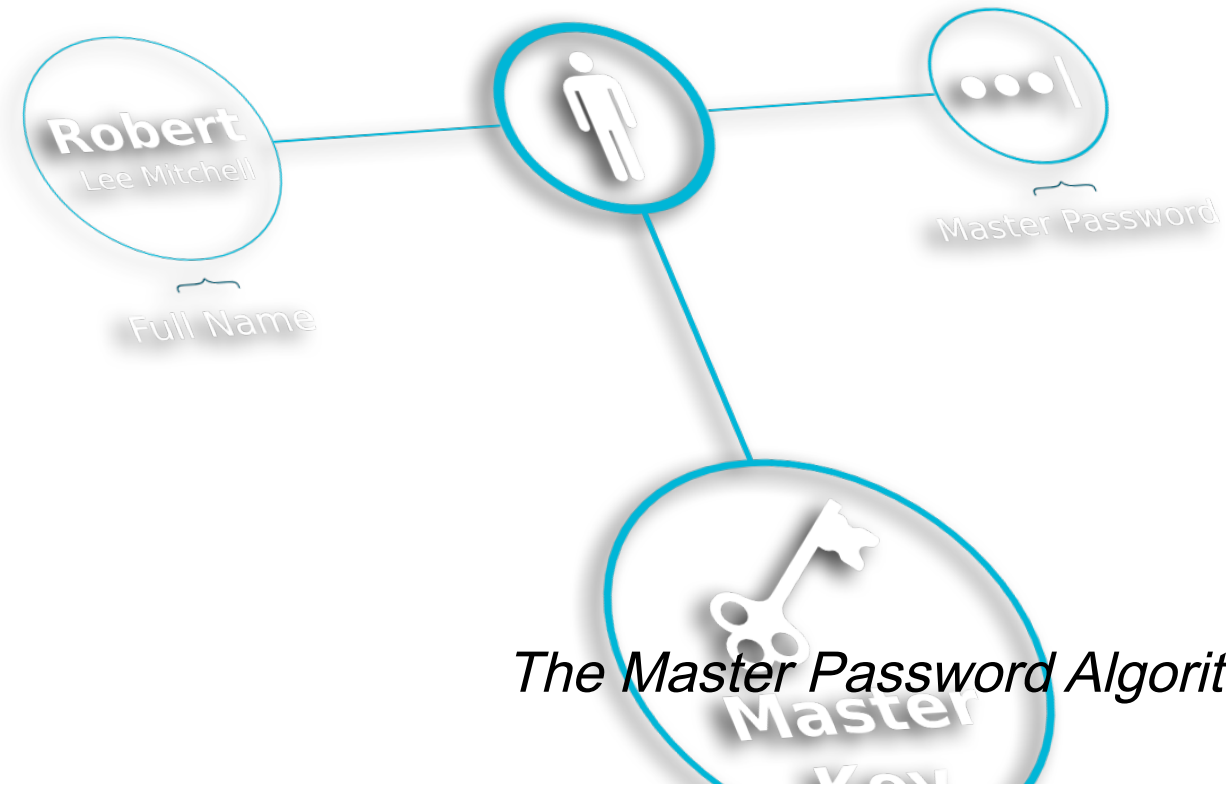
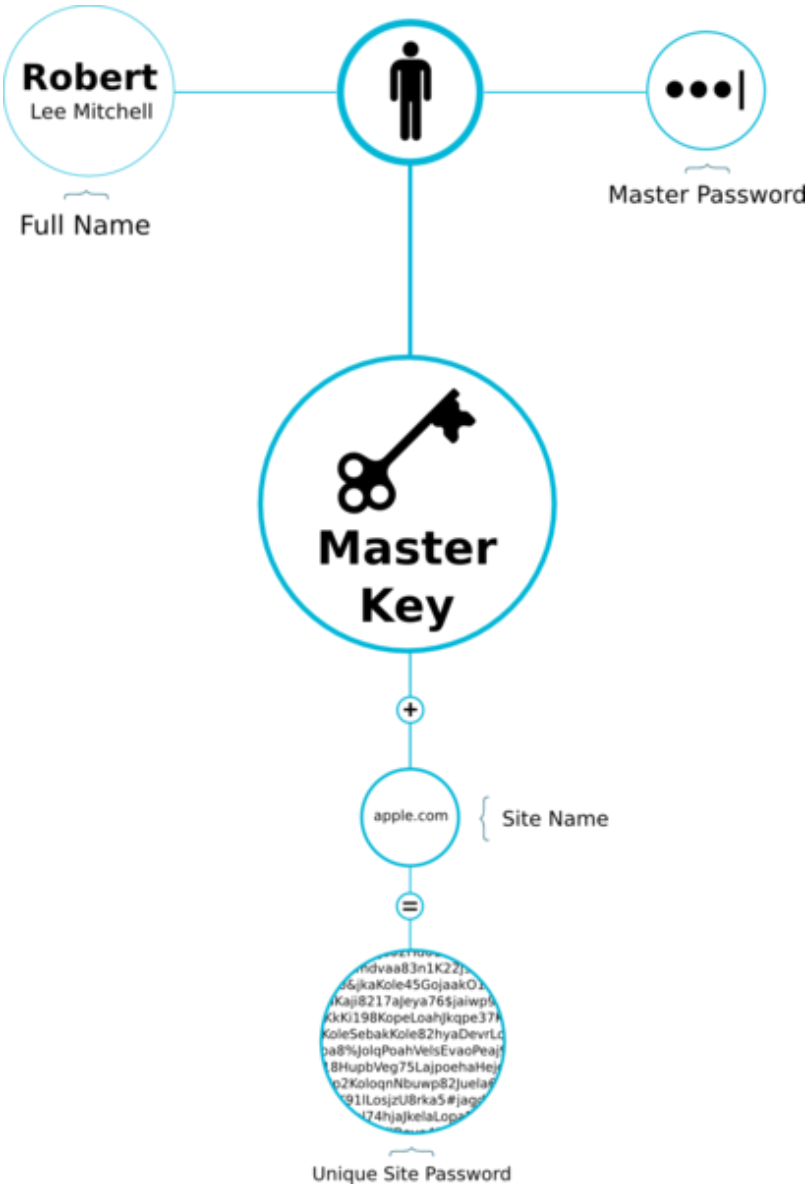


