Encryption in Practice

6.1600 Course Staff

Fall 2024

So far, we have established several constructions that allow us to hide the contents of transmissions: we created chosen-plaintext- and chosen-ciphertext-secure encryption schemes that worked with and without a shared key. We will now discuss a few practical applications of transport encryption, and why it is often difficult to get right.

Disclaimer: This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

1 *File Encryption*

Perhaps the most straightforward use of encryption is file encryption.

Example: WhatsApp Encrypted Backup. WhatsApp allows the app's users to back up their messages and contacts to the cloud. This way, a user can recover her messages if she loses or breaks her phone. To hide the user's data from WhatsApp's cloud servers, WhatsApp uses encrypted backup. To achieve this, the user's device generates a 128-bit AES key k at the time of backup and encrypt the message data (photos, messages, etc.) using AES-GCM$(k, \cdot)$ before sending the *ciphertext* to the server. In order to allow you to restore your backup on a new phone, the app allows you to export 64 decimal digits that encode the AES key used. When restoring your backup, you will enter these digits and your phone will fetch the ciphertext from the server and use these digits to reconstruct the key and decrypt your messages.

See **this document** for details on how WhatsApp encrypts backups. (The document also describes a more complicated backup scheme that uses passwords for encryption.)

This is a fairly simple application of file security. However, file encryption can be much more tricky: many applications require or desire features beyond simple encryption and decryption.

1.1 *Case Study: PDF v1.5 Encryption*

One instance of this desire for extra features that ended up going wrong was a previous version of the PDF standard, PDF v1.5.[1] This standard provided several features:

1. It is possible to password-encrypt some or all of the document. (The encryption scheme does not matter, but think of it as a secure authenticated-encryption scheme.) For example, a PDF could have an unencrypted title page and have the rest of the pages be encrypted.

[1] Jens Müller et al. "Practical decryption exfiltration: Breaking pdf encryption". In: *ACM CCS*. 2019.

2. A PDF document can contain a form that the PDF reader submits to a server via HTTP.
3. A form in a PDF document can reference other parts of a document.
4. The PDF reader may submit a form when an event happens: when the PDF is opened, closed, decrypted, etc.

Each of these features individually seems innocuous. However, when combined, there is a clever attack that allows an attacker to learn the contents of the encrypted portion of a PDF.

The attack works against a PDF document with an *unecrypted (public) title page* and an *encrypted (secret) body*. We assume that the attacker can modify the PDF file on its way to the victim.

To mount the attacker, the attacker intercepts the PDF on the way to the victim and replaces the title page with an "evil" title page. The evil title page:

- contains an invisible PDF form,
- reads the contents of the decrypted pages into a PDF form element, on the event that decryption of the PDF body succeeds, and then
- submits the form via HTTP to `evil.com`

So when the victim recipient enters her password into the PDF reader to decrypt the document, the evil title page is exfiltrates the decrypted content to `evil.com` via a PDF form.

What went wrong? The core issue here was that the unencrypted contents of the document were not authenticated. That is, an attacker could modify the unencrypted pages of the document without detection. A better design would have been to either use a MAC over all pages of the document or to use a primitive called "Authenticated Encryption with Associated Data" to authenticate the encrypted data together with the unencrypted data.

One downside of MACing the entire document is that, before rendering even a single page of the PDF, the reader would have to compute a MAC over the entire PDF. If the PDF consists of many thousands of pages, this could be expensive. Using a more sophisticated cryptographic construction: per-page MACs, plus MACs on the document metadata, etc., we might be able to construct a scheme that protects document integrity while allowing partial document rendering. But the design of such a scheme would have to carefully defend document integrity against the sort of attacks we describe here.

2 *Stream Encryption: Transport Layer Security (TLS)*

The standard Internet data transfer protocols—TCP and UDP—provide no integrity or confidentiality whatsoever. To protect the data that we send over the network, we use the *transport-layer security* (TLS) protocol. The TLS protocol runs on top of TCP and aims to provide an encrypted "tunnel" between a client and server.

HTTPS is simply HTTP run over TLS.

While designing a stream-encryption protocol may seem straightforward at first glance, the task is much more subtle than it would seem. However, as is often the case in security, features and practical requirements make the situation much more complex.

2.1 *Downgrade Attack*

We now give one example of a *downgrade attack*—the sort of subtle security issue that can arise in protocol design.

The current version of TLS is TLS 1.3. An older version of the TLS standard, called SSLv3, is vulnerable to devastating attacks that allow an attacker to recover the plaintext traffic. However, for backwards compatibility, many TLS clients and servers still supported SSLv3 until quite recently.

When a TLS client connects to a TLS server, the two parties first exchange some messages to decide which version of the TLS protocol to use. In particular, the client will try to connect to the server using the latest version of TLS it supports. The server will respond back with either a confirmation (if the server supports the client's proposed TLS version) or with garbage (if not). If garbage, the client will try to connect to the server with an older version of TLS.

An important point is that *none* of these negotiation messages are authenticated—the client and server cannot start using authentication (MACs, signatures, etc.) until they agree on which version of TLS to use! So, an active in-network attacker can simply replace all of the server responses in this protocol-negotiation phase with garbage until the client downgrades all the way to SSLv3. Once the client and server agree to use SSLv3, believing that this is their best available option, the attacker can then monitor and decrypt their traffic using known attacks on SSLv3.

The best defense against this attack is for both parties to disable support for SSLv3 completely.

