Cryptographic Engineering

working draft

Matthew D. Green
Johns Hopkins University

Excerpt: Chapter 9, Cryptographic protocols

§9.2.1 Flaws in TLS 1.2 and earlier (book pages 187–192)

Posted for 601.445/601.645 Practical Cryptographic Systems, Fall 2026
Lectures 15 and 18 (Oct 21, Nov 2): TLS attack history; oracle and downgrade attacks

`https://practicalcrypto.org`

This is an excerpt from an unfinished book draft: expect gaps, rough edges, and errors, and note that section numbers and page numbers may change in later drafts. Cross-references to other chapters point to material that is not included here. Copyright © 2026 Matthew D. Green. All rights reserved. Posted for the students of the course above; **please do not redistribute** — link to the course site instead. Built 2026-10-19.

such as a length and a type. The client and server internally record a *sequence number* for sent/received records. This is used to construct a *nonce* and the record is then encrypted using an authenticated encryption (AEAD) scheme (§5.3.2) under the appropriate key and nonce. This provides confidentiality and also ensures that any in-flight tampering will cause a decryption error: in TLS, such errors result in the termination of the TLS connection. Each ciphersuite specifies the maximum number of records or bytes that can be encrypted with a single key. When this limit is encountered, the connection must terminate or re-key with a new handshake. By default, the TLS v1.3 standard supports AES-GCM (in 128 and 256-bit key sizes) and ChaCha20-Poly1305 for symmetric encryption. In principle, other ciphers can be added as new ciphersuites.

Flaws in TLS 1.2 and earlier

The development of TLS 1.3 was a major effort that fixed many problems that plagued earlier versions of TLS and SSL. Hopefully by the time you read this, TLS 1.3 (or a successor) will be the only version of TLS on the Internet! Unfortunately, older versions of TLS/SSL have a tendency to linger in legacy devices, which means that clients such as web browsers are also likely to retain support for them. For this reason it is worth knowing a bit about past weaknesses in these protocols. Moreover, learning how protocols *break* is often the best way to learn how to build secure ones.

DRAFT

Attacks on the RSA key exchange. For various reasons, the TLS protocol (prior to 1.3) defaulted to an RSA-based key exchange protocol as its default mode for establishing keys. In this mode, clients would first generate a secret value (called a *pre-master secret* or PMS) and then encrypt the PMS under the server's RSA public key. The server would decrypt the ciphertext and use a key derivation function to combine the PMS with the "client random" and "server random" nonces, producing the protocol's *master secret*.

An obvious problem with the RSA key exchange is that it provides no forward secrecy: if a long-term RSA secret key is stolen from a server at any point, then an eavesdropper attacker can decrypt any past (or future) TLS connections. The only real fix for this is to switch away from the RSA key exchange in favor of forward-secure protocols like ephemeral (EC) Diffie-Hellman. Indeed, TLS v1.3 has done exactly this.

Bleichenbacher attacks. As bad as the RSA key exchange is in principle, it's even worse in practice. When SSLv2 was designed, the recommended padding scheme for RSA was something called PKCS #1v1.5 (§7.2.1): this scheme pads each each RSA plaintext with a short header, some random bytes, and delimiters consisting of a single "0" byte (see Figure 9.6.) When a ciphertext is decrypted by the server, older TLS/SSL servers would check the structure of the decrypted padding, and emit a *padding error* if the padding did not check out properly.

Way back in 1998, Daniel Bleichenbacher discovered that this scheme is vulnerable to an adaptive chosen ciphertext attack. If an attacker intercepts a TLS connection containing



Figure 9.6: RSA PKCS #1v1.5 padding, as used in TLS v1.2 and earlier. The full padded plaintext is as long as the RSA modulus N.

an RSA ciphertext, it can repeatedly "maul" the ciphertext by multiplying it with chosen values. The attacker can then use the original server as an "oracle": it sends each mauled ciphertext to the server within a new TLS connection request. If the server decrypts the ciphertext and indicates a padding error, the attacker learns a small amount of information about the original plaintext. By repeating this process thousands or even millions of times, the attacker can gradually decrypt the original ciphertext (see §7.3.4 for more details.)

