Attacking RSA-based Sessions in SSL/TLS

Vlastimil Klíma[1], Ondřej Pokorný and Tomáš Rosa[1,2]

[1] ICZ, Prague, Czech Republic
[2] Dept. of Computer Science and Eng., FEE, Czech Technical University in Prague
vlastimil.klima@i.cz, ondrej.pokorny@seznam.cz, tomas.rosa@i.cz

Abstract. In this paper we present a practically feasible attack on RSA-based sessions in SSL/TLS protocols. These protocols incorporate the PKCS#1 (v. 1.5) encoding method for the RSA encryption of a *premaster-secret* value. The *premaster-secret* is the only secret value that is used for deriving all the particular session keys. Therefore, an attacker who can recover the *premaster-secret* can decrypt the whole captured SSL/TLS session. We show that incorporating a version number check over PKCS#1 plaintext used in the SSL/TLS creates a side channel that allows the attacker to invert the RSA encryption. The attacker can then either recover the *premaster-secret* or sign a message on behalf of the server. Practical tests showed that two thirds of randomly chosen Internet SSL/TLS servers were vulnerable. The attack is an extension of Bleichenbacher's attack on PKCS#1 (v. 1.5). We introduce the concept of a *bad-version oracle* (BVO) that covers the side channel leakage, and present several methods that speed up the original algorithm. Our attack was successfully tested in practice and the results of complexity measurements are presented in the paper. Plugging a testing server (2x Pentium III/1.4 GHz, 1 GB RAM, 100 Mb/s Ethernet, OS RedHat 7.2, Apache 1.3.27), it was possible to achieve a speed of 67.7 BVO calls per second for a 1024 bits RSA key. The median time for a whole attack on the *premaster-secret* could be then estimated as 54 hours and 42 minutes. We also propose and discuss countermeasures, which are both cryptographically acceptable and practically feasible.

1. Introduction

In contemporary cryptography, it is widely agreed that one of the most important issues of all asymmetric schemes is the way in which the scheme encodes the data to be processed. In the case of RSA [14], the most widely used encoding methods are described in PKCS#1 [9]. This standard also underlies RSA-based sessions in the family of SSL/TLS protocols. These protocols became de facto the standard platform for secure communication in the Internet environment. In this paper we assume a certain familiarity with their architecture (c.f. §5). Since its complete description is far beyond the scope of this article, we refer interested readers to the excellent book [10] for further details. In 1998 Bleichenbacher showed that the concrete encoding method called EME-PKCS1-v1_5, which is also employed in the SSL/TLS protocols, is highly vulnerable to chosen ciphertext attacks [1]. The attack assumes that information about the course of the decoding process is leaking to an attacker. We refer to such attacks as *side channel attacks*, since they rely on *side information* that unintentionally leaks out from a cryptographic module during its common activity.

Bleichenbacher showed that it is highly probable that side information exists allowing the attacker to break the particular realization of the RSA scheme in many systems based on EME-PKCS1-v1_5. He has also shown how to use such information to decrypt any captured ciphertext or to sign any arbitrary message by using a common interaction with the attacked cryptographic module. As a countermeasure to his attack it was recommended to either use the EME-OAEP method (also defined in PKCS#1) or to steer attackers away from knowing details about the course of the decoding process. In the case of the SSL/TLS protocols it seemed to be possible to incorporate the second type of countermeasures. The story of the attack ended here by incorporating appropriate warnings in appropriate standards [9], [10], [12], and [15]. Security architects were especially instructed not to allow an attacker to know whether the plaintext *P* being decoded has the prescribed mandatory structure marks or not.

Besides being warned to carry out the above-mentioned countermeasure, architects were also instructed to carefully verify all possible marks of *P* that are specific for the SSL/TLS protocols. In particular, they were told to check the correctness of a version number (c.f. §5.2 and [12]), which is stored in the two left-most bytes of the *premaster-secret*. Unfortunately, it has not been properly specified how such a test may be combined with the countermeasure mentioned above and what to do if the "version number test" fails. Designers may be very tempted to simply issue an error message. In reality, however, such a message opened up a Pandora's box bringing a new variant of side channel attack. In this paper we present this attack and discuss its implementation details. It turns out that the version number, which was initially believed to rule out the original attack [1], even allows a relatively optimized variant of the attack if the version number check is badly implemented. Our practical tests showed that among hundreds of SSL/TLS servers randomly chosen from the Internet, two thirds of them were vulnerable to our attack (for details see §4.3).

We note that the TLS protocol may be historically viewed as an SSL bearing the version number 3.1 [12], while the SSL with the version number 3.0 is often referred to as a "plain" SSL. There are some minor changes between SSL and TLS, but these changes are unimportant for the purpose of this paper, since we rely on the general properties, which are common to both SSL v. 3.0 and TLS. Therefore, we will talk about them as about the SSL/TLS protocols. We note that SSL protocols with version numbers less than 3.0 will not be considered here, since they have already been proven to have several serious weaknesses [10], [16].

The rest of the paper is organized as follows: in §2 we introduce a *bad-version oracle* (BVO), which is a construction that mathematically encapsulates side information leaking from the decoding process. The BVO is then used for mounting our attack in §3. The attack is based on an extended variant of Bleichanbacher's algorithm from [1]. The complexity of the attack together with the statistics of the vulnerable servers found on the Internet are given in §4. In §5 we discuss some technical details behind the practical realization of the attack. Countermeasures are then proposed and discussed in §6. The conclusions are made in §7. In the appendix we recall a slightly generalized version of the original Bleichenbacher's algorithm [1].

Proposition 1 (Connection and session). *Unless stated otherwise, the term connection means the communication carried out between a client and a server. It lasts from when the client opened up a networked pipe with the server, until the pipe is closed. The term session is used to refer to a particular part of this connection which is protected under the same value of symmetrical encryption keys.*

Proposition 2 (RSA-based session). *We say that the session is RSA-based if it uses the RSA scheme to establish its symmetrical keys.*

2. Bad-Version Oracle

We start by recalling the definition of PKCS-conforming plaintext [1]. Unless stated otherwise, the term plaintext means an *RSA* plaintext. Furthermore, we denote RSA instance parameters as (N, e, d), where N is a public modulus, e is a public exponent, and d is a private exponent, such that for all x, $x \in <0, N - 1>$ it holds that $x = (x^e \bmod N)^d \bmod N$. We denote as k the length of the modulus N in bytes, i.e. $k = \lceil(\log_2 N)/8\rceil$, and the boundary B as $B = 256^{k-2}$.

Definition 1 (PKCS-conforming plaintext). *Let us denote the plaintext as P, $P = \sum_{i=1}^{k}(P_i*256^{k-i})$, $0 \leq P_i \leq 255$, where P_1 is the most significant byte of the plaintext. We say that P is PKCS-conforming if the following conditions hold:*

i) $P_1 = 0$

ii) $P_2 = 2$

iii) $P_j \neq 0$ *for all* $j \in <3, 10>$

iv) $\exists j, j \in <11, k>, P_j = 0$; *the string* $P_{j+1}||...||P_k$ *is then called as a message M or a data payload*

The definition describes the set of all valid plaintexts for the given modulus of the length k bytes. In the case of SSL/TLS protocols, however, only the subset of this set is allowed, since these protocols introduce several extensions to the basic PKCS#1 (v. 1.5) format. Therefore, we define the term S-PKCS-conforming plaintext as follows.

