TLS/SSL
(Mis)Protecting our Connections’ Security

Orr Dunkelman

Computer Science Department
University of Haifa

6th September, 2015
Outline

1. Introduction to SSL/TLS
2. What Could Possibly Go Wrong?
   - The Quick and Dirty PKI 101
   - CNNIC — If you cannot hack them, CA them!
   - DigiNotar — Why Not Hack Them?
   - MD5 — The Silent Killer
   - Real Life is Always Stronger
   - Losing all Your Security in a Heartbeat
   - No Entropy in Public Keys
3. How Not to Implement a Authenticated Encryption
   - The POODLE Attack
   - The BEAST Attack
   - The Lucky13 Attack
   - The Perfect CRIME Attack
4. Downgrade Attacks
   - Legacy Cryptosystems
   - FREAKing out TLS
   - Logjam
5. Summary

Orr Dunkelman
TLS/SSL — (Mis)Protecting our Connections’ Security
Introduction to SSL/TLS

- Developed by Netscape (mid 90’s) for E-Commerce
- SSL 1.0 never published, SSL 2.0 published in 1995 (and broken)
- SSL 3.0 published in 1996 (designed by Kocher, Karlton, Freier)
- Became IETF responsibility which re-named it TLS 1.0
- Later became TLS 1.1, 1.2, ...
Working Environment

- SSL was designed in the mid 90’s for E-Commerce
- To offer transport-level security:
  - End-to-End,
  - On top of TCP (session-/connection- services offered by TCP),
- Assumptions:
  - Server side is strong,
  - Client side is weak,
  - No forward secrecy needed,
  - Server’s identity to be verified by crypto (client — by server’s other mechanisms*),
  - Crypto-algorithm agility needed (export control, Fortezza chip, etc.)

*SSL/TLS has some mechanisms for that, which are used almost entirely in the context of VPNs)
Overview of Solution

- Composed of four sub-protocols:
  - Handshake (regular + session resumption),
  - Alert,
  - ChangeCipherSpec,
  - Heartbeat
- Handshake is the most consuming part
- Its rule is to allow two sides that never met to agree on a shared secret key
- From there, the problem is solved
SSL (or TLS) for the Programmer

Note you must check for errors! (do not use this sample in production code, or you will end in cryptohell!)

InitializeSSL();
socketfd = socket(AF_INET,SOCK_STREAM,0);
bind(socketfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr));
listen(socketfd,5);
newsocketfd = accept(socketfd,(struct sockaddr *) &cli_addr,clilen);
sslctx = SSL_CTX_new(SSLv23_server_method());
SSL_CTX_set_options(sslctx,SSL_OP_SINGLE_DH_USE);
use_cert = SSL_CTX_use_certificate_file(sslctx,"/serverCertificate.pem",SSL_FILETYPE_PEM);
use_prv = SSL_CTX_use_PrivateKey_file(sslctx,"/serverCertificate.pem",SSL_FILETYPE_PEM);
cSSL = SSL_new(sslctx);
SSL_set_fd(cSSL,newsocketfd);
ssl_err = SSL_accept(cSSL);
read(newsocketfd,buffer,nBytesToRead);

Reminder: Check for errors!

Based on:

http://stackoverflow.com/questions/7698488/turn-a-simple-socket-into-an-ssl-socket

Orr Dunkelman

TLS/SSL — (Mis)Protecting our Connections’ Security
A Standard SSL/TLS Handshake

At the end of the execution: both server and client hold a common secret string — master_secret (384-bit long). It depends on the randomness provided by both sides.
Thank you very much for your attention!
Well Known Issues with SSL/TLS

- Since day one SSL/TLS had some known issues:
  - Restricted agility (a list of possible ciphersuites),
  - Backward compatibility (⇒ weaker versions),
  - 40-bit encryption keys in use,
  - Weak public keys (short ones),
  - Relies on servers to turn on encryption (not on by default),
  - “Error” status when secure connection fails not followed by users,
  - Relies on “standard” PKI,
  - Does not fully hide packet size
The Quick (and Dirty) PKI 101

- Public Key Infrastructure (PKI) is the mechanism used to connect an identity with a public key,

- In the context of SSL/TLS, the PKI is certificate-based:
  - Each entity produces a public-key,
  - The entity approaches a certifying authority (CA) and proves its identity to the CA,
  - CA issues a certificate that connects identity and public-key

- The certificate is a statement signed by the CA,

- Signed = one needs to know the CA’s public-key,

- But how do you know the CA’s public-key?

