Solutions to Selected Problems

Guide to Internet Cryptography

Companion Material

February 12, 2026

Preface

This document provides solutions to selected problems from the book *Guide to Internet Cryptography: Security Protocols and Real-World Attack Implications*. The material is intended for educational use in courses and self-study.

Book website: https://link.springer.com/book/10.1007/978-3-031-19439-9

1 Chapter 12: Attacks on SSL and TLS

Problem 12.1 Attack target

In the following table, please indicate if the given target (HTTP session cookie, PremasterSecret, private key) can be retrieved with the given attack.

Name of Attack	HTTP Session Cookie	PremasterSecret	Private Key
BEAST			
Bleichenbacher			
POODLE			
Invalid Curve			
Lucky13			
Raccoon			

Solution

Name of Attack	HTTP Session Cookie	PremasterSecret	Private Key
BEAST	YES	NO	NO
Bleichenbacher	YES	YES	NO
POODLE	YES	NO	NO
Invalid Curve	YES	YES	YES
Lucky13	YES	NO	NO
Raccoon	YES	YES	NO

Problem 12.2 Attacker Models

(a) What is the difference between the web attacker and the Man-in-the-Browser models? Is one of these models contained in the other model?

(b) A Man-in-the-Browser has access to a (partial) encryption oracle, while a Man-in-the-

Middle doesn't have this ability. Is a Man-in-the-Browser therefore stronger?

Solution

(a) The web attacker model applies to all types of web applications, e.g., email, FTP and DNS. An attacker may set up malicious servers for any of these services, and may send messages/commands to other Internet services via the appropriate protocols. A web attacker does *not* have Man-in-the-Middle privileges, and generally he is not able to spoof IP addresses.
The Man-in-the-Browser model extends the web attacker model by the ability to run JavaScript code in the victim's browser. This is solely possible for general purpose web browsers, where the victim has to load the attacker's web page in his or her own browser. The JavaScript is executed in the context of the attacker's web origin, so this is not to be confused with Cross-Site-Scripting (XSS) attacks. Nevertheless, by invoking HTTPS URLs from the malicious webpage, the Man-in-the-Browser can trigger arbitrary many TLS handshakes, and can send known plaintext, embedded in the URL, to the target server. For adavnces Cross-Site Request Forgery (CSRF) attacks, Man-in-the-Browser privileges are also needed.

(b) No, an Man-in-the-Browser is not stronger than a Man-in-the-Middle. Although the Man-in-the-Browser can *send* known olaintext, he cannot *read* the returned answer - the Same Origin Policy (SOP) blocks this. So both attacker models are somehow orthogonal to each other. Sometimes, the two models are combined, e.g. in CRIME: The Man-in-the-Browser triggers the HTTPS requests, and the Man-in-the-Middle measures the length of the ciphertexts sent.

Problem 12.3 BEAST

For BEAST, the chosen plaintext must be inserted in the first block of a TLS record. Which parts of the HTTP request plaintext can be controlled by a Man-in-the-Browser attacker? Do these parts lie in the first plaintext block?

Solution

The Man-in-the-Browser controls the path vaule of the URL called, including the query string. The first block of the HTTPs request cannot be controlled by the BEAST attacker, because it always contains the HTTP method string (e.g., GET or POST).

Problem 12.4 Vaudenay Padding Oracle Attack

(a) Why is it essential that the encryption key k doesn't change? Which plaintext bytes could still be determined if the key k would change?

(b) Which padding is used to determine the third-last plaintext byte?

Solution

(a) In the Vaudenay padding oracle attack, the attacker *systematically* modifies the last unknown byte misusing the malleability of the cipher mode. This systematic procedure is only possible if the output of the block cipher algorithm itself does not change for the last block. But if the key k would change, this output would change, too. If the key does

change, as it is the case with padding oracle attacks on TLS, only the last byte can be determined *probabilistically*. If the plaintext doesn't change, on average we need 128 HTTPS requests with the same plaintext to recover this byte.

(b) `03 03 03`.