Definition 2 (S-PKCS-conforming plaintext). *We say that P is S-PKCS-conforming if it is PKCS-conforming and the following conditions hold:*

i) $P_j \neq 0$ *for all* $j \in <3, k - 49>$

ii) $P_{k-48} = 0$

The main restriction introduced here is the constant number of data bytes (which is equal to 48). The number of padding bytes equals k - 51. Furthermore, SSL/TLS protocols introduce a special interpretation for the first two data bytes P_{k-47} and P_{k-46}, which are respectively regarded as `major` and `minor` version numbers. This extension was introduced to thwart so-called version rollback attacks. The *data payload*, which is the concatenation of $P_{k-47} \| P_{k-46} \| P_{k-45} \| ... \| P_k$, is called a *premaster-secret* here. It is the only secret used in the key derivation process that produces the session keys used by the client and the server in the given session. An attacker, who is able to discover the *premaster-secret*, can decrypt the whole communication between the client and server which has been carried out in the session. The value of $P_{k-45} \| ... \| P_k$ is generated randomly by the client who then adds the version number P_{k-47} and P_{k-46}, encrypts the whole value of the *premaster-secret* by the server's public RSA key, and sends the resulting ciphertext C to the server. The server decrypts it and creates its own copy of the *premaster-secret*. All these steps are carried out during the Handshake sub-protocol (c.f. §5).

It is widely known that the server shall not report whether the plaintext P, $P = C^d \bmod N$, is PKCS-conforming or not. In practice, a server is recommended to continue with a randomly chosen value of the *premaster-secret* if the value of P is not S-PKCS-conforming. Obviously, the communication breaks down soon after sending a `Finished` message (c.f. §5), since the client and the server will both use different values for the session keys. However, the client (attacker) does not know whether the communication has broken down due to an invalid format of P or due to incorrect value of the *premaster-secret*. So, the attack is effectively defeated in this way. Of course, the attacker still gains some information from such an interaction with the server. She may at least try to confirm her guesses of the correct value of the *premaster-secret*. However, it has been shown by Jonsson and Kaliski [4] that it is infeasible to exploit this information for an attack.

Let us suppose that the server incorporates the above-mentioned countermeasure, the primary aim of which is to thwart Bleichenbacher's attack [1]. Furthermore, let all S-PKCS-conforming plaintexts be processed by the server to check the validity of proprietary SSL/TLS extensions according to the following proposition.

Proposition 3 (Conjectured server's behavior).

i) The server checks if the deciphered plaintext P is S-PKCS-conforming. If the plaintext is not S-PKCS conforming, the server generates a new premaster-secret randomly, thereby breaking down the communication soon, after receiving the client's `Finished` *message.*

ii) The server checks each S-PKCS-conforming plaintext P to see whether P_{k-47} = `major` *and* P_{k-46} = `minor`, *where* `major.minor` *is the expected version number which is known to the attacker (c.f. §5 for details). For instance, the most usual version numbers at the time of writing this paper were 3.0 and 3.1. If the test fails, the server issues a distinguishable error message. The test is never done for plaintexts that are not S-PKCS-conforming.*

Practical tests showed that it is reasonable to assume Proposition 3 is fulfilled in many practical realizations of SSL/TLS servers.

Definition 3 (Bad-Version Oracle - BVO). *BVO is a mapping BVO:* $Z_N \rightarrow \{0, 1\}$. *BVO(C) = 1 iff* $C = P^e \bmod N$, *where e is the server's public exponent, N is the server's modulus, and P is an S-PKCS conforming plaintext, such that either* $P_{k-47} \neq$ `major` *or* $P_{k-46} \neq$ `minor`, *where* `major.minor` *is the expected version number. BVO(C) = 0 otherwise.*

BVO can be easily constructed for any SSL/TLS server that acts according to Proposition 3. We send the ciphertext C to the server and if we receive the distinguished message from (ii), we set BVO(C) = 1. Otherwise, we set BVO(C) = 0.

Theorem 1 (Usage of BVO). *Let us have a BVO for given (e, N) and* `major.minor` *and let C be an RSA ciphertext. Then BVO(C) = 1 implies that* $C = P^e \bmod N$, *where P is an S-PKCS-conforming plaintext.*

Proof. Follows directly from Definition 3.

■

Because S-PKCS-conforming plaintext is also PKCS-conforming, it follows from Theorem 1 that we can use BVO to mount Bleichenbacher's attack. We discuss the details in §3. Now we introduce several definitions that will be useful in the rest of this paper. We use a similar notation to the one used in [1].

Definition 4 (Probabilities concerning BVO). *Let Pr(A) = B/N be the probability of the event A that the conditions (i-ii) of Definition 1 hold for randomly chosen plaintext P. Let Pr(S-PKCS|A) be the conditional probability that the plaintext P is S-PKCS-conforming assuming that A occurred for P. Let Pr(BVO|S-PKCS) be the conditional probability that BVO(*$P^e \bmod N$*) = 1 assuming that P is S-PKCS-conforming.*

For Pr(A) we have $256^{-2} < \Pr(A) < 256^{-1}$ as stated in [1]. The probability Pr(S-$PKCS$|A) can be expressed as Pr(S-$PKCS$|A) = $(255/256)^{(k-51)} * 256^{-1}$, since the length of the non-zero padding bytes must be equal to k-51. There is usually one value of the version number that is expected by BVO (see §5). Therefore, Pr(BVO|S-$PKCS$) = $1 - 256^{-2}$. Note that the value of Pr(BVO|S-$PKCS$)*Pr(S-$PKCS$|A)*Pr(A) is the probability that for a randomly chosen ciphertext C we get BVO(C) = 1.

3. Attacking *Premaster-secret*

3.1 Mounting and Extending Bleichenbacher's Attack

This attack allows us to compute the value $x = y^d \bmod N$ for any given integer y, where d is an unknown RSA private exponent and N is an RSA modulus. This attack works under the condition that an attacker has an oracle that for any ciphertext C tells her whether the corresponding RSA plaintext $P = C^d \bmod N$ is PKCS-conforming or not. Theorem 1 shows that BVO introduced in the previous part can be used as such an oracle. In the case of the SSL/TLS protocols this means that we can mount this attack to either disclose a *premaster-secret* for an arbitrary captured session or to forge a server's signature. In the following text, we mainly focus on the *premaster-secret* disclosure. Forging of signatures is discussed briefly in §3.4.

The main idea here is to employ Bleichenbacher's attack with several changes related to the specific properties of S-PKCS and BVO (§3.2). Furthermore, we employed particular optimizations, which we have tested in our sample programs, and which generally help an attacker (§3.3).

3.2 S-PKCS and BVO Properties

We show how to modify Bleichenbacher's original RSA inversion algorithm for use with the BVO and to increase its efficiency. For the sake of completeness we repeat the necessary facts from [1] in the appendix together with a brief generalization of it.

Recall that PKCS-conforming plaintext P satisfies the following system of inequalities

$$E \le P \le F,$$

where $E = 2B$, $F = 3B–1$, and $B = 256^{k-2}$. The boundaries E, F are extensively used through the whole RSA inversion algorithm. Since BVO as well as the SSL/TLS protocols deal only with S-PKCS-conforming plaintexts, we may refine the boundaries as

$$E' \le P \le F',$$

where the value of E' is obtained by incorporating the minimum value of the padding and the value of F' is computed with respect to the fixed position of the zero delimiter in the plaintext P:

$E' = 2B + 1*256^{k-3} + 1*256^{k-4} + ... + 1*256^{49} = 2B + 256^{49}(256^{k-51} - 1)/255$ and

$F' = 2B + 255*(256^{k-3} + 256^{k-4} + ... +256^{49}) + 0 + 255*(256^{47} + 256^{46} + ... + 256^{0}) = 3B – 255*256^{48} – 1.$

Substituting E', F' in place of E, F in the original algorithm (see the appendix) increases its effectiveness.

