Hindawi
Security and Communication Networks
Volume 2019, Article ID 6013846, 11 pages
https://doi.org/10.1155/2019/6013846

Research Article

The Prediction of Serial Number in OpenSSL's X.509 Certificate

Jizhi Wang [1,2,3,4]

[1] *Institute of Information Engineering, Chinese Academy of Sciences, China*
[2] *School of Cyber Security, University of Chinese Academy of Sciences, China*
[3] *Shandong Provincial Key Laboratory of Computer Networks, Shandong Computer Science Center (National Supercomputer Center in Jinan), Shandong Academy of Sciences, China*
[4] *School of Cyber Security, Qilu University of Technology, China*

Correspondence should be addressed to Jizhi Wang; wangjzh@sdas.org

Received 14 November 2018; Accepted 25 March 2019; Published 2 May 2019

Academic Editor: A. Peinado

In 2007, a real faked X.509 certificate based on the chosen-prefix collision of MD5 was presented by Marc Stevens. In the method, attackers needed to predict the serial number of X.509 certificates generated by CAs besides constructing the collision pairs of MD5. After that, the randomness of the serial number is required. Then, in this case, how do we predict the random serial number? Thus, the way of generating serial number in OpenSSL was reviewed. The vulnerability was found that the value of the field "not before" of X.509 certificates generated by OpenSSL leaked the generating time of the certificates. Since the time is the seed of generating serial number in OpenSSL, we can limit the seed in a narrow range and get a series of candidate serial numbers and use these candidate serial numbers to construct faked X.509 certificates through Stevens's method. Although MD5 algorithm has been replaced by CAs, the kind of attack will be feasible if the chosen-prefix collision of current hash functions is found in the future. Furthermore, we investigate the way of generating serial numbers of certificates in other open source libraries, such as EJBCA, CFSSL, NSS, Botan, and Fortify.

1. Introduction

Digital certificates are adopted widely in Internet, which is a basic security measurement. Many principals, such as clients and servers, depend on digital certificates to authenticate each other. If an attacker can forge other's digital certificate, he/she may impersonate other's identity and access sensitive information. This is one of serious threats for the public.

The security of digital certificates is based on the digital signature algorithms and hash algorithms. If an attack against these algorithms occurs, the digital certificates based on these algorithms cannot be trusted any more. Among attacks, collision of hash algorithms is one of the most serious threats. Since the first real MD5 collision attack was presented by Wang [1, 2] in 2004, it is possible to construct forged certificates based on the collision attack of MD5.

At Eurocrypt 2007, the different certificates with the same signature were created firstly by Stevens based on the chosen-prefix collision attack of MD5 [3–5]. This was a big event for commerce CAs and their users because the kind of forged certificates can be verified successfully. After that, many companies announced that MD5 was vulnerable to digital certificates, such as Verisign, Microsoft, Mozilla, TC TrustCenter, RSA, US-CERT, and Cisco [6]. In addition, the super-malware Flame was discovered in 2012 [7], which uses the method to forge a Microsoft's certificate [8].

The method of Stevens cannot forge a certificate from an existing certificate because the second preimage attack of MD5 is hard so far. The method needs to construct two certificates based on chosen-prefix collision attack of MD5 before submitting one of them to apply for a certificate to a CA. The implementation of the process has two key issues, one related to the collision pair construction of MD5 and the other to some fields controlled by CAs, such as serial number, in certificates, which attackers need to predict before submitting the application. Against the threat, Stevens gave two suggestions for CAs: one is to replace MD5 algorithm with other secure hash algorithms (such as SHA-256) because

chosen-prefix collision of other hash algorithms does not occur at present; the other is to add a sufficient amount of fresh randomness at the appropriate fields (such as serial number) in order to prevent attackers from predicting if MD5 cannot be replaced at once [5]. In the wild, however, many valid certificates still use MD5 [9]. In addition, we grabbed 180,000+ certificates from Internet, while 5000+ certificates are based on MD5, in other words 2.8% certificates.

In this paper, we will focus on whether the randomness of some fields in certificates is enough to prevent attackers from predicting. Since the detailed codes of business CAs are not public, we review the way of generating certificates by open source software OpenSSL to find how to predict the values of some fields in certificates. OpenSSL uses a pseudo random number generator (PRNG) to output random numbers. Some literatures related to the security of the PRNG have been proposed [10–15]. The security of OpenSSL's PRNG in Android and Debian has been reported in [10, 14]. A theory analysis of OpenSSL's PRNG was presented in [10]. However, it is not clear how the PRNG works in the procedure of generating X.509 certificates. Furthermore, we also investigated generating certificates in other open source libraries, like EJBCA, CFSSL, NSS, Botan, and Fortify.

In this paper, we have three contributions as follows:

(1) We find a vulnerability of OpenSSL that the field "not before" in certificates leaks the time of generating certificates, which is the seed of generating the field "serial number," so that it is possible to predict the value of "serial number."

(2) We give the predicting method for the field "serial number" and forge certificates based on the proposed method and Stevens's method.

(3) We investigate five other open source libraries and find similar vulnerability in two libraries, EJBCA and NSS.

The paper is organized as follows. In Section 2, some preliminaries are introduced and the problems solved by the paper are defined. Section 3 reviews the source codes of OpenSSL about generating X.509 certificates. Then, Section 4 proposes a method predicting the key fields of certificates. Some countermeasures are given in Section 5 and Section 6 investigates other open source libraries. Finally, Section 7 concludes the paper.

2. Preliminaries

In X.509 certificates, the signature of CA is the most important part to prevent from forging. Any modification of contents in certificates would make the change of CA's signature, in other words the change of Hash value. If a user A's certificate has existed, we cannot forge the certificate directly because it needs to construct the second preimage of hash value of the certificate. However, we can use other user B's identity to apply a certificate for CA, and generate a chosen-prefix collision pair, which can forge A's certificate.

Table 1: The overview of collision complexities.

