A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0

James Manger

Telstra Research Laboratories,
Level 7, 242 Exhibition Street, Melbourne 3000, Australia
James.H.Manger@team.telstra.com

Abstract. An adaptive chosen ciphertext attack against PKCS #1 v2.0 RSA OAEP encryption is described. It recovers the plaintext – not the private key – from a given ciphertext in a little over $\log_2 n$ queries of an oracle implementing the algorithm, where n is the RSA modulus. The high likelihood of implementations being susceptible to this attack is explained as well as the practicality of the attack. Improvements to the algorithm to defend against the attack are discussed.

Keywords: chosen ciphertext attack, RSA, OAEP, PKCS

1 Introduction

At CRYPTO '98 Daniel Bleichenbacher presented an adaptive chosen ciphertext attack against PKCS #1 v1.5 RSA block type 2 padding [1]. The attack needs roughly one million oracle queries to succeed for a 1024-bit RSA key. He concluded that RSA encryption should include an integrity check and that the phase between decryption and integrity verification is crucial, because any information leaking from this phase can present a security risk. Version 2.0 of PKCS #1 introduced a new algorithm RSAES- OAEP that uses Optimal Asymmetric Encryption Padding (OAEP) to counteract this attack [2][5]. It says, "a chosen ciphertext attack is ineffective against a plaintext-aware encryption scheme such as RSAES-OAEP". However, the design of RSAES-OAEP makes it highly likely that implementations will leak information between the decryption and integrity check operations making them susceptible to a chosen ciphertext attack that requires many orders of magnitude less effort than similar attacks against PKCS #1 v1.5 block type 2 padding. The attack needs roughly one thousand oracle queries to succeed for a 1024-bit RSA key.

Section 2 summarizes RSA Optimal Asymmetric Encryption Padding as defined in PKCS #1 v2.0 . [1] Section 3 describes a chosen ciphertext against this algorithm. Section 4 explores the practicality of the assumptions necessary for

[1] The same algorithm is standardized in IEEE 1363, where the relevant message encoding method for encryption is called EME1 [4]

J. Kilian (Ed.): CRYPTO 2001, LNCS 2139, pp. 230–238, 2001.

the attack to proceed. Section 5 discusses approaches for changing the algorithm or its implementation to prevent the attack and restore the intended security properties.

2 RSAES-OAEP

RSAES-OAEP encryption starts by encoding a seed, a hash, padding octets and the secret (typically a session key) into an octet string. Masking operations effectively randomize these octets before they are treated as the unsigned binary representation of an integer – the integer used in the RSA modular exponentiation operation. The number of padding octets is chosen so that the encoding consumes one less octet than required for a unsigned binary representation of the modulus. This ensures the integer is less than the modulus as required in RSA. Alteratively, the encoded messages can be considered as an octet string the same length as the modulus, but with the most significant octet set to '00'h.

Figure 1 shows the RSAES-OAEP decryption and decoding process. The ciphertext is converted to the plaintext by modular exponentiation with the private exponent followed by integer-to-octet translation. A mask generation function (MGF) uses the least significant portion of the plaintext to unmask the seed. A mask generated from the seed unmasks a hash, padding and the confidential message. The integrity of the ciphertext is verified by comparing the unmasked hash to an independently calculated hash of the parameters (and by checking the padding).

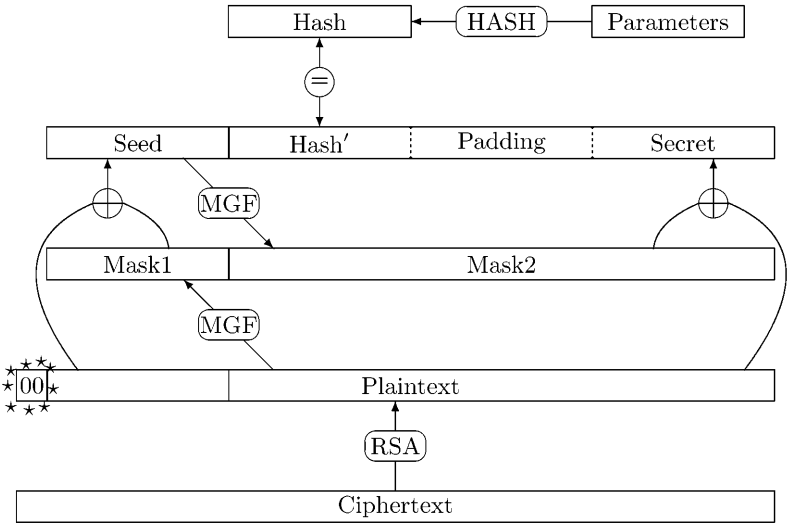

Fig. 1. RSAES-OAEP Decoding

After the private key operation the decryption operation can fail in the integer-to-octet translation (e.g. the integer is too large to fit in one fewer octets than the modulus) or in the OAEP-decoding (e.g. integrity check fails). In both instances PKCS #1 v2.0 says to output "`decryption error`" and stop.