It follows from the technical details (c.f. §5) that the attacker knows the expected value of the version number, which is checked by BVO. Therefore, when attacking the ciphertext C_0, such that BVO(C_0) = 0, carrying the *premaster-secret*, the attacker knows exactly the two bytes $P_{0,k-47}$ and $P_{0,k-46}$ of the S-PKCS-conforming plaintext $P_0 = C_0^{\,d} \bmod N$. She also knows that $P_{0,k-48} = 0$. We used this knowledge in our program to further trim the interval boundaries $<a, b>$ computed in step 3 of the algorithm (see the appendix).

3.3 Basic General Optimizations

Besides the optimizations that follow directly from §3.2, we also used the generally applicable methods described in the following subparagraphs.

Definition 5 (Suitable multiplier). *Let us have an integer C. The integer s is said to be a suitable multiplier for C if it holds that $C' = s^eC \bmod N = (P')^e \bmod N$, where P' is a S-PKCS-conforming plaintext.*

3.3.1 Beta Method

The following method (β-method) follows from a generalization of the remark mentioned in [1], pp.7 - 8.

Lemma 1 (On linear combination). *Let us have two ciphertexts C_i and C_j, such that $C_i = (s_i)^eC_0 \bmod N$, $C_j = (s_j)^eC_0 \bmod N$, where s_i and s_j are suitable multipliers for C_0. I.e. $P_i = C_i^{\,d} \bmod N = 2B + 256^{49}PS_i + D_i$ and $P_j = C_j^{\,d} \bmod N = 2B + 256^{49}PS_j + D_j$, where $0 < PS_{i,j}$ and $0 \le D_{i,j} < 256^{48}$. Then for C, $C = s^eC_0 \bmod N$ and $\beta \in \mathbf{Z}$, where $s = [(1-\beta)s_i + \beta s_j] \bmod N$, it holds that $C^d \bmod N = P$, such that $P = [2B + 256^{49}((1-\beta)PS_i + \beta PS_j) + (1-\beta)D_i + \beta D_j\,] \bmod N$.*

Proof. It suffices to observe that $P = [(1-\beta)s_i + \beta s_j]P_0 \bmod N = [(1-\beta)P_i + \beta P_j] \bmod N$, where $P_0 = C_0^{\,d} \bmod N$. $\blacksquare$

It follows from the lemma written above that once we have suitable multipliers $s_{i,j}$ for a ciphertext C, we can try to search for the next suitable multiplier s as for a linear combination of s_i and s_j. In practice, we can try small positive and negative values of β and test whether the particular linear combination s gives S-PKCS-conforming plaintext or not. Working in this way, we may hope to accelerate the algorithm in step 2b (c.f. the appendix). Since we can reasonably assume that gcd(s_j - s_i, N) = 1, there is a particular value of β for every triplet of suitable multipliers (s_i, s_j, s). However, experiments have shown that there are also differences in how much information can be obtained from such s depending on the size of β. For small values of β, it has been observed that the obtained values of s do not reduce the size of M_i as fast as the values of s obtained for β close to $N/2$. The reason is perhaps a linear dependency on $\mathbf{Z}$, which is stronger for small β. On the other hand, β close to $N/2$ clearly cannot be directly found by "brute force" searching. More precisely, we may find such β directly, but we cannot assure that obtained s will be of moderate size for further processing by the RSA inversion algorithm. Therefore, it remains to extract as much information as possible from reasonably small values of β and then to either continue with incremental searching used in the original version of the algorithm [1] or to use the Parallel-Threads (PT) method described in §3.3.2. In advance of the following discussion, we note that the source for the next incremental searching or for the PT-method is the maximum suitable multiplier s_j found, such that $s_j < N/2$.

When using the above-mentioned method with negative values of β, we may get a multiplier s that is close to N (it can be regarded as a small negative value modulo N). Such an s cannot be directly processed, since it induces a very large interval for r in the original algorithm (see step 3 in the appendix). We will show how the algorithm can be adjusted to process small positive values of s as well as small negative values of s modulo N.

Theorem 2 (On symmetry). *Let us have integers s, P, and N satisfying*

$$E_1 \leq sP \bmod N \leq F_1, \text{ where } E_1, F_1 \in \mathbf{Z}.$$

Then there is the integer v, v = N - s, satisfying

$$E_2 \leq vP \bmod N \leq F_2, \text{ where } E_2 = N - F_1, F_2 = N - E_1.$$

Proof. We have that $vP \bmod N = (N - s)P \bmod N = (-sP) \bmod N = N - (sP \bmod N)$. The upper boundary of ($sP \bmod N$) is F_1, therefore, the lower boundary E_2 of ($vP \bmod N$) is $E_2 = N - F_1$. Analogically, the upper boundary F_2 of ($vP \bmod N$) is given by the lower boundary E_1 as $F_2 = N - E_1$.

■

We use the theorem as follows: if we get a high value of s using the β-method described above, then we convert it to the corresponding symmetric value $v = N - s$ which is then processed in a modified version of step 3 of the algorithm (see the appendix). The core of the modification is using boundaries E_2, F_2 instead of the original boundaries $E_1 = E'$, $F_1 = F'$ (c.f. §3.2).

3.3.2 Parallel-Threads (PT) Method

Recall that the complexity of step 2 of the algorithm (see the appendix) for $i > 1$ depends on the size of M_{i-1}. Generally, the step is expected to be much faster if $|M_{i-1}| = 1$ than if $|M_{i-1}| > 1$. The reason is that $|M_{i-1}| = 1$ means there is only one interval approximating the value of P_0 left and therefore certain rules can be used when searching for the next suitable multiplier s_i. Experimenting with our test program, we observed that even if $|M_{i-1}| > 1$, the number of intervals was usually small enough that it was better to start a parallel thread T for each $I \in M_{i-1}$ as if it was the only interval left, i.e. it starts its own thread in step 2c of the algorithm. These threads $T_1, ..., T_w$, where $w = |M_{i-1}|$, were precisely multitasked on a per BVO call basis. They were arranged in the cycle $T_1 \rightarrow T_2 ... \rightarrow T_w \rightarrow T_1$ and stepping was done in the cycle after each one BVO call. The results obtained when thread T_j found a suitable multiplier were projected on the whole current set of intervals for all threads. After that, the threads belonging to the intervals that disappeared were discarded. We observed that the PT-method increased the effectiveness of the original algorithm.

Using a certain amount of heuristics we set the condition that directs whether we should use the PT-method or not. The PT-method is started in step i if the following inequality holds

$$|M_{i-1}| < (2\varepsilon \Pr(A))^{-1} + 1.$$

The value of ε estimates the number of passes it takes from the start of the PT-method until there is only one interval left, i.e. $|M_{i+\varepsilon-1}| = 1$, where the PT-method started in pass i. In our programs, we used $\varepsilon = 2$ which was the ceiling of the mean value observed for ε.

3.4 Note on Forging a Server's Signature

The BVO construction allows us to mount Bleichenbacher's attack without any restrictions on its functionality. As noted above, we can compute the RSA inverse for any integer y, thereby obtaining the value $x = y^d \bmod N$ for the particular server's private exponent d and the modulus N. Discussing the so-called semantics of the attack, there are only two cases in which it would be reasonable to compute this inversion.

In the first case we compute the RSA inverse for a captured ciphertext carrying an encrypted value of the *premaster-secret*. This approach allows us to decrypt the whole communication that was carried out in a given session between a client and the server. This is the main approach of this paper, which we have practically tested and optimized.