The Master Password Algorithm

Master Password is *an algorithm used to generate unique passwords* for websites, email accounts, or anything else *based only on easily reproducible input*.
The goal is a process that avoids all the problems involved with other password solutions.

The Master Password algorithm is *open*: this page describes its inner workings in detail. We believe the following is an important lesson we should all learn: Regardless of how much encryption a solution claims, if you don't know how it works, you **cannot** assume it is secure (http://www.geekzone.co.nz/foobar/5823) (at least, not the kind of secure you care about).



How Does It Work?

The user is expected to remember the following information:

- Their **full name** (eg. *Robert Lee Mitchell*):
  This is a salt for the master key generation.
- Their personal **master password** (eg. *pink fluffy door frame*):
  This is the secret for the master key generation.
- **The site name** (eg. *apple.com*):
  The user chooses a name for each site. The bare domain name is an ideal choice.
- **The site's password counter** (default: *1*):
  This is an integer that can be incremented when the user needs a new password for the site.
- **The site's password type** (default: *Long Password*):
  This type determines the format of the output password. It can be changed if the site's password policy does not accept passwords of this format.

In practice, the secret master password is the only extra thing users will actually need to remember. Their full name, they'll hopefully remember regardless. If the site is always named after the bare domain name, it needn't explicitly be remembered but can be found in the browser's address bar. The counter and type need only be remembered if they are changed from their default values.

In short, the algorithm involves the following steps:

1. Calculate the **master key** from a user's name and master password.

2. Calculate the **template seed** from the site's name and counter.
3. Encode a **site password** using the site's type template.

To ensure cross-platform compatibility, we define all data as byte streams using the following encodings for other types:

- Strings (eg. `"com.lyndir"`) are encoded as UTF-8.
- Numbers (eg. `name length`) are encoded as 32-bit unsigned integers in network byte order.

The Master Key

The master `key` is a 64-byte secret key generated by performing expensive key derivation using the user's master password salted by their full name. It represents the user's global secret.

The purpose of this process is to deter any attempts at brute-forcing a user's master password from a known site password. The key derivation is done using the scrypt (https://www.tarsnap.com/scrypt.html) algorithm, which guarantees that the process sufficiently time- and resource-consuming to make brute-forcing an infeasible attack.

The key derivation is salted by the user's full name to prevent the generation of rainbow tables on the algorithm. This salt is not secret, and the user's full name is chosen because it is an input of sufficiently high entropy while being (hopefully) impossible to forget by the user.

```
key   = scrypt( P, S, N, r, p, dkLen )
where
P     = master password
S     = "com.lyndir.masterpassword" . name length . name
N     = 32768
r     = 8
p     = 2
dkLen = 64
```

The Template Seed

With the master `key` known, we can proceed to calculate a template `seed` for the site. The template `seed` is essentially the site-specific secret in binary form.

To generate the template `seed`, we construct an authentication code for the site's `name` and `counter` using the user's master `key`.