MD5	
identical-prefix	chosen-prefix
2^{16} (2009 [4])	2^{39} (2009 [4])
SHA-1	
identical-prefix	chosen-prefix
2^{63} (2017 [17])	2^{77} (2012 [5])

2.1. Chosen-Prefix Collision Attack of MD5. According to the chosen-prefix collision, the prefixes p and p' of two message blocks are chosen. Then, the collision pair, s and s', is generated, so that $MD5(p\|s\|d) = MD5(p'\|s'\|d)$ is satisfied for any arbitrary suffix d. The two prefixes p and p' must be of equal length and their length is a multiple of the MD5 message block size. Otherwise, padding message must be added. The computing complexity of the attack is $O(2^{39})$ [4, 5] and a program was presented by Stevens [16]. For attackers, the method can be applied to forge certificates successfully.

Before that, identical-prefix collision had been studied, which is easier to be constructed than chosen-prefix collision. Although identical-prefix collision can be used to forge certificates, the kind of forgery is meaningless in practical attacks because the user's identity is in the prefix and cannot be changed.

The overview of collision complexities is in Table 1. We can see the chosen-prefix collision of MD5 is feasible in computing while the chosen-prefix collision of SHA-1 is unfeasible so far. But, in the near future, a real case of chosen-prefix collision of SHA-1 may be found, when the attack will be feasible.

2.2. X.509 Certificates. To forge a certificate, we need to know which part of certificate is as the prefix and which part of certificate the collision pair is placed on. The data structure of X.509 certificate is in Table 2.

According to the chosen-prefix collision attack, the generating collision pair is like random number, while only the field "subject public key info" is the analogy with random number. Thus, the collision pair constructed by chosen-prefix collision attack is placed in the field "subject public key info" (Table 2), and the fields from "version number" to "public key algorithm" of the certificate are as prefix chosen. Then an attacker must know the CA will chose which value to fill the fields in advance, because, before requiring the certificate for the CA, he/she must construct a collision pair and then submit the generated "public key info." Among these fields, the values of "serial number" and "not valid before" need to be forecast because they are controlled by CAs while others are easy to obtain.

2.3. The Procedure of Forging a Certificate. To forge A's certificate, we need to generate a chosen-prefix collision pair to construct two certificates, one of which is in the name of A and the other is in the name of B. Then, we submit B's identity

Table 2: The data structure of X.509 certificates.

field	comments	example
X.509 version number	Standard X.509	Version 3
Serial number	Chosen by CA	0x01000001
Signature algorithm identifier	Standard X.509	MD5withRSAEncryption
Issuer distinguished name	Identity of CA	CN="xxx CA" L="xxxxx" C="xx"
Not valid before	Controlled by CA	Jan.1,2017,00h00m01s
Not valid after	Controlled by CA	Dec.31,2017,23h59m59s
Subject distinguished name	Identity of users	CN="xxxxx" O="xxxx" L="xxx" C="xx"
Public key algorithm	Standard X.509	RSAEncryption
Subject public key info	Controlled by users	0x98765432...
Version 3 extensions	Standard X.509	...

and public key to the CA and get its signature. The signature of A's certificate is replaced, which can be verified successfully.

The flow of the forging a certificate is in Figure 1. Firstly, attackers chose a target CA. Before guessing the serial number and validity period in certificates, they need to collect/apply for enough certificates issued by the CA and look for whether the two fields have any patterns. If they find any, then the fields can be predicted. After constructing the collision pair based on chosen-prefix collision attack, attackers can submit one of the two to the CA and get its signature. If the guessed serial number and validity period are correct, it is successful! Otherwise, attackers would guess again.

In [4], Stevens reported that their targeted CA used sequential serial numbers and the validity period started exactly 6 seconds after a certification request was submitted. Thus they could predict the value of the fields easily. However, the attack becomes effectively impossible if the CA adds a sufficient amount of fresh randomness to the certificate fields, such as in the serial number. This randomness is to be generated after the approval of the certification request, so that if attackers cannot predict the value of these fields, they cannot construct the collision pair.

2.4. The Problem. Thus, in this paper, we try to answer the two questions:

(1) How do we predict the value of the field "serial number" if the CA chooses a random number as the serial number?

(2) How do we predict the value of the field "not valid before" that is in the unit of second?

To answer the two questions, we need to know how CAs generate the value of the two fields. However, the different CAs may adopt different ways to filling the fields. Since the open source software OpenSSL [18] is widely applied in generating X.509 certificates, we take it as an example to answer the two questions. For example, the open source PKI architecture OpenCA [19] is to call OpenSSL to generate X.509 certificates.

3. The Reviewing of OpenSSL

We use OpenSSL 1.1.0e to review how a certificate is generated. Before 0.9.8 of OpenSSL, MD5 was a default configuration for creating message digests [20], but after that MD5 is still supported because of compatibility.

3.1. OpenSSL's PRNG. *RAND_add()* and *RAND_bytes()* are the most important random number functions in OpenSSL.

(i) *RAND_add(void *buf, int n, double entropy)*: adds n bytes of buf into PRNG states.

(ii) *RAND_bytes(void *buf, int n)*: outputs n bytes of random number into buf.

The authors in [10–12] gave the algorithms of *RAND_add()* and *RAND_bytes()* as in Algorithms 1 and 2.

The input parameter md_0 of *RAND_add* is the IV of SHA1 algorithm. The parameter s is an array, whose initial values are zero, which is the internal states of the random number generator. The parameters p and q are location marks of array s, whose initial values are zero.

