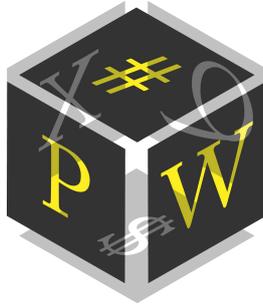# PWGen *for Windows*

- Generator of cryptographically strong passwords -

USER MANUAL

Version **2.9.0**

Licensing Information
By using, copying, distributing or modifying "PWGen for Windows" or a portion thereof you accept all terms and conditions contained in the file *license.txt* included in the package of this program.

Copyright Information
This software as a whole:
Copyright © 2002-2016 by Christian Thöing `<c.thoeing@web.de>`
Portions of this software:
Copyright © 2002-2007 by Troy Wolbrink
(*Tnt Delphi Unicode Controls*)
Copyright © 2006-2014 by Brainspark B.V.
(*implementations of AES, SHA1, and SHA-256 algorithms, base64 encoding;*
part of the "PolarSSL" library)
Copyright © 1996-2015 by Markus F.X.J. Oberhumer
("*minilzo*" *compression library*)
Copyright © 2000 by Arnold G. Reinhold
("*diceware8k*" *word list*)
Copyright © 2005-2010 by Aha-Soft
(*toolbar icons*)

World Wide Web
Homepage: http://pwgen-win.sourceforge.net/
Project page & downloads: http://sourceforge.net/projects/pwgen-win/

Contents

Introduction

The usage of a *password* is still the simplest way to control the access to specific resources. Although many other authentication factors have been developed—examples include identification cards, fingerprint or retinal patterns, voice recognition and other biometric identifiers—, password authentication systems are easier to implement for most applications, are relatively hard to break (note the term "*relatively*"!) and can thus provide accurate security, if used carefully. However, it is essential for the security that the password is *strictly kept secret*, and that it is chosen in a way that makes it hard for an attacker to guess it or to find it by try-and-error (also known as "brute force"). Both conditions are closely connected, but in a rather fatal way: Passwords which are easy to memorize for humans are for the most part *disastrous* in terms of security! Among these bad examples we find personal data (names of family members, pets, meaningful places, etc.), names and characters from favorite books, films or video games, simple words or character sequences (such as the famous "qwerty"), and so on. These passwords are for sure easy to memorize—but can often be guessed without much effort. How can we solve this dilemma?

There are many ways to choose good (i.e., *secure*) passwords—but the best way is to let a random generator choose a password. If these passwords are long enough, it will take years, if not centuries, to find them by "brute force". Computer programs like PWGen can assist you in generating random passwords, as humans are not very good at making up random numbers themselves. Unfortunately, random character sequences like `zio5FcV7J` are fairly hard to memorize (although this is possible and probably not as difficult as you might imagine), so you may want to try pass*phrases* composed of words from a word list instead: Five words from a word list with 8000 words or more are sufficient in most cases to create a high-quality passphrase; moreover, the security can easily be increased by adding some random characters.

The need for secure passwords has grown since the advent of the Internet and its many websites where the access to certain resources (message board, user account, and so on) is controlled by a user name/password pair. Fortunately, since the invention of so-called *password safes*, you don't have to remember all these passwords anymore—you just store them in the password safe which is protected by a "master password" (that must be memorized carefully, of course). As this master password is used to protect highly sensitive data, it should conform to the highest security level possible. The security level, which grows with increasing password length, is only limited by the user's ability to memorize random characters or words. With some effort, most people are certainly able to memorize a 90-bit password.

PWGen is capable of generating cryptographically secure random passwords and passphrases conforming to highest security levels. It can be used to generate master passwords, account passwords and generally all sorts of random sequences. It also offers the option to create lots of passwords at once. Just give it a try!

Program Features

- Password generation based on a **cryptographically secure pseudo-random number generator** (combination of SHA-256 and AES)

- **Entropy gathering** by measuring time intervals between keystrokes, mouse movements and mouse clicks; additionally, entropy from volatile system-specific parameters is collected in regular time intervals

- Generation of **pass*phrases*** composed of words from a word list

- **Pattern-based password generation** (formatted passwords) provides nearly endless possibilities to customize passwords to the user's needs

- Generation of **phonetic (pronounceable) passwords** based on language-specific trigram (3-letter sequences) frequencies

- Numerous **password options** for various purposes

- Generation of **large amounts of passwords** at once

- **"Password hasher"** functionality

- Secure **text encryption** (AES with 256-bit key)

- **Multilingual support**

- **Full Unicode Support**

- Runs on all 32-bit & 64-bit Windows versions (i.e., Windows 95 [OEM SR2], NT 4.0, 98, 2000, Me, XP, Vista, 7, 8, 10; Windows Server 2003, 2008, 2012)[1]

---

1 Note: I have not tested PWGen on "legacy" Windows versions (< XP). I can only state that PWGen was *designed* to be executable on these systems. Keep in mind that PWGen does not provide Unicode sup-port on Windows 9x. PWGen is developed and mainly tested on Windows 7.

Unicode Support

PWGen provides full Unicode support as of version 2.3.0. The Unicode standard can—theoretically—encode up to 1,114,112 characters, and the latest version contains a repertoire of more than 110,000 characters. PWGen aims at providing full support for all those theoretically possible 1,114,112 Unicode characters, especially in passwords (of course), text encryption, file names, and translations of the program. Note that PWGen is still compatible with older 32-bit versions of Windows (9x/Me) that provide only very limited support for Unicode.

Keep in mind that you also need a suitable font to display Unicode characters. TrueType and OpenType fonts can contain up to 65,536 characters, but most fonts on Windows contain only a subset of the first 65,536 Unicode characters. You can use Windows' character map utility (*charmap.exe*) to evaluate the fonts on your system with respect to the characters contained in the font files.

Supported Encodings

Like Windows, PWGen uses *UTF-16* (little-endian) as the default character encoding internally. UTF-16 encodes the most frequent characters in 16-bit units, but the number range $1-2^{16}$ (65,536) is actually not sufficient to encode *all* possible 1,114,112 Unicode characters, so some characters have to be encoded as 2x16-bit units. When reading or writing Unicode text from/to files, however, PWGen supports further encodings besides UTF-16 little-endian, namely *UTF-16 big-endian*, *UTF-8*, and *ANSI*.

In UTF-16 big-endian, the byte order is simply reversed compared to UTF-16 little-endian. UTF-8 is an 8-bit variable-width encoding, which means that each character is encoded as 1 to 4 bytes (8-bit units or "octets"). UTF-8 has the advantage that the first 128 Unicode characters, which are encoded as 1 byte in UTF-8, correspond exactly to the 7-bit ASCII character set, thus making ASCII texts valid UTF8-encoded Unicode, and vice versa (for the first 128 Unicode characters).

ANSI is a non-Unicode 8-bit fixed-width encoding which extends the 7-bit ASCII standard (characters 1–128) by additional 128 language-specific characters (characters 129–256). This additional character set depends entirely upon the user's codepage setting in Windows, so ANSI-encoded texts containing non-Latin characters such as *ö, é, î, Å*, etc., written on a machine with a *Western* codepage setting, looks quite different on a computer with a *Greek* codepage setting: The "special characters" would be replaced by characters from the Greek alphabet in this case, simply because the character sets which are used to display the binary codes in the range 129–256 are different on both machines! This cannot happen with Unicode-compliant encodings, since each valid binary code maps to a fixed Unicode character. As a consequence, Unicode-encoded texts look the same on all computers, irrespective of any language-dependent codepage settings in the operating system.

PWGen can identify files containing Unicode text (either UTF-16 little-/big-endian- or UTF-8-encoded) by the so-called "byte-order mark" (BOM) which is a 2-byte (UTF-16) or 3-byte (UTF-8) sequence right at the beginning of the file. If no BOM is present, PWGen assumes the file to be ANSI-encoded. PWGen is also capable of writing UTF-16- and UTF-8-encoded Unicode text files, and conversion of Unicode to ANSI characters is supported, too. The user can change the default file encoding in the *Configuration* dialog on the *Files* page. (Note: PWGen's configuration file, *PWGen.ini*, has a fixed UTF-8 encoding which cannot be changed.)

Now which encoding should you use? Well, the answer to this question depends on your language and your needs:

- **ANSI**: Using this encoding for texts containing ASCII characters exclusively is unproblematic. However, if the text contains "special" language-specific characters such as *ä, ô*, etc., you may run into trouble if the text file is read on machines with different codepage settings. Thus, ANSI encoding should only be used for Latin-based alphabets, and when the text file is read on computers with the same language settings.

- **UTF-16** and **UTF-8**: The choice between UTF-16 and UTF-8 largely depends on the character set of the text to be encoded, and also on the target application of the encoded Unicode text. UTF-8 is more efficient with respect to file size for Latin-based alphabets, whereas the alphabets of Greek, Cyrillic, Hebrew, Arabic, Coptic, Armenian, Syriac, Tāna and N'Ko require 16 bits in both UTF-16 and UTF-8. The rest of the characters of most of the world's living languages is more efficiently encoded as UTF-16 (16 bits needed) compared of UTF-8 (24 bits needed). Furthermore, in Windows, which internally uses UTF-16, reading and processing UTF-8-encoded files may be (slightly) slower in some cases because the encoding has to be converted to UTF-16 before passing the text to Windows controls. On the other hand, UTF-8 is the *de-facto* standard encoding of the Internet, and should therefore be preferred for Internet-associated document types.

- **UTF-16 little-endian** *vs.* **big-endian**: UTF-16 little-endian should be preferred on the Windows platform.

Happy unicoding!

Password Generation with PWGen –
An Overview

PWGen will assist you in generating cryptographically-strong and easy-to-memorize passwords and passphrases. The "design philosophy" behind PWGen is to provide a simple-to-use and unobtrusive, yet at the same time powerful and versatile application intended at professionals (e.g., system administrators) *and* "normal" home users.

PWGen allows you to generate various types of passwords and passphrases:

- **"Classical" passwords** such as `VLyw68JsSq0m`, i.e., random sequences of characters from a specific character set (upper-case/lower-case letters and numbers in the example).

- **Phonetic (pronounceable) passwords** such as `terweeptoton`, which are generated by evaluating the frequencies of all possible trigrams (1 trigram = 3-letter sequence; for 1 trigram, there are 26^3 = 17,576 possible letter combinations) of a certain language (English in the example).

- **Passphrases** such as `khaki cello waxy mecca verdi`, composed of randomly chosen words from a word list. Words may be easily combined with random characters to give passphrases such as `lends-ah susie-Tx chats-Hz joins-TE`.

- **Formatted passwords / passwords based on patterns**: This is the most versatile password generation feature, for it provides a variety of format specifiers to insert characters from pre-defined character sets and words from a specific word list into the resulting password. Moreover, it allows for repeating and randomly permuting sequences in the password. *Example*: The format string `%{%4u%4l%2d%s%}` means "*insert 4 upper-case letters, 4 lower-case letters, 2 digits, and 1 special symbol; permute the character sequence afterwards*", and yields passwords such as `fP4eM#mAiV2`.

- **"Hashed" passwords**, passwords based on a master password and a parameter: Similar to the website *Hashapass*, this feature enables you to reproducibly generate unique passwords using a (secret) master password and a (not necessarily secret) parameter, such as the name of a website. For example, the master password `qwerty` (*hint: don't use that one, ever!*) together with the parameter `yahoo` yields the password `W4hfOL21K3FYb8Qx`. PWGen's password hasher offers several pre-defined character sets for the user to choose from. Being independent of any encrypted databases of stored parameter–password combinations, it may thus function as a fully portable "password safe". In case you want to be independent of Windows-based applications, PWGen can also provide full compatibility with the Hashapass generator, which is accessible via the Internet and should work with any browser.

To give the user an estimation of the quality of the generated passwords, PWGen displays a "password quality bar" featuring a color range from red-orange (less secure) to dark green (more secure). A password quality/security—or, in more technical terms, *entropy*—of 128 bits is considered unbreakable by today's computer technology, and, given the huge complexity of the 128-bit key space ($\sim 3.4 \cdot 10^{38}$), I really think one can be quite certain that 128-bit passwords and keys will remain secure for the entire era of humankind on planet earth.

Step–by–Step Tutorial

In this section I want to elaborate on each step in creating a password. The main functions of the program are accessible in PWGen's main window, where you can adjust the parameters for password generation. Here we go:

Include Characters

Check **Include characters** to include characters in your password. Enter the desired length of the character string in the **Length** field (maximum 10,000).

Enter the desired character set (i.e., all the characters that may occur in your password) in the field below or choose one from the drop-down list. Note that, in a *set*, each character must occur only once. PWGen aims at providing full Unicode support as of version 2.3.0, so you can (theoretically) enter up to 1,114,112 different characters. You can use Windows' character map utility (*charmap.exe*) to insert Unicode characters which are not accessible via the keyboard.