In the second case we compute an RSA signature of a message *m* on behalf of the server. The whole attack runs in a similar way, which means that the main activity between an attacker and the server is still concerned on the phase of passing the *premaster-secret* value during the handshake procedure of the SSL/TLS protocols. However, this is only because we need to build up a BVO (c.f. §2) for computing the RSA inversion. The source of this inversion (the ciphertext *C*) will no longer be an encrypted *premaster-secret* itself, but the formatted value of $h(m)$, where *h* is an appropriate hash function. Currently, the SSL/TLS protocols sets $h(m) =^{\mathrm{def}}$ MD5(*m*) || SHA-1(*m*) and the value of $h(m)$ is further formatted according to the EMSA-PKCS1-v1_5 method from PKCS#1 ([9], [10], [12], [13], [17]). At the end of the attack we obtain C^d mod *N* which is the signature of our input *C*. It further depends on the `keyUsage` property [18] of the certificate of the server's RSA key, whether such a signature can be used for further attacks or not. At first the server's RSA key must be attributed for signing purposes. Secondly, it depends on the specific system as to how far the faked signature is important, directly implying how dangerous the attack is. From the basic properties of SSL/TLS ([10], [12]) it follows that such a signature may be abused to certify an ephemeral RSA or D-H [11] public key of a faked server. The faked server can then be palmed on an ordinary user to elicit some secret information from her. Generally speaking, this would be an attack on the authentication of a server. The necessary condition here is that the user is willing to use either the so-called export RSA key or the ephemeral Diffie-Hellman key agreement [11]. The practical situation is that some clients will - some clients will not. It strongly depends on the attention paid to the configuration of such a client. Unfortunately, these "minor" details are very often neglected in a huge amount of applications. Moreover, we emphasize that the attack described here may not be the only one possible, since the particular importance of a server's signature depends on the role that the server plays in a particular information system. The best way to avoid all these attacks is to not attribute the server's RSA key for signing purposes, unless it is absolutely necessary.

From the effectiveness viewpoint, we can estimate that using the RSA inversion based on BVO for signature forging will require more BVO calls, since we need to insert an extra masking zero-step (see appendix, step 1 of the algorithm). The number of additional BVO calls may be calculated as $[\Pr(BVO|S\text{-}PKCS)^* \Pr(S\text{-}PKCS|A)^* \Pr(A)]^{-1}$, which is given by the probability that for a randomly chosen ciphertext *C* we get BVO(*C*) = 1. Adding this value to the number of BVO calls in the former attack on *premaster-secret* (c.f. §4) gives an estimate of the overall complexity of signature forging.

4. Complexity Measurements

Basing on the elaboration from [1], we can estimate the number of BVO calls for decrypting a plaintext C_0 belonging to a S-PKCS-conforming plaintext P_0 as

$$2^*\Pr(P)^{-1} + (16k - 32)^*\Pr(P|A)^{-1}, \text{ where } \Pr(P|A) = \Pr(BVO|S\text{-}PKCS)^*\Pr(S\text{-}PKCS|A),\ \Pr(P) = \Pr(P|A)^*\Pr(A).$$

This estimation does not cover the optimization described in §3.2 and §3.3. Therefore, we regard it is as the worst-case estimation for a situation when these optimizations are not notably helping an attacker. Experiments show that the optimized algorithm is practically almost two times faster than this estimation (c.f. §4.1) for the most widely used RSA key lengths. Let us comment on the expression of the estimation now.

The first additive factor corresponds with our assumption that the attacker wants to decipher C_0 belonging to a properly formatted plaintext carrying a value of the *premaster-secret*. In such a situation, she does not have to carry out initial blinding (c.f. the appendix, step 1). According to [1], we can estimate that she needs to find two suitable multipliers $s_{1,2}$ for C_0, until she can proceed with the generally faster step 2c. This gives the first factor as $2^*\Pr(P)^{-1}$. Note that, heuristically speaking, the optimizations (§3) mainly reduce the necessity of finding s_2 in the "hard" way, thereby decreasing the first factor closely to the value $\Pr(P)^{-1}$. This hypothesis corresponds well with the results of our measurements.

The second factor is a slightly modified expression presented in [1]. It corresponds to the number of expected BVO calls for the whole number of passes through step 2c. Recall that $C_0 = (P_0)^e \bmod N$, where $2B \le P_0 \le 3B - 1$, so P_0 lays in the interval of the length *B*, $B = 256^{k-2}$. Conjecturing that each pass through step 3 roughly halves the length of the interval for P_0, we may estimate that we need $8(k - 2)$ passes. Furthermore, it is conjectured [1] that each pass through step 2c takes $2^*\Pr(P|A)^{-1}$ BVO calls. From here follows the estimation of BVO calls as $(16k - 32)^*\Pr(P|A)^{-1}$.

Finally, we note that the complexity of the attack is mainly determined by the amount of necessary BVO calls. This amount actually limits the attack in the three ways. The first one is that an attacked server must bear such a number of corrupted Handshakes (i.e. not collapse due to a log overflow, etc.). The second limitation comes from a total network delay that increases linearly with the number of BVO calls. The third limit is determined by the computational power of the server itself, which mainly means how fast it can carry out the RSA operation with a private key. Other computations during the attack are essentially faster and therefore we do not discuss them here.

4.1. Simulated local BVO

In this paragraph, we present the measured complexity of the attack with respect to the total amount of BVO calls. The data of our experiment was obtained for the four particular randomly generated RSA moduli of 1024, 1025, 2048 and 2049 bits in length. For every such modulus we implemented a local simulation of BVO that we linked together with the optimized algorithm discussed in this paper. We then measured the number of BVO calls for 1200 ciphertexts of the randomly generated and encrypted values of the *premaster-secret*.

Due to the strong dependence of the number of BVO calls on Pr(A) we see that the complexity of the attack is not strictly increasing with respect to the length of the modulus N. This discrepancy was already mentioned in [1]. It follows that one should use moduli with a bit length in the form 8r, where r is an integer, mainly avoiding the moduli with the length 8r + 1.

Table 1. Basic statistics of a measured attack complexity in BVO calls

Modulus length (bits)	**BVO calls**					
	Originally estimated (without optimizations)	**Practically measured** (with optimizations from §3)				
		Min	Max	Median	Mean	Variance
1024	36 591 001	815 835	278 903 416	13 331 256	20 835 297	$6.26258*10^{14}$
1025	979 488	630 589	105 122 011	1 197 380	1 422 176	$8.30727*10^{11}$
2048	48 054 328	2 824 986	354 420 492	19 908 079	28 728 801	$8.934*10^{14}$
2049	2 794 937	1 413 005	475 298 397	3 462 557	3 896 432	$4.04318*10^{12}$

Analyzing the measured data, we observed that the distribution of the amount of BVO calls can be approximated by a log-normal Gaussian distribution, i.e. the logarithm of the amount of BVO calls roughly follows a normal Gaussian distribution. Heuristically speaking, this means that the most basic random events governing the complexity of the attack primarily combine together in a multiplicative manner. The values of median, mean, and variance are presented in Table 1. These values were obtained using the log-normal approximation of the data samples measured. These approximations are plotted in Fig. 1 and Fig. 2. We can see that all the distributions skew to the right. Therefore, the most interesting values are perhaps given by the medians. For example, in the case of a 1024 bits long modulus, we can expect that the one half of all attacks succeed in less then 13.34 million BVO calls. Furthermore, the data in Table 1 supports our conjecture that the optimizations proposed in §3 mainly speed up the first "hard" part of the algorithm. Therefore, this speeding up is clearly notable for moduli of 1024 and 2048 bits, while there is no observable effect for the moduli of 1025 and 2049 bits.