3.2. The Serial Number of X.509 Certificates. When we use OpenSSL to generate a X.509 certificate, there are two ways to generate the serial number. In the configure file of OpenSSL "openssl.conf" (Figure 2), the term "serial" is related to the serial number. If the file "serial" in the current directory exists, the serial number can be set up in the file; that is to say, we can designate a number as the serial number in the file. For example, if we input "01" into the file "serial," the serial number will be "01." In addition, after the certificate is generated, the number in the file "serial" will be plus one and then changed into "02." In other words, the serial number of

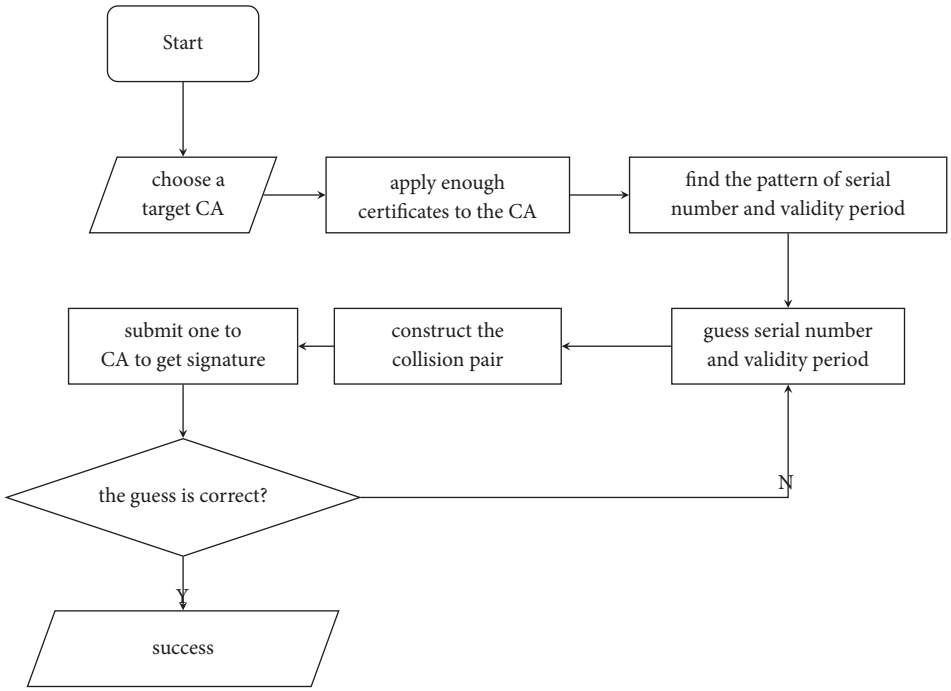

Figure 1: The flow of forging a certificate.

Input: n, b, where b is divided into 20-byte-length block bi
//entropy is 0 by default
//md is 20-byte states; s is 1023 bytes PRNG states

for i=1 to n do
  t=size(b_{i-1})-1
  md_i=SHA1($md_{i-1} \| s[p : p + t \bmod 1023] \| b_{i-1}$)
  s[p:p+t mod 1023]=s[p:p+t mod 1023] $\oplus md_i[0:t]$
  p=p+t+1 mod 1023
end for
$md_0 = md_i \oplus md_0$
q=min(q+size(b), 1023)
return md_0, s, p, q

Algorithm 1: The function of *RAND_add*. RAND_add(void *b, int n, double entropy).

Input: r, where r is divided into 10-byte-length blocks r_i
// r is defined and evaluated in the function
Output: b

for i=1 to n do
  $md_i = SHA1(md_{i-1} \| r_{i-1} \| s[p : p + 9 \bmod q])$
  $r_{i-1} = md_i[10 : 10 + size(r_{i-1}) - 1]$
  s[p:p+9 mod q]=s[p:p+9 mod q]$\oplus md_i[0 : 9]$
  p=p+10 mod q
end for
$md_0 = SHA1(md_i || md_0)$
return md_0, s, p, q

Algorithm 2: The function of *RAND_bytes*. RAND_bytes(void *b, int n).

```
[ ca ]
default_ca = myca

[ myca ]
dir =c:/openssl-1.1.0e
certificate = $dir/cacert.pem
database = $dir/index.txt
new_certs_dir = $dir/certs
private_key = $dir/private/cakey.pem
serial = $dir/serial

default_crl_days= 7
default_days = 365
default_md = md5

policy =myca_policy
x509_extensions =certificate_extensions
```

Figure 2: "openssl.conf": the configure file of OpenSSL.

the next certificate will be "02." Thus, we can forecast exactly the serial number because of the sequential serial numbers.

On the other hand, if the file "serial" does not exist, OpenSSL would use random number as the serial number of X.509 certificates. An example is in Figure 3. In this paper, we will discuss the prediction of the serial number in the way.

Reviewing the source code of OpenSSL, we can find it calls the function "*rand_serial (BIGNUM *b, ASN1_INTEGER *ai)*" in X509.c to generate the serial number (Figure 4).

After a serial of function calling, the functions "*RAND_add(const void *buf, int num, double add)*" and "*RAND_bytes(unsigned char *buf, int num)*" are called in bn_rand.c (Figure 5).

In the case, the parameter b of *RAND_add()* is "time_t" type of variable "tim," while the parameter r of *RAND_bytes()* is defined inside. We reviewed the source code of *RAND_bytes()* and found it is "FILETIME" type of variable "tv" in Figure 6. In addition, the parameter md_0 of *RAND_bytes()* depends on the "dummy seed" in Figure 6, whose value is 20 bytes of "." by default.

In summary, the serial number depends on two time variables "tim" and "tv," where "tim" is a 32-bit integer which records the number of seconds since 00:00:00 Jan. 1, 1970, and "tv" is a 64-bit integer which records the number of 100 nanoseconds since 00:00:00 Jan. 1, 1601, in Windows, while "tv" records the number of microseconds since 00:00:00 Jan. 1, 1970, in Linux. The two times are the current system time.

3.3. The Valid Time of X.509 Certificates. The valid time of X.509 certificate depends on two times: "not before" and "not after." The different time between "not before" and "not after" is the valid time.

In the source codes of OpenSSL, x509.c generates the content of a X.509 certificate (Figure 4), while the function

```
Input: x, startdate, enddate, days
Output: x

If(startdate==NULL)
   X509_gmtime_adj(s,0)
else
   ...
If (enddate==NULL)
   X509_time_adj_ex(s,days,0,0)
else
   ...
```

Algorithm 3: The function of *set_cert_time*. set_cert_time(X509 *x, const char *startdate, const char *enddate, int days).