Problem 12.5 POODLE

(a) Why does POODLE only work for SSL 3.0? Can you imagine a faulty implementation of the padding check in TLS 1.2 such that POODLE would work for this flawed implementation?

(b) Why can't POODLE be mitigated through a software patch?

Solution

(a) SSL 3.0 specifies that for CBC mode:

- The padding length byte must be correct
- But the padding bytes themselves can be any value, so they cannot be checked

This is the critical flaw. The MAC is computed over the plaintext before padding is added, so if you can manipulate the ciphertext and the server only checks the padding length (not the padding content), you can perform the POODLE attack.
TLS 1.0, 1.1, and 1.2 specify that:

- The padding length byte must be correct
- All padding bytes must equal the padding length

For example, if padding length is 5, you need: 05 05 05 05 05 05 (6 bytes total). This prevents POODLE because the only valid padding of a full AES block is 15 times the value 15. Any modified ciphertext block will result, with probabilty $1 - \frac{1}{2^{128}}$, in invalid padding bytes, causing the server to reject the message.
Here's an implementation of a flawed implementation:

```
================================================================================
FLAWED IMPLEMENTATION: "LAZY PADDING CHECK"
================================================================================

/**
 * VULNERABLE IMPLEMENTATION
 * Bug: Only checks padding_length byte, not the actual padding content
 */
bool verify_padding_FLAWED(uint8_t* plaintext, size_t length) {
    // Read padding length from last byte
    uint8_t padding_length = plaintext[length - 1];

    // Check if we have enough bytes
    if (length < padding_length + 1) {
        return false;
    }
```

```
    // BUG: Should check all padding bytes here, but doesn't!
    // This mimics SSL 3.0 behavior in TLS 1.2

    return true;  // <- VULNERABLE!
}

WHY THIS IS VULNERABLE TO POODLE:
- Attacker can manipulate ciphertext blocks
- As long as the last byte (after decryption) looks like a valid padding_length,
  the padding check passes
- The actual padding bytes are never verified
- This is exactly the SSL 3.0 vulnerability in a TLS 1.2 implementation
```

(b) The specific SSL 3.0 padding that enables the attack is part of the SSL 3.0 standard. So any implementation with a different padding would no longer be standard-compliant.

Problem 12.6 CRIME

(a) Use LZ77 to compress the string `cybersecurity and cyberwar`.

(b) Which part of the URL must be changed to include an additional character of the HTTP session cookie in the LZ77 compression window?

Solution

(a) Final LZ77 encoding

$$(0,0,\texttt{c}),\ (0,0,\texttt{y}),\ (0,0,\texttt{b}),\ (0,0,\texttt{e}),\ (0,0,\texttt{r}),$$
$$(0,0,\texttt{s}),\ (0,0,\texttt{e}),\ (0,0,\texttt{c}),\ (0,0,\texttt{u}),\ (0,0,\texttt{r}),\ (0,0,\texttt{i}),\ (0,0,\texttt{t}),\ (0,0,\texttt{y}),$$
$$(0,0,\ \ ),\ (0,0,\texttt{a}),\ (0,0,\texttt{n}),\ (0,0,\texttt{d}),\ (0,0,\ \ ),$$
$$(17,5,\texttt{w}),\ (0,0,\texttt{a}),\ (0,0,\texttt{r})$$

(b) One byte must be added before the comparison string, and one byte must be removed after the comparison string, in the path vlaue of the URL.

Problem 12.7 BREACH

Why does BREACH still work when TLS data compression is disabled?

Solution

Because HTTP compression may still be enabled.

Problem 12.8 SSL 2.0: Ciphersuite Rollback Attack

Suppose the attacker would only remove all strong ciphersuites from the ClientHello message in the SSL 2.0 handshake. Would the ciphersuite rollback attack also work in this scenario?

Solution

No, it wouldn't. In SSL 2.0, the client chooses the ciphersuite from the intersection of his set of cipehrsuites, and the ciphersuites contained in the SH message. Since the client does not rely on the (manipulated) CH message, but only on its own configuration, this tattack won't work.