3 Chosen Ciphertext Attack

Let n be an RSA modulus, with e and d the public and private exponents respectively. Let $k = \lceil \log_{256} n \rceil$ be the byte length of n and let $B = 2^{8(k-1)}$. [2]

Assume an attacker knows the public key (n, e) and has access to an oracle that for any chosen ciphertext x indicates whether the corresponding plaintext $y \equiv x^d \pmod{n}$ is less than B or not — returning "$y < B$" or "$y \geq B$". For the last assumption to hold it is sufficient for the oracle to distinguish a failure in the integer-to-octets conversion (in which case "$y \geq B$" is returned) from any subsequent failure, e.g. of the integrity check.

The attacker wishes to determine the plaintext $m \equiv c^d \pmod{n}$ corresponding to a captured ciphertext c. The basic step is to choose a multiple f and send $f^e \cdot c \pmod{n}$ to the oracle. This ciphertext corresponds to the plaintext $f \cdot m$. [3] The oracle indicates if this is in the range $[0, B)$ or (B, n) modulo n, thus providing a mathematical relationship about m that reduces the range (or ranges) in which it must lie. The aim is to reduce this range with successive oracle queries until just one value is left — m.

The approach of the attack described in this paper is to choose values of f such that the range where $f \cdot m$ could lie spans exactly one boundary between a region where $f \cdot m < B \pmod{n}$ and a region where $f \cdot m \geq B \pmod{n}$. The oracle response narrows the range to one of these regions.

Initially we know $m \in [0, B)$, as all valid messages are in this range by construction. One point to note is that since $m < B$ there is always a multiple of m that lies in any region of width B. For instance, for any integer i there is always some integer f such that $f \cdot m \in [in, in + B)$.

The following attack assumes $2B < n$. This assumption will usually be satisfied as RSA moduli are typically chosen to be exact multiples of 8 bits long making n between 128 and 256 times larger than B. Situations where this assumption does not hold are discussed toward the end of this section.

Step 1: Try multiples of $2, 4, 8, \ldots 2^i, \ldots$ in turn until the oracle returns "$\geq B$". For each multiple f_1 the possible values of $f_1 \cdot m$ span a single boundary point at B.

1.1 We know $m \in [0, B)$. Let $f_1 = 2$.
1.2 So $f_1 \cdot m \in [0, 2B)$. Try f_1 with the oracle, i.e. send $f_1^e \cdot c \pmod{n}$.

[2] Any number less than B encoded into k octets will start with a '`00`'h octet.
[3] $(f^e \cdot c)^d \equiv f^{ed} \cdot c^d \equiv f \cdot m \pmod{n}$

1.3a If the oracle indicates "$< B$":
This implies $f_1 \cdot m \in [0, B)$, so $2f_1 \cdot m \in [0, 2B)$.
Set $f_1 \leftarrow 2f_1$ and go back to step 1.2.

1.3b If the oracle indicates "$\geq B$":
This implies $f_1 \in [B, 2B)$ for a known (even) multiple f_1. Rephrasing this gives $\frac{f_1}{2} \cdot m \in [\frac{B}{2}, B)$ for a known multiple $\frac{f_1}{2}$. Now move to the next step.

Step 2: Start with a multiple f_2 such that $f_2 \cdot m$ is just less than $n + B$ for the maximum possible m. Keep increasing this multiple until the oracle returns "$< B$". For each multiple f_2 the possible values of $f_2 \cdot m$ span a single boundary point at n.

2.1 We have $\frac{f_1}{2} \cdot m \in [\frac{B}{2}, B)$. Let $f_2 = \lfloor \frac{n+B}{B} \rfloor \cdot \frac{f_1}{2}$.

2.2 So $f_2 \cdot m \in [\frac{n}{2}, n + B)$. Try f_2 with the oracle.

2.3a If the oracle indicates "$\geq B$":
This implies $f_2 \cdot m \in [\frac{n}{2}, n)$, so $(f_2 + \frac{f_1}{2}) \cdot m \in [\frac{n}{2}, n + B)$.
Set $f_2 \leftarrow f_2 + \frac{f_1}{2}$ and go back to step 2.2.