"set_cert_time(X509 *x, const char *startdate, const char *enddate, int days)" is to set the valid time (Algorithm 3).

Since the parameter "startdate" is set as NULL when the function is called, the data field "not before" of certificates is set as the current time of system. The detail code is in X509_vfy.c by a serial of functions calling (Figure 7).

In Figure 7, "not after" is got by "not before" + "days," the parameter of set_cert_times(), because the "enddate" is set as NULL.

4. The Prediction of Serial Number

From above analysis, the serial number and "not before" depend on the system time when the certificate is generated in OpenSSL. In addition, the value of "not before" is the time when generating the certificate. Thus, we know some information of the seeds of the serial number.

4.1. Low Entropy Secret Leakage. In [10], Strenzke pointed that if the seed was in a low entropy state, the output of random number generator would leak the information of the seed, which was called low entropy secret leakage (LESL). Thus, an attack can try through all the possible seeds and generate the results according to his/her instance of the random number generator. If the resulting outputs are equal to the outputs of the real random number generator, then the attacker knows the used seed of the real random number generator.

The above serial number generator of X.509 certificates in OpenSSL is an example of LESL. The current time of the day in microseconds provides about 36 bits of entropy. However, since "not before" of certificates leaks the time in seconds, as the part of seeds of serial number, we can try every 100 nanoseconds (in Windows) or microseconds (in Linux) to find which seed is used. Thus, the entropy is lost, and only 20 bits (10^6). In the next subsection, we will make the entropy reduce to 10 bits (10^3).

4.2. Testing. In [4], authors reported that the validity period started exactly 6 seconds after a certification request was submitted. To verify the issue, we selected a commercial CA

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            b2:42:84:13:4f:91:d6:87
    Signature Algorithm: md5WithRSAEncryption
        Issuer: CN = My Test CA, ST = HZ, C = CN, emailAddress = test@cert.com,
O = Root Certification Authority
        Validity
            Not Before: Nov  6 06:35:09 2017 GMT
            Not After : Nov  5 06:35:09 2020 GMT
        Subject: CN = My Test CA, ST = HZ, C = CN, emailAddress = test@cert.com,
 O = Root Certification Authority
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:ce:52:d9:91:05:d4:2b:4e:66:9d:da:de:d1:6f:
                    c0:3b:a9:a9:d9:22:b1:6b:8d:58:84:23:96:18:66:
                    6c:0b:52:1d:a8:ac:25:ea:93:a4:07:46:2a:2f:6a:
                    6a:69:b7:3d:d3:4c:c9:fe:ce:d2:ad:8c:63:e4:44:
                    61:c2:23:56:04:04:6f:79:27:71:ef:19:3b:a7:f2:
                    e5:bd:08:6d:13:38:95:d6:69:23:34:69:58:ff:3c:
                    0f:6a:61:63:a1:22:42:af:33:36:fc:f0:b7:dd:9e:
                    ce:8d:ec:3b:b4:d5:d0:72:fd:20:b5:58:f2:3b:ae:
```

Figure 3: An example of X.509 certificate.

```
544         if ((x = X509_new()) == NULL)
545             goto end;
546
547         if (sno == NULL) {
548             sno = ASN1_INTEGER_new();
549             if (sno == NULL || !rand_serial(NULL, sno))
550                 goto end;
551             if (!X509_set_serialNumber(x, sno))
552                 goto end;
553             ASN1_INTEGER_free(sno);
554             sno = NULL;
555         } else if (!X509_set_serialNumber(x, sno))
556             goto end;
557
558         if (!X509_set_issuer_name(x, X509_REQ_get_subject_name(req)))
559             goto end;
560         if (!X509_set_subject_name(x, X509_REQ_get_subject_name(req)))
561             goto end;
562         if (!set_cert_times(x, NULL, NULL, days))
563             goto end;
564
565         if (fkey)
566             X509_set_pubkey(x, fkey);
567         else {
568             pkey = X509_REQ_get0_pubkey(req);
569             X509_set_pubkey(x, pkey);
570         }
```

Figure 4: Part of x509.c.

```
17  static int bnrand(int pseudorand, BIGNUM *rnd, int bits, int top, int bottom)
18  {
19      unsigned char *buf = NULL;
20      int ret = 0, bit, bytes, mask;
21      time_t tim;
22
23      if (bits == 0) {
24          if (top != BN_RAND_TOP_ANY || bottom != BN_RAND_BOTTOM_ANY)
25              goto toosmall;
26          BN_zero(rnd);
27          return 1;
28      }
29      if (bits < 0 || (bits == 1 && top > 0))
30          goto toosmall;
31
32      bytes = (bits + 7) / 8;
33      bit = (bits - 1) % 8;
34      mask = 0xff << (bit + 1);
35
36      buf = OPENSSL_malloc(bytes);
37      if (buf == NULL) {
38          BNerr(BN_F_BNRAND, ERR_R_MALLOC_FAILURE);
39          goto err;
40      }
41
42      /* make a random number and set the top and bottom bits */
43      time(&tim);
44      RAND_add(&tim, sizeof(tim), 0.0);
45
46      if (RAND_bytes(buf, bytes) <= 0)
47          goto err;
```

Figure 5: RAND_add() and RAND_bytes() are called in bn_rand.c.