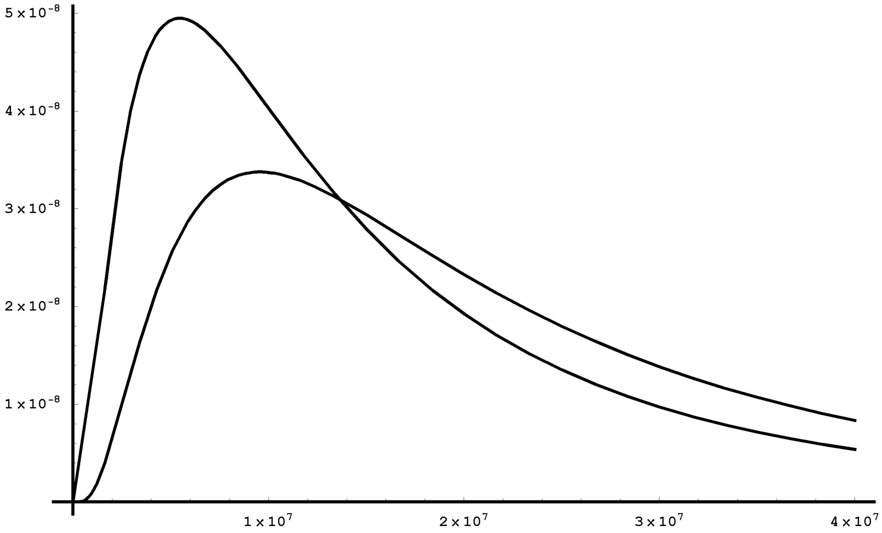

Fig. 1. Log-normal approximation of BVO calls density functions for 1024 (higher peak) and 2048 bits long moduli

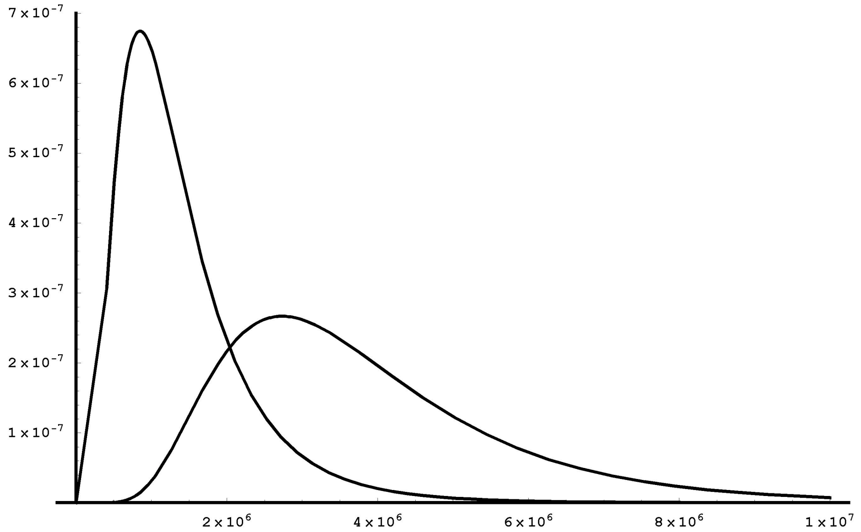

Fig. 2. Log-normal approximation of BVO calls density functions for 1025 (higher peak) and 2049 bits long moduli

4.2. Real Attack

We successfully tested the attack on a real SSL server (AMD Athlon/1 GHz, 256MB RAM) using 1024 bits long RSA key. The total number of BVO calls for decryption of a randomly selected *premaster-secret* was 2 727 042 and the whole attack took 14 hours 22 minutes and 45 seconds. It gives an estimated speed of 52.68 BVO invocations per second. The server and the attacking client were locally connected via a 100 Mb/s Ethernet network without any other notable traffic. With respect to the whole conditions of this experiment, we can conclude that this is probably one of the best practically achievable results. Therefore, we can expect that there would be few practical attacks succeeding in less then 14 hours of sustained high effort (for a 1024 bits long RSA key). Using the value of the median for 1024 bits modulus from Table 1, we can roughly expect one half of all attacks in our setup to succeed in less than 70 hours and 18 minutes. For 2048 bits long RSA key in the same setup we get an estimated speed of 11.47 BVO calls per second. Therefore, one half of all attacks should then succeed in less than 21 days.

The experiment setup described above could be slightly improved by using a more powerful server. Plugging in such a server (2x Pentium III/1.4 GHz, 1 GB RAM, 100 Mb/s Ethernet, OS RedHat 7.2, Apache 1.3.27), it was possible to achieve a speed of 67.7 BVO calls per second for a 1024 bits RSA key. The median time for a whole attack on the *premaster-secret* could be then estimated as 54 hours and 42 minutes. Note that all these estimates assume achieving and sustaining high communication and computation throughput on the server's side.

4.3. Real Vulnerability

To assess the practical impacts of the attack presented here, we had randomly generated a list of 611 public Internet SSL/TLS servers (we accepted servers providing SSL v. 3.0 or TLS v. 1.0) and then tested these servers to see whether it was possible to construct a BVO for them or not. We found that two thirds of these servers were vulnerable to our attack. We emphasize that it does not necessarily mean that the attack would always succeed on every such server. Despite the fact that all these servers can be regarded as broken from a pure cryptanalytic viewpoint, the complexity of the attack may still render it impractical in a large amount of cases. We expect that a properly administrated server (e.g. log messages are often inspected, suspicious clients are added to black-lists, etc.) should withstand the attack. Under such an administration, the attack should be recognized and the attacking client would soon be blocked. Of course, the cryptographic strength of all these SSL/TLS implementations should definitely be improved. We strongly recommend applying appropriate patches as soon as possible.

We observed an interesting anomaly for 110 out of 611 tested servers. All of them provided both SSL v. 3.0 and TLS. 26 of them were *primarily* vulnerable only through the SLL v3.0 protocol, while the remaining 84 servers were *primarily* vulnerable only through the TLS protocol. We advisedly used the word "primarily", since if these servers share the same RSA key for both protocols, which is a very common practice, then an attacker can easily assault one protocol through an interaction with the other one. Moreover, the format of the ciphertext carrying the *premaster-secret* is the same for both protocols, so this cross-attacking actually does not increase the complexity of the whole attack.

5. Technical Details

In the SSL/TLS protocols, there are several sub-protocols that are used during various stages of a connection ([10], [12]). The most interesting ones for our attack are the Handshake and the Alert sub-protocols. The Handshake is used at the beginning of each session and its aim is to establish symmetrical encryption keys for that session. The SSL/TLS protocols allow various combinations of symmetrical and asymmetrical cryptographic schemes, which are called *CipherSuites*. Our attack focuses on those *CipherSuites* that use the RSA algorithm for establishing symmetrical session keys. These suites are used by a huge amount of contemporary applications.

The messages exchanged between a client and a server during the Handshake and their mandatory order are given on Fig. 3.