Problem 12.9 SSL 3.0: Version Rollback Attack

In a version rollback attack, the attacker modifies the ClientHello message. In SSL 3.0, this message is protected by the MACs contained in ClientFinished and ServerFinished. So why does this modification remain undetected?

Solution

If ther version is rolled back to SSL 2.0, then only SSL 2.0 checks will be applied during the handshake. And in SSL 2.0, the version number is not protected by the MACs.

Problem 12.10 PKCS#1 variants

Consider the following PKCS#1 variants. Would their use prevent Bleichenbacher-like attacks? Which problems would occur? Please calculate the probability of a random plaintext being compliant with these variants.

(a) Use the PKCS#1 coding for digital signatures also for public-key encryption. In this case, many static padding bytes 0xFF can be checked, and these checks would reduce the probability of finding a PKCS-compliant ciphertext to nearly zero.

(b) Modify PKCS#1 as follows: Half of the padding bytes are chosen randomly, and half have the value 0xFF. At least 16 bytes must be padded.

(c) Use 8 bytes 0x00 as a separator between the random, non-zero padding and the message m.

Solution

Let the RSA modulus length be k bytes. A random plaintext is uniformly distributed in $\{0,1\}^{8k}$.
Recall:

- PKCS#1 v1.5 encryption format:

  $$0x00 \| 0x02 \| PS \| 0x00 \| m,$$

  where PS consists of at least 8 non-zero random bytes.
- Bleichenbacher's attack requires a *valid/invalid padding oracle*. If the probabilty for receiving a 'valid' answer becomes too small (e.g., smaller than 2^{-64} since this is an online attack), we consider the PKCS variant secure.

(a) Use the signature padding format (0x00 0x01 FF...FF 0x00 m).
Format:

$$0x00 \| 0x01 \| \underbrace{0xFF \ldots 0xFF}_{\ell \text{ bytes}} \| 0x00 \| m$$

Probability that a random plaintext is compliant:

- First byte must be $0x00$: probability 2^{-8}
- Second byte must be $0x01$: probability 2^{-8}
- Each padding byte must be $0xFF$

If the padding length is ℓ, the probability is

$$P_a = 2^{-16} \cdot (2^{-8})^{\ell} = 2^{-8(\ell+2)}.$$

For realistic key sizes (e.g. $\ell \geq 8$), this probability is extremely small.

Does it prevent Bleichenbacher? Yes, because $P_a \leq 2^{-8(8+2)} = 2^{-80} < 2^{-64}$.

Additional problem: This format is deterministic (no randomness). RSA encryption becomes deterministic and therefore insecure against chosen-plaintext attacks (no semantic security).

(b) Half random padding, half 0xFF (at least 16 padding bytes).
Suppose the padding length is $\ell \geq 16$, with $\ell/2$ random non-zero bytes and $\ell/2$ fixed $0xFF$ bytes.
Probability:

- First two bytes fixed: 2^{-16}
- Each fixed $0xFF$: 2^{-8} per byte
- Each random non-zero byte: $255/256$

Hence:

$$P_b = 2^{-16} \cdot (2^{-8})^{\ell/2} \cdot \left(\frac{255}{256}\right)^{\ell/2}.$$

Since $\frac{255}{256} \approx 1$,

$$P_b \approx 2^{-16-4\ell}.$$

Again extremely small, but nonzero.

Does it prevent Bleichenbacher? Yes. Because $P_b < 2^{-16-4\cdot 16} = 2^{-80} < 2^{-64}$

(c) Use 8 bytes 0x00 as separator.
Format:

$$0x00 \| 0x02 \| PS \| \underbrace{0x00 \ldots 0x00}_{8 \text{ bytes}} \| m$$

Probability:

- First byte $0x00$: 2^{-8}
- Second byte $0x02$: 2^{-8}
- 8 zero bytes: $(2^{-8})^8 = 2^{-64}$

Ignoring the non-zero requirement for PS,

$$P_c \approx 2^{-16} \cdot 2^{-64} = 2^{-80}.$$

Does it prevent Bleichenbacher? Yes, since the probabilty for 'valid' answers is small enough.