2.2 *TLS Structure*

TLS consists of two main phases:

1. **Handshake:** In this phase, the client and server use a key-exchange protocol to agree on a shared key to use to encrypt

their application-layer traffic. This step uses public-key cryptography, since the client and server initially have no shared secret.
2. **Record protocol:** This phase is where the actual application-layer communication happens. This phases uses the secret key that the client and server agreed upon in the handshake phase for authentication and encryption.

2.3 TLS Handshake Properties

In our definitions of public-key encryption, we had only two (relatively simple) properties: correctness and security. The TLS handshake, however, has a much more complicated set of goals:

- **Correctness:** Both parties agree on the same session key at the conclusion of the handshake.
- **Security:** adversary "learns nothing" about the secret key that the parties agree upon at the conclusion of the handshake.
- **Peer authentication:** At the end of the handshake, each party believes that they are talking to the other party.
- **Downgrade protection:** The parties agree on the same version of TLS that they would agree on absent an in-network attacker.
- **Forward secrecy with respect to key compromise:** if an attacker steals the secrets stored on the client or the server, the attacker cannot decrypt past traffic.
- **Protection against key-compromise impersonation:** If an attacker steals a client's secret key, the attacker should not be able to impersonate other servers to the client.
- **Protection of endpoint identities:** The public keys of the two parties should never flow over the wire in the clear. For example, if a client is connecting to a website that uses a content-distribution network, such as Akamai or Cloudflare, an attacker should not be able to tell which website the client is connecting to—only that it is hosted on Akamai or Cloudflare.

To provide forward secrecy, modern cryptographic protocols use long-term secrets only for signing—not for encryption. These protocols use *ephemeral* (one-time-use) cryptographic keys for key exchange and encryption. Protocol participants delete these ephemeral keys on connection teardown and/or they rotate these keys often.

This way, if an attacker compromises a party's secret key, the attacker can only sign messages on behalf of that party; the attacker cannot use the secret to decrypt past messages.

2.4 TLS Handshake

The TLS handshake is very carefully designed to achieve these properties. A grossly simplified version looks something like the following:

1. At the start of the handshake, the TLS client holds the public pk_{CA} of a certificate authority. The TLS server (for example, MIT) holds its secret signing key sk_{MIT} and a public-key certificate cert_{MIT} binding its public key pk_{MIT} to its domain `mit.edu`.

2. **Client Hello**: The client sends the following values to the server:
   - a random nonce,
   - list of supported ciphersuites, and
   - an ephemeral Diffie-Hellman public key. (The client constructs this Diffie-Hellman public key using its preferred ciphersuite. If the server does not support the ciphersuite the client picked, the client will have to re-run this step using a different ciphersuite.)
3. **Server Hello**: The server sends several values to the client, choosing a ciphersuite to use and completing the Diffie-Hellman key exchange. That is, the server sends:
   - a random nonce,
   - a ciphersuite to use,
   - and an ephemeral Diffie-Hellman public key
4. Both partices compute a shared session key k using Diffie-Hellman key agreement on the ephemeral keys they exchanged.
5. Under encryption using the keys derived from the session key k, the server sends the certificate for `mit.edu` as well as a signature over all messages sent so far, using its long-term secret key $\mathsf{sk}_{\mathrm{MIT}}$. The client then checks that:
   - the certificate has been signed by one of the client's trusted CAs, and
   - the signature from the server matches their own record of the messages.
6. Finally, the client and server run the TLS Record Protocol to exchange encrypted and authenticated application data.

A *ciphersuite* contains all of the cryptographic parameters needed to perform key exchange, hashing, authentication, and encryption. For example, one possible ciphersuite for TLS 1.2 is `ECDHE-RSA-AES256-GCM-SHA384`. This indicates use of ephemeral elliptic-curve Diffie-Hellman key exchange (`DHE`) with RSA signatures (`RSA`), 256-bit AES encryption in GCM mode (`AES256-GCM`), and SHA2-384 as the hash function (`SHA384`).

This simplified toy version of the TLS handshake does not provide many of the features that the real TLS handshake provides. But it should give you a flavor of what the real handshake looks like.

3 *Properties that TLS does not provide*

Authenticated End-of-File TLS does not provide any end-of-file authentication, or "clean closure." To explain what this means by example:

A popular tool to install the toolchain for the trendy systems programming language Rust is `rustup`. To use the tool and install the Rust toolchain, the recommanded method is to run the command "`curl https://sh.rustup.rs | sh`." This downloads a shell script from the internet over HTTPS and immediately runs it using the shell `sh`.

Imagine that the contents of the downloaded script create a tem-

porary directory, copy things into it, install some things, and finally delete the temporary directory with something like `rm -r /tmp/install`. An in-network attacker, who knows the structure of the `rustup` install script, could drop all of the packets in the stream immediately after the characters `rm -r /`. Eventually, the TLS connection will timeout, an shell will run the command `rm -r /`, deleting the user's entire file system.

To protect against this, script writers try to design their scripts such that if the stream is cut off in the middle of a download, nothing happens. For example, the install script might consist of a single function definition that is called at the very end of the function.

Plaintext Length Obfuscation As we have discussed, encryption reveals exactly the length of the plaintext. If there is data that is not encrypted that is then included inside the encrypted data as well, this can cause a vulnerability—see the CRIME attack.

References

Müller, Jens et al. "Practical decryption exfiltration: Breaking pdf encryption". In: *ACM CCS*. 2019.