2.3b If the oracle indicates "$< B$":
This implies $f_2 \cdot m \in [n, n + B)$ for a known multiple f_2. Now move to the next step.

As f_2 increases at iterations through step 2.3a the lower bound on $f_2 \cdot m$ increases, eventually exceeding n when $f_2 = \lceil \frac{2n}{B} \rceil \cdot \frac{f_1}{2}$. Branch 2.3b must occur at or before this multiple. That is, step 2 will always terminate — taking at most $\lceil \frac{n}{B} \rceil$ oracle queries.

Step 3: Try multiples f_3 that give a range for $f_3 \cdot m$ about $2B$ integers wide and spanning a single boundary point. Each oracle response will half the range back to a width of about B integers, so the next multiple is approximately twice the previous value.

3.1 We have $f_2 \cdot m \in [n, n + B)$.
Rephrasing, we have a multiple f_2 and a range $[m_{\min}, m_{\max})$ of possible m values, where $m_{\min} = \lceil \frac{n}{f_2} \rceil$, $m_{\max} = \lfloor \frac{n+B}{f_2} \rfloor$ and $f_2 \cdot (m_{\max} - m_{\min}) \approx B$.

3.2 Choose a multiple f_{tmp} such that the width of $f_{\text{tmp}} \cdot m$ is approximately $2B$.
$f_{\text{tmp}} = \lfloor \frac{2B}{m_{\max} - m_{\min}} \rfloor$. This value is about double the previous multiple.

3.3 Select a boundary point, $in + B$, near the range of $f_{\text{tmp}} \cdot m$.
$i = \lfloor \frac{f_{\text{tmp}} \cdot m_{\min}}{n} \rfloor$.

3.4 Choose a multiple f_3 such that $f_3 \cdot m$ spans a single boundary point at $in + B$.
$f_3 = \lceil \frac{in}{m_{\min}} \rceil$. This gives $f_3 \cdot m \in [in, in + 2B)$ (though the upper bound is only approximate). f_3 is approximately equal to f_{tmp}. Try f_3 with the oracle.

3.5a If the oracle indicates "$\geq B$":
This implies $f_3 \cdot m \in [in + B, in + 2B)$.
Set $m_{\min} \leftarrow \lceil \frac{in+B}{f_3} \rceil$ and go back to step 3.2.

3.5b If the oracle indicates "$< B$":
This implies $f_3 \cdot m \in [in, in + B)$.
Set $m_{\max} \leftarrow \lfloor \frac{in+B}{f_3} \rfloor$ and go back to step 3.2.

Each answer from the oracle in step 3 selects either the top or bottom half (approximately) of the $f_3 \cdot m$ range, halving the range of possible m values. Eventually the range in which m lies narrows to a single number, which is the desired plaintext. At this point $f_3 \approx B = 2^{8(k-1)}$.

The description of step 3 above does not provide a proof that those particular choices of multiples, boundary points and interval widths will always work for any key or message. Minor variations on these choices can make the attack algorithm marginally more efficient. See [1] for a more mathematically rigorous analysis of a closely related problem.

3.1 Complexity

Steps 1 and 3 approximately halve the range of possible m values with each iteration so between them they take about $\log_2 B = 8(k-1)$ oracle queries. [4] Step 2 takes at most $\lceil \frac{n}{B} \rceil$ oracle queries (which must be ≤ 256), and half this number on average.

RSA moduli are typically chosen to be exact multiples of 8 bits long, e.g. 1024-bit moduli are far more prevalent than, say, 1021-bit moduli. Hence, for typical keys $\lceil \frac{n}{B} \rceil$ is in the range (128, 256], so step 2 will typically take on the order of 100 oracle queries.

For a 1024-bit RSA key the attack requires about 1100 oracle queries, for a 2048-bit key about 2200.

3.2 When $n < 2B$

The attack procedure described above assumes $2B < n$. If this is not the case, an indication from the oracle of "$< B$" when $f = 2$ narrows the range in which $f \cdot m$ lies not to a single region, but to a pair of regions: $f \cdot m \in [0, B) \bigcup [n, 2B)$. The range in which m is known to lie is reduced is total size, but is no longer confined to a single interval. This somewhat complicates the decision about which multiples to try but an adaptive chosen ciphertext attack will still work. The chosen ciphertext attack against RSA block type 2 padding had a similar issue — see [1] for a full analysis.