- Either you know the CA in advance, or the CA got a certificate that he is a CA from some other CA...
Implications of Certificate-Based PKI

- Any CA you trust can sign whatever certificate he wishes (mistakes, on purpose),
- Any CA your trusted CA trusts, can generate certificates as he wishes,
- Any CA who has a certificate by a CA which is trusted by your trusted CA, can generate certificates as he wishes,
- ... 
- Happened many times:
  1. Someone obtained a “Microsoft” certificate once (from Verisign),
  2. DigiNotar,
  3. CNNIC
CNNIC — The Chinese are After You

- In 2010 the Chinese CNNIC was added to the list of trusted CAs of Firefox,
- In other words, any Firefox trusts certificates issued by CNNIC,
- Including for gmail. Or bankofamerica.com,
- In other words, a CA can issue certificates “incorrectly”, and these certificates will be accepted by you,
- Allows for Meet in the Middle attacks
DigiNotar — The Iranians are After You

- In 2011, a Dutch CA, DigiNotar was taken over by the Dutch government,
- Apparently, their systems were hacked,
- And their private key was used to sign rouge certificates for several domains (mostly google related),
- These certificates were used to spy on Iranian activists,
- After the forensics, DigiNotar was shut down
Also Cryptographic Attacks

- [S+08]:
  - At CCC08 Stevens et al. reported that they successfully generated a “real-life” certificate using an MD5-collision,
  - As MD5 is a weak hash function, signatures based on it are weak,
  - Also certificates based on it,
  - Including RapidSSL’s certificates . . .

- Flame:
  - Microsoft admitted the Flame was also using rogue certificates,
  - Analysis suggests that again the attack was concerning producing collisions in MD5,
  - The collisions seem to be sub-optimal for a single-CPU attack, but optimized for large scale automation
The Real Issue

- Obviously, joining the CA roots or hacking into a CA invalidates the entire security model.
- However, there are better attack vectors:
  - Users accept all certificates (self-signed, expired, etc.)
  - Users can be easily tricked to not use secure connections.
  - Users . . .
- But also developers are to blame . . .
The Real Issue (cont.)

- Not all applications check certificates.
- In [F+13] it was found out that:
  - Of about 13,500 applications in google play that use SSL/TLS, only 17 implemented the full certificate validation correctly.
- Common errors:
  - Accept all certificates (89%)
  - Only check expiration date (7.5%)
  - Break SSL
Heartbleed

- Heartbeat is a sub-protocol of SSL/TLS to maintain connections alive,
- Roughly speaking, it’s ping with variable datagram size,
- Due to a buffer overflow in OpenSSL, one could obtain the full memory of the other side, including keys,
- Yes, a buffer overflow. Again.
- Though the main culprit is why a ping protocol needs a variable size datagram?
- and yes, a buffer overflow. In 2014.
The Debian Bug — OpenSSL

- OpenSSL is the most common open source cryptographic suite (implements SSL/TLS).
- It handles its own key generation, on top of the /dev/random offered by the system.
- In September 2006, a Debian developer (kroeckx) commented out the following line:

  \[ MD\_Update(&m, buf, j); \]

- (Actually, he commented this line twice).
- The reason: Valgrind complained about using an uninitialized data structure — buf.
The Debian Bug — OpenSSL (cont.)

- One problem — buf contained some “random” leftovers.
- Without it, the only “randomness” the PRNG of OpenSSL was seeded with was the process id.
- One of $2^{15} = 32768$ possible values...
Impact

- If there are only 32,768 seeds, there are at most 32,768 different random sequences that may be produced.
- Even in the key generation phase of OpenSSL (and of OpenSSH).
- Meaning: whoever produced a public key between 2006 and the discovery (2008), used low-entropy keys.
- Which can be factored, reversed (signatures), etc.
- Lots and lots of affected systems. Including small network devices.
In Theory there is no Difference between Theory and Practice . . .

