crypto_toolkit Documentation

Release 0.1

Adarsh Saraf

February 12, 2017

Contents:

CHAPTER 1

Crypto Guidelines

1.1 Contents

1. *Random Number Generation*
2. *Password Storage*
3. *Key Generation*

1.1.1 Random Number Generation

`/dev/random` or `/dev/urandom` are considered very good sources for random numbers. From the Linux man page:

> The random number generator gathers environmental noise from device drivers and other sources into an entropy pool. The generator also keeps an estimate of the number of bits of noise in the entropy pool. From this entropy pool, random numbers are created...

`/dev/random` is blocking until environmental noise is available. `/dev/urandom` is non-blocking and can reuse the internal pool to produce more pseudo-random bits when new ones are not available. For futher info on these look at the man-page.

`/dev/urandom` can be accessed through the python `os` module as follows:

```
import os
random_number = os.urandom(16)
```

For further info look at `‘os.urandom` <https://docs.python.org/3/library/os.html>‘__.

In this toolkit we have provided the following method for users to easily get pseudorandom numbers:

```
get_random_number(size = 16):
    '''
    Get a random number.
    The size parameter specifies the number of bytes in the random number generated.
    The default size of 16 is acceptable for salts, etc.
    Returns the Base58 encoded random number.
    '''
```

Currently it’s a wrapper around `os.urandom` but can be updated in the future to support better random number generators.

1.1.2 Password Storage

It is never advisable to store user passwords in plaintext in any manner. Any user password must immediately be garbled to safeguard it's security. The current practices require that we store a hash of the password generated. This can be done using either of `PBKDF2`, `bcrypt`, or `scrypt` cryptographic tools. While `PBKDF2` is secure, it is vulnerable to ASICs/GPUs based attacks since it does not use more memory but just repetitive computations. It is suggested that passwords be hashed using `scrypt` which has larger memory requirements. All these methods use a unique salt per password to prevent against *rainbow attacks*, which involves the creation of inverse hash tables. The use of the salt makes it difficult to precompute inverse hashes since now the salt varies and therefore any attacker will have to compute the hashes based on this salt, which is effectively a brute-force attack and is made very difficult since finding collisions for cryptographically secure hash funtions is computationally difficult.

`PBKDF2` can take any pseudorandom function like cryptographic hash, ciphers or hash-based message authentication code to garble the given password using the salt. For more details, see ‘`PBKDF2` <https://en.wikipedia.org/wiki/PBKDF2>‘__.

In the given toolkit, we provide the following two methods:

```
generate_storage_hash_from_password(password, salt = None, length = 128, n = 2**14, r = 8, p = 1)
verify_storage_hash_from_password(storage_hash, password, salt, length = 128)
```

These can be used to provide the functionalities of generating and verifying storage hashes for passwords. The user can supply the salt, or a random salt is generated using `get_random_number`.

1.1.3 Key Generation

CHAPTER 2

README

2.1 crypto_toolkit

A set of cryptographic tools exposed in a simple user interface for most common usages. We also provide a set of guidelines for common cryptographic uses tying them to the methods provided in this toolkit.

2.1.1 Background

We create this simple toolkit in order to enable users exploit cryptographic techniques for data security without actually having to know about them. We provide simple APIs for common use scenarios using the Python `cryptography` module.

2.1.2 Requirements

You should have the Python `` `cryptography `` <https://pypi.python.org/pypi/cryptography>`__ module installed in the environment you are working. It you have `pip` installed in your system, this can be installed using:

```
[sudo] pip install cryptography
```

We recommend the use of `` `virtualenv `` <https://pypi.python.org/pypi/virtualenv>`__ to create a separate virtual environment for your project. It can be installed using:

```
[sudo] pip install virtualenv
```

2.1.3 Usage

We are currently maintaining a single module under this project for easy import into your project. Download this project and then import the `crypto_toolkit` module:

```
import crypto_toolkit
```

It currently has the following functions to handle passwords: * `generate_key_from_password` * `verify_key_from_password` * `generate_storage_hash_from_password` * `verify_storage_hash_from_password`

The names of the functions are intuitive. The above functions are necessary since it is never advisable to store passwords. Any password must immediately converted into a key using a key derivation function (kdfs). Based on our explorations, we found that the common practice is to use **PBKDF2** for key generation, that is use the password to derive a key that can be used further with various encryption techniques, and **scrypt** to generate hashes of passwords that can be stored for password verification.

CHAPTER 3

Code Documentation

CHAPTER 4

Indices and tables

- genindex
- modindex
- search