Furthermore, you may use the following placeholders to abbreviate the sequence (some placeholders may be further abbreviated to save space in the input box):

Placeholder(s)	Meaning	Character set
\<AZ\>	upper-case letters	ABCDEFGHIJKLMNOPQRSTUVWXYZ
\<az\>	lower-case letters	abcdefghijklmnopqrstuvwxyz
\<09\>	digits	0123456789
\<Hex\>	upper-case hexadecimal symbols	0123456789ABCDEF
\<hex\>	lower-case hexadecimal symbols	0123456789abcdef
\<base64\>/ **\<b64\>**	base64 symbols	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
\<easytoread\>/ **\<etr\>**	like *\<AZ\>\<az\>\<09\>*, but without similar-looking ("ambiguous") characters (B8G6I1l|0OQDS5Z2 by default). See *Advanced Password Options* for more options.	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789 *without ambiguous characters*
\<symbols\>/ **\<sym\>**	additional symbols accessible via the keyboard	!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
\<brackets\>/ **\<brac\>**	brackets	()[]{}<>
\<punctuation\>/ **\<punct\>**	punctuation marks	,.;:

<highansi>/ <high>	higher ANSI characters (symbols #127 to #255)	(characters depend on the current ANSI code page)
<phonetic> <phoneticx>	generate phonetic (pronounceable) passwords based on trigram frequencies of the English (or a user-specified) language *(see notes below)*	*<phonetic>*: lower-case letters *<phoneticx>*: mixed-case letters *(frequencies of the letters are language-dependent; English by default)*

You may provide a **comment** included in square brackets `[...]` at the beginning of the entry. All characters in this comment will be ignored by PWGen. For example, entering `[This is a comment.]<AZ>` will add only upper-case letters to the character set. Note that the comment must be placed right at the beginning; otherwise, it will be treated as a normal sequence of characters!

If the text in the box is equal to `<phonetic>` or `<phoneticx>` (other characters are *not* allowed!) PWGen will execute a special algorithm to generate **phonetic passwords**, i.e., passwords which are likely to be pronounceable. The `<phonetic>` version generates passwords composed of lower-case letters exclusively, whereas passwords generated with the `<phoneticx>` version can contain both lower-case and upper-case (i.e., mixed-case) letters. The algorithm is based on the language-specific frequencies of so-called trigrams, which consist of three letters from the English alphabet (a–z, 26 symbols). Phonetic passwords look like this: `rationeterbonte`. You can also generate phonetic passwords based upon another language by loading a "trigram file" containing the trigram frequencies which are characteristic of that language (note, however, that this is only possible for languages the alphabets of which are Latin-derived!). For example, using the trigram frequencies stored in the file *German.tgm* yields passwords such as `gergenfortenman`. To load a trigram file, you have to modify the *Advanced Password Options*. You may also create a trigram file yourself by evaluating a dictionary, word list, or any other text of your choice (see *Tools* / *Create Trigram File (F6)* in the main menu).

The algorithm for generating phonetic passwords is sensitive to the options "*Include at least ...*" (see: *Advanced Password Options*), so you may include an upper-case character, digit, and/or special symbol in the resulting password in order to increase the security. It is, however, not possible to combine the `<phonetic(x)>` password rule with other character sets (digits, symbols, …) because this would destroy the "pronounceability" of the sequence. Alternatively, you can use the placeholder `%q` in a format sequence and combine it with any other character set (see: *Format Password*).

The character set will be updated as soon as you leave the input field, e.g., by clicking on another interaction element. The entropy (number of bits per character) is displayed on top of the *Character set* field: For "normal" (non-phonetic) passwords, it is calculated as $\log_2 N$, where N is the number of characters and $\log_2$ is the logarithm base 2; for phonetic passwords, the entropy is lower than $\log_2 26$ because the letters all have different frequencies (~3.6 bits per character for the default trigrams).

Clicking on the **symbol** ? opens a quick help box containing the most important information.

Clicking on the **symbol** ⓘ opens a message box where each character of the currently loaded set is displayed.

Errors occur when the set contains less than two unique characters. Keep that in mind if you want to generally exclude ambiguous characters (see below).

Include Words
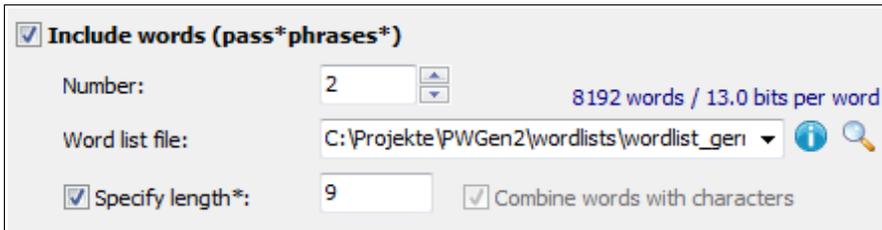
Check **Include words** if you want to include words from a word list in your password. Enter the desired number of words into the **Number** field (maximum 100).

By default, PWGen uses an internal English word list containing 8192 (2^{13}) words. If you enter `<default>` into the **Word list file** field or if you simply leave this field blank, PWGen will use its internal list. However, you may enter the name of your own word list file into this field; alternatively, you can click on the **symbol** 🔍 to browse through your folders and select a file. Note that PWGen supports Unicode word lists and is capable of reading Unicode text files encoded as UTF-16 or UTF-8 if the file contains the corresponding byte-order mark (BOM) at the beginning of the file. (Of course, ANSI-encoded text files are supported, too.)

The word list will be loaded as soon as you leave the input field, e.g., by clicking on another interaction element.

To **reload a word list** from an already loaded file, load the default list, then load the previous file again by selecting it from the drop-down list.

By default, PWGen accepts words that have at most 30 (Unicode) characters; however, you can reduce this limit in the _Advanced Password Options_ dialog. The number of bits per word is calculated as $\log_2 N$ (with N being the number of words) and will be displayed on top of the _Word list file_ field. The size of word lists is limited to 1,048,576 (2^{20}) words.

Clicking on the **symbol** ⓘ will show some information about the currently loaded word list.

Errors occur when the word list cannot be opened (for whatever reason), or when it contains less than two valid words.

Select **Combine words with characters** to generate passwords as a combination of words with characters. This means that each word will be combined with one or more characters, depending on the number of words and the number of characters, respectively. Example of 3 words combined with 8 characters by a "-" symbol: `putty-fuV umbra-Sxq faint-fT`.

Select **Specify length** to specify a length range for the passphrases in the edit box on the right. This option may be helpful for word lists the words of which exhibit a wide range of lengths, as in case of the default English word list. If activated, only passphrases with lengths conforming to the specified length range are allowed, whereas non-conforming passphrases will be discarded. Here, "length" refers to the *total number of characters* in the resulting passphrase, including all words, characters from the password, spaces, etc. Depending on the length specification, it may take some time until a randomly generated passphrase finally meets the length criterion; in the worst case, there is no passphrase combination at all that meets the criterion. When the number of failed attempts reaches 1000, a progress window with a *Cancel* button will be shown, which allows the user to cancel the process to avoid infinite loops.

The length can be specified in various **formats:**

Format (*M,N*=number)	Meaning
N	Length must be exactly *N* (characters)
M-N	Length must be at least *M* and at most *N*
>N	Length must be greater than *N* (at least *N*+1)
>=N	Length must be at least *N*
<N	Length must be less than *N* (at most *N*−1)
<=N	Length must be at most *N*

Note that a length of 0 is not allowed. Apart from the check for 0, PWGen does *not* check the validity of the specified length range for the current password/passphrase settings. It is the user's responsibility to provide a length range which "makes sense" with the current settings.

The label of the checkbox is marked with an asterisk (*) to indicate that this option can decrease the password security. The extent of this reduction depends on the width of the restriction in most cases: The shorter the passphrase and the narrower the length range, the greater the potential decrease in security. Yet filtering out short passphrases may actually be beneficial overall because short passphrases, when treated as pass*words* (i.e., sequences of characters), tend to be insecure compared to long passphrases. Thus using a length specification such as ">*N*" (see table above) can effectively increase the security by avoiding short and insecure passphrases.

Format Password

Select **Format Password** to format your password or passphrase generated via *Include Characters* or *Include Words*, respectively. Additionally (or alternatively), you can use format specifiers and placeholders to introduce random characters from various character sets, and random words from the word list.

Format strings can contain two types of characters, **format specifiers** and **literal characters.** Literal characters are copied verbatim to the resulting password. Format specifiers begin with a "%" character and can be used to insert random characters/words or to manipulate the input sequence (e.g., by randomly permuting a character sequence). A literal percent sign can be inserted into the resulting password by the sequence %%.

Format specifiers have the following form: **%[*][number](specifier)**. All format specifiers begin with a "%" character. Arguments enclosed with square brackets "[]" are optional, those enclosed with brackets "()" are required.

- The optional **number** argument indicates how many times the format command is to be repeated; it can be a decimal number in the range from 1 to (theoretically) 99,999. If *number* is not specified, "1" is assumed by default. Alternatively, this argument can also be specified as a **number range** in the form *N1-N2*; then, a randomly chosen number between *N1* and *N2* will be inserted.

- The **specifier** argument is the actual format specifier or placeholder for a random character from a certain character set (see below for a list of format specifiers) or for a random word from the currently loaded word list.

- The optional **asterisk (*)** argument instructs the program to use each character/ word in the following sequence only once. In this case, the number of characters/ words is limited to the size of the character set/word list. This argument can be placed *anywhere* between the percent sign and the format specifier. Note that if the asterisk is specified but the *number* argument is omitted, the size of the character set or of the word list, respectively (depending on the format specifier), will be used as the default number.

Examples:

- %u means "*Insert 1 random upper-case letter*"

- %20a means "*Insert 20 random lower-case alphanumeric characters*"

- %*5-10d means "*Insert between 5 and 10 (exact number chosen randomly) random digits, each digit must occur only once in the sequence*"

- %*d means "*Generate permutation of digits 0-9*".

The **length of formatted passwords** is currently limited to 16,000 (Unicode) characters.

Some format specifiers have **special meanings** ("*[N]*" means that the *number* argument may be specified optionally):

- `%P`: Insert the password generated via *Include characters* and/or *Include words*. The password can be inserted only once! A "warning" message will be displayed on top of the input field, if ...

  ○ *Include characters* or *Include words* is checked, but **%P** is not specified in the format string ("%P is not specified.");

  ○ **%P** is specified, but neither *Include characters* nor *Include words* is checked ("%P: password not available.");

  ○ the password is too long to insert it, i.e., inserting the full password exceeds the size limitation of the formatted password ("%P: password too long.").

  Example: Entering `This is your password: "%P".` may result in a password like this: `This is your password: "Zq3wu9gL".`

- `%[N]w` or `%[N]W`: Insert random word(s) from the currently loaded word list (see above). In case of multiple words (*N* > 1), the words are separated by a space if the specifier is **w** (lower-case), or they are not separated at all if the specifier is **W** (upper-case). Note that *Include words* doesn't have to be checked in order to use the word list for formatted passwords.

  Example: `%5w` to `dod zion belt xylem avery`

- `%[N][...%]`: Repeat the sequence included in the brackets **%[** and **%]** *N* times. Repeat sequences may be nested up to 4 times (e.g., `%5[...%4[...%3[...%2[...%] %] %] %]`).

  Example: `%5[%a%d%]` to `u9o2p9r2a8`

- `%[N]{...%}`: Randomly permute the sequence included in the brackets **%{** and **%}** and insert *N* characters from the permuted sequence. Nesting is not possible (because it doesn't make any sense). Please note that, when PWGen calculates the quality/security of the resulting password, it does *not* take into account the security gain caused by the random permutation, which is very likely unless the permuted sequence consists of characters from only one single character set (in the latter case, there wouldn't be any increase in security). An exact calculation of this security gain is not trivial and can be computationally expensive, so this feature is currently not available. However, you can try to calculate the gained security bits by this formula:

$S = \log_2(N! / (n_1! \, n_2! \, \ldots \, n_i!))$, where N is the total number of characters in the permuted sequence, and n_i are the numbers of characters from a specific character set.

Examples: (1) `%{hello world!%}` to `ldloor w!lhe`
(2) `%{%9l%d%}` to `euvk8bfifg`

- `%[N]<...%>`: Treat the sequence included in the brackets **%<** and **%>** as a character set, and choose *N* random characters from this set. Note that within this sequence, format specifiers don't have any effect (except for **%>** which ends the sequence); each character will be used "as-is". Note that it is possible to use abbreviation codes, i.e., placeholders for character sets (see *Include Words*), such as `<AZ>`, `<09>`, and so on.

  Example: `%10<<az>0123%+/%>` to `r0/bpya%0y`

Comments enclosed in square brackets `[...]` may be provided at the beginning of the sequence (and only there!).