- [H+12] gathered 12.8M TLS public keys and 10.2M SSH public keys.
- Using some quick algorithms (DJB’s algorithm) they found pairs of keys that share prime numbers.
- Such pairs of keys allow using $gcd(\cdot)$ to find the prime numbers themselves (i.e., factorizing the RSA key)
- Which is a bad thing...
### Summary of \([H+12]\) Results

<table>
<thead>
<tr>
<th></th>
<th>TLS</th>
<th>SSH</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total # of Keys</strong></td>
<td>12,828,613</td>
<td>10,216,363</td>
</tr>
<tr>
<td><strong>Repeated Keys (RKs)</strong></td>
<td>7,770,232 (60.5%)</td>
<td>6,642,222 (65.0%)</td>
</tr>
<tr>
<td><strong>Vulnerable RK</strong></td>
<td>714,243 (5.57%)</td>
<td>981,166 (9.6%)</td>
</tr>
<tr>
<td><strong>Default Keys</strong></td>
<td>670,391 (5.23%)</td>
<td></td>
</tr>
<tr>
<td><strong>Low-entropy RK</strong></td>
<td>43,852 (0.34%)</td>
<td></td>
</tr>
<tr>
<td><strong>Factored RSA keys</strong></td>
<td>64,081 (0.5%)</td>
<td>2,459 (0.03%)</td>
</tr>
<tr>
<td><strong>Compromised DSA keys</strong></td>
<td></td>
<td>105,728 (1.03%)</td>
</tr>
<tr>
<td><strong>Debian weak keys (!)</strong></td>
<td>4,147 (0.03%)</td>
<td>53,141 (0.52%)</td>
</tr>
<tr>
<td><strong>512-bit RSA keys</strong></td>
<td>123,038 (0.96%)</td>
<td>8,459 (0.08%)</td>
</tr>
</tbody>
</table>
What Went Wrong? (partial list)

- Sites using default keys. with certificates(!)
- Citrix servers using shared keys (again some with certificates).
- Most repeated keys — ok (used in hosting services). Some — low entropy of the PRNG.
- Many routers, server management cards, VPN devices, VoIP products, and network storage devices suffered from these issues.
Questions?

Thank you very much for your attention!
Let’s Step Back

- Assume we have a server which uses secure ciphers with long keys,
- and that the PKI is tight (including certificate pinning),
- and that encryption is on by default (referring from http:// into https://)
- and that you use a secure implementation of SSL/TLS,
- and the user does not click automatically “yes”,
- and add here whatever security requirement you wish

Are you secure?
Padding Oracle Attacks 101

- Recall 1: Multiple error messages leaks internal secrets ("user does not exist" vs. "user/password combination does not exist"),
- Recall 2: Different timings leaks internal values (comparing strings one character at a time, and stopping when they differ reveals information about the strings),
- Padding Oracle Attacks: Play with padding of cryptographic messages — either cause decryption error (which will be handled fast) or context error (decryption succeeds, but the result is junk that is later discarded by server)
Cipher Block Chaining (CBC) Mode of Operation

- Given a plaintext $P$, pad $P$ to be a multiple of the block size,
- Divide the padded message into $n$-bit blocks $P_1, P_2, \ldots$,
- Set $C_0 = IV$, and encrypt $C_i = E_k(P_i \oplus C_{i-1})$
Decryption Errors in CBC

- Decryption in CBC is performed as follows:

\[ P_i = D_k(C_i) \oplus C_{i-1} \]

- Hence, an error \( \epsilon \) appears in \( C_i \) (i.e., the decrypted value is \( C_i' = C_i \oplus \epsilon \)) there are only **two** message blocks affected:
  - \( P_i' \) which will contain a complete nonsense,
  - \( P_{i+1}' \) which will be \( P_{i+1} \oplus \epsilon \),

- This allows “selecting” the change in \( P_{i+1}' \) (at the cost of a complete non-sense in \( P_i' \)
CBC Padding in SSL 3.0

- Add 1 to $L$ bytes ($L$ being the block size in bytes),
- All but last byte can have whatever value needed.
- Last byte contains encoding of the number of added bytes.
- Padding not authenticated by the MAC used in SSL
Basic Padding Oracle Attack against CBC with SSL 3.0 Padding

Client: $C = AESCBC_k(M)$

Man in the Middle:

- $C'$
- $C''$
- Alert
- Alert
- ...