The fix for this issue is to stop using PKCS #1v1.5! Naturally legacy compatibility reasons made this impossible, at least before TLS v1.3. Hence most pre-1.3 TLS implementations fix the problem via a series of "band-aids": servers must carefully disguise the fact that a padding error has occurred, by internally generating a random string in place of the decrypted PMS and proceeding through the remainder of the handshake process until a different error occurs. This mostly works, but occasionally a researcher will find a new way to exploit the thing, and some new band-aids have to be added to the pile [74, 285].

Attacks on non-AEAD symmetric encryption. A pre-historic aspect of older TLS/SSL versions is that they do not mandate modern authenticated encryption. While TLS v1.3 requires secure AEAD ciphersuites for encryption, older versions of TLS support CBC mode encryption and unauthenticated stream ciphers such as RC4. In this approach, each record is (1) appended with a MAC, then (2) padded to a multiple of the cipher's block size (or byte, in the case of RC4), then (3) the full record, MAC and padding are encrypted using one of these two ciphers.

Padding oracle attacks. The symmetric encryption construction used in TLS was found to be vulnerable to a *padding oracle attack*, discovered by Serge Vaudenay back in 2002 [317]. In these attacks, an attacker intercepts a CBC-encrypted ciphertext that it wishes to decrypt, then "mauls" the ciphertext by flipping some bits. Finally, it sends the modified ciphertext to a server, which decrypts it and only reveals if the padding is valid or invalid. Using this single bit of information, a clever attacker can gradually decrypt the plaintext by repeating the attack dozens or hundreds of times. See §5.3.1 for a more detailed description of the attack.

As with the broken RSA encryption padding, the correct solution is to rip out this legacy encryption mode and replace it with a proper authenticated encryption mode: this is precisely what TLS 1.3 has done. Unfortunately, this proved challenging in earlier TLS/SSL versions (for legacy support reasons), and hence the "fix" was for the server to hide the existence of padding-check failures from any client. The result was a game of cat-and-mouse,

DRAFT

where researchers would periodically find a new way to exploit the flaw, and defenders would write increasingly byzantine code to stop them [11, 174, 197].

Predictable IVs for CBC mode. A second problem with TLS's CBC mode encryption, at least up through TLS 1.0, is related to the way that Initialization Vectors (IVs) are chosen. In TLS v1.1 and later, these IVs are generated at random, and the generated IVs are stored within the encrypted records. To save bytes, TLS 1.0 and SSL use an "optimization": the IV for record N is set equal to the final block of the ciphertext for record $N - 1$.[24]

This small flaw exposes TLS 1.0 to *blockwise chosen plaintext attacks* [30]. Unlike the padding oracle attack, these attacks require that the attacker can convince a TLS client (or server) to encrypt messages of its choosing. Fortunately for attackers, this is easy to do in the browser ecosystem: an eavesdropper attacker simply needs to convince a target's browser to run some Javascript code of its choosing. The Javascript code can then initiate connections to a third-party website, and the attacker can intercept the resulting ciphertexts.

The best example of this vulnerability was captured by the BEAST attack [**?**]. Here, the attacker's goal is to obtain confidential session cookies for some website, *e.g.,* Google. To do this, they inject some Javascript code that crafts custom requests to google.com: each HTTPS request contains some attacker-chosen data, which will be followed by a confidential session cookie that the attacker code does not know. If the attacker can predict the IV that the browser will use for encryption, they can craft an attack that systematically decrypts the secret cookie, one byte at a time. The full attack is described in §5.4.5.

RC4 biases. Prior to TLS v1.3, the most popular stream cipher in TLS was the RC4 cipher (§3.1.1.) This was largely because RC4 was extremely efficient to implement in software. RC4 was also proposed as one "fix" for the BEAST attack, since RC4 does not suffer from IV vulnerabilities like CBC mode encryption does.

Unfortunately, RC4 has its own set of vulnerabilities that make it inappropriate for use in TLS. Critically, the pseudorandom keystream produced by RC4 contains small *biases*: specific byte positions in which the output stream tends to be slightly non-uniform [13]. If the same plaintext is encrypted many times with different RC4 keystreams, these biases can be used to decrypt most bytes of the plaintext. This attack can be leveraged to obtain the value of fixed secrets such as passwords and session cookies, albeit only after seeing several million encryptions of the same plaintext.

Compression attacks. Up until 2012, most implementations of TLS v1.2 (and earlier) used the default *compression* option as part of the TLS standard. This option enabled native DEFLATE compression for TLS-encrypted data, reducing the bandwidth of TLS sessions prior to encrypting them.