Here is a complete list with the format specifiers which are currently supported by PWGen (most placeholders for the character sets are identical to those available in the built-in password generator of the application *KeePass*):

Format specifier/ placeholder	Meaning	Character set
1) Placeholders for random characters from various character sets *(asterisk and number argument may be specified optionally in all cases)*		
%x	insert random character(s) from the user-defined character set (as given in the input field below *Include characters*)	*(user-defined)*
%a	lower-case alphanumeric	abcdefghijklmnopqrstuvwxyz0123456789
%A	mixed-case alphanumeric	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghij klmnopqrstuvwxyz0123456789
%U	upper-case alphanumeric	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
%E	mixed-case alphanumeric, but without ambiguous characters	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefg hijklmnopqrstuvwxyz0123456789 *without ambiguous characters*
%d	digit	0123456789
%h	lower-case hexadecimal	0123456789abcdef
%H	upper-case hexadecimal	0123456789ABCDEF
%l	lower-case letter	abcdefghijklmnopqrstuvwxyz
%L	mixed-case letter	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghij klmnopqrstuvwxyz
%u	upper-case letter	ABCDEFGHIJKLMNOPQRSTUVWXYZ

%v	lower-case vowel	`aeiou`
%V	mixed-case vowel	`AEIOUaeiou`
%Z	upper-case vowel	`AEIOU`
%c	lower-case consonant	`bcdfghjklmnpqrstvwxyz`
%C	mixed-case consonant	`BCDFGHJKLMNPQRSTVWXYZbcdfghjklmnpqrstvwxyz`
%z	upper-case consonant	`BCDFGHJKLMNPQRSTVWXYZ`
%p	punctuation marks	`,.;:`
%b	brackets	`()[]{}<>`
%s	special symbols (may be user-defined)	`!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~` *(or user-defined)*
%S	mixed-case alphanumeric and special symbols	`ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!"#$%&'()*+,-./:;<= >?@[\]^_`{|}~` *(special symbols may be user-defined)*
%y	higher ANSI characters (symbols #127 to #255)	*(characters depend on the current ANSI code page)*
%q	generate phonetic password composed of lower-case letters exclusively	`abcdefghijklmnopqrstuvwxyz` *(frequencies of the letters are language-dependent; English by default)*
%Q	generate phonetic passwords composed of mixed-case letters	`ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz` *(see notes above)*
2) Special format specifiers *(possible usage of optional arguments is indicated in italics)*		
%P	insert password generated via *Include characters* and/or *Include words*; password may be inserted only once	
%[*][N]w	word from word list; multiple words are separated by a space	
%[*][N]W	word from word list; multiple words are concatenated without any separators	
%[N][	repeat input sequence included in the brackets *N* times; maximum nesting depth is 4	
%]		
%[N]{	randomly permute formatted sequence included in the brackets and keep *N* characters from the permuted sequence	
%}		
%[N]<	treat the character sequence included in the brackets as a character set and insert *N* characters from this set	
%>		

With the format option at hand, you can easily define your own **rules for creating passwords for specific purposes**. Some examples are given in the following table.

Rule	Format sequence
4 upper-case letters, 4 lower-case letters, 2 digits, 1 special symbol *in random order*	`%{%4u%4l%2d%s%}`
8 alphanumeric characters, of which at least 1 is a letter and at least 1 is a digit	`%{%6A%L%d%}`

4 artificial words, where each word consists of 6 letters and is composed of alternating consonants and vocals	`%4[%3[%c%v%] %]`
3 artificial "phonetic" words (generation is based on trigram frequencies) with 8 letters each	`%3[%8q %]`
phonetic password interspersed with digits and special symbols	`%6q%2d%s%6q%2d%s`
5 words from the currently loaded word list, each combined with 2 digits	`%5[%w-%2d %]`
random permutation of upper-case letters	`%*u`
random number in the range 0123-9876, each digit must occur only once	`%*4d`
10 alphanumeric characters, first character must not be a lower-case letter	`%U%9A`
random product ID	`%5d-OEM-%7d-%5d`
hexadecimal 128-bit key (e.g., WEP key)	`%32h`
random MAC address (48-bit)	`%5[%2h-%]%2h`
12 random base64 symbols	`%12<<base64>%>`
128-bit key in binary notation	`%128<01%>`
8 random characters from the German alphabet	`%8<<AZ><az>ÄÖÜäöüß%>`
3 artificial *German* words, where each word consists of 6 letters and is composed of alternating consonants (including *ß*) and vocals (including *ä*, *ö* and *ü*)	`%3[%3[%<bcdfghjklmnpqrstvwxyzß%>%<aeiouäöü%>%] %]`
At least 4, at most 8 words from the currently loaded word list, each word must occur only once	`%*4-8w`

Advanced Password Options

Advanced

Clicking on this button opens a dialog where you can activate or deactivate some "advanced" password options:

- **Exclude ambiguous characters:** Excludes those characters from character sets that might be confused with other similar-looking characters (that is, by default, `B8G6I1l|0OQDS5Z2`). As this option reduces the size of the character set, the password security will be reduced accordingly. This option also applies to formatted passwords and affects all character sets encoded by the placeholders %a, %A, %U, etc. This option does not affect phonetic passwords.

- **First character must not be a lower-case letter:** Activate this option if you don't want the first password character to be a lower-case letter (a–z). This might be useful when copying passwords to certain word processors or e-mail programs that automatically convert the first character to upper-case in case it is a lower-case letter, thereby manipulating the original password. If the first character belongs to a word, it will simply be converted to upper-case. If it is, however, a ran-

dom character, PWGen will choose a character that is *not* a lower-case letter (for example, if the character set consists of lower-case letters and numbers, the first character will be a number). If the character set does not contain any non-lower-case letters, the first character will be an upper-case letter. Note that activating this option (slightly) reduces the password security.

- **Don't separate words by a space**: By default, PWGen inserts spaces to separate words in passphrases. Select this option to deactivate this behavior.

- **Don't separate words and characters by a '-' character**: By default, PWGen inserts a minus sign ("`-`") between a word and a character (when the option *Combine words with characters* is selected). Select this option to deactivate this behavior.

- **Reverse default order of character/word combinations**: When you generate passwords/passphrases consisting of both characters and words, PWGen by default puts the characters at the beginning and appends the words. If *Combine words with characters* is activated, a subset of the character sequence is appended to each word (e.g., `word1-3wp word2-k8#`). If you activate this option, however, the order is reversed in each case: If *Combine ...* is activated, each word is appended to a subset of characters (e.g., `3wp-word1 k8#-word2`); otherwise, the words are put first, and the characters are appended.

- **Include at least ...**: Activate these options to include at least one character from the given character sets, upper-case letters (A–Z), lower-case letters (a–z), digits (0–9), and special symbols (`!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~`, or user-defined symbols, see below). If the user-specified character set does not contain any characters from a predefined set, the resulting passwords will contain exactly one character from this predefined set. Otherwise, they will contain *at least* one character from this set. For example, if you specify a `<az>` character set and activate all of the aforementioned options, PWGen will generate passwords like this one: `t]cXy7cu`. (If the desired password length is less than 4, not all characters can be included, of course.) Depending on the user-specified character set, selecting these options can—rather slightly—reduce (but in some cases also *increase*) the password security. Note that these options do not affect the generation of pass*phrases* or formatted passwords.

- **Only include characters from custom character set**: This option refers to the *Include at least …* options explained above. If activated, characters from the user-defined (custom) character set will be used exclusively for generating passwords. For example, if the character set is defined as `abcdEFGH012345+/` and all four *Include at least …* options are activated, generated passwords will contain ≥1 lower-case letter from the set `abcd`, ≥1 upper-case letter from the set `EFGH`, ≥1 digit from the set `012345`, and ≥1 special symbol from the set `+/`. If the custom set does not contain any matching characters from a given *Include …* character set, charac-

ters from that set will not be included in the resulting password. This would be the case for the custom character set `abcdEFGH012345` and the option *Include at least one special symbol*: As the custom set does not contain any special symbols, generated passwords will not contain any special symbols either. Also, note that the "special symbols" character set may be redefined (see below), which would also affect this option with respect to matching characters in the custom character set. The option has no effect on phonetic passwords, i.e., these passwords always contain characters from the default *Include …* character sets if the corresponding options are selected.

- ***Exclude repeating consecutive characters*:** Choose this option if you want to avoid repeating sequences of the same character, as in `r3aa5ZiK` or `E0u555Wp`, for example. If this option is activated, each character in the sequence will be different from the previous one. This option certainly reduces the security of the passwords, but the extent of this reduction strongly depends on the size of the user-defined character set: For large character sets, this effect becomes almost negligible. For example, if your set contains 64 characters (6 bits per character) and the password length is 16, the loss of entropy is only 0.34 bits of 96 bits (0.35%). The effect is more pronounced for small character sets: If your set contains only 4 characters (2 bits per character) and the password length is 48, the loss of entropy is 19.5 bits of 96 bits (20.3%). This option also applies to formatted passwords.

- ***Exclude duplicate entries in password lists*:** Select this option if you want to generate "unique" password lists where each entry occurs only once. Generating unique lists may take longer than usual due to the extra check whether the entries to be added already exist in the list. It also requires roughly the double amount of memory. In certain cases, it might not be possible to generate complete password lists using the parameters given by the user; for example, if the user chooses `0123456789` as the character set (= 10 possibilities) and "3" as the password length, it is not possible to generate a list of more than 1000 (= 10^3) unique entries, since the total number of possible password permutations is limited to this exact number. As a consequence, the security of the entire password list is decreased, because the number of possibilities decreases from top to bottom in the list. However, when a single password independent from the rest of the list is considered, it retains its designated security. Note that you can cancel the generation process any time.

- ***Convert all words in word lists to lower-case*:** When loading word lists, each word in the list must occur only once, and the words are compared in a case-sensitive manner. So if you don't want the list to contain both upper-case and lower-case words, you can activate this option and thus force PWGen to convert each letter in a word to lower-case.

- ***Each character/word must occur only once*:** In passwords, each character will occur only once; in passphrases, each word will occur only once. Note that this op-

tion limits the password length and the number of words to the size of the character set and of the word list, respectively; if you enter a number exceeding this limit, you will receive a warning message, and the number will be decreased automatically. Also note that this option *always* decreases the password security: For example, if the size of the character set is given by N, and M characters are to be inserted from this set, the entropy of the resulting sequence can be calculated according to $S = M \log_2 N$ or $S_{red} = \log_2(N!/(N-M)!)$ if this option is *de*activated or activated, respectively. Due to the factorial terms S_{red} is always smaller than S (for $M > 1$). The effect of entropy reduction becomes less pronounced with increasing N and comparably small M ($N \gg M$): For $N = 26$ and $M = 12$ (e.g., 12 random letters), S is 56.4, whereas S_{red} is 52.0, corresponding to an entropy reduction of 8%. However, for $N = 8192$ and $M = 12$ (e.g., 12 random words from the standard word list), the effect is practically negligible (156 vs. 155.99, corresponding to a reduction of 0.006%). When this option is activated or the "asterisk" (*) argument in formatted passwords (see: *Format Password*) is used, PWGen calculates the reduced entropy S_{red} for the resulting password.

Most of these options can potentially—but not necessarily!—reduce the security of the resulting passwords. See the corresponding notes for the options above for more details. If one or more of these options are activated, the caption of the *Advanced* button is marked with the sequence "(!)".

Tip: Right-clicking on the list of options opens a menu where you can (de)select all items or invert the current selection.

***Redefine ambiguous characters*:** The set of ambiguous characters is defined as `B8G6I1l|0OQDS5Z2` by default. Redefining this set also affects the other pre-defined character sets (i.e., *<AZ>, <az>, <09>,* etc., as well as all the character sets accessible via the *Format Password* option). You can also supply this character set in *groups* of similar-looking characters, separated by a space. Then, each group of this set is only excluded from the user-defined character sets if they contain *two or more* characters from this group; if only one character is used, the group will *not* be excluded. Please note that

- a group must consist of at least two characters;

- each character must only occur once in the whole sequence, i.e., it cannot be part of more than one group;

- you can still exclude the space character itself by supplying it at position 1 or 2 in the sequence (e.g., `1 0OQD` is not a valid definition of groups, so the sequence will be treated as a "normal" set of ambiguous characters, including the space character);

- group definitions exclusively apply to user-defined character sets (i.e., via option *Include Characters* and via format specifier `%<...%>`). For all pre-defined character

sets, *all ambiguous characters* present in the sequence will be excluded, independent from any group definitions!

Example: Enter `B8 G6 I1l| 0OQD S5 Z2` to define *six groups* of similar-looking characters. If your character set is defined as `<az><09>` (i.e., lower-case letters and digits), only the characters `1` and `l` from group 3 will be excluded, whereas all the characters from the other groups remain unaffected, because your character set doesn't contain any other characters from the groups which might be confused with the characters in the set. All groups will be excluded, however, if you add upper-case letters to the set.

Redefine special symbols: The set of special symbols is defined as `!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~` by default. Redefining this character set affects the *<symbols>* character set, the option *Include at least one special symbol,* and the corresponding placeholder character sets in the *Format Password* option.

The field **Maximum length of words in word lists** should be self-explanatory. When loading a word list, only those words the lengths of which do not exceed this value will be added to the list. The maximum word length can range between 1 and 30 (default). Changing this value is particularly useful to filter out very long words, or to reduce the size of the entire word list. Note that this option does not apply to the default word list but only to lists from an external source.

Trigram file for generating phonetic (pronounceable) passwords: Here you can specify a special "trigram file", i.e., a file containing the frequencies of all 17,576 (26^3) possible trigrams (= 3-letter combinations: *aaa, aab, …, zzz*) for a given language. This file should have the extension ".tgm". PWGen uses the trigram frequencies to generate phonetic passwords (see *Include Characters*). Trigram files can be downloaded from the PWGen download page, but you may also create your own files via the menu item *Tools / Create Trigram File (F6)* in the main menu. If the input field is left blank, PWGen will use its internal (English) trigram table.

Generate Multiple Passwords

If you want to generate more than one password and display the list in a separate window, enter the exact number into this field (maximum 2,000,000,000) and click on **Generate**. The program will open a window showing the password list. Right-click into the window to access the context menu with additional functions:

- **Copy:** Copies the current text selection to the clipboard.

- **Encrypt & Copy:** Encrypts the selected text and copies the ciphertext to the clipboard (see also: *Encrypt/Decrypt Clipboard (F3/F4)*)

- **Select All:** Selects the entire text.

- **Save As File:** Writes the text to a file of your choice.

- **Change Font:** Changes the text font.

You may also **drag & drop** the password list or a selection thereof by left-clicking on a text selection, holding the mouse button, and dragging the text to other applications which have been registered as "drop targets".

