BTI 4202: From Symmetric Encryption to Secure Channels

Christian Grothoff

Berner Fachhochschule

30.4.2021
Learning Objectives

Example: Attack on CBC Stateful IV

Beyond IND-CPA

Real-world use of cryptographic primitives (exercise)

Symmetric key establishment protocols

Secure channels
Part I: Attack on CBC
Goal: confirm “Kimberly” was sent!
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.
The CTR random IV symmetric encryption scheme is IND-CPA secure.

The CTR stateful IV encryption scheme (ensuring no IV re-use) is IND-CPA secure.

The CBC random IV symmetric encryption scheme is IND-CPA secure.
A pseudo random function (PRF) is a function that is (computationally) indistinguishable from a true random function.

The previous positive results are true under the assumption that the block cipher used (e.g. AES) is a PRF.

Assumption really means that this is a commonly shared belief of the crypto community. No proof exists!

Breaking any of these schemes thus means breaking the PRF property of the underlying block cipher.

The crucial security property of a secure block cipher is that it is a PRF!
Part II: Chosen Ciphertext Attacks
IND-CPA vs. Chosen Ciphertext

IND-CPA is **not** the strongest security model!

- The adversary does not have access to a *decryption* oracle
- With a decryption oracle, an adversary can be allowed to ask for *some* messages of its choice to be decrypted.
- Security is achieved only if *other* messages still remain indistinguishable.
Indistinguishability under Chosen Ciphertext Attacks (IND-CCA)

The adversary’s goal is the same as in IND-CPA (determine \( b \) given \( \text{enc}(k, M^i_b) \)) for sequences of messages \( M^i_{0,1} \).

**Setup** Generate random key \( k \), select \( b \in \{0, 1\} \).

**Oracle E** Given \( M \), return \( C := \text{enc}(k, M) \).

**Oracle D** Given \( C' \), return \( M := \text{dec}(k, C') \).

The additional restriction \( C' \neq C \) must be imposed on the use of **Oracle D**: The adversary is not allowed to ask for decryption of a ciphertext \( C \) that was previously returned by the encryption oracle.
Examples for IND-CCA Insecure Schemes

CTR schemes are IND-CCA insecure:

“Say \( \langle r, C \rangle \) is a ciphertext of some \( l \)-bit message \( M \), and we flip bit \( i \) of \( C \), resulting in a new ciphertext \( \langle r, C' \rangle \). Let \( M' \) be the message obtained by decrypting the new ciphertext. Then \( M' \) equals \( M \) with the \( i \)-th bit flipped. Thus, by making a decryption oracle query of \( \langle r, C' \rangle \) one can learn \( M' \) and thus \( M \).”

–Symmetric Encryption by Mihir Bellare and Phillip Rogaway
IND-CCA does not provide authenticity!
Real-world security

- Schemes providing authenticated encryption are IND-CCA secure.
- For details, see presentation linked from course Web site at https://grothoff.org/christian/teaching/2021/4202/
Break
Part III: Real-world symmetric encryption
GCM encryption

\[\text{Enc}_k(\text{Counter}_0) \rightarrow \text{Enc}_k(\text{Counter}_1) \rightarrow \text{Enc}_k(\text{Counter}_2)\]

\[\text{Plaintext}_1 \rightarrow \text{Ciphertext}_1 \rightarrow \text{Plaintext}_2 \rightarrow \text{Ciphertext}_2\]

\[\text{Auth Data}_1 \rightarrow \text{Auth Tag}\]

\[\text{mult}_H(\text{Auth Data}_1) \rightarrow \text{mult}_H(\text{Auth Data}_2) \rightarrow \text{Auth Tag}\]

\[\text{len}(A) || \text{len}(C) \rightarrow \text{mult}_H\]
Using encryption APIs

GNU libgcrypt is a C library offering a wide range of cryptographic primitives.

1. # apt install libgcrypt20-dev
2. # apt install gcc gdb valgrind emacs
3. Download source templates (exercise.txt) from course Git
Example: AES256 GCM (encrypt.c)

```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)

```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)

```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;
}
```
Tasks (1/3)

- Use the provided encrypt and decrypt programs to encrypt “Hello world” text using AES256+GCM and then decrypt it.
- Study the libgcrypt documentation. Use it to switch the program to use AES256+CBC instead.
- Switch back to AES256+GCM. Extend the program to obtain, transmit and verify the authentication tag.
- Extend the program to authenticate additional plaintext data that is not at all encrypted.
Tasks (2/3)

- Write a new program `hash.c` to compute the SHA-256 hash of the data read from `stdin`. Output the result in HEX and compare to `sha256sum`.
- Modify your program to use SHA-512 instead.
- Write a new program `kdf.c` to compute the SCRYPT key derivation function. Output the result in HEX.
Tasks (3/3)