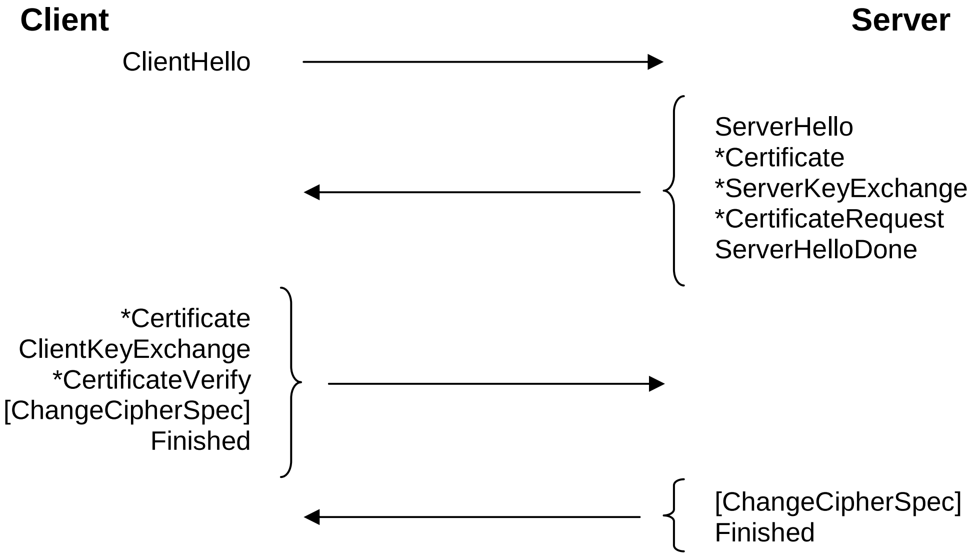

Fig. 3. SSL/TLS Handshake

The messages marked with * are optional, however, they must be in their given places if they are issued. The main purposes of the `ClientHello` message are announcing supported *CipherSuites* and protocol versions together with presenting a client's random value used for the derivation of session keys. The server replies with the `ServerHello` message in which it selects the *CipherSuite* and the protocol version giving the highest possible security level for this session. In RSA-based sessions, the `ClientKeyExchange` message carries an encrypted value of the *premaster-secret* that was formatted to be S-PKCS-conforming before its encryption (c.f. §2). The sequence of messages `[ChangeCipherSpec]` and `Finished` denotes the end of the Handshake. Note that the `ChangeCipherSpec` is not properly part of the Handshake sub-protocol [10], but it is an unimportant technical detail here. The `Finished` messages are the first messages protected by the newly negotiated *CipherSuite* and session keys. They also include a cryptographic checksum of all the exchanged data that has been part of the Handshake sub-protocol. The checksum is computed using the PRF method [10] under the new session keys. Under normal operation, the Handshake sub-protocol passes to the Application Data sub-protocol, which serves as a transparently secured data pipe between the client and the server. The connection may return to the Handshake phase later on whenever the client decides to start a new session (the current session is then closed). We note that our attack is entirely focused on analyzing the first session in a captured connection. The second and next sessions will have their Handshakes encrypted under the keys belonging to their "parent" session. It is still possible to decrypt these sessions however, the attacker must do this on a session-by-session basis starting from the first one.

Possible fatal errors and warnings arising during SSL/TLS communication are reported using the Alert sub-protocol. Its messages transparently flow between packets of other sub-protocols. Once the session is well established, the Alert messages are also cryptographically protected.

5.1 Constructing BVO

From the technical viewpoint, each BVO(C) call means starting a new RSA-based session with the Handshake described above. The attacker does mainly the following: she asks for an appropriate RSA-based session, sets the version number in the `ClientHello` (c.f. §5.2), and incorporates the challenge C into the `ClientKeyExchange` message. She completes the Handshake using the session keys derived for a randomly guessed value of the *premaster-secret'*. Then she waits for the server's reaction, which will be sent using the Alert sub-protocol. The core of her approach is based on the Alert message she gets. The attack relies on the possibility of identifying the case in which the decrypted plaintext P, $P = C^d \bmod N$, is S-PKCS-conforming with a bad version number. For example, the OpenSSL implementation [7] sends the "handshake failure" (for SSL v. 3.0) or "decode error" (for TLS v1.0) Alert message in such a situation. This message is unencrypted, since the session has not been set up yet, and it precisely distinguishes the situation described above from other possible erroneous states at the end of the Handshake. We note that the Alert message may not be the one and only way in which the BVO can be constructed. There may be other side channels that allow a BVO construction for instance, a timing side channel. The use of this side channel in connection with a SSL/TLS server was already demonstrated in a different attack [2].

5.2 Version Number

According to the version number included in the *premaster-secret*, the documents [10], [12] say the following: "*Upon receiving the premaster-secret, the server should check that this value matches the value transmitted by the client in the* `ClientHello` *message*" (c.f. [12], p. 44). However, Rescorla noted (c.f. [10], p. 79) "*...but in practice many clients use the negotiated version instead...*". So, there may be two correct values of the version number for the given Handshake. As may be expected, from a pure cryptological point of view, the situation becomes a bit messy when moving from theory to a practice. It solely depends on the particular server whether it requires the version number to be set to the value offered in `ClientHello` (type-I server) or if it allows this value to be also set to the negotiated version from `ServerHello` (type-II server). However, such a situation may be checked before an attack and the particular behavior of the server may be then well estimated. With this knowledge, we can construct a BVO so that there is only one correct value of the version number expected. If the server is of type-I then there is no problem with this. If it is of type-II then all we have to do is to find the setting under which the version number negotiated in `ServerHello` becomes the same as the number offered in `ClientHello`. Then the server behaves exactly like a server of type-I.

We can also use the version number that is known from the `ClientHello` and `ServerHello` to narrow the size of intervals searched for in the attack (c.f. §3.2).

First, we compare the version numbers from `ClientHello` (ver_1) and `ServerHello` (ver_2) belonging to the Handshake of the analyzed session. If they are the same, then this common value must also have been included in the *premaster-secret*. If these values are different then we change the value in `ClientHello` so that the value chosen by the server would not change. We start a new session using this `ClientHello` message. In place of the `ClientKeyExchange` we use directly the `ClientKeyExchange` from the analyzed session. If the server terminates the connection due to a bad version number, then we know that there is ver_1 in the attacked *premaster-secret*. Otherwise, it is ver_2.

6. Countermeasures

Due to the compatibility demands, it does not seem possible to simply leave the EME-PKCS1-v1_5 method and use its successor EME-OAEP. Note that even the EME-OAEP method must be implemented carefully (c.f. [5], [6]). On the other hand, it has been recently shown by Jonsson and Kaliski in [4] that the EME-PKCS1-v1_5 can offer reasonable security (the proof was carried out for the TLS protocol) assuming that it is implemented properly – i.e. mainly that side channels are avoided. What remains is to show what a proper implementation should look like. The current guidelines in [12] together with [15] are obviously insufficient and should be updated to avoid weaknesses like the one discussed in this paper. Moreover, it seems that the edge between secure and insecure implementation of EME-PKCS1-v1_5 is very sharp. This implies that the standards regarding its implementation must really be very precise.

6.1 Promising Countermeasures which Are Cryptographically Odd

The following countermeasures can hinder our attack, but each one exposes at least a theoretical cryptographic weakness.

6.1.1 Testing Randomly Generated Payload

It may be a tempting idea to introduce the following general countermeasure: at first, the plaintext *P* being decoded is checked to see if it is S-PKCS-conforming. If it is not, then the data payload $P_{k-47} \| ... \| P_k$ is randomly generated from scratch. Therefore, at the end of the first phase, we should have a data payload, no matter how it was obtained - whether by extracting or by random generating. The data payload would then be the subject of all successive checks, especially the version number test. Note that with a little effort, this countermeasure can be read between the lines on page 44 of RFC 2246 [12].

We will show that such a countermeasure is not cryptographically good, since the successive checks work on small parts of the data payload. By examining a sequence of Handshake invocations, we can distinguish whether the server generates the payload randomly or if it uses the result of decoding a properly formatted plaintext *P*. Thereby, we know if the original plaintext *P* is S-PKCS-conforming or not.