3.3 Comparison to the RSA Block Type 2 Attack

Analysis in [1] of the number of oracle queries required for a chosen ciphertext attack found an expression with two terms: the first term inversely proportional

[4] Reduction of the range of possible m values in step 2 slightly reduces the number of oracle queries required during steps 1 and 3, but this number also slightly increases (by a few percent) as the ranges in step 3 not being exactly centred on boundary points.

to the probability that a random integer from $[0, n)$ conforms to the encoding format; the second term proportional to $\log_2 n$. The first term dominates for RSA block type 2 padding (making the number of required queries quite dependent on various implementation issues, i.e. how the encoding format is checked). For RSAES-OAEP the first term corresponds to the number of oracle queries in step 2, which is an order of magnitude less than the second term.

4 Likelihood of Susceptibility

The chosen ciphertext attack described in the previous section starts with an assumption that the attacker can distinguish a failure in the integer-to-octets conversion from any subsequent failure, e.g. of the integrity check during OAEP-decoding. PKCS #1 v2.0, however, recognizes this risk by explicitly stating "it is important that the error messages output in steps 4 [integer-to- octets conversion] and 5 [OAEP decoding] be the same". [5] This section investigates why, in spite of this statement, it is likely that many RSAES-OAEP implementations will be susceptible to chosen ciphertext attack.

4.1 Spelling

Simply misspelling a word, including a full-stop or starting with a capital letter at one point is sufficient to distinguish two error messages that are otherwise the same. Having to relying for security on the absence of any such trivial occurrence in an implementation should not be necessary.

4.2 Logs

Even when a system avoids revealing error details in, say, its protocol response it is likely to reveal more detailed error descriptions in its logs. [6] "`Integer too large`" and "`decoding error`" – included in PKCS #1 v2.0 as error messages from sub-routines used by RSAES-OAEP – are just the sort of details a log may contain yet their presence is sufficient for the attack to proceed. Requiring access to system logs clearly lessens the risk of an attack but it is still an attack that must be considered. Logs are typically available to a much larger set of people than have direct access to a private key and logs will be given less protection (and should not be required to have the same protection as a private key).

[5] PKCS #1 v2.0, section 7.1.2 *Decryption operation*, last paragraph.

[6] Divulging less detail and only very general error indications is a well-known security technique, but it does come at a cost. Less information for an attacker also means less information for developers, support staff and users to understand the state of a system and respond appropriately.

4.3 Other Error Conditions

There are many possible errors that are not mentioned in the definition of RSAES-OAEP in PKCS #1 v2.0. This seems sensible as most are implementation issues but it becomes problematic when, due to the algorithm's design, these errors can have serious security implications. Consider what could happen when an unsupported mask generation function (MGF) is specified (by the attacker, along with his chosen ciphertext). Though not explicitly considered in PKCS #1, some sort of error must result, say "`unsupported algorithm`", and it may not be detected until the MGF is first used – in the OAEP-decoding stage. Any indication that the OAEP-decoding stage has been reached, however, is sufficient for the attack to proceed as it implies the previous integer-to-octet conversion stage was successful, i.e. plaintext $< B$.

4.4 Timing

Even identical error responses can be distinguished if they take different amounts of time to occur. For instance, detecting an integrity error during OAEP-decoding takes at least the time of two mask generation operations longer than detecting an error in the integer-to-octet conversion. Though this time difference may be small compared to the total response time (e.g. the modular exponentiation is likely to take much longer) it is still likely to be measurable, even if extra oracle queries and statistical analysis have to be employed.

RSAES-OAEP offers an even bigger target for a timing attack. The integrity check compares a hash from the OAEP-decoding to a locally calculated hash of the parameters. The parameters can be an octet string of arbitrary length chosen by the attacker. The hash is only needed in the OAEP-decoding stage and it is reasonable to assume many implementations would calculate it during this stage (as the standard suggests), but this point is after the integer-to-octet conversion. An attacker can achieve whatever time difference he or she requires to distinguish the relevant error sources by using a sufficiently large octet string for the parameters — set the parameters to be 10MB long and do the attack with a wristwatch.

This use of the hash operation to attack RSAES-OAEP illustrates the algorithm's fragile nature. The hash does not involve the private key or the secret in the plaintext at all, so even a diligent implementer is unlikely to expect its operation to impact the security. Performing the hash operation before the integer-to-octet conversion eliminates its usefulness in a timing attack.

4.5 Summary

An algorithm that relies on identical responses to errors (despite their disparate sources), no access to logs, a specific (undocumented and not obvious) order of sub-tasks and attention to timing must be considered quite fragile. Though it is possible some implementations of RSAES-OAEP will be immune, it is quite likely that many others will be susceptible to the chosen ciphertext attack described in

this paper. To some degree RSAES-OAEP achieves security through obscurity — obscurity of the source of errors, of implementation details and of timing information. Obscurity, however, is widely recognized as a poor principle for designing an algorithm.