```
287 /* time value for various platforms */
288 #ifdef OPENSSL_SYS_WIN32
289     FILETIME tv;
290 # ifdef _WIN32_WCE
291     SYSTEMTIME t;
292     GetSystemTime(&t);
293     SystemTimeToFileTime(&t, &tv);
294 # else
295     GetSystemTimeAsFileTime(&tv);
296 # endif
297 #elif defined(OPENSSL_SYS_VXWORKS)
298     struct timespec tv;
299     clock_gettime(CLOCK_REALTIME, &ts);
300 #elif defined(OPENSSL_SYS_DSPBIOS)
301     unsigned long long tv, OPENSSL_rdtsc();
302     tv = OPENSSL_rdtsc();
303 #else
304     struct timeval tv;
305     gettimeofday(&tv, NULL);
306 #endif
307
399         while (n > 0) {
400 #if MD_DIGEST_LENGTH > 20
401 # error "Please adjust DUMMY_SEED."
402 #endif
403 #define DUMMY_SEED "...................." /* at least MD_DIGEST_LENGTH */
404             /*
405              * Note that the seed does not matter, it's just that
406              * rand_add expects to have something to hash.
407              */
408             rand_add(DUMMY_SEED, MD_DIGEST_LENGTH, 0.0);
409             n -= MD_DIGEST_LENGTH;
410         }
411         if (ok)
412             stirred_pool = 1;
413     }
450         if (curr_time) {        /* just in the first iteration to save time */
451             if (!MD_Update(m, (unsigned char *)&curr_time, sizeof curr_time))
452                 goto err;
453             if (!MD_Update(m, (unsigned char *)&tv, sizeof tv))
454                 goto err;
455             curr_time = 0;
456             if (!rand_hw_seed(m))
457                 goto err;
458         }
```

Figure 6: The source code of RAND_bytes().

that provides personnel with free certificates. We used ten different E-mail addresses to apply to the CA for certificates. The submitting time was recorded and the value of "not before" was checked after receiving the certificate. We can find that the difference between the two times is 5 seconds fixed.

Obviously, according to the difference of the two times, attackers can control the time when a CA generates a certificate because the value of "not before" directly shows the time. Furthermore, the serial number depends on the time in seconds and in nanoseconds in OpenSSL (Figures 3 and 4). Then attackers know the time in seconds while not knowing the time in 100 nanoseconds. Thus, for attackers, to predict the serial number of certificates, a natural idea is to brute force every 100 nanoseconds in the second according to Algorithms 1 and 2. The computation complexity is $O(10^7)$. However, in real computer systems, can the timing precision be 100 nanoseconds? We test the parameter "tv" in Figure 4

```
ASN1_TIME *X509_gmtime_adj(ASN1_TIME *s, long adj)
{
    return X509_time_adj(s, adj, NULL);
}

ASN1_TIME *X509_time_adj(ASN1_TIME *s, long offset_sec, time_t *in_tm)
{
    return X509_time_adj_ex(s, 0, offset_sec, in_tm);
}

ASN1_TIME *X509_time_adj_ex(ASN1_TIME *s,
                            int offset_day, long offset_sec, time_t *in_tm)
{
    time_t t;

    if (in_tm)
        t = *in_tm;
    else
        time(&t);

    if (s && !(s->flags & ASN1_STRING_FLAG_MSTRING)) {
        if (s->type == V_ASN1_UTCTIME)
            return ASN1_UTCTIME_adj(s, t, offset_day, offset_sec);
        if (s->type == V_ASN1_GENERALIZEDTIME)
            return ASN1_GENERALIZEDTIME_adj(s, t, offset_day, offset_sec);
    }
    return ASN1_TIME_adj(s, t, offset_day, offset_sec);
}
```

Figure 7: The default value of "not before" is the current time of system.

Table 3: The testing result for timing precision.

Operation system	Timing Precision	Computation Complexity
Windows XP	10ms	$O(10^2)$
Windows 7	1ms	$O(10^3)$
Ubuntu 14.04	1ms	$O(10^3)$

in different operation systems. We installed three operation systems in the same computer (Intel Core i7 2GHz) and tested the time jumping. We can see that every time jumping is larger than 100 nanoseconds. The result is shown in Table 3.

From Table 3, we can see the computation complexity in reality is much smaller than the one in theory.

According to the above discussion, attackers can predict the serial number and "not before" of a certificate. To verify the conclusion, we use Algorithm 4 to predict the serial number and "not before."

In Windows XP, the time precision is 0x18730 100nanoseconds (=100144). So in Step 5, we select randomly a value of m; the success probability is 0.01; in other words, we submit the application more than 69 times; the success probability is more than 50%. In Ubuntu, the time precision is 0x3f0 microseconds (=1008). So the success probability is 0.001.

The testing result shows that the real serial number of the certificate is one of the candidate serial numbers that we predict (in Table 4).

5. Countermeasure

Since the value of "not before" leaks the time of certificates' generation, attackers can limit a narrow range of the seeds for generating serial numbers in OpenSSL. The problem shows that the entropy of the seed is too low, which cannot guarantee the randomness of serial numbers. Thus a natural idea is to add entropy of the seed. In Figure 4, a dummy seed is defined but it is a fixed 20 bytes ".". Obviously, if the seed is a variable secret, the entropy will be increased. This is the simplest method to deal with the problem.

The other idea is that the value of "not before" should be set a future time instead of the current system time. For example, the value can be set as 00:00:00 of the second day after the day of application. Thus, attackers cannot know the exact time when the certificate is generated.

```
        Date firstDate = new Date();

        // Set back startdate ten minutes to avoid some problems with wrongly set clocks.
        firstDate.setTime(firstDate.getTime() - (10 * 60 * 1000));

        Date lastDate = new Date();

        // validity in days = validity*24*60*60*1000 milliseconds
        lastDate.setTime(lastDate.getTime() + (validity * (24 * 60 * 60 * 1000)));
```

Figure 8: The valid time of certificates in EJBCA.

