

The passphrase FAQ, version 1.04

HTML coded and last revised on 01-13-1997 by Randall T. Williams.

Contents

- 1.0 Introduction
 - 1.0.1 Where do I find the Passphrase FAQ?
 - 1.1 How things are used in this document
 - 1.2 How do I get random numbers?
 - 1.2.1 Psuedo-random number generators
 - 1.2.2 Hardware random number generators
 - 1.3 How hard is it to crack the IDEA key?
 - 1.4 How hard is it to crack RSA?
 - 1.5 What is MD5?
- 2.0 How long should the passphrase be?
 - 2.0.1 What if I use another language?
 - 2.1 What if I use common phrases or quotes?
 - 2.2 What happens if I combine phrases and nonsense phrases?
 - 2.3 Does odd spelling, punctuation and capitalization help?
 - 2.4 What if I use random words?
 - 2.4.1 Can I use a key generator dictionary?
 - 2.5 What if I use all random letters?
 - 2.6 What if I use all random characters?
 - 2.7 How long does it take to attack a passphrase?
 - 2.7.1 Something anyone can do
- 3.0 How do I make a strong passphrase?
- 4.0 How strong is my passphrase?
 - 4.1 Passphrase examples
- 5.0 Who might try to get my passphrase and how?
 - 5.1 Law enforcement and the government
 - 5.2 Multitasking systems
 - 5.3 Multi-user systems
 - 5.4 Tempest and electronic surveillance
- 6.0 How do I securely store my passphrase?
 - 6.1 Should I write my passphrase down?
- 7.0 How do I use the passphrase switch or variable?
 - 7.1 Setting the environment variable PGPPASS
 - 7.2 Using the -z switch
 - 7.3 Passphrases in batch files
- 8.0 Copyrights, numbers, glossary and other leftovers
 - 8.1 The really big numbers
 - 8.2 Glossary

1.0 Introduction

This is The Passphrase FAQ for PGP. I tried to include everything I've seen asked on alt.security.pgp along with some extras to cover other things like passwords and different key lengths. Most people who have had college algebra or higher should be able to follow the math. Check the glossary in section 8.2 to help with some of the terms and how they are used.

MD5 and IDEA are based on 128 bit blocks. It should be trivial to change to a 56 bit DES key or keys of other sizes. Passwords are different than passphrases due to length. The same ideas will work for analyzing your password or passphrase.

This FAQ is under construction (aren't they all?). Changes will be made as I learn about them, generate them or have time. Applied Cryptography [1] leads me to believe that there hasn't been much published research in this area. Password cracking is covered, but little is said about passphrases. Some of the math involved is not normally a part of what I do. I'm an electronics engineer by training. Cryptography is just a hobby. Forgive anything that may seem trivial that was overlooked. I created most of this from my head. Comments, corrections, additions, encouragement and positive criticism can be emailed to ac387@yfn.yosu.edu , but don't expect much of a reply. Flames should be sent to /dev/null.

1.0.1 Where do I find the Passphrase FAQ?

This is a catch 22. Since you are reading this, you found it. If you can't find it, this section probably won't help. So far there is only three official places you can find the Passphrase FAQ. Thanks go to Galactus, Don Henson and Patrick Finerty for the web sites. A PGP signed copy of the FAQ also gets posted to alt.security.pgp about the middle of every month (most of the time).

The web sites are:

- Galactus' page (Netherlands)
- Don Henson's page (USA Texas)
- Patrick Finerty's page (USA Utah)

1.1 How things are used in this document

The first thing is about the numbers used in this document. Most math here was performed on a Texas Instruments TI-60 pocket calculator and a couple programs on my PC for verification. The number of significant digits is limited to just a few places because a string of almost 40 digits isn't needed and they are really beyond the comprehension of many. Look at section 6.0 for the exact value of some of the big numbers used here. I use the notation where $3.4E38 = 3.4 * 10^{38} = 2^{128}$. This was easier to type and it saves a little space. Also note that $\log(x)$ (base 10) is being used here. Since we are just using the exponents, don't worry about using $\ln(x)$ (base 2.718 or e) if you don't have $\log(x)$.