5 Directions towards a Solution

The attack relies on distinguishing different actions of the oracle resulting from a decision about the structure of the plaintext. This suggests two possible approaches for a solution: ensure the actions are indistinguishable; or avoid any decision based on the structure of the plaintext. The former approach uses obscurity to achieve security, while the latter approach offers better hope of reducing the security dependence on seemingly innocuous implementation choices.

PKCS #1 v2.0 makes a basic effort at obscurity by outputting the same error message for all identified errors. PKCS #1 v2.1 draft 2 enhances this effort by noting that errors from integer-to-octet conversion and OAEP-decoding must be indistinguishable and, importantly, that execution time must not reveal which error occurred [3]. [7]

A naive solution for avoiding a decision about the structure of the plaintext is to simply ignore its structure, i.e. ignore its most significant octet (after converting integer m to k octets). Ignoring this octet during decryption allows it to be set to any value (e.g. a random value) during encryption (subject to the restriction $m < n$). As it stands, however, this is not a good solution because these modifications mean the algorithm is no longer plaintext-aware — destroying the security proof that OAEP offered. An operation on a ciphertext that only altered the most significant octet of the corresponding plaintext would produce a different, but still valid, ciphertext without requiring knowledge of the plaintext. How to perform such an operation is an open question (at least to the author), as is the question of how such an ability would affect security in practice.

Another open question is how to modify RSAES-OAEP to eliminate the last vestige of structure from the plaintext, yet retain a proof of its security against chosen ciphertext attack in the random oracle model. Not only would such a solution avoid decisions based on plaintext structure – it would ensures no such decision could reasonably be made (even inadvertently) as there is no structure upon which to make it. [8]

5.1 Best Practise

Though the check that $m < B$ is the basis of the attack, it is other details (such the time a hash operation takes) that allow the attack to proceed. This rein-

[7] PKCS #1 v2.1 draft 2, section 7.1.2 *[RSAES-OAEP] Decryption operation*, see the note at the bottom of page 18.

[8] Such an inadvertent decision (i.e. a software bug) has been noticed by the author in one RSAES-OAEP implementation. It never explicitly checked if the plaintext "`integer [was] too large`", but just assumed it would fit in $k - 1$ octets and suffered buffer overflow problems when this was not the case.

forces Bleichenbacher's conclusion that the "integrity check must be performed in the correct step of the protocol – preferably immediately after decryption" [1]. Moving any processing that does not have to occur between the decryption and integrity check to another location is a practical step towards satisfying this criterion, hence lessening the exposure to chosen ciphertext attacks (though it does not, by itself, eliminate the threat). Processes that could be performed before the decryption operation in RSAES-OAEP include hashing the parameters, confirming relevant MGF and hash algorithms are supported and allocating memory required during mask generation and OAEP-decoding. Rearranging these processes should occur in implementations and also in standards defining algorithms, as the latter are the specification from which implementations are built.

6 Conclusion

Optimal Asymmetric Encryption Padding adds an integrity check and masks the structure of the message being encrypted to achieve plaintext-awareness and consequent protection against chosen ciphertext attack. However, translating the octet-aligned OAEP process into integers modulo n in RSAES-OAEP reintroduced sufficient structure to make an adaptive chosen ciphertext attack possible, with a high likelihood, in many implementations.

Acknowledgements. I thank the Director of Research, Telstra Research Laboratories, for supporting this work. I also thank the reviewers of this paper for highlighting the risks of simply ignoring the structure in RSAES-OAEP.

References

1. D. Bleichenbacher: Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In Hugo Krawczyk (ed.), *Advances in Cryptology – CRYPTO '98*, pages 1–12, Berlin, Springer, 1998 (Lecture Notes in Computer Science, vol. 1462).
2. PKCS #1 v2.0: *RSA Cryptography Standard*, 1 October 1998. `http://www.rsasecurity.com/rsalabs/pkcs/`
3. PKCS #1 v2.1 draft 2: *RSA Cryptography Standard*, 5 January 2001. `http://www.rsasecurity.com/rsalabs/pkcs/`
4. IEEE 1363 draft 13: *Standard Specifications for Public Key Cryptography*, 12 November 1999. `http://grouper.ieee.org/groups/1363/`
5. M. Bellare and P. Rogaway: Optimal Asymmetric Encryption Padding — How to Encrypt with RSA. In *Advances in Cryptology — EUROCRYPT '94*, pages 92–111, Springer-Verlag, 1994.