Note: PWGen is a 32-bit application and thus cannot handle more than 2 gigabytes (2,147,483,647 bytes) of RAM. Considering a safety factor of ~4 for various buffering operations and Unicode transcoding, PWGen limits the size of memory buffers for password lists to 500,000,000 bytes. Generating very large lists may exceed the available memory (of your system or of the program) and cause an "out of memory" error. If the option *Exclude duplicate entries in password lists* (see: *Advanced Password Options*) is activated, the memory requirement is roughly doubled due to the creation of a binary search tree, which allows for a fast search of the existing password entries.

If you want the list to be **directly written to a file** instead of being displayed in a window, click on the button 🖺 and specify a file where the list is to be stored. PWGen will write each password to this file without any limitations regarding RAM requirements. Instead, this process only requires negligible 64 kilobytes of memory, and the amount of data that can be generated is only limited by the free disk space on the device. Thus, you can theoretically generate several gigabytes of data (and more … but note that I have not tested this myself!). Note that this low memory requirement does *not* apply to unique lists generated with the option *Exclude duplicate entries*.

As of version 2.3.0, PWGen is capable of writing Unicode-compliant text files encoded as UTF-16 or UTF-8 (depending on the corresponding setting under *Files* in the configuration; 8-bit non-Unicode ANSI conversion is supported, too). Each generated password will be encoded accordingly before being written to the file.

If the destination file already exists, you have the possibility of extending the list by appending new passwords to the file. This function is not compatible with the option *Exclude duplicate entries*, i.e., the newly generated passwords will not be checked against the already existing passwords in the file. Instead, a new unique list will be generated and appended to the file.

If you generate a larger amount of passwords (more than, say, 1000 passwords), PWGen shows a progress window with a *Cancel* button. Click on *Cancel* to stop the process and show a list of the passwords which have been generated so far.

Generate Single Passwords
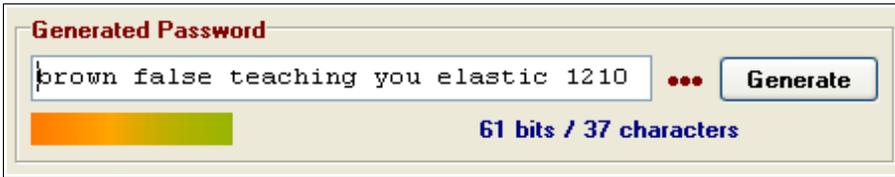
If you want to generate one single password, click on **Generate**. PWGen will display the password in the box.

The estimated security of the password is indicated by the colored "**security bar**" below the password box. The color of this bar ranges from red-orange to light green, i.e., from "*low security*" to "*high security*", where the limit for "*highest security*" has been—more or less arbitrarily—set to 128 bits (which is just some kind of a "magic number" for cryptographers). Always stay on the green side when generating "serious" passwords!

Drag & Drop: Left-click on the security bar or on the "*Generated Password*" group box (on top of the password box) and hold the mouse button to start a drag & drop operation. Keep the button held down while dragging the password into other applications which have been registered as "drop targets". If the password box contains a selection of characters, only this selection will be dragged & dropped. Otherwise, the entire password will be used. Conversely, you can also drop text from other applications into the password box, provided that the *Editable* option (available in the context menu of the box) is enabled.

Note that the *security of a single password is limited to 256 bits* because the "random pool" which is used to generate passwords has an effective size of 256 bits (which corresponds to the message-digest length and the key length of SHA-256 and AES, respectively). However, if the random pool hasn't been filled with enough entropy before password generation, the actual security is lower than the value shown in this field—to be more precise here, the pool must have been filled with *at least N* bits with *N* being the password security. For more information on this topic, see: *How shall I interpret the information about the random pool?*

If—for whatever reason (severe paranoia?)—you want to create a password with a security of more than 256 bits, you may concatenate two or more 256-bit passwords to a 512-bit (or more) password. This is done as follows: First, collect entropy for a 256-bit password, generate it and store it somewhere; then collect entropy for a second password and concatenate that with the first one. Note, however, that there are *absolutely no concerns* that it will *ever* be possible to break 256-bit keys.

PWGen also offers you the possibility to test your own password creations. Right-click in the password box to open the context menu and activate the options **Editable** (allows editing the contents of the text box) and **Enable Password Testing**. Now you can type into the box or paste text from the clipboard, and PWGen *estimates*(!) the security/quality (in bits) of the entered sequence. A leading asterisk symbol (*) will be displayed in the

label (i.e., "*xx bits / xx characters"), indicating that the displayed security value in bits is roughly estimated and therefore not exact. Note that the underlying algorithm is quite simple and does not make use of any sophisticated statistical tests for randomness (e.g., autocorrelation, runs test, entropy estimation, etc.), since passwords—even though they may be chosen randomly—are typically much too short for these kinds of tests to be applicable. As a consequence, the algorithm tends to *under*rate passwords (consisting of random characters) and to *over*rate passphrases (consisting of random words from a word list). Furthermore, please note that PWGen does *not* test for popular (= bad) passwords like "qwerty", "1234", and so on. So please apply some common sense when judging passwords!

Clicking on the **symbol** ••• hides the password characters behind asterisks (***).

You can change the font which is used to display the password by right-clicking in the box and selecting the entry ***Change Font*** from the context menu.

Random Pool
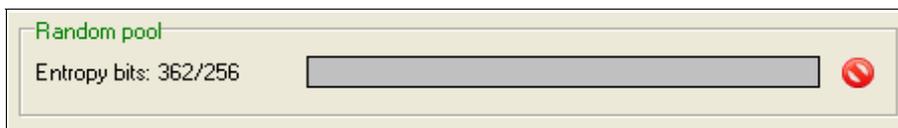
This progress bar informs you about the current state of the random pool, i.e., the amount of entropy bits the pool can currently provide. Whenever you generate one or more passwords, $N·p$ bits will be "consumed" from the pool (N = number of passwords, p = bits of entropy present in every password).

You can provide entropy by moving your mouse, by clicking with your mouse and by typing on your keyboard. Furthermore, the program regularly collects entropy from various system parameters.

As stated above, the random pool can yield a maximum Shannon entropy of 256 bits. However, PWGen also shows the non-limited entropy amount (possibly *exceeding* 256 bits), which might be of interest if you don't trust PWGen's estimations regarding the entropy content of the different sources. So the message "*Entropy Bits: 416/256*" means that you have provided 416 entropy bits so far, and that the random pool has reached maximum entropy *according to PWGen's estimations*. The non-limited entropy counter is increased (up to 9,999 maximum) as long as you don't generate any passwords. Whenever you "consume" entropy from the pool, the "total entropy" counter will assume a value less than 256 bits, of course. Additionally, there is a *total entropy counter* which counts the amount of entropy gathered by the program during its runtime (up to 999,999,999 maximum), and which is *not* decreased after consuming entropy from the pool. The value of this counter is displayed in the context menu of the progress bar (right-click). You can reset the counters by selecting *Reset Counters* from the menu (this will not affect the actual contents of the random pool in any way).

If the blinking of the progress bar irritates you, click on the **symbol** 🚫 to hide the entropy progress.

Main Menu   <kbd>File Tools Options Help</kbd>

Note that some of the following functions can also be accessed via the toolbar buttons right below the main menu.

File

Profile

Shows a submenu containing all password generation profiles. To load a profile, select the respective menu item or press the designated key combination[2] (<Shift>+<Alt>+<0>, +<1>, …, +<9>, +<A>, …, +<Z>). All the settings for generating passwords (including advanced password options, if saved additionally) will be restored according to the selected profile.

Profile / Profile Editor (<F10>)
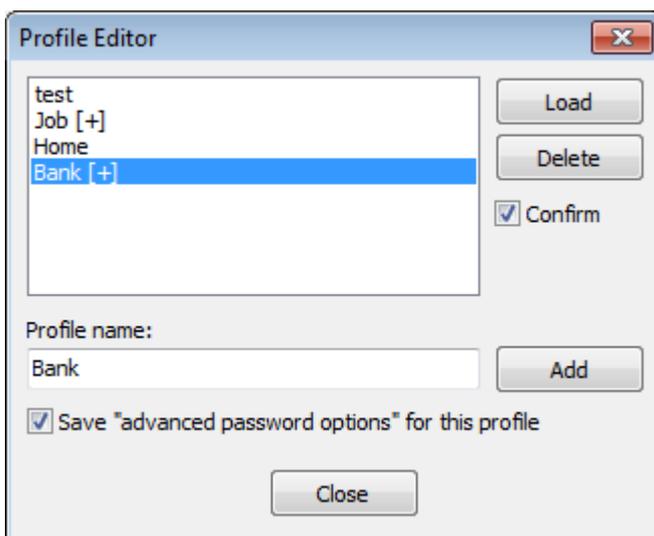
In this dialog, you can manage your profiles, i.e., add/overwrite, load and delete profiles. Note that the number of profiles is limited to 50, and that profiles which contain additional advanced password options are marked with "[+]".

Add/overwrite profiles: Enter the name of the profile into the ***Profile name*** box and click on ***Add***. Activate ***Save "advanced password options" for this profile*** if you want to explicitly store the advanced options with this profile; if you leave this option deactivated instead, the current advanced options will never be modified when loading this profile.

---

2 The key combination used to be <Ctrl>+<Alt>+… in previous versions, but the <Ctrl>+<Alt> combination equals the <AltGr> key in Windows, which leads to ambiguities when the user tries to access alternative characters on the keyboard using the <AltGr> key.

If a profile with an identical name already exists and **Confirm** is activated, you will be asked to confirm the overwriting.

Load a profile: Select the profile of your choice and click on **Load**, or double-click on the entry.

Delete profiles: Select the profile(s) you want to delete and click on **Delete**. If **Confirm** is activated, you will be asked to confirm the deletion.

Note that there are no *OK* and *Cancel* buttons in this dialog; all changes to the profiles are made active instantaneously, and there is no possibility to undo an accidental deletion within a PWGen session. To restore the profile settings to the "start-up state" (initial set-tings loaded from the configuration file on start-up), close the window, make sure that the *Options / Save Settings on Exit* menu item is deactivated, and restart the program.

Exit (<Alt>+<Q>)

Closes the application.

Tools

Clear Clipboard (<F2>)

Clears the text contents of the clipboard.

### Encrypt/Decrypt Clipboard (<F3>/<F4>)

Encrypts or decrypts the clipboard text and stores the result in the clipboard. At first you have to enter a password which can be of any length. The clipboard text (if there's any) will then be en-/decrypted using a secure 256-bit key derived from this password. The encryption algorithm is AES with a key length of 256 bits. Note that the text is com-pressed before encryption. (For more information about the exact encryption procedure, see: *Text Encryption*.) The ciphertext will be converted to the base64 format consisting of letters (A-Z, a-z), numbers (0-9) and the symbols "+", "/" and possibly "=". In the result-ing text, lines are automatically wrapped after 76 characters. It is, of course, strongly recommended to check the encryption before discarding the plaintext. If decryption fails, you probably entered a wrong password. Another possibility is that the text is corrupted, i.e., it has been modified in some (bad) way. Note that decryption will always fail if one or more characters in the ciphertext have been altered (except for characters involved in line breaks). Inserting or removing line breaks in the ciphertext does *not* pose a problem.

The clipboard encryption is not intended for encrypting very long texts. (Please use a file encryption utility for this purpose.) To avoid memory problems, PWGen currently limits the text length to 128 megabytes.

Important note: The encryption scheme has been changed (again) in version 2.3.0 to provide Unicode support. The plaintext is converted to UTF-8 before compression and en-

cryption. For backwards compatibility, i.e., to decrypt ciphertexts generated with older versions, you can select the option *PWGen <2.3.0* in the password dialog when decrypting texts. It is strongly recommended to re-encrypt any existing ciphertexts with the new version.

Create Random Data File (<F5>)
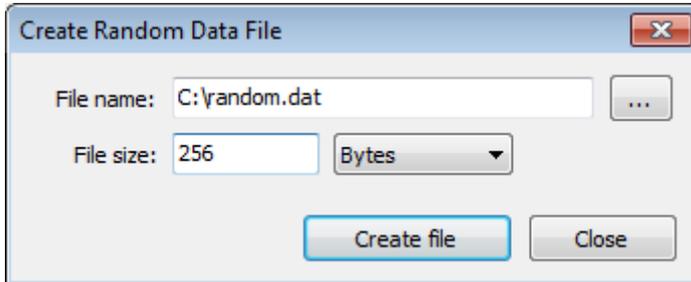
Creates a file consisting of purely random (i.e., *cryptographically random*) data. First, select an existing file (by clicking on the **Browse** "**...**" button) or enter the **name of the file** where the random data is to be stored. Then enter the desired **file size** in *bytes*, *kilobytes* (1,024 bytes) or *megabytes* (1,048,576 bytes) and select the appropriate list entry. Note that you can enter floating-point numbers (e.g., "0.5", "3.14", "2.25e3", etc.), but the number of bytes to write will be converted to an integer, of course. In case of a file size of more than 1 megabyte, you have the possibility to stop the creation process by clicking on the *Cancel* button in the progress window.

Note: PWGen cannot handle file sizes >2 gigabytes. Entering file sizes greater than 2,147,483,647 bytes will cause an error. If you need to generate very large files, please contact me. I will then consider adding this feature in a future release.