Rounding where it effects security are rounded up to the next highest whole number to be safe. Other numbers are just rounded to the nearest decimal place. The text should be clear on this in most cases.

References are numbered and enclosed in []. Sub references have a lower case letter added. So with that out of the way, on to other things.

1.2 How do I get random numbers?

Random numbers are very hard to generate. Quite often non-random events effect the randomness of devices or circuits. A suggestion would be to make 1 to N markers and place them in a very good mixer. You might want to try coin flipping, but if you have a person involved, a coin flip can be biased enough to skew the results over the long term. A ball method like several lotteries use is a good random source, but don't use the numbers from the lottery. Lottery numbers are a little too obvious and it is easy to try them. A pool/billiards game uses a set of balls in a bottle that allows only one ball to be extracted at a time. This is a cheap and inexpensive source for random numbers. I leave it to the reader to figure out how to get the 16 balls translated to something useful for any particular application. Those who are familiar with Dungeons and Dragons and the other role playing games may already have a set of dice numbered in a variety of sizes. The one caution with dice is that adding dice (Eg. 2 six sided dice) will change the output to a median number (odds of 7 are 1 in 6) and the extremes (odds of 2 and 12 are 1 in 36) are less likely to occur, losing some randomness. Be sure that you have random dice. The quality is sometimes not very good and may cause non-random results. You might want to look at RFC1750 [6] for more information on generating random numbers for secure purposes.

1.2.1 Psuedo-random number generators

Using a PRNG is in most cases a bad way to generate random numbers. The problem with PRNGs is the numbers are generated by a function. This includes the BASIC RND() function, the C rand() function or any other language that has a random function. Programmers have used this simple and relatively fast method in programs and games for years. The reason for this is because of the way PRNGs work. A simple PRNG will use code something like $R = (A * R1 + B) \text{ mod}(C)$; $R1 = R$; $R = R / C$. Primes are usually used for constants A, B, and C. Most languages have provisions for placing a seed value in R1 before calling the PRNG but it isn't needed and some PRNGs may not bother with the additive constant B. What makes a PRNG easy to break is that many only use 16 bits to store the values. That means we can brute force a 16 bit PRNG key space in $65536 * N$ attempts where N is the number of psuedo-random elements used. Almost anyone can probably search a standard PRNG key space in a day. A worst case search will probably last less than a week even on the average home computer.

If you are lucky and have a good PRNG, then the search space may be 2^{32} which isn't a whole lot better. Note that 40 bit keys can be brute forced by an individual with access to enough computing power in about a week or less and places like the NSA don't mind 40 bit keys. Look at [1e] for more information and other references on random sequence generators.

1.2.2 Hardware random number generators

Hardware generators can be made using the noise from a variety of semiconductor PN junctions. A good example of this is simply amplified noise from a zener diode. Other noise sources are high value resistors and a number of commercial chips that use a variety techniques to make noise. A caution with hardware sources of random information is that they could be influenced by noise or other signals that are not random. Most places are saturated with 50 or 60 Hz noise from power, clock signals and other digital noise from computers, television and radio, and a variety of other types of electronic equipment. For safety, you may want to encrypt or hash the output of a hardware source. A good hash function or encryption will hide any undiscovered patterns. An inexpensive random bit source can be built for \$10.00 (U.S.) [7] and mass produced for under \$5.00 (U.S.).

