

PWGen for Windows



Generator of cryptographically strong passwords

USER MANUAL

Version 2.08

Licensing Information

By using, copying, distributing or modifying PWGen 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 © 2002-2012 by Christian Thöing <c.thoeing@web.de>.

Portions of this software:

Copyright © 2006-2010 by Brainspark B.V.

(implementations of AES and SHA-256 algorithms, base64 encoding; part of the PolarSSL library)

Copyright © 1996-2008 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

PWGen is hosted at sourceforge.net.

Homepage: <http://pwgen-win.sourceforge.net/>

Project page: <http://sourceforge.net/projects/pwgen-win/>

Contents

Introduction.....	3
Program Features.....	5
Why should I prefer PWGen to Password Generator XY?.....	5
Step-by-Step Tutorial.....	5
Include Characters.....	6
Include Words.....	7
Advanced Password Options.....	8
Generate Multiple Passwords.....	11
Generate Single Passwords.....	11
Random Pool.....	12
Main Menu.....	13
File.....	13
Tools.....	13
Options.....	15
Help.....	16
Additional Menus.....	16
System Tray Menu.....	16
Password List Menu.....	17
Questions & Answers.....	17
Which security level is appropriate for my password?.....	17
Which security measures should I take when generating a strong password?.....	20
Is it possible to memorize those random passwords?.....	20
What about pronounceable passwords?.....	21
Can I use PWGen as a password safe?.....	21
Which kinds of word lists does PWGen accept?.....	22
How shall I interpret the information about the random pool?.....	22
Technical Details.....	24
Random Pool.....	24
Text Encryption.....	26
Contact.....	28
Translations.....	28
Please donate!.....	28
Acknowledgements.....	28

Introduction

The usage of a *password* is still the simplest way to control the access to a specific resource. 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 called “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 favourite 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 (that is, *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 attacks. 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 *passphrases* 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; the security can easily be increased by adding some random characters.

[Here is an interesting article](#) about problems regarding passwords chosen by humans.

The need for secure passwords has grown since the advent of the Internet and its many websites where the access to a certain resource (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 any more—you just store them in the password safe which is protected by a “master password” (that must be remembered, 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. (You will find more information about this topic later in the manual.)

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 opportunity to create many passwords at once. Just give it a try!

Program Features

- Password generation based on a cryptographically secure pseudo-random number generator (AES encryption algorithm)
- Entropy gathering by measuring time intervals between keystrokes, mouse movements and mouse clicks; entropy from system-specific parameters is also collected in regular time intervals
- Secure memory management: memory allocated by the program is filled with binary zeros before deallocation
- Generation of *passphrases* composed of words from a word list
- Generation of large amounts of passwords
- Secure text encryption (AES with 256-bit key)
- Multilingual support
- Runs on all 32-bit & 64-bit Windows versions

Why should I prefer PWGen to Password Generator XY?

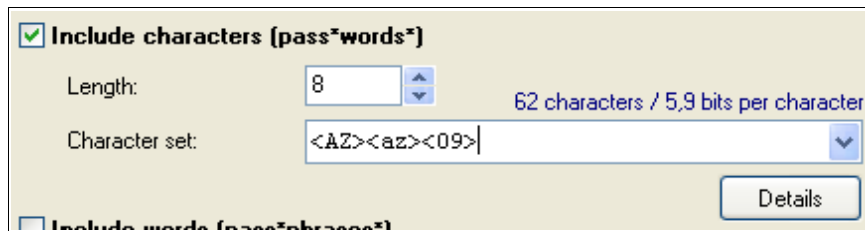
Because PWGen is more intuitive, more powerful, more user-friendly; because it has a better random pool design, a better entropy gathering function and is capable of generating passphrases containing real words. Moreover, it uses strong encryption, it's completely free and Open Source software. And, most important, PWGen just looks more beautiful than all the others! (OK, joke.)