Create Trigram File (<F6>)
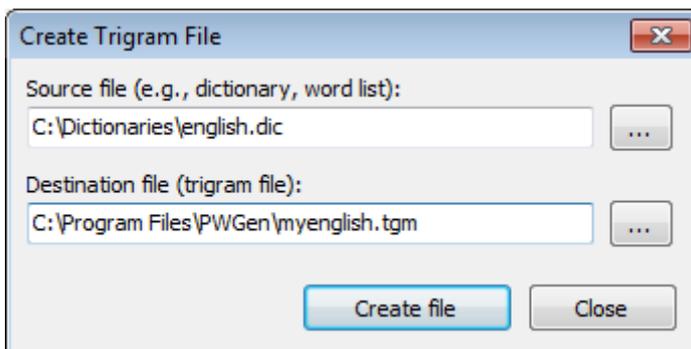
Here you can create a "trigram file", i.e., a file containing the frequencies of all possible $26^3 = 17,576$ trigram (3-letter) combinations (*aaa*, *aab*, ..., *zzz*). The **Source file** may be any (Unicode or ANSI) text file with words consisting of letters (lower-/upper-case) of the English alphabet (A–Z, a–z); all other symbols—even derived letters such as *ä, è, î*, etc.— are ignored! When clicking on **Create file**, PWGen will analyze the entire source file and write the trigram statistics to the **Destination file**, which should have an extension of *.tgm*. This file may be loaded via the *Advanced Password Options* dialog to generate custom phonetic passwords. If the destination file could be created successfully, PWGen

shows some information about the number of trigrams evaluated and the amount of en-
tropy per letter. If all trigrams had the same occurrence, the entropy per letter would be
$\log_2 26 \approx 4.7$. However, as every language has its own "phonetic rules", the trigram fre-
quencies tend to exhibit large variations, which lead to a decrease in the amount of en-
tropy.

Word lists, and, even better, dictionaries are especially useful for the trigram analysis.
Novels or other long texts are suitable, too, but the resulting entropy may be lower com-
pared to dictionaries: In texts, the different frequencies of words must be taken into ac-
count, which increases the frequencies of certain trigrams and thus *decreases* the total
entropy even further. (The resulting passwords might be better pronounceable, though).
Typical entropy values are ~3.5 for dictionaries and ~3.0 for long texts (tested for English
and German).

Example: In order to generate phonetic passwords in a specific language with Lat-
in-based letters, it would be constructive to analyze a large dictionary (>1 megabyte) of
that language. Since all non-Latin characters are ignored by the algorithm, it may be
helpful to convert all umlauts, accented letters, etc. to allowed letters (a–z).

MP Password Generator (<F8>)

"MP Password Generator" (or even shorter: MPPG) stands for "Master-Password based
Password Generator". This tool allows you to create (practically) unique passwords for
various applications, such as websites, bank accounts, etc. Basically, it works as follows:
The user provides a *master password* and a *parameter* (e.g., the name of the website).
Then, PWGen uses strong cryptographic algorithms to derive a password of variable
length from these user-supplied data: HMAC-SHA-256 for deriving a 256-bit key, and
AES-256 in counter mode as cryptographically-strong pseudo-random number generator
(CSPRNG) for generating passwords. Thus, the MPPG can serve as a fully portable "pass-
word safe", functioning without any encrypted files (which are usually essential for ac-
cessing one's passwords). Instead, each password can be directly accessed by its unique
identifier (= parameter). By default, PWGen uses its own method to generate passwords;
however, an option is provided which enables full compatibility with the online password
generator tool *Hashapass*. Let's have a look at the details of the usage:
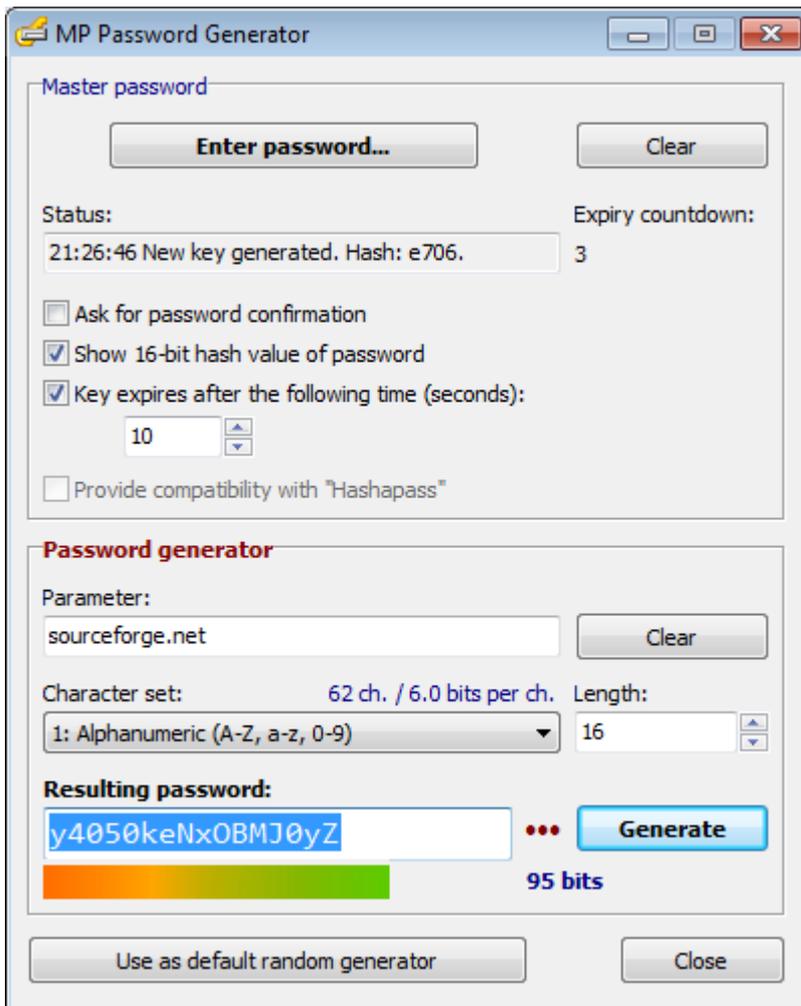
Set master password: Clicking on the button **Enter password** opens a dialog where you can enter your master password. If the option **Ask for password confirmation** is activated, you have to enter the password twice to avoid using a wrong password due to a typing error in the input string.

Clear password: Click on the **Clear** button to clear the key cache, the parameter, and the lastly generated password.

Once the master password has been set, a new message in the **Status** field is displayed, featuring the current time at the beginning. The time stamp is added so as to inform the user that the password has been updated successfully. If the option **Show 16-bit hash value of password** is activated, the status message additionally contains a 16-bit hash value of the password in hexadecimal notation. This hash may be helpful for quickly checking whether the password has been entered correctly. Checking a short hash sequence may be more convenient than always having to enter the password twice. The 16-bit hash has a complexity of $2^{16} = 65,536$, so the probability that a wrongly typed password has the same hash as the correct one is about 0.0015%, which may be considered negligible if the amount of typing errors in the password is limited to a few positions. A possible downside is that you have to remember another character string along with your

password. Note that it is not possible to derive any information about the master password from this hash.

Key expiry: For reasons of convenience, PWGen can store the master password in memory for easy re-use. This so-called "key cache" is encrypted with a random 128-bit key, which is itself stored in a different memory location. If the option **Key expires after the following time** is *de*activated, the key "expires" only when the user clears it manually, or when the window (or the whole application) is closed. If the option is activated, the key cache will be cleared some time after a new master password has been entered. You can enter a value (in seconds) in the range 0–32,767 in the corresponding input box:

- A special case applies to the number "0", which means that the key will not expire before a new password has been generated for the first time. In this case, only the master password will be cleared, whereas the parameter string and the generated password will remain unaffected.

- In all other cases, a countdown for the expiry of the key will be displayed in the field next to the *Status* box as soon as the user has entered a master password. When the countdown reaches 0, the key cache, the parameter and the lastly generated password will be cleared.

Hashapass compatibility: Some users may want to be independent of PWGen with respect to the MPPG; for example, when traveling, or when using other peoples' computers, it may not be possible to install or even execute PWGen (although the application should run on legacy Windows versions like Windows 98, and should also run on Unix-like OS via [Wine](#)). For these users, PWGen offers the option **Provide compatibility with Hashapass**: The functionality is comparable to PWGen's MPPG (although I think that the latter is more secure—but my opinion may be slightly biased; *see table below for a detailed comparison*), but Hashapass has the advantage that its [website](#) is accessible via the world wide web and should be compatible with all popular browsers, independent of the operating system (Windows/Unix/Mac OS/...). Hashapass generates passwords in offline mode "*in the safety of your web browser, without any information being transmitted over the Internet*" (cited from the website). Unfortunately, the password length is limited (fixed) to 8 characters (= 48 bits of security), and key caching is not supported, so the master password always has to be re-entered by the user to generate a new password. Another downside is that the master password has to be stored in memory (albeit in encrypted form), whereas with PWGen's own method, a 256-bit hash value of the master password is cached, which does not reveal any information about the master password itself.

Note: Due to the differences in key caching the selection of the *Hashapass* option cannot be changed anymore as soon as the master password has been set. To change the selection, clear the key cache (via the *Clear* button), (de)select the option and enter the password again.

Enter a unique **Parameter** into the corresponding input box. It can be any kind of character string, for example, the name of a website or application, a URL, etc. It is best to choose a distinct, yet easy to memorize parameter. Note that the box can act both as a drop target (receive drag & drop text from other applications) and as a drop source by clicking on the *Parameter* label and dragging the text into other applications.

Choose a **Character set** from the drop-down list. Currently, PWGen provides 6 different sets, some of them are available without ambiguous characters (`B8G6I11|0OQDS5Z2`). You may wonder why there isn't any option to define custom character sets. The reason is that the MPPG functionality should be as portable as possible, that is, it should work without having to remember a plethora of settings such as the exact definition of the character set. Even if you don't carry a personalized version of PWGen with you all the time, you can download a new copy from the Internet and execute it on Windows (or via *Wine;* installation is not necessary!). The only settings you have to remember are the type/index of the character set and the password length. If you don't know which character set and/or which password length to choose, simply use the default settings.

Enter the desired **Password length** (number of characters) into the corresponding input box. For security reasons, I recommend using the default value of 16 characters (or more; at least 12). Unfortunately, some applications limit the password length to only 8 characters, which may be too short for sensitive data or applications (e.g., online banking accounts). Simply use the first 8 characters of the generated password in such cases.

Click on the **Generate** button to generate the password which results from the unique combination of the master password and the parameter string. The quality of the password (measured in bits) is displayed as a number and illustrated by the length of the "password quality bar" (see: *Generate Single Passwords* for more details) below the password box. Note: The font for displaying the password is the same as that for the password box in the main window.

Drag & Drop: Passwords can be dragged & dropped just like those generated in the main window. Move your mouse to the security bar or to the "*Resulting password:*" label (on top of the password box) and hold the left mouse button to initiate a drag & drop operation. Conversely, you may also drop text from other applications into the box.

Click on the **symbol** ••• to hide the password behind asterisks (***).

The button **Use as default random generator** enables you to make PWGen's random generator completely "deterministic" by replacing the default generator (random pool) with the pseudo-random number generator (PRNG) which is used for the MPPG. In this case, all passwords and random data files generated with PWGen are "deterministic" in a sense that they can be reproduced *exactly* by using the same master password and parameter again. (Note that the output of this PRNG is still cryptographically-secure, since it is based on AES with a 256-bit key.) In contrast, the random pool was designed to never generate the same output twice (at least the probability of a repeat should be so low that

it is practically impossible)[3]. By using this option you can extend the "password hasher" functionality *practically to the entire application*. Note that the random pool will still be active and gather entropy from user inputs and system-specific parameters, but it will not be used for generating passwords and random data files.

For technical reasons, this button is only accessible if the *Hashapass compatibility* option is *not* activated. Otherwise, you have to clear the key, deactivate the option and re-enter the password to gain access to it.

Please read the following notes carefully:

- **WARNING 1:** I cannot guarantee that the algorithms for generating passwords in the main window ((phonetic) passwords, passphrases, formatted passwords) will never change again in future program versions. So it *might* be possible that some kinds of passwords generated in the main window using the MPPG option cannot be reproduced in some future version of PWGen. However, this guarantee is given for the MP Password Generator itself; if the MPPG algorithm has indeed to be changed due to a (severe) bug in the code, I will at least provide an option for backwards compatibility.

- **WARNING 2:** Never forget to deactivate the deterministic random generator as soon as you don't need it anymore (a warning message is displayed in the window caption to remind you of that). It can be deactivated via the menu item *Tools / Deterministic Random Generator / Deactivate*, or by pressing F7. Upon deactivation, the random pool will be re-used as default random generator.

Reset: The deterministic random generator can be reset (i.e., set to its initial state right after activation) via the menu item *Tools / Deterministic Random Generator / Reset*, or by pressing <Ctrl>+<F8>.

The button **Close** first clears the key cache and all sensitive input fields (parameter and password box) before closing the window. Note that the deterministic random generator may still be active and has to be turned off separately (simply press <F7>).

CAUTION: I strongly recommend using a "physical" password safe such as *KeePass* or *Password Safe* in addition to the MPPG. This double-tracked approach protects against memory loss regarding parameters (though not the master password, of course), and ensures that a complete list of all your parameter–password combinations (maybe with additional information like user names and personal comments) is available as a backup. If you don't want to use such a password safe, you can also keep a list of all parameters in a simple (encrypted?) text file.

---

3   From a theoretical standpoint, the random pool is also a "pseudo-random" number generator because it uses deterministic algorithms to generate random numbers. However, the entropy provided by the user and the computer system (especially the high-resolution timer provided by the CPU) make the construction more and more "indeterministic".