1.3 How hard is it to crack the IDEA key?

PGP uses IDEA as the conventional cipher. The key for IDEA is 128 bits. We can calculate the brute force key space with $2^{128} = 3.4E38$. A special hardware based key cracker for IDEA that can try one billion ($1E9$) keys per second will take $1.08E22$ years to go through all possible keys. You can expect to get most keys in about half that time which will take $5.39E21$ years. It is estimated that the sun will go nova in $1E9$ years. Since the algorithm is secure, the cryptanalyst has to go after other things like RSA or your passphrase. It is currently beyond our technology to crack an IDEA key by brute force.

1.4 How hard is it to crack RSA?

Factoring is an easier problem than brute force search of the key space. The only current practical factoring methods for RSA size numbers are the Multiple Polynomial Quadratic Sieve (MPQS) and its variations, and the Number Field Sieve (NFS). Estimates for the MPQS run around $3.7E9$ years for a 200 digit/664 bit number [1d]. I should include that no one knows how long it will take to factor numbers larger than about 130 digits/429 bits except for some special cases. Some net references on numbers that have been factored are RSA129 and The 384 Bit Blacknet Key. You should note that it took a lot less time and computing power to factor a 116 digit/384 bit key than it took to factor a 129 digit/426 bit key. The NFS factored RSA130 a 130 digit/430 bit key even faster than RSA129 was factored. RSA is probably the weakest link in PGP, but currently no one knows a good way to factor numbers over 155 digits/512 bits without building special hardware.

1.5 What is MD5?

MD5 is what takes your passphrase and scrambles it into an IDEA key. In theory, MD5 should generate a different output for every possible bit combination as long as your key space is equal to or larger than 2^{128} . Proving that MD5 will generate all 2^{128} outputs from a given key space equal to 2^{128} is practically impossible. This would be about the same as a brute force search on the IDEA key. An interesting problem is that theoretically you can produce an equivalent passphrase by searching any given key space that is 2^{128} or larger.

In light of the attack on MD5, wait and watch. While a weakness has been found, the jury is still out on using unmodified MD5. A move to SHA or other hash function may be in the future for PGP.

2.0 How long should the passphrase be?

The rule of thumb is that you use one character per bit of key needed. You really get about 1.2 bits per English text character [1c] for key usage. Modifying the key size means $128 / 1.2 = 106.667$ or 107 letters of text are needed. This assumes normal English structure, only lower case letters and spaces for the passphrase and for the calculation purposes, all spaces are ignored in the passphrase [1a]. Few of us are willing to type out a line and a half of text every time we use PGP though. This is where security fails and we use weak passphrases.

2.0.1 What if I use another language?

Using your native language is probably an obvious choice. Throughout this FAQ, data and statistics apply to English text. Using another language or combining languages will change the numbers some. It will not make your passphrase harder to guess. Attacking a different language or even multiple languages is still the same. The search space is roughly the size of the language or grows by adding the size of the average size of the vocabulary of the added language. Dictionary attacks in another language would run in the same manner as a dictionary attack in English.

2.1 What if I use common phrases or quotes?

The short version on common phrases is don't use them ever. A book of quotes may contain 40,000 quotes [5]. You could probably set an old PC XT in a corner and have common phrases checked in a relatively short amount of time without any special hardware. Simple phrases will be the first ones checked. If you are a Star Trek fan, "Beam me up Scottie" is a bad phrase to use. If you can find the phrase in any published work then don't use it. A simple background search will reveal what kind of music, books, TV shows, movies, games, hobbies, and everything else you might use. All the common phrases will be tried on the first pass of a key search. You can try 40,000 quotes using unmodified PGP in about 2.4 days. See section 2.7.1

2.2 What happens if I combine phrases and nonsense phrases?

Combining phrases extends the phrase search some. Nonsense phrases will also slow down a brute force search [4]. A smart attack would take advantage of normal phrase structure. Ordering nouns, verbs, adverbs, adjectives and all the other components of a sentence would be tried in a natural order. A good nonsense phrase begins to appear to be random as far as a brute force search goes, but it isn't really random.