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



Include characters (pass*words*)

Length: 62 characters / 5,9 bits per character

Character set:

Include words (pass*words*)

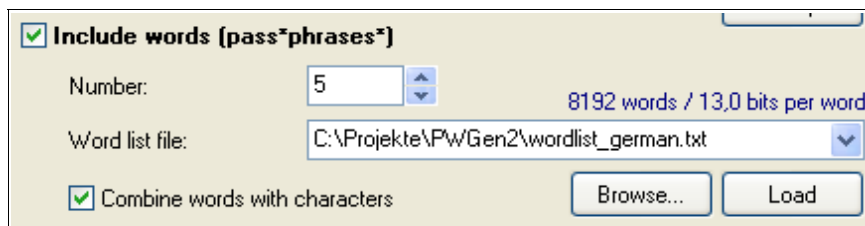
- Check “*Include characters*” if you want to include characters in your password. Enter the desired length of the character string in the “*Length*” field (maximum 20,000).
- Enter the desired character set (that is, all the characters that may occur in your password) in the field below or choose one from the drop-down list. Note that each character may only occur once—otherwise, it will not be accepted. PWGen only accepts 255 ASCII characters (symbols #1 to #255, excluding #0), for it does not support Unicode. You can use the following codes to abbreviate the sequence:
 - `<AZ>` or `<az>`: Upper-case/lower-case letters from A to Z
 - `<09>`: Numbers from 0 to 9
 - `<Hex>` or `<hex>`: Hexadecimal symbols (0123456789ABCDEF), where the letters may be upper-case (`<Hex>`) or lower-case (`<hex>`), respectively
 - `<b64>`: Base64 symbols; equal to: `<AZ><az><09>+/-`
 - `<symbols>`: Additional symbols accessible via the keyboard: `!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~`
 - `<easytoread>`: Like `<AZ><az><09>`, but without similar-looking (“ambiguous”) characters: `B8G6I1l0OQDS5Z2`. If you want to generally exclude ambiguous characters from character sets, you

can activate the corresponding option in the “*Advanced Password Options*” dialog (see below).

You may provide a *comment* included in square brackets “[]” at the beginning of the entry. All characters of 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!

- The character set will be updated as soon as you leave the input field, e.g. by clicking on another interaction element. The number of bits per character is displayed on top of the “*Character set*” field; it is calculated as $\log_2 N$, where N is the number of characters and \log_2 is the logarithm base 2. Due to the limitation of 255 characters, the number of bits per character is limited to 7.9 (note that PWGen rounds down the first number after the decimal point). If you click on “*Details*”, the program will show you the actual characters that have been accepted.

Include Words

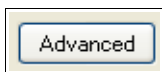


- Check “*Include words*” if you want to include words from a word list in your password. Enter the desired number of words into the “*Number*” field (maximum 200).
- By default, PWGen uses an internal English word list containing 8192 (2^{13}) words. If you enter “<default>” into the “*Word list file*” 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 “*Browse...*” to browse through your folders and select a file.

- Then click on “*Load*” to load the contents of this file into memory. The program will show you the number of words that were accepted (maximum 65,536 [2^{16}] words). By default, PWGen accepts words that are not longer than 30 characters; however, you can change this number in the *Options* menu (see below). 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.
- Check “*Combine words with characters*” to generate passwords as a combination of words with characters, that is, every word will be combined with one or more characters, depending on the number of words and the number of characters, respectively. Example (3 words combined with 8 characters by a “-” symbol): putty-fuV umbra-Sxq faint-fT.

Advanced Password Options



- 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, B8G6I1l|00QDS5Z2). For this option to take effect, you have to reload the current character set. As this option reduces the size of the character set, the password security will be reduced accordingly.
 - *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's 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 random 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 a passphrase. Select this option to deactivate this behaviour.
- *Don't separate words and characters by a '-' sign*: 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 behaviour.
- *Include at least ...*: Activate these options in order to force the program 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 [!"#\$%&'()*+,-./:;<=>?@[\]^_`{|}~]). If the user-specified character set does not contain any characters from a pre-defined set, the resulting passwords will contain exactly one character from this pre-defined 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 < 4, not all characters can be included, of course.) Depending on


the user-specified character set, selecting these options can—rather slightly—reduce the password security. Note that these options do not affect the generation of *passphrases*.

- *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, every character in the sequence will be different from the previous one.
- *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 considerably longer than usual because PWGen has to search through the entire list before adding a new password. Note that, in certain cases, it might not be possible to generate a complete password list 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's not possible to generate a list of more than 1,000 (= 10^3) unique entries, since the total number of possible password permutations is limited to this exact number. Note that you can cancel the generation process any time. Moreover, PWGen will stop automatically after ~ 2 billion (2^{31}) futile searches.
- There are two additional fields for redefining special character sets:
 - *Redefine ambiguous characters*: The set of ambiguous characters is pre-defined as `B8G6I1I0OQDS5Z2` by default. Redefining this set also affects the other pre-defined character sets (i. e., `<AZ>`, `<az>`, `<09>`, etc.).
 - *Redefine special symbols*: The set of special symbols is pre-defined as `!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~` by default. Re-

defining this character set affects the *<symbols>* character set, as well as the option *Include at least one special symbol* (see above).