It has long been known that the use of compression (prior to encryption) can create information leaks that reveal information about plaintext data [188]. For example, the size of a compressed file can leak information on the compression ratio, which in turn can reveal

[24] The first IV is generated as part of the handshake process.

DRAFT

information about the nature of the file's contents. However, this does not immediately lead to a direct attack that enables the decryption of TLS-encrypted data.

In 2011, Adam Langley proposed the following attack scenario [196]: imagine that a TLS plaintext string contains two portions of data concatenated together: (1) a string chosen by an attacker (such as Javascript code running in a victim's browser), and (2) a secret value, like a session cookie. Langley noted that algorithms such as DEFLATE [102] are designed to detect and eliminate repeated strings. Hence, if the inputs (1) and (2) contain common substrings, then the size of the concatenated data will tend to be smaller than cases where they *don't* contain common substrings. An attacker who could see the size of the resulting ciphertexts would therefore be able to conduct experiments that would reveal the value of string (2). Imagine that string (2) contains the value `cookie: twid=<secret>`. where `<secret>` is some string of bytes. Then if the attacker submits the string `cookie: twid=A`, DEFLATE will exactly match the substrings iff `<secret>` begins with the character `A`; however, it will only partially match if `<secret>` begins with a different character, resulting in a detectable difference in the size of the compressed string between the two experiments. By repeating this experiment with many chosen substrings, the attacker can gradually learn the secret value. This attack was implemented as the CRIME attack by Julian Rizzo and Thai Duong [140].

Export-grade rollbacks. Back in the 1990s the U.S. government maintained a two-tier system for cryptography. Strong encryption was export-controlled, which means that it could not be sold to people outside of the United States. This absurd arrangement meant that some web browsers would only support *export ciphersuites* that used absurdly short keys, such as 512-bit RSA or 40-bit symmetric keys. Fortunately for the Internet, these laws were loosened in 2000 [313]. However export-grade ciphersuites continued to be widely-supported well into the 2010s: by 2015, as many as 37% of measurable TLS servers were still offering them as an option [115].

In a perfect world, server support for export-grade ciphersuites should not have been a problem. Unfortunately, until 2015, many implementations of TLS 1.2 and earlier were vulnerable to *ciphersuite rollback attacks*: an MITM attacker could tamper with negotiation messages, causing the connection to "downgrade" to an export-grade cipher that neither side would use under ordinary conditions. The attacker could then silently decrypt or even modify TLS data.

FREAK [61]. The FREAK attack exploited a vulnerability in TLS software implementations, not the protocol itself. This is cold comfort, however, since the flaw existed in nearly every major TLS software stack. The conditions for the attack required that a client would request the (standard) RSA handshake, and that the server would also support the weakened "export RSA" variant. In this case, an MITM attacker could simply replace the client's list of requested ciphersuites with a single option requesting the weakened export variant. In principle, the client should have immediately detected this attack: the server's response message explicitly notes that it is using export-RSA. In practice, most TLS/SSL

DRAFT

implementations became confused and went forward with a 512-bit RSA server public key. Even worse, since most servers re-used the same export RSA key for long periods of time, an attacker could factor this short key at its leisure. With knowledge of the secret key, it could decrypt any RSA key exchange and "fix up" the TLS Finished message in real time.[25]

Logjam. The Logjam attack is similar to FREAK, but does in fact rely on a flaw in the protocol itself. In TLS v1.2 and earlier, the parameters for the ephemeral Diffie-Hellman (DHE) key exchange (the values p, g, g^a) are supplied by the server. In normal Diffie-Hellman, the modulus p should be 1024 bits or higher. In "export Diffie-Hellman", it is reduced to a much weaker 512 bits. In the Logjam attack, an MITM attacker re-writes the Client Hello to indicate that export-DH is desired. If the server supports export-DH, it will respond with a weak 512-bit Diffie-Hellman key. The server's key share is signed: however, nothing in the signed message indicates which ciphersuite the server has selected. The client thus knows it has received an unusually short Diffie-Hellman key, but does not realize this is the result of a downgrade attack.[26] Even if the client does not support export-grade Diffie-Hellman, therefore, it can be downgraded to use weak keys.

