6797
  o Ref: http://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security
• Uses header instead of magic cookie:
  o Strict-Transport-Security: max-age=7884000; includeSubDomains
• Turns HTTP links into HTTPS links.
• Prohibits user from overriding SSL/TLS errors (e.g., bad certificate).
• Optionally applies to all subdomains.
  o Why is this useful?
  o non-Secure and forged cookies can be leaked or set on subdomains.
• Optionally provides an interface for users to manually enable it.
  • Implemented in Chrome, Firefox, and Opera.
  • Bootstrapping largely unsolved.
    o Chrome has a hard-coded list of preloads.
  • IE9, Firefox 23, and Chrome now block mixed scripting by default.
    o Ref: [http://blog.chromium.org/2012/08/ending-mixed-scripting-vulnerabilities.html](http://blog.chromium.org/2012/08/ending-mixed-scripting-vulnerabilities.html)
    o Ref: [https://blog.mozilla.org/tanvi/2013/04/10/mixed-content-blocking-enabled-in-firefox-23/](https://blog.mozilla.org/tanvi/2013/04/10/mixed-content-blocking-enabled-in-firefox-23/)

Another recent experiment in this space: HTTPS-Everywhere.
• Focuses on the "power user" aspect of ForceHTTPS.
  • Allows users to force the use of HTTPS for some domains.
  • Collaboration between Tor and EFF.
  • Add-on for Firefox and Chrome.
  • Comes with rules to rewrite URLs for popular web sites.

Other ways to address problems in SSL/TLS
• Better tools / better developers to avoid programming mistakes.
  o Mark all sensitive cookies as Secure (#4).
  o Avoid any insecure embedding (#3).
  o Unfortunately, seems error-prone..
  o Does not help end-users (requires developer involvement).
• EV certificates.
  o Trying to address problem 5: users don’t know what to look for in cert.
  o In addition to URL, embed the company name (e.g., "PayPal, Inc.")
  o Typically shows up as a green box next to the URL bar.
  o Why would this be more secure?
  o When would it actually improve security?
  o Might indirectly help solve #2, if users come to expect EV certificates.
• Blacklist weak crypto.
• Browsers are starting to reject MD5 signatures on certificates
  o (iOS 5, Chrome 18, Firefox 16)
• and RSA keys with < 1024 bits.
  o (Chrome 18, OS X 10.7.4, Windows XP+ after a recent update)
• and even SHA-1 by Chrome.
• OCSP stapling.
  o OCSP responses are signed by CA.
  o Server sends OCSP response in handshake instead of querying online (#2).
- Effectively a short-lived certificate.
- Problems:
  - Not widely deployed.
  - Only possible to staple one OCSP response.
- Key pinning.
  - Only accept certificates signed by per-site whitelist of CAs.
  - Remove reliance on least secure CA (#2).
  - Currently a hard-coded list of sites in Chrome.
  - Diginotar compromise caught in 2011 because of key pinning.
  - Plans to add mechanism for sites to advertise pins.
    - Ref: [http://tack.io/](http://tack.io/)
  - Same bootstrapping difficulty as in ForceHTTPS.

Other references:
- [http://www.imperialviolet.org/2012/07/19/hope9talk.html](http://www.imperialviolet.org/2012/07/19/hope9talk.html)