- The field *Maximum length of words in word lists* should be self-explanatory. When you load 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. It may also be helpful in reducing the word list size in case of lists longer than 65,536 words. Note that this option does not apply to the default word list but only to lists from an external source. You may have to reload the word list to make this option effective.

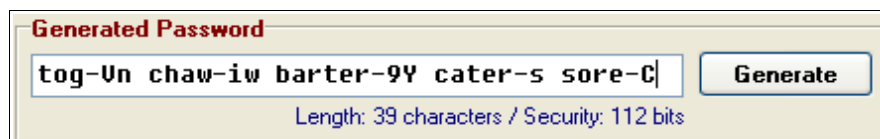
Generate Multiple Passwords



Number of passwords:

- If you want to create more than one password, enter the exact number into this field (maximum 1,000,000) and click on “*Generate*”. The program will open a window containing the password list.
- If you generate a larger amount of passwords (more than, say, 1,000 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 were generated so far.

Generate Single Passwords



Generated Password

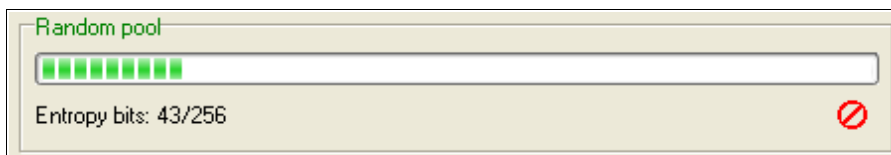
Length: 39 characters / Security: 112 bits

- If you want to create one single password, click on “*Generate*”. PWGen will display the password in the box.
- The length of the password and its “security”—that is, the estimated

amount of entropy provided by the password—will be shown in the bottom line. Note that the security of a single password is limited to 256 bits, since the “random pool” which is used to generate passwords has an effective size of 256 bits (which corresponds to the message-digest length of SHA-256). 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 about how to interpret these values, see *Questions & Answers*.

- 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.
- 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, that is, the amount of entropy bits the pool can currently provide. Whenever you generate one or more passwords, $N \cdot p$ bits will be “consumed” from the pool (N =number of passwords, p =amount 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 *total* entropy amount (possibly *exceeding* 256 bits), which might be of your interest if you don't trust PWGen's estimations regarding the entropy content of the different sources. So "*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 "total entropy" counter is increased (up to 65,536 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.
- You can reset the counters by right-clicking on the progress bar and selecting "*Reset Counters*" (this won't affect the actual contents of the random pool in any way). If the blinking of the progress bar irritates you, click on the symbol on the right below the bar in order to hide the entropy progress.

Main Menu

File Tools Options Help

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

File

- **Exit:** Exits the program.

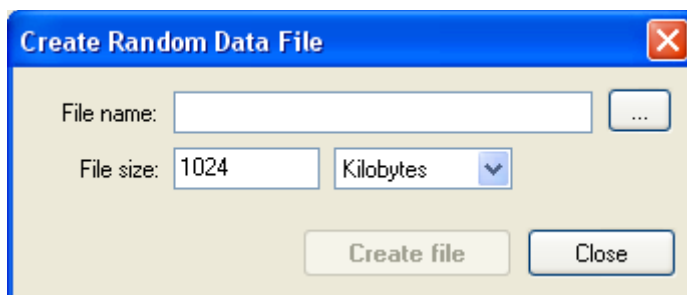
Tools

- **Clear Clipboard (F2):** Securely clears the text contents of the clipboard. The text buffer is overwritten with binary zeros before deal-

location.

- **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 compressed before encryption. (For more information about the exact encryption procedure, see *Technical Details* below.) The ciphertext will be converted to the base64 format consisting of letters (A-Z, a-z), numbers (0-9) and the symbols "+" and "/". Non-base64 symbols present in the ciphertext will be ignored during the decryption process, so you can insert line breaks etc. But beware not to change the actual base64 symbols that make up the ciphertext! 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, that is, 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.

- **Create Random Data File (F5):**



Creates a file consisting of purely random (that is, *cryptographically random*) data. First, select an existing file (by clicking on the *Browse* "..." button) or enter a 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. In case of a file size of more than 1 MB, 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 GB. If you enter a desired file size of more than 2 GB, the resulting file will be definitely less than 2 GB in size. If there is a need to generate very large files, please contact me. I will then consider adding this feature in a future release.