Once the attacker has succeeded in reducing the strength of the handshake to 512 bits, it compute a discrete logarithm of the server's weak Diffie-Hellman share and completely break the security of the handshake, including forging TLS Finished messages. The short-term fix for this attack was for clients to simply reject all short Diffie-Hellman keys, something that proved slightly more challenging than it sounds![27] The long-term repair was to sign the entire handshake (including ciphersuite choices), as is done in TLS v1.3.

Weak Diffie-Hellman moduli. Exploiting Logjam requires the attacker to compute a discrete logarithm on a 512-bit Diffie-Hellman key $g^a \ mod \ p$. In total, this process required up to a week of core-time in 2015, which seems to make real-time attacks infeasible. However, there is a shortcut: if the modulus p is re-used for many keys, then the vast bulk of the work can be done in a pre-computation phase that depends only on p! Only the final phase (called the "descent") depends on the specific g^a, and this can be done in as little as 30 seconds [8].

This raised interesting questions about whether such an attack could be launched against non-export Diffie-Hellman. A feature of TLS servers in the pre-v1.3 period was that most TLS (and IPsec) implementations used 1024-bit Diffie-Hellman as the standard for non-export security. Worse, servers routinely selected their prime moduli from a small list,

[25] Earlier versions of TLS did not include a digital signature on the transcript, in the manner of TLS 1.3's Client Verify message. This meant that any tampering with the handshake messages could only be detected after the key exchange completed. If the key exchange becomes insecure as a result of tampering, then even this protection does not work anymore.

[26] This is not the first downgrade attack on this mechanism to be discovered. In much earlier work [320], Wagner and Schneier proposed to downgrade Diffie-Hellman to an ephemeral RSA mode, with similar consequences.

[27] At the time, most client implementations of Diffie-Hellman would happily accept even very small Diffie-Hellman keys, in some cases as short as one byte. While this was just stupid, the second problem was that many clients actually needed to support Diffie-Hellman with 512-bit key sizes in order to connect to ancient legacy devices. Fixing Logjam made a significant number of these older devices unusable.

DRAFT

```
struct {
    opaque dh_p<1..2^16-1>;
    opaque dh_g<1..2^16-1>;
    opaque dh_Ys<1..2^16-1>;
} ServerDHParams;     /* Ephemeral DH parameters */

dh_p
   The prime modulus used for the Diffie-Hellman operation.

dh_g
   The generator used for the Diffie-Hellman operation.

dh_Ys
   The server's Diffie-Hellman public value (g^X mod p).
```

Figure 9.7: The data structure used for sending Diffie-Hellman parameters from server to client in TLS v1.2. This structure is signed, but the signature is not shown. Note that the structure does not include the ciphersuite identifier. Source: RFC 5246 [**?**].

meaning that huge portions of the Internet used common moduli, and hence could *theoretically* be vulnerable to a scaled-up pre-computation attack. Based on some heuristic estimates, my colleagues calculated that in 2015 such an attack would require approximately 45 million core years of pre-computation (for a single prime p) and only 30-core days of calculation for a specific key.

Now, it must be noted that 45 million core years is a huge number! However, with hardware optimizations, research estimates note that such a procedure could be achieved within one year, at a cost of hundreds of million dollars. This would put such attacks within the resource budget of an organization such as the NSA. And indeed, several slides from the Snowden files strongly indicate that the NSA has some unexplained cryptanalytic capability that would be largely explained if they had conducted this attack [8]. We will likely never know whether these concerns are absurd or well-founded; nonetheless, the TLS community decided to err on the side of safety. TLS 1.3 bans 1024-bit Diffie-Hellman and forces users to select from a small set of much safer Diffie-Hellman groups.

DRAFT

9.2.2 Wireguard

Sometime around 2016, a researcher named Jason Donenfeld called to ask my advice about Wireguard, a new network encryption protocol he had developed. I was busy and skeptical, and gave him some version of the advice I gave earlier in this chapter: *are you sure the world needs another encryption protocol?* As a compromise, I suggested he send his ideas to a top network security conference, where I felt the hard work of checking his work could be outsourced to Reviewer B.

Fortunately for the world, I was wrong: the world did, in fact, need a new encryption protocol. Jason's paper was published at NDSS 2017 [111], and Wireguard has subsequently become one of the most popular and carefully-reviewed protocols for implementing virtual private networking tunnels (VPNs) across many platforms (notably it is included in the Linux kernel from v5.6 onward.) Although Wireguard is not the only protocol to serve this purpose — alternatives such as IPsec and OpenVPN's TLS-based approach are available —