Server:

1. With the SSL padding rule,
2. $C$ is obtained by the MitM,
3. Adversary parse $C$ as $C_1, C_2, \ldots, C_\ell$. 

Orr Dunkelman
The Magic

- If the decryption of $C'$ is successful, then the padding rule was checked to be correct,
- Namely $P'_{\ell-1}$ was decrypted such that the last byte contains a number smaller than $L$,
- In other words

$$P'_{\ell-1} = D_k(C_{\ell-1}) \oplus C'_{\ell-2}$$
$$= P_{\ell-1} \oplus C_{\ell-2} \oplus C'_{\ell-2}$$
$$= P_{\ell-1} \oplus \epsilon$$

- One can play with the value $\epsilon$ (in the last byte of $C'_{\ell-2}$) to learn something on the last byte of $P_{\ell-1}$.
- Attack can also be used to learn other bytes in the ciphertext...
The POODLE Attack

- The POODLE Attack is a padding oracle attack against SSL 3.0 [MDK14],
- It is composed of three parts:
  - Downgrading the connection from TLS 1.x into SSL 3.0,
  - Applying a padding oracle attack,
  - on cookies (similar to BEAST we’ll describe next)
The POODLE Attack (cont.)

- Downgrading a connection is not so hard,
- Just interfere as a man in the middle in the handshake,
- Most SSL/TLS implementations will try the strongest possible version,
- Then, they will slowly “step down” the security into legacy,
- When secure connection is finally established, the downgraded protocol (unlike downgraded ciphersuites) is not detected by cryptographic means.
The POODLE Attack (cont.)

- SSL/TLS applies MAC to the payload,
- Despite the right order being Encrypt-then-MAC, SSL/TLS uses MAC-then-encrypt:

```
SQN||HDR Payload
```

```
MAC
Payload tag Pad
```

Based on Kenny Paterson’s presentation
The BEAST Attack

- In SSL 3.0/TLS 1.0, the IV is chosen to be the last message’s last ciphertext block,
- This saves the need to communicate the IV, and generates security issues,
- The key idea — one can detect whether a plaintext block has some value or not,
- The attack [DR11] heavily relies on the fact that the IV is known in advance

†First observed as a theoretical issue by Rogaway in 1995, suggested to be used against TLS in 2004 [DM04], extended by [B06] to message recovery.
How to Recover Text in BEAST

- Assume you suspect that the first ciphertext block $C$ encrypts the value $P$.
- Ask for the encryption of $(P, P \oplus IV \oplus C)$.
- If $C$ is the encryption of $P$, namely, $C = E_k(P \oplus IV)$, then the two ciphertext blocks will be the same(!)
- If you know most of $P$, you can just try the remaining few bits (by exhaustive search),
- Allows for a byte-by-byte (or bit-by-bit) recovery of plaintexts
Obtaining Chosen Plaintext Capabilities in SSL/TLS

- The adversary needs to run a JavaScript on your machine (e.g., for evil.com),
- The adversary tries to decrypt a cookie you are using for the site target.com,
- Assumes the adversary sees the communication to evil.com (which is feasible),
- And the JavaScript needs to communicate with evil.com online
Obtaining Chosen Plaintext Capabilities in SSL/TLS (cont.)