```

        // Serialnumber is random bits, where random generator is initialized with Date.getTime() when this
        // bean is created.
        byte[] serno = new byte[8];
        SecureRandom random;
        try {
            random = SecureRandom.getInstance("SHA1PRNG");
        } catch (NoSuchAlgorithmException e) {
            throw new IllegalStateException("SHA1PRNG was not a known algorithm", e);
        }
        random.setSeed(new Date().getTime());
        random.nextBytes(serno);
```

Figure 9: The serial number of certificates in EJBCA.

6. Other Libraries

We have investigated other open source libraries generating certificates, EJBCA [21], CFSSL [22], NSS [23], Botan [24], and Fortify [25], to find whether similar problems exist when generating serial numbers of certificates.

6.1. EJBCA. EJBCA is an open source PKI Certificate Authority software based on Java technology. We reviewed the source codes of EJBCA Community 6.10.1.2. In EJBCA, a tool called CertTool is provided to generate certificates, where is in *\ejbca\modules\cesecore–common\src\org\cesecore\util\CertTool.java*. We reviewed the file to find how the valid time and serial number of certificates are generated.

From Figures 8 and 9, we can conclude that the default value of "not before" is set as "current time - 10 minutes" (in milliseconds), and "not after" is set as "current time + 24 hours" (in milliseconds). The generation algorithm of "serial number" is "SHA1PRNG" and the seed is set as "current time" (in millisenonds).

Obviously, the problem of EJBCA is similar to OpenSSL. We can get "not before" of certificates easily, then know the seed of "SHA1PRNG," and predict the serial number.

6.2. CFSSL. CFSSL is an open source PKI/TLS toolkit developed by CloudFlare. We reviewed the source codes of CFSSL 1.2 in order to find how the valid time and serial number of certificates are generated.

From Figure 10, we can see that the default value of "not before" is set as "current time." The "serial number" is generated by the function "rand" in the package "crypto/rand" of Go. "rand.Reader" is a global shared instance of a cryptographically PRNG, which reads from /dev/urandom on Unix-like systems or from CryptGenRandom API on Windows systems; i.e., the seed of the PRNG is from operation systems.

Table 4: Forged certificate.

Field	Submitted Certificate	Forged Certificate
Serial number	cd e9 6e da bc a0 04 f4	cd e9 6e da bc a0 04 f4
Signature algorithm identifier	MD5withRSAEncryption	MD5withRSAEncryption
Issuer distinguished name	My Test CA	My Test CA
Not valid before	0x59c5905d	0x59c5905d
Not valid after	0x5f692add	0x5f692add
Subject distinguished name	My Tes1, tes1@cer1.com	My Tes2, tes2@cer1.com
Public key algorithm	RSAEncryption	RSAEncryption
Subject public key info (2048bit)	BE 4F C4 66 2B AB 69 FB B9 50 78 55 12 33 9C E3 25 7B B3 9A A4 2F D9 F6 C7 56 C9 9A 38 D8 08 5A 00 00 00 00 C8 4C B9 00 6F E2 2B E0 91 09 8F F6 9C EB 64 14 35 B3 01 47 DC FC C1 81 DD 96 93 9E 61 07 07 0E 3B 5F F7 C3 B8 FF AE AB 40 32 56 2B 21 21 CC B7 CB 4D DD C4 78 5D C1 02 02 83 09 88 26 DD 3D 51 7A 5D 4A E7 7D 53 4E B3 B4 D5 D0 72 FD 20 B5 58 F2 3B AE 06 D7 17 B5 FD DB 02 22 DC 2A BD B8 D8 9B ED B7 D1 B0 83 F6 8F 98 69 BD 8E 9B 0D 44 71 ED 86 A6 80 1A A6 39 5D E7 88 E0 CE 0B F5 C5 F9 D6 5C 27 35 A0 F0 65 93 FE CA D3 DA 42 AC 0A 98 AB B9 49 70 28 85 8C 46 31 B7 3F 9D 28 32 19 5E 45 7C 79 36 81 D6 04 9C 40 3E AA FA AA AD 19 1A 78 82 4C D2 52 06 0B E4 05 CF 4A 39 97 41 FD 43 AB 90 A3 0C 20 59 C7 EF DD 5B 70 0E 82 79 54 AD 5E 2D 30 95 54 97 C6 10 4F CA 20 59	00 48 C7 2A F7 D3 19 0C C9 24 1D 43 D5 CB B4 6C E4 AD 87 60 4E 74 F1 C6 41 23 D8 17 7C 85 20 DB 00 00 00 00 B3 73 81 B5 62 8C BD 7A 91 09 8F F6 9C EB 64 14 35 B3 01 47 DC FC C1 81 DD 96 93 9E 61 07 07 0E 3B 5F F7 C3 B8 FF AE AB 40 32 56 2B 21 21 CC B7 CB 4D DD C4 78 55 C1 02 02 83 09 88 26 DD 3D 51 7A 5D 4A E7 7D 53 4E B3 B4 D5 D0 72 FD 20 B5 58 F2 3B AE 06 D7 17 B5 FD DB 02 22 DC 2A BD B8 D8 9B ED B7 D1 B0 83 F6 8F 98 69 BD 8E 9B 0D 44 71 ED 86 A6 80 1A A6 39 5D E7 88 E0 CE 0B F5 C5 F9 D6 5C 27 35 A0 F0 65 93 FE CA D3 DA 42 AC 0A 98 AB B9 49 70 28 85 8C 46 31 B7 3F 9D 28 32 19 5E 45 7C 79 36 81 D6 04 9C 40 3E AA FA AA AD 19 1A 78 82 4C D2 52 06 0B E4 05 CF 4A 39 97 41 FD 43 AB 90 A3 0C 20 59 C7 EF DD 5B 70 0E 82 79 54 AD 5E 2D 30 95 54 97 C6 10 4F CA 20 59