The following table lists the differences between PWGen's MP Password Generator and the Hashapass password generator:

Parameter	PWGen's MPPG	Hashapass
password length	1–32,767 characters	8 characters (fixed)
password character set	Currently 6 sets to choose from	Base64 (*A-Z, a-z, 0-9, +, /, =*)
password security	Up to 256 bits (variable)	48 bits (fixed)
mode of password generation	1) Derive a 256-bit hash *H* from the master password by applying HMAC-SHA-256 (to avoid caching the password as plaintext); 2) Derive a 256-bit key *K* from *H* and a user-defined parameter *P* by applying PBKDF2 (based on HMAC-SHA-256) with 8192 iterations; 3) Initialize AES with *K* and use AES in counter mode to generate random numbers; change key after every 1 megabyte of random output	1) Apply HMAC-SHA-1 to the master password and the parameter; 2) Take the first 48 bits and convert it to base64, which yields the final 8-character password
cryptographic primitives used	1) SHA-256 (*256-bit one-way hash function*) 2) AES (*128-bit block cipher with 256-bit key*)	SHA-1 (*160-bit one-way hash function*)
information stored in encrypted key cache	256-bit hash *H* of the master password	Master password
Unicode support	Full support	Partial support (only the lowest significant byte of each Unicode character is used, so some non-ASCII characters are not distinguishable when used together in a string)
availability	PWGen for Windows (32-bit & 64-bit), on other operating systems possibly via *Wine*	Website, should work with all popular browsers; PWGen and possibly other applications

Deterministic Random Generator

This menu controls the so-called "deterministic" random generator, which can be activated in the [MP Password Generator](#) dialog via the button *Use as default random generator*.

Reset (<Ctrl>+<F8>): Resets the random generator to its initial state, thus allowing you to reproduce the entire sequence of random numbers generated so far.

Deactivate (<F7>): Destroys this random generator and resets the random pool as the default random generator.

Provide Additional Entropy

As Text (<F9>)
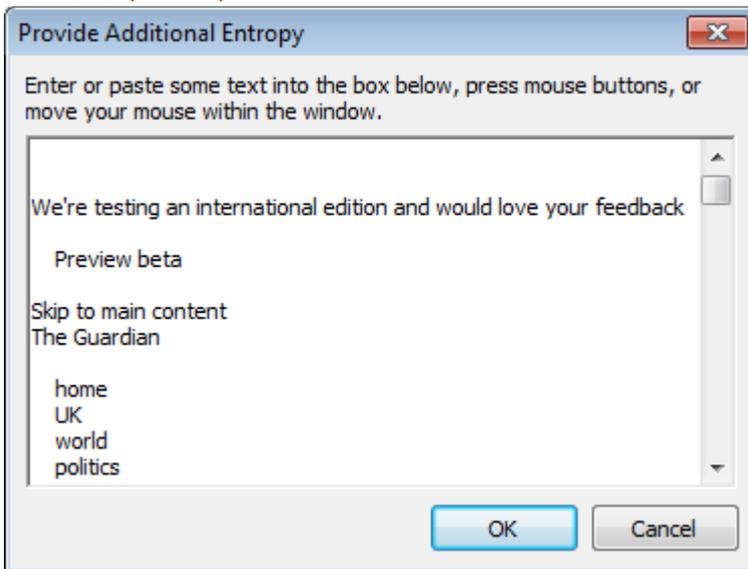
Opens a window with a text box where you can enter some text (by pressing keys, pasting text from the clipboard, or via drag & drop from other applications), press mouse buttons, and move your mouse in order to feed the random pool with entropy (i.e., data which is indeterministic to a certain extent). When you're finished, click on **OK** to add the text to the random pool and evaluate it with respect to its entropy content: The text is first converted to UTF-8 encoding and compressed; the compressed data is then fed into the random pool, and the length of the compressed data block in bits, downgraded by a safety factor, is added to the entropy counter of the random pool. Clicking on **Cancel** skips this procedure.

Note that each interaction with your keyboard or mouse—no matter in which window—is added to the pool instantaneously, indicated by discrete increments in the "Entropy bits" counter (displayed in the main window). For these kinds of events, the entropy is mainly determined by the quality/precision of their time stamps. The text in the text box as a whole may contain some additional entropy, provided that the content is "indeterministic" to a certain degree. To fulfill the last requirement, it is recommended to paste longer texts from volatile, non-static sources (e.g., log files, chat protocols, frequently updated web pages, …) into the box. The longer the text, the more precise the entropy estimation will be.

From File (<Ctrl>+<F9>)

Select a file the contents of which you want to incorporate into the random pool. The file is read in 64-KB chunks, which are compressed and then added to the pool (similar to the procedure described above). PWGen assesses each compressed byte with 4 bits of entropy (corresponding to a safety factor of 0.5).

Options

Language

Changes the program language. You will receive a warning message if the language version is not compatible with the program version. The program has to be restarted to load the language file. If you downloaded a language file from the PWGen homepage, extract it into the program directory. PWGen will find it on start-up and add the language as an item to this menu.

Configuration (<F11>)

Allows you to access the main configuration of the program. In the dialog, you can select between three pages, *General*, *Updates*, and *Hot Key*.

General

Change font for the GUI controls**:** Changes the font which is used to display the text in most of the GUI controls in the application. The font of the password box in the main window and of the password list window cannot be changed via this function; please use the corresponding functions in the popup menus of these controls for this purpose. The default font is Tahoma with a size of 8 pt.

Clear clipboard after the following time (seconds)**:** When copying sensitive data such as passwords to the clipboard, it is prudent in many cases to clear the clipboard automatically after a certain time (a few seconds should be sufficient). If this option is activated, a countdown is started whenever any text from password boxes is copied to the clipboard. As soon as the countdown reaches zero, the clipboard is cleared automatically.

Show system tray icon constantly**:** Shows PWGen's program symbol in the system tray (next to the system clock). Clicking on the symbol will open a menu where several functions of the program are accessible.

Minimize program to system tray**:** If activated, PWGen's taskbar button will be hidden when the program is minimized, and the program symbol will be displayed in the system tray instead. If *Show system tray icon constantly* is deactivated, the symbol will be removed when restoring the application.

Files

Character encoding for text files**:** Unicode characters are encoded as UTF-16 internally in Windows, but when writing Unicode text to a file, it may be necessary to transcode the characters in order to ensure compatibility with other applications/ systems. PWGen supports ANSI (*caution*: non-Unicode 8-bit encoding!), UTF-16 (Little Endian), UTF-16 Big Endian, and UTF-8 (see *Unicode Support* for more details on Unicode and the different types of encodings).

Note that in Windows (which also supports UTF-16 and UTF-8 encoding of text files), the UTF-16 encoding is referred to as "Unicode", which is misleading because UTF-8 is as "Unicode" as UTF-16; both are valid Unicode character encodings!

Updates

Automatic check for updates: Enables automatic update checking at regular intervals, i.e., *daily*, *weekly*, or *monthly*. Checks are performed on start-up, and a message will be displayed if a new version is available. You can also *disable* this option and check for updates yourself (see: *Help* / *Check for Updates*).

Hot Key

The "hot key" option allows you to associate a special key combination with PWGen. Whenever you press the designated key(s), PWGen is called and performs an action of your choice: The main window can be shown or restored (if minimized), and/or password(s) can be generated.

Activation: Activate ***Associate the following hot key with PWGen***, place your cursor inside the input box on the right and press the desired key combination, for example <Ctrl>+<Alt>+<P>, <Ctrl>+<F12>, <Alt>+<F8>, etc.

Actions: Select the action(s) which are to be performed when you press the hot key on the keyboard. Note that if *Generate password and show it in a message box* is selected from the list, the main window will always be restored, so that the message box can be placed in the foreground; when you close the box afterwards, the program will be minimized again if necessary.

Note that some key combinations may already be registered by other applications and thus not be accessible.

Always on Top

Select this option to make PWGen the top-most window on the screen, meaning that the main window and subordinate windows will stay on top even when the application is deactivated and another application gets focus.

Known issues: When this option is actively *disabled* during runtime, the "Quick help" window, which is opened by clicking on one of the corresponding buttons in the main window, may be empty. Please click on the button again to display the help text correctly.

Save Settings on Exit

If checked, all program settings will be saved in an .ini file (default: *PWGen.ini*) when the user exits the program.

Save Settings Now

Saves the settings instantaneously.

Help

Open Manual (<F1>)

Opens this manual.

Visit Website

Opens the homepage of PWGen in your default web browser.

Donate

Donations to the project are highly appreciated!

Check for Updates

Connects to the Internet to check if a new program version is available. If so, you may be directed to the download page of this version.

Timer Info

Shows information about the high-resolution timer (HRT) called by PWGen whenever you press a key, move your mouse, or click with your mouse (refer to *High-Resolution Timer* for more in-depth information). "*N/A*" means that a HRT is not available on your system; a low-resolution system date and time value is used instead, providing a resolution in the millisecond range (roughly 1-15 ms). Note that it is recommended to run PWGen on systems where a high-resolution timer (*RDTSC* or *QueryPerformanceCounter*) is available. The current 64-bit value (ranging from 0 to 18,446,744,073,709,551,615) of the timer is also displayed in the message box.

About

Shows copyright information about the program.

Additional Menus

System Tray Menu

The system tray menu, which opens when you click on the PWGen symbol in the system tray, allows you to access some useful functions of the program. In addition to the self-explanatory functions and those already explained above, there are two new menu items that are only available in this menu:

Generate Password

This function quickly generates a password using the settings given in PWGen's main window and copies it to the clipboard. It's like clicking on the *Generate* button with the difference that the password is not displayed but copied to the clipboard instead. This quick generation may be useful whenever you quickly need a password.

Generate and Show Password

Generates a password (see previous menu item) and displays it in a message box before copying it to the clipboard. In the message box, choose *Yes* to copy the password to the clipboard, *No* to generate and display a new password, or *Cancel* to cancel the process. Note that if the password is very long (longer than 1000 characters), it won't be shown and will just be copied to the clipboard.

Configuration File (*PWGen.ini*)

All program settings are stored in an .ini file named *PWGen.ini*, which you can also view and modify manually in a conventional text editor. The location of this file depends on the mode in which the program is run, i.e., *portable* or *installation* mode[4].

In the portable mode, the file is located in the program folder together with *PWGen.exe*, e.g., `C:\Program Files\PWGen`. In the installation mode, the file is located in the *%APPDA-TA%* folder, which is specific for each user on the computer. On Windows 7, for example, *%APPDATA%* is defined as `C:\Users\{user}\AppData\Roaming` (with {*user*} as the place-holder for a specific user name).

When PWGen is installed via *PWGen-xxx-Setup.exe*, the `UseAppDataPath` flag in the *PW-Gen.ini* file located in the installation folder is set to "1" (TRUE) during setup, and PWGen will store all future settings in a *PWGen.ini* file located in the *%APPDATA%* folder (if the file doesn't exist, it will be created). If the flag is set to "0" (FALSE) or doesn't exist, which is the case when PWGen is extracted from *PWGen-xxx.zip* file into a folder without any previous installations of the program, PWGen will store all settings in the program folder.

In previous versions of the program (<2.5.1), PWGen could exclusively be run in the portable mode and thus stored its .ini file always in the program folder. However, this leads to problems when Windows prevents applications from writing to the `Program Files` folder for security reasons (e.g., when the program is installed with administrator rights and executed as a normal/limited user afterwards). This conflict should be solved by the introduction of the installation mode and usage of the *%APPDATA%* folder instead of the program folder for storing application-specific data. You can easily switch between both modes by editing the *PWGen.ini* file in the program folder and setting the `UseAppDataPath` flag to "0" (= portable mode) or "1" (= installation mode). If the entry doesn't exist yet, simply open the file in your favorite text editor and insert a new line into the "[Main]" section of the file:

```
[Main]
UseAppDataPath=1
```

Thus you can easily make an installed version portable by copying the program folder to a USB flash drive, for example, and changing the `UseAppDataPath` flag in the *PWGen.ini* file to "0".

---

4   This setting also applies to the *randseed.dat* file containing 32 bytes of entropy from the random pool (see: *Random Pool*).

Command Line Options

Command line switches have to be prefixed by either one or two minus characters (`-` or `--`), or by a slash character (`/`). PWGen currently supports the following switches:

Switch	Meaning
`ini=<filename>`	Uses the .ini file specified by *<filename>* for reading and storing the settings of the program (instead of using the default .ini file, *PWGen.ini*). Include the file path in quotation marks ("") if it contains any spaces. If the file does not exist, PWGen loads the default settings on start-up and will try to create the file when the settings are to be saved.
`readonly`	Prevents any automatic writing to the disks/volumes on the computer, i.e., PWGen will not *automatically* save any data to the .ini file and the random seed file (*randseed.dat*). The user can still save the program settings by actively calling the corresponding function in the main menu. If no .ini file is specified by the user, PWGen will load the default *PWGen.ini*; if this file does not exist, PWGen will load its default settings stored internally. The random seed file will be read on start-up but otherwise will not be modified in any way.

Example: `PWGen.exe /ini="C:\My Passwords\my.ini" /readonly` loads the settings from the file `my.ini` in the folder `C:\My Passwords`, and prevents any automatic writing to the disks/volumes on the computer.

Questions & Answers

Which security level is appropriate for my password?