Overall Conclusion
All of these variants prevent Bleichenbacher-like attacks.
However, (a) re-introduces chosen-plaintext attacks due to the deterministic nature of the padding. RSA encryption using this variant would no longer be CPA-secure.
The Proposed countermeasure is to use:

$$\boxed{\text{RSA-OAEP (provably CCA-secure)}}$$

instead of modifying PKCS#1 v1.5 heuristically.

Problem 12.11 Bleichenbacher attack

For an RSA modulus, n with $|n|$ =1025 bits compute the number of integers in the interval $[2B, 3B)$.

Solution

In Bleichenbacher's setting, let the RSA modulus have bitlength

$$|n| = 1025.$$

For PKCS#1 v1.5, we define

$$k = \left\lceil \frac{|n|}{8} \right\rceil$$

as the modulus length in bytes.
Since

$$1025 = 8 \cdot 128 + 1,$$

we get

$$k = 129 \text{ bytes.}$$

The value B is defined as

$$B = 2^{8(k-2)}.$$

Thus,

$$B = 2^{8(129-2)} = 2^{8 \cdot 127} = 2^{1016}.$$

We are asked to compute the number of integers in the interval

$$[2B, 3B).$$

The size of this interval is

$$3B - 2B = B.$$

Hence, the number of integers in the interval is

$$\boxed{2^{1016}}.$$

Problem 12.12 Bleichenbacher attack: Signature forgery

The message to be signed is formatted to be PKCS#1 compliant and is then treated as an RSA ciphertext. So if this ciphertext is already PKCS#1 compliant, will the decrypted plaintext also be PKCS#1 compliant?

Solution

No. First, we are talking about two completely different versions of PKCS#1 here. Second, the 'RSA decryption' of the PKCS#1 (for signatures) formatted 'ciphertext' is a digital RSA signature. This is just an integer modulo n and has no additional structure.

Problem 12.13 ROBOT

Which side channels were used to implement the Bleichenbacher oracles in ROBOT?

Solution

ROBOT ("Return Of Bleichenbacher's Oracle Threat", 2017) showed that many TLS servers still provided a Bleichenbacher-style padding oracle for RSA PKCS#1 v1.5 key exchange.
The oracle was not implemented intentionally, but leaked through *side channels*. The main side channels used were:

1. Different TLS alert messages. Some servers responded with different TLS alert codes depending on whether:

- the RSA PKCS#1 padding was invalid, or
- the padding was correct but later checks failed (e.g., wrong premaster secret version).

Distinguishable error messages directly revealed padding validity.

2. TCP-level behavior. Even when TLS alerts were unified, differences appeared at the TCP layer:

- immediate connection termination vs. graceful shutdown,
- different TCP FIN/RST behavior.

These differences acted as a padding oracle.

3. Timing side channels. Some implementations performed different processing steps depending on whether the padding check succeeded:

- early abort on padding failure,
- further processing (e.g., key derivation) if padding was valid.

This caused measurable timing differences, allowing a timing oracle.

4. Subtle protocol-state differences. In certain cases, servers continued the handshake differently depending on padding correctness, leading to observable differences in subsequent messages.

Conclusion. ROBOT exploited observable differences in:

alert messages, TCP behavior, timing, and handshake state

to reconstruct a Bleichenbacher padding oracle and perform adaptive chosen-ciphertext attacks against RSA key exchange.

Problem 12.14 Raccoon

For which prime moduli p with $1000 < |p| < 2000$ could the Raccoon attack possibly work? Which hash function must be negotiated in TLS 1.2?