2.3 Does Odd spelling, punctuation and capitalization help?

Using "Odd sp3LLing5 and CaPitaliZaTiOn" will extend the search by about 1 million tries [1b] per word. Modifying the numbers for passphrases means you probably get more than 8 million (1 million per word) for a decent passphrase. Capitalization at random will cause word length dependent permutations. Adding a single digit 0-9 to a word multiplies the dictionary size by 10. This is a small gain but in some cases may be worth the trouble. Substituting 3 for E, 1 for I, 5 for S and 2 for Z adds the numbers to the possible alphabet. Adding the numbers 0-9 increases the alphabet to 36 characters. Switching letters, letter rotations, letter shifts, and other word scrambling won't help the randomness but they do slow the brute force search some. You can approach a random looking passphrase in this manner.

2.4 What if I use random words?

A dictionary [3] has around 74,000 words in it. Using the 128 bit key size we then need, $\log(2^{128}) / \log(74,000) = 7.91$, random words from our dictionary. Rounding up, you will then need 8 random words to make the passphrase harder than the IDEA key. A brute force dictionary attack will then take slightly longer than a brute force attack on the IDEA key. This is a decent way to generate a passphrase except that it is kind of hard to remember sometimes. This is pretty easy to type though.

2.4.1 Can I use a key generator dictionary?

A smaller dictionary can be searched much faster. Just having one around is enough of a clue to start with that instead of the normal searches. So, you better be sure your key generation system is really random. Programs can be compromised, written poorly or simply monitored. Try Diceware [8] for a good random passphrase generation system. It is irrelevant if the dictionary has any tricks that make the construction of the words more random. In the end, the search space is all that counts. The random number source may not be random and further reduce the search. For these reasons, you need to be sure your key generator is really random.

Here is what effect different size dictionaries have. Using a 10,000 word dictionary, (from section 2.7) $\log(3.16E13) / \log(10,000) = 3.37$ or about 4 words are needed to last more than the average 6 months.

Using the same dictionary to create an IDEA equivalent passphrase gives us $\log(2^{128}) / \log(10,000) = 9.63$ or 10 words are needed. Using a 25,000 word dictionary means $\log(2^{128}) / \log(25,000) = 8.76$ or 9 words. A 50,000 word dictionary needs $\log(2^{128}) / \log(50,000) = 8.20$ or 9 words.

2.5 What if I use all random letters?

The standard alphabet has 26 letters in it. Doing the math again we get $\log(2^{128}) / \log(26) = 27.23$ random letters are needed. Rounding up will mean using 28 letters to make it harder than the IDEA key. Memorizing the 28 random letters would be tough to do, but it isn't impossible. This isn't too bad to type though.

2.6 What if I use all random characters?

If we use all possible printable ASCII characters we end up with 95 possible characters to work with. Punching buttons we end up needing $\log(2^{128}) / \log(95) = 19.48$ random characters for this method. Rounding up again, 20 random characters are needed to make this method harder than the IDEA key. Memorizing 20 random characters is still a tough job, and it is kind of hard to type.

2.7 How long does it take to attack a passphrase?

We can assume that a 1 million key per second key cracker is possible. A Pentium executes about 1 instruction per clock cycle with pipelining [3]. Using a 200Mhz Pentium and minimal instructions shows us that a small program will run 1 million times per second. The Cyrix 6x86 is faster for an identical clock speed and RISC chips are even faster. This means that without stretching current technology much, we can program a desk top computer and try $1E6 * 60 * 60 * 24 * 365.25 = 3.15576E13$ keys per year. A key of random words must be $\log(3.16E13) / \log(74,000) = 2.77$ or 3 words to last longer than an average of 6 months. The random 3 word key has all keys searched in about 1 year. In the end, what we are really trying to do is stop a dumb computer attack. The smarter the computer gets, the slower the computer gets. We can always build custom hardware and just use the computer as a monitor or controller.