As always, this depends on your specific needs. Passwords corresponding to the lowest security level—for example, account passwords for not-so-important websites (there are certainly many of this kind)—should have *at least* 40–48 bits. A higher (say, "intermediate") security level is provided by 64-bit passwords. For sensitive data, a password of at least 72 bits is reasonable. To protect *really* sensitive data, the security should be at least 90 bits. Data that have to be protected for several centuries (if not thousands of years) should be encrypted with a password of at least 112, better 128 bits. Currently, it's technically infeasible to break 128-bit keys, and this statement will hold for many, many years. (In fact, it's *quite* unlikely that there will be ever found a practical way to search a 128-bit key space by brute force. If you don't trust in this assumption, go with 256-bit passwords—they are practically unbreakable.)

The following table shows the lengths of *N*-bit passwords created by different character sets and word lists.

N (bits)	*A*) letters A-Z (4.7 bpc)	*B*) upper-/ lower-case letters, numbers 0-9 (5.9 bpc)	*C*) B + including 37 "special characters" (6.6 bpc)	*D*) words from list with 8192 words (13 bpw)	*E*) combination of D and B (examples)
40	9 (=42 bits)	7 (=41 bits)	6	3 (=39 bits)	2 w., 3 ch.
48	10 (=47 bits)	8	7-8	4 (=52 bits)	3 w., 2 ch.
64	14 (=65 bits)	11 (=65 bits)	10 (=66 bits)	5 (=65 bits)	3 w., 5 ch. / 4 w., 3 ch.
72	16 (=75 bits)	12 (=71 bits)	11	6 (=78 bits)	3 w., 6 ch. / 4 w., 4 ch.
90	19 (=89 bits)	15 (=89 bits)	14 (=92 bits)	7 (=91 bits)	4 w., 7 ch. / 5 w., 5 ch.
112	24	19 (=113 bits)	17	9 (=117 bits)	5 w., 8 ch. / 6 w., 6 ch.
128	27 (=127 bits)	22 (=131 bits)	20 (=132 bits)	10 (=130 bits)	5 w., 11 ch. / 6 w., 9 ch. / 7 w., 7 ch.
256 (max.)	54 (=253 bits)	43	38 (=251 bits)	19 (=247 bits)	10 w., 21 ch.

In the table header, "bpc" means "bits per character" and "bpw" means "bits per word". The table shall give you an idea of the lengths of passwords composed of different character sets. It shows how the size of the character set or the size of the word list, respectively, influences the overall password length. It's obvious that the higher this size (i.e., the more items there are to randomly choose from), the shorter the resulting password for a given "security" in bits.

The next table shows the times to search the entire key spaces of *N*-bit passwords (given by 2^N); that is, for every *N*-bit password, it shows the time to "break" it. The speed (number of keys per second) with which this "brute force attack" can be carried out is limited by the attacker's financial resources (the amount of money the attacker can dispense), as well as by current technology. Note that neither of these variables can grow infinitely: First, limited financial resources are commonplace; second, the computer power is limited by the propagation speed of electromagnetic waves. Considering this second assessment, it becomes evident that attacks on 128-bit keys are infeasible (at least in this universe ...).

N (bits)	10^6 s^{-1}	10^9 s^{-1}	10^{12} s^{-1}	10^{15} s^{-1}	10^{18} s^{-1}	10^{21} s^{-1}	10^{24} s^{-1}
40	12.7 days	18.3 min	1.1 s	< 1 s	< 1 s	< 1 s	< 1 s
48	8.9 years	3.2 d	4.7 min	< 1 s	< 1 s	< 1 s	< 1 s
64	5.8E5 y	584.9 y	213 d	5.1 h	18.4 s	< 1 s	< 1 s
72	1.5E8 y	1.5E5 y	150 y	54.6 d	1.3 h	4.7 s	< 1 s
90	3.9E13 y	3.9E10 y	3.9E7 y	3.9E4 y	39 y	14.3 d	20.6 min
112	1.6E20 y	1.6E17 y	1.6E14 y	1.6E11 y	1.6E8 y	1.6E5 y	165 y
128	1.1E25 y	1.1E22 y	1.1E19 y	1.1E16 y	1.1E13 y	1.1E10 y	1.1E7 y

To give you an idea of the computer power available nowadays: The network *distributed.net* broke a 64-bit key in 1757 days (searching 83% of the key space) with a rate of 10^{11} keys per second. More than 70,000 computers took part in this challenge. A similar project, aimed at breaking a 72-bit key, is still running with a current rate of ~$5 \cdot 10^{11}$ keys per second. Provided that the rate remains constant over the entire running time, it would still take over 200 years to search the entire key space. So, given an attacker who has access to a computer power comparable to that of *distributed.net* with a rate of 10^{12} keys per second, a 72-bit key may be considered secure. To put it another way, a 72-bit key is sufficient for protection against attackers who lack large financial resources. Now imagine an attacker who can afford a search rate which is *one million times* higher than that of *distributed.net*, resulting in approximately 10^{18} key trials per second. A 72-bit key is not sufficient here, so we should resort to a key of at least 90 bits in size. If the attacker is able to afford even more money and thus more computer power, a key size of 112 bits and more should be strongly preferred.

Of course, these incredibly big numbers for key spaces and their complexity are pretty impressive—but don't be misled by the conception that a sufficiently long, randomly chosen key automatically provides 100% security! Only megalomaniac fools try to break keys that are far beyond the scope of computer power. Clever attackers know that there are easier, faster and more effective methods to get hold of a password. Think of spyware, keyloggers, computer viruses, security flaws in operating systems, spies, double agents, truth drugs, torture... only to name a few malicious non-academic methods to find a key (you may call them "real-life brute force attacks").

Which security measures should I take when generating strong passwords?

First, you should make sure that your system is not infected with malicious software (like spyware, etc.). Before generating a "really strong" password, make sure that the random pool is "full of entropy", i.e., the "total entropy bits" counter has a value of at least 256 bits. When you copy passwords to the clipboard, clear the clipboard if you don't need them anymore (click on the *Clear Clipboard* button or select the corresponding menu item in the system tray menu). To clear the password box in the main window, just click on *Generate*, so that the contents of the box will be overwritten. To clear the password list, just close the window. PWGen automatically overwrites the text in this list when the window is closed or when a new password list is generated. Don't save sensitive passwords *as plaintext* anywhere on your hard disk. Instead, it is recommended to encrypt them (use the text encryption utility for this purpose, see: *Tools*) and save them as ciphertext.

Last but not least, I recommend you to regularly wipe the free space on your hard disk, since it may contain remains of sensitive data which were swapped out by Windows some time ago. A first-rate tool for accomplishing this task is *Eraser*.

Is it possible to memorize those random passwords?

Yes, it is, although it may appear rather impossible on first sight. Maybe pass*phrases* composed of random words are easier to memorize for you than random characters. Try to memorize the passphrase by "visualizing" the words or by making up a funny "story" where these words play a role. Random characters can be memorized by really learning them by heart; if necessary, write them down somewhere for a *short time*, and as soon as you have succeeded in truly memorizing them, destroy the material *irreversibly*. You could also try to make up a story, the words of which begin with a character of the password. You see, lots of possibilities. So with some effort, you will eventually succeed in memorizing a strong password generated by PWGen.

In most cases, it's sufficient to memorize one or two really secure passwords (that is, having a security of at least 72 or better 90 bits); the others can be easily stored in a password safe (see: *Can I use PWGen as a password safe?*).

What about pronounceable passwords?

As of version 2.3.0, PWGen allows generating phonetic (pronounceable) passwords based on language-specific trigram frequencies; this feature is accessible by entering `<phonetic>` into the *Character set* field (see: *Include Characters*). However, note that the trigram-based password generation is restricted to languages with Latin-based alphabets (letters a–z)! Alternatively, you may generate artificial, pronounceable "words" by applying the *Format Password* option. Passwords generated by the rule `%4[%3[%c%v%] %]`, for

example, will look like this: `sapifu kixezu cehiwu lukuro`. You can play around with the placeholders for consonants and vocals to create passwords that suit your needs.

Can I use PWGen as a password safe?

Yes, this is possible via the text encryption function of PWGen (see: *Tools*). You can use a simple text editor (like *Notepad++*, but Windows' rudimentary *Notepad* does it, too) to manage your "password safe". Whenever you need a password, for example, for an Internet service, use PWGen to create a random password of any length and store it together with the user name (or the URL as an alternative) in the safe. To "close" the password safe, copy the whole text block to the clipboard, encrypt it with a secure master password and replace the plaintext of your password safe with the ciphertext in the clipboard. This may not be as comfortable as using a real password safe (like *KeePass* or *PasswordSafe*), but it works and is secure (as long as your master password is secure, of course).

Alternatively—or in addition to a "classical" password safe—, you may want to have a look at the *MP Password Generator* in PWGen, which encapsulates a "password hasher" functionality: The user provides a secret master password and a unique parameter string (such as the name of the website for which the password is to be created), and the program generates a (practically) unique password from this data. Thus, an encrypted database is not required anymore; instead, each password can be reproduced by entering the user's master password and the specific parameter.

Which kinds of word lists does PWGen accept?

In principle, PWGen accepts all (Unicode or ANSI) text files containing some kind of "words". These words must be separated by a line break, a space, or a tabulator. The upper limit for the word length is 30 characters (you may decrease this length down to 1, see: *Advanced Password Options*). Word lists must contain at least 2 different words. Note that PWGen does not accept duplicate words; the procedure that checks for duplicates is case-sensitive, i.e., it does *not* ignore upper-case and lower-case letters! However, you can force PWGen to convert all letters to lower-case via *Advanced Password Options*. PWGen will not add more than 1,048,576 words—upon reaching this limit, the program will stop the progress. If your word list contains more words, you can try to select shorter words by decreasing the maximum word length.

As explained before, the program accepts text files consisting of words. Here is an example which uses the words in *license.txt* to create the following passphrase:

    original, (and number. (whether judgment

Note that PWGen does not remove punctuation marks or other non-letter symbols.

How to interpret the information about the random pool?

Whenever you generate "indeterministic" events by pressing keys, clicking with your mouse or by moving your mouse within PWGen's windows, the application collects "entropy" from messages sent by Windows, as well as from a high-performance counter (RDTSC processor instruction, if available; alternatively, the program calls functions provided by the Windows API; see: *High-Resolution Timer*). Additionally, the program derives entropy from several system-specific parameters, which is done in regular intervals in the background.

The actual "random pool" has a size of 32 bytes and can thus provide a maximum Shannon entropy of 256 bits. As explained in the *Step-by-Step Tutorial* above, the pool is completely filled with entropy if the progress bar is full—but the entropy counter can exceed this limit anyway. Note that this counter only serves informational purposes—it's absolutely impossible for the random pool to yield more than 256 bits of Shannon entropy! The counter just informs you about the total amount of entropy bits added to the pool, which might be of your interest if you consider PWGen's entropy estimations as too optimistic.

If you generate passwords, PWGen uses the pool contents as key for the AES (*Advanced Encryption Standard*) cipher which is then used for generating random numbers (see: *Random Pool* for more technical details). This means that a certain amount of entropy is "consumed", so to speak, from the random pool. As a consequence, the entropy counters will be decreased by the entropy content of the generated password(s). So generating five 48-bit passwords will consume 240 bits of entropy from the pool. (Of course the counters cannot fall below 0.)

Mind you: this loss of entropy only applies to the case that you actually *use* the generated password(s). If you discard a password and generate a new one without refilling the random pool, the entropy of the discarded password has *not* been lost. However, PWGen presumes that you use every password you generate. So once you have filled the random pool with entropy, you may effectively ignore the loss of entropy when generating passwords, as long as you don't use them. This can be explained as follows: Whenever you generate passwords or, more generally, random data using a fixed 256-bit state of the random pool, the entropy of the generated random sequence *can never exceed 256 bits*. If the apparent (!) entropy of the sequence exceeds 256 bits, it would be "easier" (in purely theoretical terms) for an attacker to find the 256-bit state of the random pool than to find the actual random sequence originating from this very 256-bit state. To put it another way, each 256-bit data block within the sequence has an *independent entropy of 256 bits* when treated separately from the rest, but adding more data from the same sequence to a 256-bit block doesn't increase its security. This does *not* mean that only the first 256 bits of random output are secure and the further output is deterministic/insecure and thus useless. It simply means that you cannot increase the entropy of the random output beyond 256 bits. *Theoretically*, an attacker could predict the output following the first 256 bits, *provided that he is able to break the 256-bit state of the random pool*.

However, as emphasized multiple times within this manual, breaking 256-bit keys is practically unfeasible by any standards (well, neglecting the exploitation of time travels, black holes, parallel universes, etc. here...).

Note: PWGen uses additional time stamps, independent of the 256-bit state of the random pool, when generating random numbers, which can increase the entropy up to 384 (256+128) bits. Nevertheless, since such security-relevant estimations should be made in a rather conservative way in the realm of cryptography, the internal limit is set to a fixed value of 256 bits.

Technical Details

Random Pool

Whenever you interact with your computer—for example by pressing a key, by clicking with your mouse or by moving your mouse—you create an event that is, unlike most internal computer events, to a certain degree indeterministic or unpredictable. The term "indeterministic" means here that these events can afford a certain amount of randomness—and let it be just a few bits. (Due to their random character, indeterministic events are sometimes—rather loosely—referred to as "*entropy*"; this is not to be confused with the entropy term in thermodynamics!) By calling a high-performance timer (like the RDTSC processor instruction available on all modern CPUs, see: *High-Resolution Timer*) when the user generates such an event we can efficiently make use of its random character. Moreover, the operating system Windows provides additional (and again partly indeterministic) information such as the type of Windows control where the message was sent to, the cursor position and the (low precision) date and time when the message was sent.