Solution

To derive the MasterSecret, the PremasterSecret is used as a key in an HMAC construction. When using prime order groups in TLS-DH, the PremasterSecret is usually much longer than the allowed key length in the HMAC construction. Therefore, the PremasterSecret isn't directly used as the key, but its hash value is.
Before the PremasterSecret is hashed, at least 9 additional byte are appended to it, containing the padding length (1 byte) and a padding which is at least 8 bytes long. This padding is added to make the resulting bytestring a multiple of the block length of the hash function used both in the HMAC function and to compress the PremasterSecret.
For the Raccoon attack to work, this padding should result in one hash function block less when the leading byte of the PremasterSecret is 0 (and is therefore stripped) compared to the situation where this leading byte is not equal 0.
Which prime moduli p with $1000 < |p| < 2000$ would allow this, for the hash functions SHA-1, SHA2-256, and SHA2-512?
The internal blocks of SHA-1 have size 160 bit; those of SHA2-256 have 256 bits; and those of SHA2-512 have 512 bits. SHA2-224 and SHA2-384 are truncated versions of SHA2-256 and SHA2-512, resp., and therefore have the same internal block size.
For Raccon to work optimally, the bit size x of the prime number plus $9 \cdot 8 = 72$ bits from the minimal padding must be just 1 more than a multiple of the block size:

$$x + 72 \equiv 1 \bmod blocksize$$

For SHA-1 with $blocksize = 160$ this yields: We solve

$$x + 72 \equiv 1 \pmod{160}.$$

Step 1: Isolate x.

$$x \equiv 1 - 72 \pmod{160}$$

$$x \equiv -71 \pmod{160}$$

Since

$$-71 \equiv 160 - 71 = 89 \pmod{160},$$

we get

$$x \equiv 89 \pmod{160}.$$

Step 2: General solution.

$$x = 89 + 160k, \quad k \in \mathbb{Z}.$$

Step 3: Restrict to $1000 \leq x \leq 2000$**.** We solve:

$$1000 \leq 89 + 160k \leq 2000.$$

Lower bound:

$$89 + 160k \geq 1000$$

$$160k \geq 911$$

$$k \geq 6.$$

Upper bound:

$$89 + 160k \leq 2000$$

$$160k \leq 1911$$

$$k \leq 11.$$

Step 4: Compute solutions. For $k = 6, 7, 8, 9, 10, 11$:

$$\begin{aligned} k = 6: &\quad x = 1049 \\ k = 7: &\quad x = 1209 \\ k = 8: &\quad x = 1369 \\ k = 9: &\quad x = 1529 \\ k = 10: &\quad x = 1689 \\ k = 11: &\quad x = 1849 \end{aligned}$$

Final Answer

$$\boxed{1049,\ 1209,\ 1369,\ 1529,\ 1689,\ 1849}$$

Problem 12.15 Cross-protocol attack: TLS-1.2 and TLS 1.3

(a) Why does the adversary know the discrete logarithm of the DH share in the ServerKeyShare extension?

(b) Which of the values that are hashed-and-signed in the ServerKeyShare extension can be chosen by the adversary, and which are chosen by the client? Which of these values change between TLS sessions?

Solution

(a) Because in Figure 12.32, the MITM adversary creates the DH share contained in the ServerKeyShare extension himself.

(b) In TLS 1.3, the server's signature is contained in the CertificateVerify message (CV) sent by the server. Here, the whole tanscript of the handshale up to this point is signed. This transcrit contains static values (e.g., the list of ciphersuites and extensions sent by the client), ephemeral values chosen by the client (especialyy r_C and the client's DH share), and values chosen by the server/adversary. The ephemeral values chosen by the client and server change between TLS sessions, and the attacker cannot influence the epemeral values of the client.

Problem 12.16 Cross-protocol attack: DROWN

In the SSL2 handshake (Figure 11.1), in addition to the adaptively chosen RSA ciphertext c in the CMK message, the adversary always has to send a MAC mac_C computed with the client write key cwk. Since he cannot calculate this MAC, the handshake will always abort, both for PKCS#1 compliant and non-compliant plaintexts. How can he nevertheless distinguish PKCS#1 compliant and non-compliant plaintexts?

Solution

As noted in the DROWN paper https://drownattack.com/drown-attack-paper.pdf, all SSL 2.0 servers they tested responded with a SV message *immediately* after receiving the CMK message. This the attacker didn't have to prepare the CF message. This is a deviation from Figure 11.1, and the SSL 2.0 standard is unclear about this message ordering.