RSA Padded Oracle Attack

JP Scaduto, Will Bowditch

Department of Computer Science, Boston College

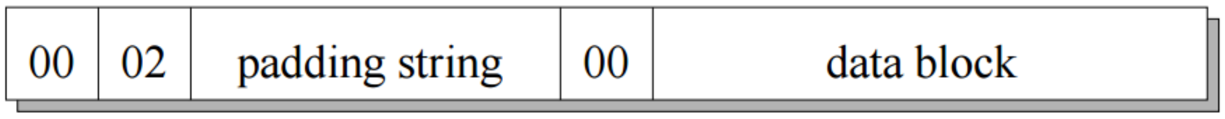
Abstract

This project reproduces a chosen cipher-text attack against an RSA cryptosystem with PKCS 1.5 padding. The reference here is the original paper ”Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS1” by Daniel Bleichenbacher. We show that an RSA private-key operation can be performed when the server informs the attacker that a message has been padded incorrectly prior to encryption. After implementing the algorithm explained by Bleichenbacher in Python, the paper discusses the advantages of threading the search for the modifier s' in order to lower the the attack's runtime.

Introduction

Let n, e be an RSA public key, and let d be the corresponding secret key. The attacker has access to an oracle such that for any chosen cipher-text c, the oracle indicates whether the corresponding plain-text $c^d\ mod(n)$ has the correct PKCS 1.5 padding. We assume that the attacker has an oracle that for each time a ciphertext is sent to the oracle, it returns whether the decrypted version of the ciphertext is conforming to the proper padding or not. This allows us to get part of the corresponding correctly padded plaintext to the corresponding cipher-text. PKCS 1.5 padding is implemented in the following manner. First, a padded string p is generated randomly with non-zero numbers, with a byte length of at least 8. Given the plain-text m, the padded message in base hex is initialized as $00|02|p|00|m$, then converted to an integer such that the encrypted text c is equivalent to $x^e\ mod\ n$. The oracle attacked for this project confirms whether, after decryption, the two high-order bytes are 00 02. We show that we can use this oracle to compute $c^d\ mod\ n$ and thus retrieve the plain-text m for any chosen integer c.

Figure 1: PKCS 1.5 padding format.



Implementation

We now will now explain the attack. The attacker chooses an integer s, and computes $c' \equiv sc^e \ mod\ n$ to send to the oracle. If the oracle says that c' has correct padding, then we know that the first two bytes of ms are 00 and 02. Let k be the length of public key n in bytes, and $B = 2^{8(k-2)}$. If ms is conforming, then we know that $2B \leq ms$ $mod\ n \leq 3B$.

The attack can be divided into four steps. In the first step, called Blinding, the cipher-text c_0 is found corresponding to the unknown message m_0 by searching for a correctly padded ciphertext while $c(s_0)^e$ mod n. If you begin with a ciphertext that is known to be correctly padded, such as we were in the given problem with the correctly padded ciphertext, you can skip step 1 by setting $s_0 = 1$. When this initialization is achieved, you can set c_0 to $c(s_0)^e$ mod n, the set M_0 to $[(2B, 3B-1)]$ and $i = 1$.

The second step is divided into 3 cases, one of which is carried out each loop through, depending on the situation. In general, the attacker tries to find small values s_i such that $c'(s_i)^e$ mod n is correctly padded. For each successful value for s_i, the attacker computes, using previous knowledge about the plain-text m, a set of intervals that must contain the plain-text.

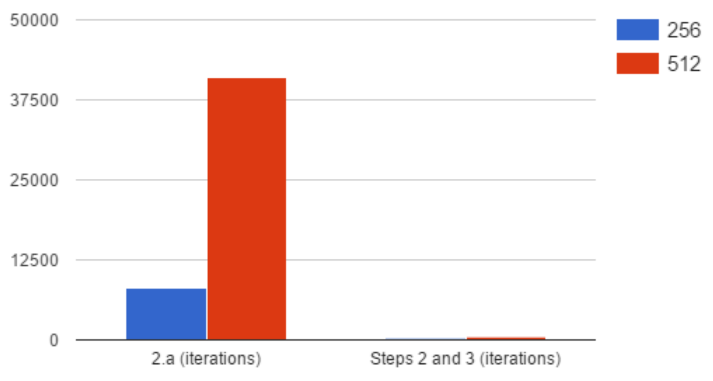
For the first case 2.a) when $i = 1$ you begin the search for s_1 such that $c_0(s_1)^e$ mod n is correctly padded. The bounds begin at $s_1 > n/3B$ and increment $s_1 = s_1 + 1$ until a proper s_1 is found. This step was found to be the most time consuming step to find the initial s_1. The condition for the next case for step 2 is if $i > 1$ and M_{i-1} is at least 2 elements.

If these conditions are satisfied step 2.b) is carried out by searching for the integer $s_i > s_{i-1}$ so that $c_0(s_i)^e$ mod n is correctly padded. The last case, 2.c, is if M_{i-1} has only one interval left. In this case we choose the integer r_i such that $r_i > 2\frac{bs_{-1}-2B}{n}$ which we then use to find s_i within the newly defined range $\frac{2B+r_in}{b} \leq s_i < \frac{3B+r_in}{a}$ in which a and b are the elements in the single range in M_{i-1}. After establishing the range for s_i you can iterate through as many possible i until $c_0(s_i)^e$ mod n is conforming to the specified padding.