In our attack, such a countermeasure would mean that an attacker has to change her strategy. Now, there is a high probability that a randomly chosen ciphertext *C* gives a bad version number. The attacker will generate random *C* and wait until the server responds that the version number is correct. With this result she still does not know whether $P = C^d \bmod N$ is a correct S-PKCS-conforming plaintext (with a correct version number) or if the version number was accidentally correct in the random payload. However, she can decide between these two variants by sending the same *C* again. If it again gives the correct version number, then it is highly probable that *P* is S-PKCS-conforming, since the probability that a consecutive randomly generated payload again gives the correct version number is close to 256^{-2} here. The probability that even a third consecutive invocation gives the correct version number, given that *P* is not S-PKCS-conforming, is close to 256^{-4}, etc. Let us say that she would always carry out these two checking invocations for each ciphertext of a possibly S-PKCS-conforming plaintext. If *X* denotes the number of necessary oracle calls in the BVO-attack, then she needs approximately $X*(2^{16}+2)$ oracle calls now. Using only one control invocation gives the estimate as $X*(2^{16}+1)$ oracle calls. We note that even this generally low complexity may successfully thwart the attack in many practical implementations. On the other hand, such a protocol still cannot be regarded as cryptographically secure.

6.1.2 Treating the Version Number as a PKCS Mark

Another seductive countermeasure is to regard a failure of the version number check as a failure of EME-PKCS1-v1_5 decoding. Therefore, seeing an incorrect PKCS#1 format or an incorrect version number, the server would randomly generate new data payload $P_{k-47} \| P_{k-46} \| ... \| P_k$ and continue with it through the rest of the Handshake (of course omitting the version check). Such a countermeasure would effectively thwart the attack described here.

This countermeasure is nearly perfect, but we have to note that there is a weakness, even if it is highly theoretical and impractical. The disadvantage is in the following: Suppose the attacker has a ciphertext *C*, such that $C = P^e \bmod N$ for a S-PKCS-conforming plaintext *P*, and at the same time she already knows $\mu = P_{k-45} \| ... \| P_k$. Then she can discover the values of P_{k-47} and P_{k-46}.

She does so by sending the `ClientHello` (*ver*) messages for different versions *ver* according to her choice together with the ciphertext *C* in the `ClientKeyExchange` message. She sends these messages until the server uses the *premaster-secret* from *P*. Since she knows the rest of the *premaster-secret*, she can compute `Finished` message correctly. When the server accepts her `Finished` message, she hit the correct version *ver*, originally unknown to her. There is probably no practical attack based on this construction. However, from a pure cryptographic viewpoint, such a property is not good and should be avoided.

The approach described above is recommended by Rescorla ([10], p. 170), who attributes it to RFC 2246 [12]. However, from a closer cryptographic examination, it may seem that RFC 2246 rather recommends the measure from §6.1.1. Evidently, this is another reason why this standard deserves a certain amount of additional explanation.

6.2 Countermeasure which Is Both Practically and Cryptographically Bearable

Demands: Recall that a good countermeasure should mainly:

i) ensure that tampering with the version number is detected,
ii) hide partial information about the RSA plaintext being decoded as much as possible,
iii) not allow new attacks.

Proposal: Under these demands, we propose processing the ciphertext C from the `ClientKeyExchange` message to get the *premaster-secret* as follows:

i) decrypt C on P as $P = C^d \bmod N$
ii) check if P is S-PKCS-conforming, if it is not, replace the values $P_{k\text{-}45}$ ... P_k by 46 bytes of random data
iii) in any case securely discard P_1 ... $P_{\text{k-48}}$
iv) in any case replace $P_{k\text{-}47}$ and $P_{k\text{-}46}$ by the expected `major` and `minor` version numbers, respectively (c.f. §5.2)
v) set *premaster-secret* = $P_{k\text{-}47} \| ... \| P_k$

The server may optionally check the version number from the original plaintext P against the expected values and log the result of this test. It may also log the check result from step (ii). However, all these logs should then be regarded as sensitive values.

There is a minor problem relating the tolerant type-II servers (c.f. §5.2) in that such a server will do the substitution in step (iv) twice, since two values are expected for the version number. It also means that the server must compute two *premaster-secrets* as well as sets of session keys and hold them until seeing the client's `Finished`. After then the server decides which one should be used and which one can be discarded.

We conjecture that the countermeasure presented above meets our demands. To support this hypothesis, we present the following arguments.

Let us consider tampering with a version number so that the offered number from a client is not the one received by the server. Such tampering will be detected after exchanging `Finished` messages, since the client and the server will both use different values of the *premaster-secret*. We may reasonably assume that it would be infeasible for an attacker to also tamper with these `Finished` messages, which do not only carry a cryptographic checksum, but are also protected using session keys derived from the *premaster-secret*.

According to the server's behaviour proposed above and the possible chosen ciphertext attack, an attacker is only able to try to distinguish whether the server uses a random or the original value of the *premaster-secret* in step (ii). Assume the attacker has a ciphertext C, where $C = P^e \bmod N$. Let us denote $\mu = P_{k\text{-}45} \| ... \| P_k$.

- If she does not know the value of μ, she cannot get any new notable partial information about P, since she is unable to distinguish whether the server uses a random or the original (*version number* || μ) value of the *premaster-secret*.
- If she knows the value of μ, she has an oracle O_μ: $\boldsymbol{Z_N} \rightarrow \{0, 1\}$, so that O_μ (C) tells her whether C decrypts to a S-PKCS-conforming P or not. In this way she gets certain particular information about the plaintext of this specific ciphertext C. Such information does not seem to be of notable merit provided she can only get it for some singular ciphertexts. For instance, she cannot learn the exact values of $P_{k\text{-}47}$ and $P_{k\text{-}46}$ as in §6.1.2, since the server does not use the values $P_{k\text{-}47}$ and $P_{k\text{-}46}$ in any way. Let us assume that the attacker is able to use O_μ (C) for any ciphertext C. This means that she can know the appropriate value of μ for every such ciphertext. From here and the theorem of RSA individual bits [3], it follows that she can invert the RSA permutation $x \rightarrow (y = x^e \bmod N)$ for any integer y. Proof of the reduction from a partial-RSA problem to a gap-RSA-P1 problem in [4] even shows an optimized algorithm for such an inversion (see [4], appendix A, proof of Lemma 1). It follows that getting the possibility of routine O_μ usage (i.e. "un-keying" it for any C) is as hard as inverting the whole RSA. Therefore, we conjecture that leaking partial information about P is minimized.

From step (iv) it follows that an active attacker can use messages from a captured session to tamper with the server using various version numbers for the *premaster-secret*. However, all she can do is to make the server set $P_{k\text{-}47}$ and $P_{k\text{-}46}$ at arbitrary values of her choice (using `ClientHello`, see §5.2) and then wait to see if the server accepts a tampered content of `Finished` belonging to the captured session. With regard to how the computation of `Finished` together with the derivation of session keys are carried out ([10], [12]), one can hardly expect that successful attacks would be constructed in this way.

7. Conclusions

We have presented a new practically feasible side channel attack against the SSL/TLS protocols. When Bleichenbacher presented his attack on PKCS#1 (v. 1.5) in 1998 [1], it was generally assumed that the attack was impractical for the SSL/TLS protocols, since these protocols add several proprietary restrictions on the plaintext format, which increase the complexity of the attack. Of course, the protocols could not be called secure from a pure cryptographical viewpoint. Therefore, a special countermeasure was introduced and generally adopted [10], [12]. However in this paper, we have shown that problems with Bleichenbacher's-like attacks on the SSL/TLS protocols are still not properly solved. We have identified a new possibility of a substantial side channel occurring during an SSL/TLS Handshake. The side channel originates when a receiver checks a version number value stored in the two left-most bytes of the *premaster-secret*. Based on the receiver's behavior during this check, we have defined its mathematical encapsulation as a *bad-version oracle* (BVO, c.f. §2). Such a check is widely recommended for SSL/TLS servers, but unfortunately it is not properly specified how it should be performed. Practical tests showed that two thirds of randomly chosen Internet servers carried out the test wrongly, thereby allowing the construction of BVO resulting in a new attack on RSA-based sessions. The attack itself may be viewed as an optimized and generalized variant of the original Bleichenbacher's attack [1]. The most obvious target of our attack would probably be discovering the *premaster-secret*, thereby decrypting a captured RSA-based session. It is also possible (with an additionally increased complexity, c.f. §3.4) to compute the signature of any arbitrary message on behalf of the server.