Options

- **Change 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 startup and add the language as an item to this menu.
- **System Tray Icon:**
 - *Show 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 checked, 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 Constantly" is unchecked, the symbol will be removed when restoring the application.
- **Save Settings on Exit:** If checked, all program settings will be saved in an *.ini* file when the user exits the program.
- **Save Settings Now:** Saves the settings instantaneously.

Help

- **Open Manual (F1)**: Opens this manual.
- **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 (X characters)**: This function quickly generates a password of *X* characters (with *X* depending on the settings in PWGen's main window) from the current character set and copies it to the clipboard. The inclusion of words is not possible. This quick generation may be useful when you need a password for a website or so. Although the passwords generated by this function are cryptographically secure, it is generally recommended to use the main window to create "strong" passwords.
- **Generate and Show Password**: Generates a password of *x* characters (see previous menu item) and displays it in a message box. Click on "Yes" to copy the password to the clipboard, click on "No" to generate and display a new password, or click on "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.

Password List Menu

In the *password list* window, you can access some useful functions by right-clicking with your mouse:

- **Copy (Ctrl+C)**: Copies the current text selection to the clipboard.
- **Select All (Ctrl+A)**: Selects the entire text.
- **Save As File (Ctrl+S)**: Writes the text to a file of your choice.
- **Change Font**: Changes the text font.

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 (that is, 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, we're all frequently short of money, this is a commonplace **sigh**; 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 ...).

<i>N</i> (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. Over 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 $1.4 \cdot 10^{11}$ keys per second. Provided that the rate remains constant over the entire running time, it still takes over 1000 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 a strong password?

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", that is, 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 above) and save them as ciphertext.

Last but not least, I recommend you to regularly wipe the free space on your hard disk, because it could contain remains of sensitive data which were swapped out by Windows some time. 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 *passphrases* 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 below).

What about pronounceable passwords?

Yes, pronounceable passwords are certainly a nice feature, but their quality in terms of security is difficult to assess, because their generation has to obey very specific phonemic rules. Thus not all character patterns are equally probable, which results in an entropy reduction. While pronounceable passwords are certainly better than those based on personal data, I would prefer randomly chosen passwords or passphrases anyway. Pronounceable passwords are an interesting gimmick for the future development of PWGen, but chances are rather low that I'm up for implementing it...

Can I use PWGen as a password safe?

Yes, this is possible via the text encryption function of PWGen (see above). 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 [KeepPass](#) or [PasswordSafe](#)), but it works and is secure (as long as your master password is secure, of course).

Which kinds of word lists does PWGen accept?

In principle, PWGen accepts all 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 above). Word lists must contain at least 2 different words. Note that PWGen does not accept duplicate words; when checking for duplicates, PWGen ignores upper-case and lower-case letters! (Nevertheless, the words are stored in the original letter format.) PWGen will not add more than 65,536 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. However, non-text ASCII characters may be ignored when reading the file, because it is read in “text mode”.

How shall I 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). 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. This means that a certain amount of entropy is “consumed”, so to say, from the random pool. As a consequence, the entropy counters will be decreased by the entropy bits in 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 it (for example, by simply generating a new one), the entropy 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. What's the reason for this? Imagine you generate a 90-bit password using a full random pool of 256 bits. After this process where 90 bits are consumed from the pool, it cannot provide the original 256 bits any more, but only 166 bits, because otherwise it would be (in theory!) easier for an attacker to “guess” the 256 bits of the random pool than to guess $90+256=346$ bits! (Remark: This calculation is a bit simplified, because PWGen uses additional time stamps, independent of the random pool, when generating random numbers. This makes a brute-force attack much more difficult.)

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. 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) 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 if 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 (that is, 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 pass-

word. 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: SHA-256 is used as secure hash function. It accepts an input of any length and yields a *message digest* of 256 bits (32 bytes). To distil randomness and to generate a new pool, the old pool contents and the entropy data are concatenated and “hashed” afterwards:

$$(1) \quad P_{\text{new}} = H(P_{\text{old}}, B),$$

where P is the random pool (32 bytes), H is the hash function and B is the entropy data of any length. 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 binary zeros) afterwards.