- Modify your programs to perform 10000 iterations each time before generating any output.
- Measure the time the various operations take.
- Modify your programs to process 1 MB of input instead of the 11 bytes of “Hello world”.
- Again, measure the time the various operations take.
- Change the IV length from 96 bits to 128 bits for AES256+GCM and measure again.
Break
Part IV: Symmetric key establishment protocols
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 Transport**: One party generates and distributes a secret key
- **Key Agreement**: 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.*
Once upon a time...
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

1: $A, \{T_a, B, K_{ab}\}_{K_{as}}$

2: $\{T_s, A, K_{ab}\}_{K_{bs}}$

TTP

Bob

Alice
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

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

TTP
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

1: $A, B$

2: $\{T_S, L, K_{ab}, B, \{T_S, L, K_{ab}, A\}_{K_{bs}}\}_{K_{as}}$

3: $\{T_S, L, K_{ab}, A\}_{K_{bs}}, \{A, T_A\}_{K_{ab}}$

4: $\{T_A + 1\}_{K_{ab}}$

Bob - TTP - 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.
Otway-Rees protocol

1: $M, A, B, \{N_a, M, A, B\}_{K_{as}}$

2: $M, A, B, \{N_a, M, A, B\}_{K_{as}}, \{N_b, M, A, B\}_{K_{bs}}$

3: $M, \{N_a, K_{ab}\}_{K_{as}}, \{N_b, K_{ab}\}_{K_{bs}}$

4: $M, \{N_a, K_{ab}\}_{K_{as}}$

Bob

TTP

Alice
Otway-Rees protocol

- Only 4 messages as Kerberos, but completely different messages.
- Does not require clock synchronization.
- Has a number of problems ⇒ Homework!
Station to station key agreement protocol

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

Alice

1. \(x_A \in [0, n-1]\)
   \(y_A = g^{x_A}\)
   \(CERT_A, y_A\)

Bob

2. \(x_B \in [0, n-1]\)
   \(y_B = g^{x_B}\)
   \(CERT_B, sig_B, y_B\)

3. verify\((A\parallel B\parallel y_B\parallel y_A, sig_B, PK_B)\)
   \(sig_A = sign(A\parallel B\parallel y_A\parallel y_B, SK_A)\)
   \(y_{AB} = y_B^{x_A}\)
   \(A, sig_A\)

4. \(y_{AB} = y_A^{x_B}\)
   verify\((A\parallel B\parallel y_A\parallel y_B, sig_A, PK_A)\)

- 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
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)
Break
Part V: 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

Alice \((k, m)\)

c \(:=\) secure-send\((k, m)\)

Eve (aka Malroy)

c

Bob \((k)\)

If \(m :=\) secure-receive\((k, c)\) \(\neq\) FAIL
Then \(m\) is valid message
Secure channel - Secure send

secure-send(m, k_E, k_M) {

    STATIC msgsnt := 1

    IF (msgsnt ≥ MAX_MSGS) THEN RETURN ⊥

    c := ENC(k_E, m)

    ŵ := msgsnt||LENGTH(c)||c

    t := MAC(k_M, ŵ)

    SEND(ŵ||t)

    msgsnt := msgsnt + 1
}

Secure channel - Secure receive

\[
\text{secure-receive}(C, k_E, k_M) \{ \\
\text{STATIC} \quad \text{msgrcvd} := 0 \\
(mgsnt, len, c, t) = \text{PARSE}(C) \\
\text{IF} \quad (t \neq \text{MAC}(k_M, mgsnt||len||c)) \text{ THEN RETURN } \perp \\
\text{IF} \quad (mgsnt \leq \text{msgrcvd}) \text{ THEN RETURN } \perp \\
m := \text{DEC}(k_E, c) \\
\text{msgrcvd} := mgsnt \\
\text{RETURN } m \\
\}
\]
Remarks

- The *freshness property* based on counters guarantees the following: If $m_1, m_2, \ldots, m_n$ denote the messages sent using secure-send(), then secure-receive() can guarantee that the messages $m_1, m_2, \ldots, m_n$ being received are a 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
Break
Part IV: Extended Security Objectives for Secure Channels
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)

\[ S_A(T_A) \]  \hspace{1cm} (1)
\[ S_B(T_B) \]  \hspace{1cm} (2)
\[ HKDF(DH(T_A, T_B)) \]  \hspace{1cm} (3)
The OTR protocol protects the above KX by wrapping it inside another ephemeral key exchange:

\[ K_1 = DH(T^1_A \| T^1_B) \] (4)

\[ E_{K_1}(S_A(T^2_A)) \] (5)

\[ E_{K_1}(S_B(T^2_B)) \] (6)

\[ K_2 = HKDF(DH(T^2_A, T^2_B)) \] (7)

To achieve forward secrecy, OTR keeps rolling out new keys \( T^i_{A,B} \). 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?
With 3DH, what happens if Alice’s private key \((a, T_a)\) is compromised?

\[ M: K = HKDF(DH(T_a, T_G)||DH(T_a, G)||DH(a, T_G)) \]
\[ A: K = HKDF(DH(T_a, T_G)||DH(T_a, G)||DH(a, T_G)) \]
Forward secrecy

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