Cybersecurity MSc: Applied Cryptography 2023-24

2nd Project: Deterministic RSA key generation (D-RSA)

November 2, 2023 Due date: December 24, 2023

Changelog

- v1.0 – Initial version.

1 Introduction

In most cases, RSA key pairs are randomly generated for some dimension of their module. This means that, ideally, one cannot reproduce the generation process later, in order to discover the private key. On the other hand, private keys need to be kept with the appropriate protection, which means confidentiality guaranties, access control mechanisms to prevent others than the owner to use it and protection against loss.

Alternatively, one can use a deterministic key pair generation. In this case, the same key pair is always generated from a set of parameters, which are analogous to a password that protects a private key stored in an ordinary file. We will use this approach to build a secure, deterministic RSA key pair generation (D-RSA).

D-RSA already exists, although not with this name. In fact, devices such as a TPM (Trusted Platform Module), which exist in most laptops today, are able to recreate RSA key pair this exact way, i.e. from some user-provided secret. However, a TPM also uses something that is not easily available to an ordinary program: an internal secret, unique to the TPM device. To overcome this limitation, we will use a time-based alternative. In a nutshell, we will make the process slow, so that guessing attacks against the secrets that seed the D-RSA process could not thrive. Furthermore, we will make it a variable, stop-and-go process, so that no parallel machines or precomputation databases could be used.

2 Homework

The work consists on implementing three fundamental components:

- A pseudo-random byte generator, with a N-byte seed. The initialization of this generator will be the fundamental contribution to the time to generate a D-RSA key pair;
- A complete, and secure, RSA key pair generator using some arbitrary precision integer representation.

These components will be driven by 3 parameters: a **password**, a **confusion string** and an **iteration count**. These, all together, must assure that the setup of a randomness source for the RSA key pair generator takes a long time (possibly, minutes). Remember, this time is not critical to the key pair creator, but is devastating for an attacker.

2.1 Pseudo-random byte generator

The pseudo-random byte generator should be completely specified in the project, possibly using standard functions, such as hash or cipher functions. Consider, for instance, using the generator of a stream cipher.

The setup of such a generator should be long and complex, in order to complicate its cryptanalysis (discovery of the actual seed). For that, we will use the following strategy:

1. Compute a bootstrap seed from the password, the confusion string and the iteration count. Consider, for instance, using the PBKDF2 method;
2. Transform the confusion string into an equal length sequence of bytes (confusion pattern). These resulting bytes should be able to have any value;
3. Initialize the generator with the bootstrap seed;
4. Use the generator to produce a pseudo-random stream of bytes, stopping when the confusion pattern is found;
5. Use the generator to produce a new seed and use that seed to re-initialize the generator;
6. Repeat the last two steps as many times as the number of iterations defined by the user.

This strategy can take an arbitrarily high computation time by acting either on the length of the confusion string and on the number of iterations.

2.2 Applications

Create two applications, one for evaluating the time taken to set up the described pseudo-random generator for different input parameters (`randgen`) and anther one that uses it to deterministically generate an RSA key pair (`rsagen`). Use the `randgen` application to produce charts ilustrating the contribution of the two input parameters – confusion string and number of iterations – to the setup time of the pseudo-random number generator.

For `rsagen`, use by default the ordinary value chosen for the RSA exponent e: $2^{16}+1$. Also, use standard formats, such as W3C XKMS 2.0 or PEM, to store the key pair. Alternatively, you can create your own format (e.g. with JSON), but you should avoid it for interoperability sake. Furthermore, standard formats will be valued over non-standard ones.

Consider the implementation of `rsagen` using the `stdin` as random byte source. This allows you to experiment your application with many different randomness sources (e.g. `/dev/urandom` or other) just by redirecting the application's `stdin`. Consider as well the implementation of `randgen` with an option for either performing speed tests or generate a random byte sequence to the `stdout`. With this approach, you can easily combine both tools with a shell pipeline. And you can as well perform statistical randomness tests on your generator's output (this is not required, though).

3 Evaluation

The project will be evaluated as follows:

- Implementation of the pseudo-random generator proposed for D-RSA (in any language): 25%;
- Implementation of the RSA key pair generator (in any language): 25%;
- Implementation of the applications: 5% for `randgen`, 5% for `rsagen`;
- Implementation of D-RSA in a 2nd language, producing the same result in both languages: 10%;
- Written report, with complete explanations of the strategies followed and the results achieved: 30%

4 Homework delivery

Send your code to the course teachers through Elearning (a submission link will be provided). Include a small report, with no more than 6 pages, describing the implementation (not a complete copy of the code developed!). Code snippets may be used to illustrate your implementation.

Every piece of code imported from anywhere must be stated in the report and in the code itself. Failure to do so will be penalised.

5 References

- *PBKDF2*, `https://en.wikipedia.org/wiki/PBKDF2`
- *RSA cryptosystem*, `https://en.wikipedia.org/wiki/RSA_(cryptosystem)`
- *Arbitrary-precision integers*, `https://rosettacode.org/wiki/Arbitrary-precision_integers_(included)`
- *C/C++ GNU MP (multiple precision arithmetic library*, `https://gmplib.org/manual`
- *OpenSSL BIGNUM*, `https://www.openssl.org/docs/man1.1.1/man3/BN_bin2bn.html`
- *Python gmpy2 module*, `https://gmpy2.readthedocs.io/en/latest/intro.html`
- *Java 14 Class BigInteger*, `https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/math/BigInteger.html`
- *JavaScript BigInt*, `https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/BigInt`