We employ the HMAC-SHA-256 (https://tools.ietf.org/html/rfc4868) algorithm to obtain a large enough `seed` for the encoding step that follows.

```
seed = hmac-sha256( key, "com.lyndir.masterpassword" . site name length . site name . counter )
```

The Site Password

The template `seed` is a site-specific secret for our user, but it's in a binary form which is not useful as a password. To convert this byte string into a password, we need to encode it as a string of characters.

We have two additional problems that need to be solved: The output password should be easy for a user to read from a screen and type using a keyboard or smartphone. Additionally, it should also be compatible with most site's password policies. These policies often restrict the kind of passwords users can assign to their accounts in

an attempt to foil bad password habits but often have the opposite effect, especially on secure passwords. Commonly, they are a side-effect of a site's bad password handling (eg. storing clear-text passwords in a database).

Master Password addresses these problems by introducing password type templates. Each password type describes what an output password must look like and maps to a set of `templates`. Templates describe the resulting output password using a series of character groups mappings.

By default, Master Password uses the *Long Password* type for any new passwords. The user is able to choose a different password type, which is normally only done if the site's password policy is incompatible with the output password produced by this type.

To create the output password, the bytes in the template `seed` are encoded according to the `template`. The first `seed` byte is used to determine which of the type's templates to use for encoding an output password. We take the byte value of the first `seed` byte modulo the amount of `templates` set for the chosen password type and use the result as a zero-based index in the `templates` list for the password type.

```
templates = [ "CvcvCvcvnoCvcv", "CvcvnoCvcvCvcv", "CvcvCvcvCvcvno", ... ]
template  = templates[ seed[0] % count( templates ) ]
```

Now that we know what `template` to use for building our output password, all that's left is to iterate the `template`, and produce a character of password output for each step. When we iterate the `template` (index `i`), we look in the character group identified by the character (string `passChars`) in the `template` at index `i`.

We use the `seed`'s byte value at index `i + 1` modulo the amount of characters in the character class to determine which character (`passChar`) in the class to use for the output password at index `i`.

```
passChar    = passChars[ seed[i + 1] % count( passChars ) ]
passWord[i] = passChar
```

The result is an encoded `passWord` string that contains the password generated for the site, such as:

CuzaSasy3*Rimo

Password Type Templates

Master Password defines the following password types and their templates:

- Type: **Maximum Security Password**
  - `anoxxxxxxxxxxxxxxxxx`
  - `axxxxxxxxxxxxxxxxxno`
- Type: **Long Password**
  - `CvcvnoCvcvCvcv`
  - `CvcvCvcvnoCvcv`
  - `CvcvCvcvCvcvno`
  - `CvccnoCvcvCvcv`
  - `CvccCvcvnoCvcv`
  - `CvccCvcvCvcvno`

    - `CvcvnoCvccCvcv`
    - `CvcvCvccnoCvcv`
    - `CvcvCvccCvcvno`
    - `CvcvnoCvcvCvcc`
    - `CvcvCvcvnoCvcc`
    - `CvcvCvcvCvccno`
    - `CvccnoCvccCvcv`
    - `CvccCvccnoCvcv`
    - `CvccCvccCvcvno`
    - `CvcvnoCvccCvcc`
    - `CvcvCvccnoCvcc`
    - `CvcvCvccCvccno`
    - `CvccnoCvcvCvcc`
    - `CvccCvcvnoCvcc`
    - `CvccCvcvCvccno`
- Type: **Medium Password**
    - `CvcnoCvc`
    - `CvcCvcno`
- Type: **Short Password**
    - `Cvcn`
- Type: **Basic Password**
    - `aaanaaan`
    - `aannaaan`
    - `aaannaaa`
- Type: **PIN**
    - `nnnn`

Where each of the letters above expand any of the characters in their respective character group:

- Template character: `V`
    - `AEIOU`
- Template character: `C`
    - `BCDFGHJKLMNPQRSTVWXYZ`
- Template character: `v`
    - `aeiou`
- Template character: `c`
    - `bcdfghjklmnpqrstvwxyz`
- Template character: `A`
    - `AEIOUBCDFGHJKLMNPQRSTVWXYZ`
- Template character: `a`
    - `AEIOUaeiouBCDFGHJKLMNPQRSTVWXYZbcdfghjklmnpqrstvwxyz`
- Template character: `n`

    - `0123456789`
- Template character: `o`
    - `@&%?,=[]_:-+*$#!'^~;()/.`
- Template character: `x`
    - `AEIOUaeiouBCDFGHJKLMNPQRSTVWXYZbcdfghjklmnpqrstvwxyz0123456789!@#$%^&*()`