```
94          var notBefore time.Time
95          if len(*validFrom) == 0 {
96                  notBefore = time.Now()
97          } else {
98                  notBefore, err = time.Parse("Jan 2 15:04:05 2006", *validFrom)
99                  if err != nil {
100                         fmt.Fprintf(os.Stderr, "Failed to parse creation date: %s\n", err)
101                         os.Exit(1)
102                 }
103         }
104
105         notAfter := notBefore.Add(*validFor)
106
107         serialNumberLimit := new(big.Int).Lsh(big.NewInt(1), 128)
108         serialNumber, err := rand.Int(rand.Reader, serialNumberLimit)
109         if err != nil {
110                 log.Fatalf("failed to generate serial number: %s", err)
111         }
```

Figure 10: The valid time and the serial number of certificates in CFSSL.

It is hard to predict the output of random number generators of operation systems so far.

6.3. NSS. NSS is a set of libraries supporting cross-platform network security services and developed by Mozilla. We reviewed the source codes of NSS 3.38 to find the way that the valid time and serial number of certificated are generated. The tool creating certificates is in *\nss\cmd\certutil\certutil.c*.

From Figure 11, we can see that the default value of "not before" is set as "current time." From Figure 12, "serial number" is not a random number. "LL_USHR" is a macro defined in "prlong.h" to logically shift the second operand right by the number of bits specified in the third operand. "PRTime" is a 64-bit structure in microseconds. Thus, "serial number" is 64-bit "current time" shifted right by 19 bits. Obviously, we can predict "serial number" easily.

6.4. Botan. Botan is an open source cryptography library written in C++. We reviewed the source codes of Botan 2.6 to find the way that the valid time and serial number of certificated are generated.

Form Figure 13, the default value of "not before" (start_time in Figure 13) is set as "current time." The "serial number" is the second parameter of the function "sign.request", i.e., "rng()", which is defined in the header file /botan/src/cli/cli.h in Figure 14.

There are 5 kinds of random number generators in Botan, which is dependent on the command parameters "rng –system –rdrand –auto –entropy –drbg –drbg-seed=

Step 1: Attacker applies for a certificate to target CA and records the submitting time T_s in seconds.
Step 2: Attacker receives the certificate, checks the time Tb of "not before" in certificate and the time Ta of "not after", and computes $T_d = T_b - T_s$ and $T_v = T_a - T_b$.
Step 3: According to the Algorithms 1 and 2, attacker brutes force the every 100-nanosecond (for Windows) or every microsecond (for Linux) in T_b to find which time seed T_{bn} is used to generate the certificate (totally 10^7 or 10^6)
Step 4: Attacker selects a future time T_f as the time of "not before" of the target forged certificate
Step 5: Attacker randomly selects a value of m, which satisfies the condition:
$T_f(inseconds) \le T_{bn} + 100144m(in100nanoseconds) < T_f + 1(inseconds)$
(for Windows XP)

$T_f(inseconds) \le T_{bn} + 1008m(inmicroseconds) < T_f + 1(inseconds)$
(for Ubuntu)
Step 6: According to the Algorithms 1 and 2, attacker computes the candidate serial numbers with the seed of T_f (in seconds) and T_{bn}+100144m(in 100nanoseconds) or T_{bn}+1008m(in microseconds).
Step 7: Attacker uses the candidate serial numbers, T_f as "not before", and $T_f + T_v$ as "not after", to generate forged certificates according to the Stevens's method [3].
Step 8: Attacker submits the application at the time $T_f - T_d$ and get the signature of the certificate.

Algorithm 4: Guess the serial number.

```
1924    now = PR_Now();
1925    PR_ExplodeTime(now, PR_GMTParameters, &printableTime);
1926    if (warpmonths) {
1927        printableTime.tm_month += warpmonths;
1928        now = PR_ImplodeTime(&printableTime);
1929        PR_ExplodeTime(now, PR_GMTParameters, &printableTime);
1930    }
1931    printableTime.tm_month += validityMonths;
1932    after = PR_ImplodeTime(&printableTime);
1933
1934    /* note that the time is now in micro-second unit */
1935    validity = CERT_CreateValidity(now, after);
```

Figure 11: The valid time of certificates in NSS.

```
3006    /*  If making a cert, need a serial number.  */
3007    if ((certutil.commands[cmd_CreateNewCert].activated ||
3008         certutil.commands[cmd_CreateAndAddCert].activated) &&
3009        !certutil.options[opt_SerialNumber].activated) {
3010        /*  Make a default serial number from the current time.  */
3011        PRTime now = PR_Now();
3012        LL_USHR(now, now, 19);
3013        LL_L2UI(serialNumber, now);
3014    }
```

Figure 12: The serial number of certificates in NSS.

```
75          auto now = std::chrono::system_clock::now();
76
77          Botan::X509_Time start_time(now);
78
79          typedef std::chrono::duration<int, std::ratio<86400>> days;
80
81          Botan::X509_Time end_time(now + days(get_arg_sz("duration")));
82
83          Botan::X509_Certificate new_cert = ca.sign_request(req, rng(), start_time, end_time);
84
85          output() << new_cert.PEM_encode();
```

Figure 13: The valid time and the serial number of certificates in Botan.

```
162
163         Botan::RandomNumberGenerator& rng();
164
```

Figure 14: The definition of rng() in Botan.

*bytes." The parameter "–system" means using the RNG of operation systems, such as /dev/(u)random in Linux-like systems. The parameter "–rdrand" means using the instruction RDRAND from Intel x86 on-chip hardware random number generator. The parameters "–auto" and "–entropy" use the system RNG or else a default entropy source to input seeds. The parameter "–drbg" uses a PRNG complied with NIST SP 800-90A, whose seed is designated by "–drbg-seed." There are no known security vulnerabilities of those RNGs for predicting their outputs so far.

6.5. Fortify. Fortify is an open source application supported by the CA Security Council. We reviewed the source codes of Fortify 1.0.17 to find the way that the valid time and serial number of certificated are generated.

Form Figure 15, the default value of "not before" is set as "current time." The "serial number" is generated by the function "crypto.getRandomValues," which is from Web Crypto API and is a cryptographically strong RNG.

Concluding the above analysis on OpenSSL, EJBCA, CFSSL, NSS, Botan, and Fortify, we can compare the way generating valid time and serial number of certificates in Table 5.

7. Conclusion

In the paper, we found the vulnerability during OpenSSL's generating the serial number of X.509 certificates. It is possible to forge certificates based on the method presented by Stevens. Similarly, EJBCA and NSS have the same vulnerability among other 5 open source libraries.

Although MD5 has been replaced by CAs now, with the development of technology, new attacks for current hash algorithm adopted by CAs, such as SHA-256, will probably occur in the future. If the chosen-prefix collision of some hash algorithm occurs, the threat will work again probably. In that case, attackers still need to predict the value of fields

Table 5: The comparison of 6 open source libraies.

	Programming Language	Not Before	Serial Number	
			Is random?	Is predictable?
OpenSSL	C	current time	√	√
EJBCA	Java	current time - 10minutes	√	√
CFSSL	go	current time	√	×
NSS	C	current time	×	√
Botan	C++	current time	√	×
Fortify	Typescript	current time	√	×