2.7.1 Something anyone can do

In an experiment on a 486DX2-66 w/128k cache, a RAM drive was set up, Smartdrv, an *unmodified* copy of MIT PGP262, and all other files needed were loaded. RAM shadows were enabled, video and BIOS cacheable and any other setting that made it all run faster. A program was written in QBASIC (it comes with DOS 5 and 6.x) to try a passphrase using the passphrase environment variable to send new passphrases to PGP and check exit error codes. PGP was executed with +batchmode.

Using this method, it is possible to try almost two passphrases every second (1.8125 actually). PGP has beeps and delays when errors are detected, but were minimized by some of the settings used. In order to seriously attack a passphrase, you would need to modify PGP to eliminate the delays and speed it up.

The moral is anyone can get a single random word from a small dictionary in about an hour. Most larger dictionaries can be searched in less than a day. Just about anyone has all the tools needed for this attack. The program and the settings needed to do the work are simple enough for any decent high school hacker.

3.0 How do I make a strong passphrase?

The answer depends on how secure your passphrase needs to be. Start with a normal phrase and then with a bit of random help, distort it. Make a nonsense phrase by changing words. Remember to switch the sentence structure around in a random fashion. Add a few random words or characters to enhance the security. The goal is to create something you can remember and last as long as a brute force attack on the IDEA key.

The phrase, "my unbreakable super pass phrase can't be beat", is weak by itself. So what if we change it some? "mile unbraking stupor past froze can tent bee beets" is all well and good except that in an attack, a homophone dictionary may be used. On the other hand, in one pass we have a nonsense phrase that has a different structure and words that don't quite logically connect. Add several random characters to make it impossible to guess by any means other than brute force and you are done. The phrase is fairly easy to remember because you used a normal phrase to construct it. If you forget the actual phrase you will probably be able to reconstruct it. Being human, we tend to do things the same in a predictable manner.

For more security, you can generate fully random phrases or character sequences. This will take time and may be difficult to remember. Your level of security is easy to control by limiting the key length. One nearly foolproof method is Diceware [8].

4.0 How strong is my passphrase?

Now using what we know of absolute minimums and maximums of a passphrase, we can make up a little formula to calculate how secure any given passphrase is. For purposes here, random means really random. Psuedo-random methods like rnd() and linear congruential generators don't count here. The constants are based on the needs of PGP. You may need to change them for your use.

PS = passphrase security

FF = fudge factor

this is an attempt to include variables like nonsense phrases,
odd spelling, punctuation, capitalization and numbers.

RW = random words (Don't count as a nonsense phrase)

RC = random characters

RL = random letters

OC = odd characters (other than lower case letters)

LC = total character count

(letters in whole words, spaces ignored) (don't count if a totally random system is used.)

Note: fudge factors may change when more work is done.

F1 = 0.5 = nonsensical phrases hooked together

F2 = ? = odd spelling/misspelling, punctuation and capitalization

This is a permutation dependent on the number of characters changed and the length of the words used. To simplify use $F2 = 4 * OC / LC$

F3 = .09 = random numbers (exclude if F2 is used)

$FF = 1 + F1 + F2 + F3$

$PS = RW/8 + RC/20 + RL/28 + LC/107 * FF$

Calculating the passphrase security (PS) should be a simple matter for most people. A $PS > 1$ means it will be easier to attack the IDEA key before your passphrase will crack. A $PS < 1$ means that it is probably easier to attack your passphrase instead of the IDEA key. If you have a PS under 1, you may still have a secure passphrase. An estimate is that PS values less than .35 can be broken in less than a

year. The formula is under construction and is only a guide number. There is hope that any errors are on the conservative side and it is probably possible to fool the formula.

4.1 Passphrase examples

These are examples of passphrases and the PS numbers associated with them. If you can work through these and get the same numbers, then you are well on your way to understanding how to make passphrases good or bad.

.855 Nonsense phrase

betty was smoking tires in her peace of pipe organs and playing tuna fish.