To generate cryptographically secure random numbers, PWGen uses the AES (*Advanced Encryption Standard*) encryption algorithm instead of SHA-256. AES is set up with a 256-bit key and operates on 128-bit blocks. Here, the pool contents P functions as the 256-bit AES key:

$$(2) \quad R_i = E_P(R_{i-1} + T).$$

R_i is the next 128-bit random number, R_{i-1} is the previous random number, T is a time stamp from a high-resolution timer, and E_P is the encryption function. As the attacker doesn't know the key P , he cannot predict the next number R_i , even if he knows the entire last sequence R_1, \dots, R_{i-1} . R_0 is a secret timer-based *initialization vector* (IV). The time stamp T adds to the security of the construction, since its value is to a certain degree random. After one megabyte of output, the key will be changed; this is done by rehashing the pool (eqn. (1)) and using the new value as key.

All sensitive data (random pool, entropy buffer, random numbers) is held in RAM to prevent it from being swapped out to the hard disk. For per-

formance 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 20 seconds, which yields about 24 bits of entropy. Note that these estimations are rather conservative—the “real” quality of these entropy sources is likely to be higher.

Text Encryption

The text encryption can be divided into four steps: Text compression, encryption, HMAC generation and base64 conversion. First, the text is compressed using the [LZO1X-1](#) compression algorithm. Second, the compressed text is encrypted using AES with a 256-bit key. To derive the AES key from the password provided by the user, a randomly chosen 128-bit IV and the password are hashed together 8192 times using SHA-256. This procedure, which is sometimes referred to as “key crunching”, is performed to complicate dictionary attacks: the attacker has to compute this cycle for every chosen password, which slows down the attack dramatically. In mathematical terms:

$$(1) \quad S_i = H(S_{i-1}, X),$$

where X is the user password, H is the hash function, S_i is the i th message digest and S_0 is the random IV. So the AES key K is given by S_{8192} . The text which is to be encrypted consists of the length of the uncompressed text (4-byte value) and the compressed text (variable length). This assembly is encrypted in CBC (*Cipher Block Chaining*) mode in order to hide patterns in the plaintext:

$$(2a) \quad C_i = E_K(P_i \text{ xor } C_{i-1}).$$

C_i describes the i th block of ciphertext (that is, the encrypted text) and P_i the i th block of plaintext (that is, the unencrypted text). C_0 is the IV also

used in key crunching (S_0 in eqn. (1)). Again, E_K is the encryption function with K as key. "xor" denotes an "exclusive-or" bit operation. In decryption mode, eqn. (2a) can be reversed in a simple way:

$$(2b) \quad P_i = D_K(C_{i-1}) \text{ xor } C_{i-1},$$

where D_K is the decryption function.

The third step involves calculating a so-called HMAC (*keyed-Hash Message Authentication Code*). The purpose of this value is to check 1) if the key is correct, and 2) if the ciphertext has been altered in some way (data integrity). In general terms, the HMAC can be considered as a message digest of the ciphertext with K as parameter. As the attacker doesn't know the key, he cannot compute valid HMACs himself. Any modifications he makes will be recognized during the decryption process. PWGen uses SHA-256 to generate the HMAC of the entire [IV,ciphertext] block. The 256-bit output is truncated to the first 128 bits, which should be sufficient for this purpose.

In the last step, the binary data generated in the previous three steps are converted to the base64 format (a character set consisting of letters, numbers, "+" and "/") in order to make it "readable" by text editors, e-mail programs, etc.

The decryption procedure is essentially the reverse of the encryption procedure: The ciphertext is converted from base64 characters to 8-bit bytes, the HMAC is generated and verified (if the verification fails, the password is incorrect, or the ciphertext has been modified), the ciphertext is decrypted to give the compressed plaintext which is decompressed to give the original plaintext.

Contact

If you have further questions, critical remarks, suggestions for the future development, bug reports or something else to tell, 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 you to use the [PWGen Translation Utility](#) that I've created especially for this purpose. This program uses the file *German.Ing* (which accompanies all releases of PWGen) as a template for your own translations.

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!*

Acknowledgements

I would like to thank the following people:

- *Brainspark B.V.* for the implementation of AES, SHA-256 and base64 encoding; these source files are part of the [PolarSSL](#) package.
- *Markus F.X.J. Oberhumer* for his fast and easy-to-use [miniLZO](#) code.
- *Arnold G. Reinhold* for the [Diceware](#) 8k word list; this is the internal

word list that PWGen uses by default.

- *Sun Microsystems* for OpenOffice.org Writer; this document was created with this program.
- [Aha-Soft](#) for the nice (free) toolbar icons.

Thank *you* for using PWGen!