The attack was carried out in practice and its efficiency was measured (§4). The amount of time the attack takes in practice is mainly determined by the amount of BVO calls. Each BVO call corresponds to one attempt to establish a SSL/TLS connection with an attacked server. If the server uses a typical 1024 bits long RSA key, then we can expect that roughly 50% of attacks succeed in less than 13.34 million BVO calls. For a practical estimation, the particular server speed must be known. For instance, in one of our testing setups we achieved a speed that allowed us to expect that 50% of attacks succeeded in less then 54 hours and 42 minutes. This load may be further spread as 2 hours of these interactions per day, thereby spreading the whole attack over roughly one month, etc. The attack is not limited to running on a single computer, so it can be distributed. The main aim would not be speeding up the attack, but making its localization and blocking harder. Although the complexity presented here is definitely very low from a pure cryptographic viewpoint, there may still be technical measures that can thwart the attack in a practice. For instance, each BVO call should produce at least one log record on the server's side. If these logs are well maintained and appropriately inspected, then the attack should be recognized in time. Unfortunately, there also seem to be poorly administrated servers where SSL/TLS audit messages are almost ignored. These servers remain protected solely by their network and computational throughput, which is obviously alarming.

Finally, we conclude that even those well-administrated servers should be patched to thwart the attack in a primarily cryptographic rather than a pure technical way. For this case, we have discussed various possible countermeasures in §6. There are three countermeasures presented the strength of which can be commented on as follows. Measure §6.1.1 is obviously weak and should be avoided. Measure §6.1.2 effectively thwarts our attack, however leaving a (purely) theoretical weakness. The third countermeasure presented in §6.2 seems to be both cryptographically and practically optimal. However, for those implementations that are already using the measure from §6.1.2, we do not think it is necessary for them to immediately follow §6.2. This should be used mainly in new implementations of the SSL/TLS protocols.

Acknowledgements

We are grateful to Jiří Hejl for technical support and consultations. We also appreciate technical help of Roman Kalač and Libor Kratochvíl. The third author is grateful to his postgraduate supervisor Dr. Petr Zemánek for continuous support in research projects.

References

1. Bleichenbacher, D.: Chosen Ciphertexts Attacks Against Protocols Based on the RSA Encryption Standard PKCS#1, *in Proc. of CRYPTO '98*, pp. 1 - 12, 1998
2. Canvel, B.: Password Interception in a SSL/TLS Channel, http://lasecwww.epfl.ch/memo_ssl.shtml, February, 2003
3. Håstad, J., Näslund M.: The Security of Individual RSA Bits, *in Proc. of FOCS '98*, pp. 510 - 521, 1998
4. Jonsson, J., Kaliski, B., S., Jr.: On the Security of RSA Encryption in TLS, *in Proc. of CRYPTO '02*, pp. 127 -142, 2002
5. Klíma, V., Rosa, T.: Further Results and Considerations on Side Channel Attacks on RSA, *in Proc. of CHES '02*, August 13 - 15, 2002
6. Manger, J.: A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1, *in Proc. of CRYPTO'01*, pp. 230-238, 2001
7. OpenSSL: OpenSSL ver. 0.9.7, http://www.openssl.org/, December 31, 2002
8. PKCS#5 ver. 2.0: Password-Based Cryptography Standard, RSA Laboratories, March 25, 1999
9. PKCS #1: RSA Encryption Standard, An RSA Laboratories Technical Note, Version 1.5, Revised November 1, 1993
10. Rescorla, E.: SSL and TLS: Designing and Building Secure Systems, Addison-Wesley, New York, 2000
11. RFC 2631: Rescorla, E.: Diffie-Hellman Key Agreement Method, June 1999
12. RFC 2246: Allen, C., Dierks, T.: The TLS Protocol, Version 1.0, January 1999
13. RFC 1321: Rivest, R.: The MD5 Message-Digest Algorithm, April 1992

14. Rivest, R., L., Shamir, A., Adleman L.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, *Communications of the ACM*, 21, pp. 120-126, 1978

15. RSA Labs: Prescriptions for Applications that are Vulnerable to the Adaptive Chosen Ciphertext Attack on PKCS #1 v1.5, RSA Laboratories, http://www.rsasecurity.com/rsalabs/pkcs1/prescriptions.html

16. Schneier, B., Wagner, D.: Analysis of the SSL 3.0 Protocol, *The Second USENIX Workshop on Electronic Commerce Proceedings*, USENIX Press, November 1996, pp. 29 - 40

17. Secure Hash Standard, FIPS Pub 180-1, 1995 April 17

18. X509: ITU-T Recommendation X.509 (06/97) - Information Technology - Open System Interconnection - The Directory: Authenticantion Framework, ITU, 1997

Appendix

For the sake of completeness we enclose here the algorithm from [1]. For our purposes we define directly a slight generalization and modification of it. Recall that in the original text $E = 2B$, $F = 3B–1$, where $B = 256^{k-2}$. In our variant, we will use the refined values E' and F' (c.f. §3). The modified algorithm is as follows.

Step 1: Blinding. Given an integer c, choose different random integers s_0; then check, by accessing the oracle, whether $c(s_0)^e \bmod N$ is PKCS conforming.

For the first successful s_0, set

$$c_0 \leftarrow c(s_0)^e \bmod N$$
$$M_0 \leftarrow \{[E, F]\}$$
$$i \leftarrow 1.$$

Step 2: Searching for PKCS conforming messages.

Step 2.a: Starting the search. If $i = 1$, then search for the smallest positive integer $s_1 \geq N/(F+1)$, such that the ciphertext $c(s_1)^e \bmod N$ is PKCS conforming.

Step 2.b: Searching with more than one interval left. Otherwise, if $i > 1$ and the number of intervals in M_{i-1} is at least 2, then search for the smallest integer $s_i > s_{i-1}$, such that the ciphertext $c(s_i)^e \bmod N$ is PKCS conforming.

Step 2.c: Searching with one interval left. Otherwise, if M_{i-1} contains exactly one interval (i.e. $M_{i-1} = \{[a, b]\}$), then choose small integer values r_i, s_i such that

$$r_i \geq \lceil 2(bs_{i-1} - E)/N \rceil$$

and

$$(E + r_iN)/b \leq s_\mathrm{i} < (F + r_iN)/a$$

until the ciphertext $c(s_i)^e \bmod N$ is PKCS conforming.

Step 3: Narrowing the set of solution. After s_i has been found, the set M_i is computed as

$$M_i \leftarrow \cup_{(a,b,r)} \{[\max (a, \lceil (E + rN)/s_i \rceil), \min (b, \lfloor (F + rN)/s_i \rfloor)]\}$$

for all $[a, b] \in M_{i-1}$ and $(as_i - F)/N \leq r \leq (bs_i - E)/N$.

Step 4: Computing the solution. If M_i contains only one interval of length 1 (i.e., $M_i = \{[a, a]\}$), then set $m \leftarrow a(s_0)^{-1} \bmod N$, and return m as solution of $m \equiv c^d \pmod N$. Otherwise, set $i \leftarrow i + 1$ and go to step 2.