1.05 A random bunch of characters.

A6:o@6 Ls+\` uGX%3y[k

1.34 Odd capitalization/punctuation and nonsense.

Web oF thE Trust is BrokEn cAn You Glue it Back ToGether? and give it xRays.

.280 An average phrase

There is a sucker born every minute.

1.125 Random words

paper factors difference votes behind chain treaties never group

.761 Phrases with some random letters.

Ignorance is bliss. spgemxk Education cures ignorance.

5.0 Who might try to get my passphrase and how?

Why would anyone want your passphrase? For almost all of us, no one is really interested in what we encrypt. The worst "enemy" we might normally face is a family member that is poking around where they don't belong or maybe the system administrator where your internet account is. Most family members these days probably wouldn't know where to begin attacking a passphrase and even 256 bit RSA would be safe from the computer illiterate crowd. For the really paranoid or fringes of society, the FBI or other major law enforcement agency might be looking. Everyone who knows what they are doing will try to get the passphrase without trying a brute force attack.

5.1 Law enforcement and the government

If you are investigated by a law enforcement agency, then this is what you might get from the various sources. All your communications would be monitored. When they think they have enough information, the law enforcement agency will hand you a search warrant and they will go away with your computer and disks and probably a lot of other stuff as evidence. They will probably already have copies of plaintext traffic from and to you. While they are at it, they will probably take you in for questions. Once they have your computer, they will make copies and search the hard drive. If any or all of it is encrypted, they will try to decrypt it including any deleted files that might remain on the hard drive. If your passphrase is anywhere on the hard drive then they have the key to all of the files encrypted to you. Law enforcement has their own computer experts and can call in professionals as needed. Your individual experiences may vary depending on what country you are in.

5.2 Multitasking systems

You can't trust Windows 3.x, Windows 95, OS/2, and any other operating system that swaps memory to the hard drive or that uses virtual memory. For Mac users, the RAM disk may be saved to the hard drive automatically. Several windows users have found their passphrase in the swap file. It should be safe to run PGP in a DOS shell from Windows as long as Windows is inactive or in other words, no

DOS windows. Windows programs that shell to DOS seem to directly write the passphrase into the swap file. There are several programs that will search the entire surface of a disk with little more than point and click. It is also pretty trivial to write a simple program that searches a file for text strings. More serious attacks and deleted files may require one of the many services that recover data from an unreadable disk. The main problem with multitasking systems is one of control. You simply can't effectively control what happens with the things in memory.

5.3 Multi-user systems

On the bigger multi-user systems, it is trivial for anyone with enough access to install snooping programs, make copies of files, and in some cases even directly monitor a user. You can also include networked PCs. On a network, you can control things remotely with the right software. Some network software may even come with programs that allow limited snooping. Using the computer at work could be handing your passphrase to a variety of people. Many people try to get around this problem by using a separate key on the multi-user system and a secure home key.

5.4 Tempest and electronic surveillance

It is pretty well known that the electronic noise from computers can be monitored and even used. Every wire acts as an antenna radiating any signals that might be on it. The tricky part could be finding the one computer among several identical computers. If there is only one computer, then the spy job is pretty easy. In some cases, it is much easier to shield a room than to buy specially shielded equipment. The hardest part may be identifying the leaks and plugging them. Every wire into a room could carry a signal out of the room no matter how well the shielding is constructed. You would have to be pretty important to a major government or corporation before you need to worry about a tempest attack. Some tests with some really basic equipment showed that quite a bit of noise came from a monitor, very little noise was around a steel cased computer, and the keyboard allowed some noise. All cables used during the testing appeared to be shielded and the computer was idle with a variety of data shown on the screen. The detection equipment wasn't very sensitive so there may be more noise than was actually detected.