We can gather all this entropy from user-generated events in a so-called *random pool*. At this point it is essential to note that this data does not have to be purely random—it is sufficient that it contains some bits of uncertainty (the more, the better, of course). The "trick" is now to apply a *cryptographic hash function* to this entropy in order to "distil" randomness from this data. This operation yields data that looks truly random, although it cannot contain more information than the original partly-random data. If we now collect entropy amounts large enough to provide (in total) sufficient uncertainty, we can generate sequences that do not only look random, but in effect *are* random. For example, to generate a highly secure 128-bit password, we fill the random pool with enough user inputs (i.e., entropy) first—by entering some text on the keyboard, or by moving the mouse, etc. Then we use this data to distil the required 128 bits of truly random data from the pool and convert it to a secure password. This is the basic concept of random number generation in PWGen. The "real implementation" is a bit different.

In mathematical terms, the process looks as follows: HMAC-SHA-256(K, M) is used as secure hash function. The HMAC algorithm uses an additional key K to compute the 256-bit message-digest of a message M. (Note that in this case, the "key" does not have a special purpose here; we simply use the HMAC algorithm instead of the plain hash function because the former is thought to have better "randomness extraction" properties compared to the latter.) To distil randomness and to generate a new pool, the old pool contents and the entropy data are processed in the following way:

$$(1) \qquad P \leftarrow HMAC(P, S \;||\; T),$$

where P is the random pool (32 bytes), *HMAC* is the HMAC hash function, S is the entropy data of any length, and T is a time stamp from a high-resolution timer. "||" denotes simple concatenation of the sequences. PWGen uses a memory block of a defined length to

buffer subsequent entropy inputs; when this buffer is full, it will be hashed according to eqn. (1) and cleared (overwritten with random data) afterwards.

To generate cryptographically secure random numbers, PWGen uses the AES (*Advanced Encryption Standard*) encryption algorithm in counter (CTR) mode. AES is set up with a 256-bit key and operates on 128-bit blocks. In order to derive a key K for AES, the pool is first updated (eqn. (1)), then the pool is hashed:

$$(2) \quad K \leftarrow HMAC(T, P).$$

This procedure ensures that P cannot be derived from K, and vice versa. Random numbers are then generated by encrypting a 128-bit block counter C (3a), which is incremented by 1 afterwards (3b):

$$(3a) \quad R \leftarrow E_K(C),$$

$$(3b) \quad C \leftarrow C + 1.$$

C is initially derived by encrypting two 64-bit time stamps. R is the next 128-bit random output, and E_K is the encryption function with key K. As the attacker neither knows K nor C, he cannot predict the next number R, even if he knows the entire previous sequence. Furthermore, K and C are changed regularly to ensure that, even in case of a *state compromise* (leakage of sensitive data in K and C to the attacker), the attacker cannot reconstruct those random blocks generated before the key change ("forward secrecy"). Using time stamps when updating the pool and when generating secret counter values C adds to the security of the construction, since they provide a certain degree of unpredictability.

All sensitive data (random pool, entropy buffer, AES key, counter, …) are held in RAM to prevent it from being swapped out to the hard disk. The pool is made indistinguishable from random by clearing all buffers with random data after use. This will make it difficult to locate the pool in RAM by means of any easily identifiable sequences.

For performance reasons, PWGen delays the recognition of mouse movements. For every mouse input, PWGen counts a maximum of 10 bits of entropy (max. 8 bits for the timer plus 2 bits for cursor coordinates); for every keystroke, PWGen counts a maximum of 9 bits (1 bit for the key code). In addition to these user inputs, PWGen collects entropy from various system parameters every 15 seconds, which yields 24 bits of entropy in PWGen's estimations. Note that these estimations are rather conservative—the "real" quality of these entropy sources is likely to be higher.

In order to preserve the entropy collected during runtime, PWGen writes a random seed file (*randseed.dat*) in the application folder. This 32-byte file is read on start up, its contents are fed into the random pool, and the seed file is immediately overwritten with new data afterwards. PWGen updates this file in regular intervals while running as well as on exit. If you don't want PWGen to write this file, you can use the */readonly* command line switch (see: *Command Line Options*).

Text Encryption

The text encryption can be divided into five steps: UTF-16 to UTF-8 Unicode conversion, Text compression, HMAC generation, encryption and base64 conversion.

1) The UTF-16-encoded text as received from the Windows API is converted to **UTF-8 encoding**.

2) The text is **compressed** using the LZO1X-1 compression algorithm.

3) The **256-bit HMAC** of the compressed text is generated.

4) The compressed text is **encrypted** using AES with a 256-bit key.

5) The encrypted text (ciphertext) is converted into "readable" characters (**base64 character set**).

Let's start with an overview of the encryption procedure: To derive the 256-bit key from the password provided by the user, a version of the key derivation function *PBKDF2* is used, which applies the hash function HMAC-SHA-256 to the user password along with a randomly chosen 128-bit initialization vector (IV; also called "salt" in this context). This process is repeated many times (8192 iterations) in order to make password cracking much more difficult.

The text which is to be encrypted consists of a header structure (3-byte "magic" identification string, version number and length of the uncompressed text) and the compressed text (variable length). This assembly is encrypted in CBC (*Cipher Block Chaining*) mode in order to hide patterns in the plaintext:

$$(1a) \quad C_i \leftarrow E_K(P_i \oplus C_{i-1}).$$

C_i describes the ith block of ciphertext (i.e., the encrypted text) and P_i the ith block of plaintext (i.e., the unencrypted text). C_0 is the IV which is also used in key derivation. Again, E_K is the encryption function with K as key. The symbol "$\oplus$" denotes an "exclusive-or" bit operation. In decryption mode, eqn. (1a) can be reversed in a simple way:

$$(1b) \quad P_i \leftarrow D_K(C_{i-1}) \oplus C_{i-1},$$

where D_K is the decryption function.

Before encrypting the plaintext, a so-called HMAC (*keyed-Hash Message Authentication Code*) of the text is calculated. In general terms, the HMAC can be considered as a message digest of the ciphertext with K as parameter. It serves to check the decryption process in two ways: 1) *Is the key correct?*, and 2) *Has the ciphertext been altered in any way (data integrity)?* Note that PWGen cannot distinguish between these questions, i.e., it can only check if the HMAC is correct or not, but it cannot tell you on the basis of the HMAC if the decryption failed because of a wrong key or because of a corrupt cipher-

text. As the attacker doesn't know the key, he cannot compute valid HMACs himself; in particular, he cannot modify the ciphertext and then make any valid modifications to the HMAC due to the strong non-linearity of the HMAC algorithm. Any modifications of the ciphertext will be recognized during the decryption process. PWGen uses HMAC-SHA-256 to generate a 256-bit HMAC of the entire plaintext block. This 32-byte sequence is appended to the ciphertext after encryption.

In the last step, the binary encrypted data generated in the previous three steps are converted to the base64 format (a character set consisting of 64 "readable" characters: mixed-case letters, numbers, and the characters +, / and possibly =) in order to make it "readable" by text editors, e-mail programs, etc. In the encoding scheme used by PWGen, lines are wrapped after 76 characters to allow for a smooth display of the text in editors. (Some editors, such as Windows' *Notepad*, may crash [or take a *very long* time] when the user attempts to paste very long texts consisting of only one single line. Therefore, it seems to be prudent to create multi-line texts with shorter but more numerous lines.)

The decryption procedure is essentially the reverse of the encryption procedure. There are several "checkpoints" in this procedure, which serve to verify the validity of the key ("*Did the user enter the correct password?*") as well as data integrity ("*Is the ciphertext still in its original state or has it been modified in any way?*").

1. The ciphertext is converted from base64 (6-bit) characters to 8-bit bytes. **Check #1:** If there are illegal characters in the text, or if the resulting text is shorter than 64 bytes, or if the text length is not a multiple of 16, the ciphertext structure is invalid, which means that the text is either corrupted or not encrypted by PWGen.

2. The first 16 bytes of the ciphertext are decrypted and the "magic string" in the header structure is checked. **Check #2:** If the decrypted identification string and the default string do not match, then either the key is wrong, or the beginning of the ciphertext (first 32 bytes) has been modified.

3. The remaining ciphertext is decrypted to give the compressed plaintext.

4. The HMAC is generated and checked. **Check #3:** If this verification fails, either the key is wrong, or there were modifications *somewhere* within the entire ciphertext. However, as the verification of the identification string in the header must have succeeded in step (2), it is actually more likely that the ciphertext has been modified.

5. Finally, the plaintext is decompressed and converted from UTF-8 to UTF-16 to give the original plaintext. [**Check #4:**] Note that the decompression and the Unicode conversion can *theoretically* fail, although this should *never* happen under "normal" circumstances. If this error really occurs, the most probable cause is a heavily manipulated ciphertext, i.e., a text that has *not* been generated by PWGen.

CAUTION: *Always check the correct decryption of the ciphertext before discarding the plaintext!*

High-Resolution Timer

If possible, PWGen uses a high-resolution timer (HRT) the value of which is added to the pool when you press a key, move your mouse, or click with your mouse. It is also used when generating cryptographically strong random numbers to increase the security of the generator. Due to the high resolution of this timer every value possesses a certain degree of unpredictability, particularly when measuring the timer intervals between user-generated events such as keystrokes and mouse clicks. Therefore, it can provide a relatively high amount of entropy. Moreover, consider the following worst-case scenario that an attacker got hold of the entire state of the random pool ("*state compromise*") and thus could theoretically predict the entire sequence of random numbers that will be generated. Adding the HRT value before generating the next random sequence offers a certain protection against this scenario: The attacker can read the timer value himself, of course, but it is actually very unlikely that he reads the same value due to the high time resolution of the timer (typically nanoseconds on modern CPUs). Since the HRT is available very quickly, it is used in every critical situation of the random pool, i.e., when updating the pool and when generating random numbers.

PWGen can use the following timers (listed with descending priority):

1. **Time stamp counter (RDTSC instruction):** The RDTSC processor instruction returns the number (64-bit) of cycles since reset. The time resolution therefore depends on the clock rate of the processor: On a 3 GHz processor, for example, each cycle takes ~0.3 nanoseconds. The RDTSC instruction is present on all x86 processors since the Pentium and provides an excellent high-resolution, low-overhead way of getting CPU timing information.

2. **QueryPerformanceCounter (Windows API):** According to the *Windows SDK*, this function returns the current value (64-bit) of the high-performance counter. This is probably just a wrapper for the time stamp counter on most systems, but the return value may be different on multicore computers. Calling this function is slower than executing RDTSC.

3. **Low resolution timer (Windows API):** Alternatively, if neither RDTSC (1) nor QueryPerformanceCounter (2) are available, PWGen retrieves the current date and time of the system. The resolution of this timer is only in the millisecond range and thus *much* lower (by several orders of magnitude) than that of the high-resolution timers (1) and (2), but may still be sufficient for measuring keyboard and mouse events.

On start up, PWGen checks if the RDTSC instruction (1) is available, which should be the case on all modern computers (Pentium processor onwards). If this check fails, PWGen

tests QueryPerformanceCounter (2). If both tests fail, PWGen uses the current system date and time (3). However, if (1) or (2) are allegedly available, PWGen briefly checks if they can really hold the high resolution; this test might fail when PWGen runs on emulated systems, for example, and if it fails, PWGen uses (3) instead.

Note that PWGen constantly checks the resolution of the timer values and decreases the entropy rating if the resolution is low (i.e., in the millisecond region). Although the security of the random generator does *not* entirely depend on the entropy of the timer values because a lot of other parameters are additionally incorporated into the pool, it is strongly recommended to run PWGen on systems where a high-resolution timer (1 or 2) is present due to the higher entropy and due to the increased security in critical situations of the random generator. You can check which timer is used by selecting the item *Timer Info* from the *Help* menu.

Contact & Further Information

Contact

If you have further questions, critical remarks, suggestions for the future development, bug reports or something else to tell—comments are always welcome!—, feel free to contact me via e-mail:

```
c.thoeing@web.de
```

Translations

Translations of PWGen into languages other than English and German are always very welcome. For a list of translations that are available so far, see here. If you want to update an existing translation file or create a new one, I recommend using the *PWGen Translation Utility* that I have created especially for this purpose. This program uses the file *German.lng* (which accompanies every release of PWGen) as a template for your own translation. Please always use the latest version of the Translation Utility.

Word Lists and Trigram Files

Have a look at the PWGen project page to access additional word lists and trigram files suitable for generating passphrases and phonetic passwords, respectively. If you have created a word list or trigram file and want to share it with other users, please send me the file, so that I can publish it on the web. Alternatively, you can find special word lists in several languages on the "Diceware" homepage; however, you may want to extract the actual words (without the numbers/indices before the words) first before using the "Diceware" lists in PWGen.

Please Donate!

As you can certainly imagine, developing and maintaining a software project like PWGen requires a lot of effort and commitment, especially if you have other responsibilities in your regular job as well as at home. If you like PWGen and use it frequently, please donate to the project. Your donations will encourage me to further develop the application and keep it bug-free. *Thanks!*

Acknowledgment

Thanks to Brainspark B.V., M.F. Oberhumer, A.G. Reinhold, Aha-soft, D. Reichl, S. Tolvanen, M. Hahn, T. Ts'o, R.J. Jenkins, J. Brown, and W. Dai for code & inspiration! Thanks to all the users for their donations, helpful comments and suggestions!