The third steps begins after the s_i is chosen from one of the three possible step 2 conditions. The purpose of this step is to narrow down the set of possible solutions M_i. $M_i = U\left\{[max(a\frac{2B+rn}{s_i}), min(b\frac{3B-1+rn}{s_i})]\right\}$ while r is defined as $\frac{as_i-3B+1}{n} \leq r \leq \frac{bs_i-2B}{n}$ for all [a,b] in M_{i-1}.

As s_i increases the ranges [a,b] gradually decrease. When there is only one interval left in the set following step 3 and the length of the interval is 1, so $M_i = \{[a,a]\}$ then plaintext m, is $a(s_0)^-1$ mod n and m can be returned as the solution to $m \equiv c^d$. If there is more than one range in M_i then the attacker is directed back to the top of the loop at step 2.

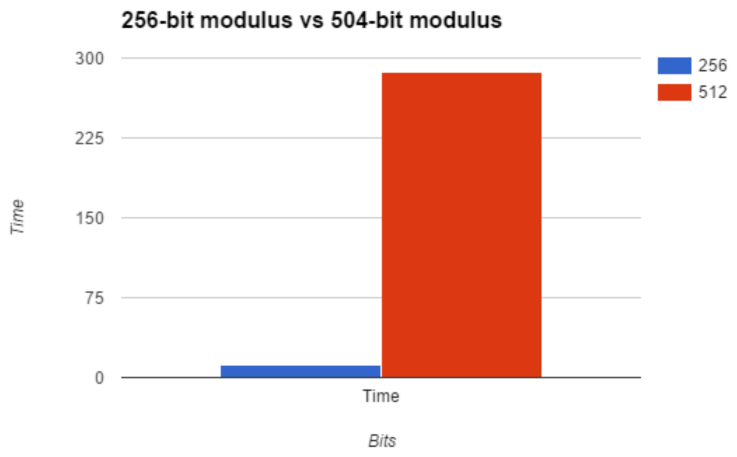
Figure 2: Number of iterations required for each step of Bleichenbachers' algorithim.



Results and Improvements

The results of the final algorithm presented results for both 256 bit keys and 512 bit keys. The Plaintext recovered from the cipher-text with the small modulus was "Get me out of here!" and the plain-text recovered from the 512 bit plain-text was "Help! I'm a prisoner in a padded cell." For the 256 bit key, part 2.a, the most time consuming part found a s_i to correctly pad it in about 8028 iterations. Overall to find the full plain-text, it was required for the steps two and three to iterate 226 times, with a time duration of about 11.8 seconds. For the 512 bit key, part 2.a took about 41016 iterations and steps two and three required 482 steps, and 286.6 seconds overall.

Figure 3: Time comparison between 256-bit modulus and 504-bit modulus for RSA oracle attack.



When beginning to forge the algorithm from Bleichenbachers paper, we used integer division through all of the algorithm and after running it, it ran slowly to the point that it took tens of minutes to carry out fully on the 256 bit run. Instead, by using the built in *divmod* function provided in python, we were able to carry out the divisions much quicker making the local 256 bit key runs on average 11.14 seconds with a different cipher-text start every time. The 512 bit keys took about 27.6 seconds on average with a different cipher-text each run.

Also, another way we tried to increase efficiency and run speed is by implementing the attack in C using threading search. By using a threading method in C we aimed to minimize the time it took to find the s_1 in step 2.a which was the most time consuming step to find the first initial $c_0(s_1)^e$ mod n with correct padding. However, this require a padding oracle implemented in C which due to time constraints we could only implement the threaded number search that iterates through a range of numbers given to it (instructions in the header of file ”threadPad.c”). This function can cut the runtime of the searching steps by using the threading.

Conclusion

Since the initial search for s' could hypothetically run forever, it is hard to determine the time-complexity of this algorithm. We will now approximate the probability $P(P)$ that a randomly chosen integer is conforming. Let $P(A)$ be equal to the probability that the first two bytes of a randomly chosen integer are 00 and 02. Since our modulus n is less than $2^{16}B$ and greater than $2^{8}B$, we can denote that $2^{-16} < P(A) < 2^{-8}$. The probability that the padding block p, defined in the introduction, contains at least 8 non-zero bytes followed by a zero byte is $\frac{255}{256}^{8} \times (1-(\frac{255}{256}^{k-10}))$. Assuming a modulus n of 512 bits, we have $0.18 < P(P|A) < 0.97$, and therefore $0.18 \times 2^{-16} < P(P) < 0.97 \times 2^{-8}$, a time-feasible probability of finding a conforming integer.

By implementing Bleichenbachers algorithm in Python, we were able to demonstrate an irrefutable security vulnerability in any website or services that uses PKCS 1.5 padding with an oracle that signals to the user correct or incorrect padding. This vulnerability allows an attacker to decrypt cipher-text in a reasonable amount of time without expensive equipment or software.

Instructions for Running Files

The files padAttack.py and padAttackLocal.py can be run by using python in the command line by running: ”python padAttack.py”. You will then be prompted to enter 256 or 512 to establish which attack you would like to carry out. The attacks will then run and return the message and how long it took to preform. For threadPad.C you can compile with cc and ignore any warnings. To use the file after compilation you can then use ”./a.out 0 50000” which is the executable followed by the lower and upper bound of the search. The files rsa-po-tools.py and conversions.py were supplied to us and are used in supplement to the given files not to be run alone.

References

[1] Bleichenbacher, Daniel *Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS 1.* http://archiv.infsec.ethz.ch/education/fs08/secsem/bleichenbacher98.pdf