6.0 How do I securely store my passphrase(s)?

The best way is probably a key splitting technique. You need to distribute pieces of a passphrase that protects all your regular passphrases. There is a number of ways to do this that will safeguard your keys even if you lose a few friends. A simple method would be to break up the key passphrase into 3 pieces. Then give the pieces to 6 different friends. To reconstruct your passphrase you need only 3 of your friends and you have backups. Do the same thing with your actual passphrase file. The individual friends can't reconstruct your passphrase and they can't assemble the pieces unless all 3 of them cooperate. The security of this method improves if you use more people, but the most important part is having copies of your keys distributed in a way that you can recover them and no one else can. You should have at least one copy of PGP and your keys some place other than your house. Remember to limit your risks. See [1] for more on key splitting techniques and other references.

6.1 Should I write my passphrase down?

I'll contradict myself now. For total security, you shouldn't write your passphrase down anywhere in any form, ever. Using the above key splitting technique isn't perfectly safe.

Writing your passphrase is a breach of security if care is not taken. Many ordinary disposal methods hand your written passphrase to anyone looking. A simple technique with an ordinary pencil will grab

a passphrase from a pad of paper after the top sheet where the actual writing took place is removed. Throwing the copy of your passphrase in the trash gives your passphrase to the dumpster divers. Even trash from your house can be searched without much trouble. A wallet isn't a good place if you get hurt or your wallet gets stolen. There are many other problems with things that are written down.

7.0 How do I use the passphrase switch or variable?

It is recommended that you don't use these methods. The reason is that it becomes a huge security hole unless you are extremely careful. Misusing them or making common mistakes will leave you vulnerable to single word dictionary searches or hand your passphrase to an attacker. Double check using PGP in manual mode and a test case to be sure your batch process is working correctly before using it on sensitive data.

The primary system for this section is an MSDOS PC. UNIX, Mac, and others will be different. The primary purpose here is to show you the possible risks. It is highly recommended that you read the PGP manual and the operating system manual for your system before using these methods. Even that isn't enough sometimes. Some manuals are pretty obscure or just don't have the information.

7.1 Setting the environment variable PGPPASS

Many people have developed some good and some bad methods to try to limit the security risk involved with using PGPPASS. My method for running serious batch programs is setting a dummy passphrase to allocate more environment space than you need in the autoexec.bat. If you don't allocate enough space then you may get an out of environment space error later. Then the batch program, usually QBASIC, changes the environment setting from the program through user prompts. The program process runs, and then resets PGPPASS to filler space. The security in this is that everything gets over written in memory. Your passphrase is never written to disk.

7.2 Using the -z switch

The command line switch is a convenience for some users and batch processing. Under MSDOS, you are limited to a 128 character command line. A good passphrase can be over 80 characters in length and limits the usefulness of this. Additionally, if you have spaces in your passphrase, you will only get the first word or up to the first space if you don't enclose it in quotes. Many have found that their perfect passphrase was completely useless when PGP was only getting the first word.

7.3 Passphrases in batch files

The best recommendation is don't do it. If the batch file is found then they have your passphrase. It gets kind of complex keeping this method secure. Set a dummy passphrase in your autoexec.bat. Now in a batch file, prompt for user input of the passphrase, set the real passphrase, execute the PGP commands, overwrite the passphrase, and then exit the batch file. Always make sure the real passphrase gets overwritten before you exit the batch file. Be careful about using quotes around passphrases with spaces in them and test everything.

8.0 Copyrights, numbers, glossary and other leftovers

Copyright © 1995-97 by Randall T. Williams <ac387@yfn.yzu.edu> This is free to distribute where it might be useful and not for profit as long as this notice remains attached.

8.1 The really big numbers and other numbers

These are here to show how big these numbers really are. They are hard to work with and there is no good reason to use them other than to try and put things into scale. You need more than a pocket calculator to work with them in this form. Take note of the length of 2^{128} . This is the size of a 128 bit number. A 512 bit modulus is about 4 times as long.