- Evil.com JS code is being run,
- The user sends an encrypted cookie to target.com,
- Adversary knows the first few bytes (e.g., “username = XXXX”),
- Adversary captures the ciphertext of the cookies, and sends to the JS,
- JS code uses the above attack to find an unknown byte of the first ciphertext block,
- Adversary tells the evil.com website when the two ciphertext blocks are the same, which means the JS can move to the next byte,
- This can be done by shifting the ciphertext a bit,

Mitigation: Switch to TLS 1.1/1.2 (IV is randomized)
The Lucky13 Attack

- Padding in CBC for TLS has changed,
- The padding is either one byte of value $00_x$, two bytes of value $01\ 01_x$, three bytes of value $02\ 02\ 02_x$, etc.
- Oh — and you can pad more than a single block, up to 256 bytes of the form $FF\ FF\ \ldots\ FF_x$,
- Allows for a full padding oracle attack that recovers the full plaintext,
The Lucky13 Attack (cont.)

- Main issue: for the attack to work, adversary needs to identify what error occurred,
- Fortunately, there are different error message (padding failed or MAC failed),
- Unfortunately, these error messages are encrypted (not accessible to the adversary),
- But different timing,
- So manual suggests that both errors will take exactly the same time:
  - Some implementations compute MAC even if padding fails,
  - Some implementations just remove the “suggested” number of bytes, and running the MAC,
- The last option still allows for an attack
The Lucky13 Attack (cont.)

- The MAC in use is usually HMAC, usually based on SHA256,
- The performance depends on the number of bytes in the message,
- For messages up to 55 bytes, it uses 4 compression function calls,
- For messages of 56 to 119 bytes, it uses 5 compression function calls,
- For messages of 120 to 183 bytes, it uses 6 compression function calls,
- ... 
- We can learn how many bytes were removed from the message! And recover plaintext,
- Becomes practical combined with BEAST
The CRIME Attack

- SSL/TLS offers compression of the plaintext (better performance),
- But compression depends on the actual data (and is stateful),
- For example, high redundancy plaintexts have better compression ratios,
- BEAST + picking the correct padding games ⇒ message recovery! [DR12],
- Look out for BREACH as well (as HTTP also has compression mechanism)
The Legacy of SSL (Export Restrictions)

A long time ago in a country far, far away. It was a period of liberty unrest. Liberal cryptographers, striking from their universities and companies, have won their first victory against the export restrictions. During the battle, these cryptographers came to realize the empire’s ultimate weapon, the legal case for export restrictions, a set of restrictions with enough power to make any protected communications accessible to the NSA. As a result, SSL has several export-level primitives available, from 40-bit keys to short public-keys.
FREAK — Short RSA Keys

- Long time ago, only 512-bit RSA keys were allowed,
- Today, a simple Number Field Sieve can factor such an RSA moduli at the cost of 100 US$ on a cloud service provider,
- So, [B15] just showed that it is easy to attack such servers...
- Mitigations: Shut down SSL 3.0; Disable legacy crypto; Use larger primes.
Logjam — Short Diffie-Hellman Keys

- SSL 3.0 allows using “export-level” Diffie-Hellman keys,†
- Today, such keys are insecure. More precisely, finding $x$ from $g^x \mod p$ for 512-bit prime $p$ is feasible,
- If $p$ is known in advance, the process can be done in practical time (less than 2.5 minutes),
- Identified by Adrian et al. [A+15] logjam is a cryptographic attack on such keys,
- Improved success due to the fact that two common primes (one by Apache, and one by mod_ssl), “cover” 92% of the “market share”
- Mitigations: Shut down SSL 3.0; Disable legacy crypto.

†About 8% of the top 1M internet sites support that.
Summary

- SSL/TLS is one of the greater successes in computer security.\(^\S\)
- Widely deployed and used,
- Crypto agility which is easy to use,
- Multiple implementations to pick from,
- YOU MUST USE IT whenever possible.

\(^\S\) Conditions apply
Summary (cont.)

- Just remember to turn off legacy options, both on server side and client side,
- Fix PKI (or be very very careful with it),
- Turn off compression (both for TLS and for HTTP),
- Switch to TLS 1.2, and the moment TLS 1.3 is out the door, switch to it,
- Set your IDSes to detect renegotiation and downgrade attacks,
- Use very long (and secure) public-keys,
- Does not protect against traffic analysis (message sizes usually not protected),
- BETTER TO NOT USE IT unless you must.
Thank you very much for your attention!