```
const certificate = new pkijs.Certificate();

// region Put a static values
certificate.version = 2;
const serialNumber = crypto.getRandomValues(new Uint8Array(10));
certificate.serialNumber = new asn1js.Integer();
certificate.serialNumber.valueBlock.valueHex = serialNumber.buffer;

const commonName = new pkijs.AttributeTypeAndValue({
  type: "2.5.4.3", // Common name
  value: new asn1js.PrintableString({ value: process.env.FORTIFY_SSL_CN || "127.0.0.1" }),
});

certificate.subject.typesAndValues.push(commonName);

certificate.issuer.typesAndValues.push(new pkijs.AttributeTypeAndValue({
  type: "2.5.4.3", // Common name
  value: new asn1js.PrintableString({ value: "Fortify Local CA" }),
}));

// Valid period is 1 year
certificate.notBefore.value = new Date(); // current date
const notAfter = new Date();
notAfter.setFullYear(notAfter.getFullYear() + 1);
certificate.notAfter.value = notAfter;

certificate.extensions = []; // Extensions are not a part of certificate by default, it's an optional array

// Extended key usage
const extKeyUsage = new pkijs.ExtKeyUsage({
  keyPurposes: ["1.3.6.1.5.5.7.3.1"],
});
certificate.extensions.push(new pkijs.Extension({
```

Figure 15: The valid time and the serial number of certificates in Fortify.

controlled by CAs in order to construct forged certificates. Thus, the randomness of the serial number is important for CAs too.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The author declares that they have no conflicts of interest.

Acknowledgments

The project is supported by Key Research and Development Plan of Shandong Province, China (NO.2017CXGC0704), and Fundamental Research Fund of Shandong Academy of Sciences, China (NO.2018:12-16).

References

[1] X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu, "Collisions for hash functions md4, md5, HAVAL-128 and RIPEMD," *IACR Cryptology ePrint Archive*, vol. 199, 2004.

[2] X. Wang and H. Yu, "How to break MD5 and other hash functions," in *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, vol. 3494 of *Lecture Notes in Computer Science*, pp. 19–35, Springer, Berlin, Germany, 2005.

[3] M. Stevens, A. Lenstra, and B. de Weger, "Chosen-prefix collisions for MD5 and colliding X.509 certificates for different identities," in *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, vol. 4515, pp. 1–22, Springer, Berlin, Barcelona, Spain, 2007.

[4] M. Stevens, A. Sotirov, J. Appelbaum et al., "Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate," in *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference*, vol. 5677, pp. 55–69, Santa Barbara, CA, USA, 2009.

[5] M. Stevens, A. K. Lenstra, and B. de Weger, "Chosen-prefix collisions for MD5 and applications," *International Journal of Applied Cryptography*, vol. 2, no. 4, pp. 322–359, 2012.

[6] J. Appelbaum, A. Lenstra, D. Molnar et al., "Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate," in *Advances in Cryptology - CRYPTO 2009*, vol. 5677 of *Lecture Notes in Computer Science*, pp. 55–69, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, http://www.win.tue.nl/hashclash/rogue-ca/.

[7] Lab. Kaspersky, *The Flame: questions and answers*, 2012.

[8] M. Fillinger and M. Stevens, "Reverse-engineering of the crypt-analytic attack used in the flame super-malware," in *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Part II*, vol. 9453, pp. 586–611, Springer, Heidelberg, Auckland, New Zealand, 2015.

[9] Netcraft., "14% of SSL certificates signed using vulnerable MD5 algorithm," https://news.netcraft.com/archives/2009/01/01/14_of_ssl_certificates_signed_using_vulnerable_md5_algorithm.html.

[10] F. Strenzke, "An analysis of openssl's random number generator," in *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 644–669, Springer, Berlin, Germany, 2016.

[11] S. H. Kim, D. Han, and D. H. Lee, "Predictability of android openssl's pseudo random number generator," in *Proceedings of*

the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, pp. 659–668, Berlin, Germany, 2013.

[12] F. Dörre and V. Klebanov, "Practical detection of entropy loss in pseudo-random number generators," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 678–689, Germany, October 2016.

[13] T. Yoo, J.-S. Kang, and Y. Yeom, "Recoverable random numbers in an internet of things operating system," *Entropy*, vol. 19, no. 3, p. 113, 2017.

[14] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage, "When private keys are public: results from the 2008 debian openssl vulnerability," in *Proceedings of the 2009 9th ACM SIGCOMM Internet Measurement Conference, IMC 2009*, pp. 15–27, Chicago, Illinois, IL, USA, 2009.

[15] S. H. Kim, D. Han, and D. H. Lee, "Practical effect of the predictability of android openSSL PRNG," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E98A, no. 8, pp. 1806–1813, 2015.

[16] M. Stevens, "hashclash project," https://github.com/cr-marc-stevens/hashclash.

[17] M. Stevens, P. Karpman, and T. Peyrin, "Freestart collision for full SHA-1," in *Advances in Cryptology – EUROCRYPT*, M. Fischlin and J. S. Coron, Eds., vol. 9665, pp. 459–483, Springer, Berlin, Berlin, Heidelberg, 2016.

[18] OpenSSL., https://www.openssl.org.

[19] OpenCA., https://www.openca.org/projects/openca.

[20] CVE-2005-2946., http://cve.mitre.org/cgi-bin/cvename.cgi.

[21] EJBCA., https://www.ejbca.org.

[22] CFSSL., https://github.com/cloudflare/cfssl.

[23] nss., https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS.

[24] botan., https://github.com/randombit/botan.

[25] Fortify., https://github.com/PeculiarVentures/fortify.