1 million	=	1,000,000
1 billion	=	1,000,000,000
1 trillion	=	1,000,000,000,000
3.15576E13	=	31,557,600,000,000
2^{128}	=	340,282,366,920,938,463,463,374,607,431,768,211,456
$74,000^8$	=	899,194,740,203,776,000,000,000,000,000,000,000,000
95^{20}	=	3,584,859,224,085,422,343,574,104,404,449,462,890,625
26^{28}	=	4,161,536,836,220,038,342,098,551,818,958,537,752,576

These are the $\log(x)$ numbers used through out the FAQ. Mostly it is an attempt to make the math easier even if you don't understand what $\log(x)$ is.

$\log(2^{128})$	=	38.53183945
$\log(3.16E13)$	=	13.49910397
$\log(74,000)$	=	4.86923172
$\log(50,000)$	=	4.69897004
$\log(25,000)$	=	4.397940009
$\log(10,000)$	=	4.0
$\log(95)$	=	1.977723605
$\log(26)$	=	1.414973348

8.2 Glossary

This is to clear up a few things in case the context isn't clear. More definitions will be added as needed.

attacker = This is anyone who might want your passphrase. It could be your little brother or sister, wife, friends, hacker down the street, law enforcement and many others.

brute force search = This is a search of the entire key space. Every possible combination will be tried in sequence. Eg. Briefcase combination locks have a key space of 1000 and will be searched (000, 001, 002... 999).

key size = The actual size of the key. Eg. IDEA has a key size of 128 bits.

key space = The number of possible combinations a key can have. Key space is sometimes tricky to compute if there are methods of attack other than trying every possible combination. Eg. IDEA has a key space of 2^{128} .

psuedo-random = a mathematical sequence or other repeatable sequence that appears to be random.

random = A sequence that can not be reproduced by any means other than replaying a recording of the sequence.

search space = The size of the search needed to break a key. Sometimes keys have a much smaller search space than the key size might dictate. Eg. A 40 digit/130 bit hard number, (toy RSA), is bigger than the 39 digit key space of IDEA but can be factored in a few minutes or less using one of the faster factoring methods.

9.0 References

There are other books that could be included like statistics and books on calculating odds. I also may have missed a few references. I used [1] a lot in this document because of it's encyclopedic nature instead of including a long list of separate references.

[1] Bruce Schneier, Applied Cryptography. John Wiley & Sons, 1994

(1st edition paperback)	(2nd edition paperback)
[1a] p. 144-5 and p. 190-1	p. 173-5 and p. 234
[1b] p. 141-3	p. 170-3
[1c] p. 190 (1.2 bits Shannon)	p. 234 (1.3 bits Cover)
[1d] p. 212	p. 256 (No specifics)
[1e] p. 347 Chapter 15	p. 369 Chapter 16

[2] The Random House Dictionary. Ballantine Books, 1980
(paperback, about 1.5 inches (3.8cm) thick with "over 74,000 entries")

[3] Nick Stam, Inside the Chips. PC Magazine Feb. 21, 1995
p. 190-199

[4] Grady Ward, Creating Passphrases From Shocking Nonsense

[5] The Oxford Dictionary of Quotations. ???
("over 40,000 quotations" from a sales add)

[6] RFC1750 Randomness Recommendations For Security
One source is: <http://www.clark.net/pub/cme/html/ranno.html>

[7] Randall T. Williams, A Simple Random Noise Source, July 01, 1995
Posted to sci.crypt and alt.security.pgp 9/95 and 10/95

[8] Arnold Reinhold, Diceware (A Passphrase generation system)
<http://world.std.com/~reinhold/diceware.html>

HTML 3.2 Checked!

Last modified: 23 Mar 1997

Author: Randall T. Williams <ac387@yfn.ysu.edu>

Comments: galactus@stack.nl

This document was generated with Orb v1.3 for OS/2.