A Little Privacy

for the Little Guy

Computer-savvy methods
to obstruct totalitarian
surveillance of ideas

by

Douglas Lowry

https://marpx.com

Dedicated to the young people who

peacefully, but relentlessly,

protest régimes that deny human dignity

THIS BOOK IS FOR ...

Computer-savvy youth under repressive régimes

Governments of many nations feel threatened by dissent and use all manner of techniques to maintain control. There is a fairly steady flow of news reports from around the world: "Authorities in ____ [fill in a country name] have instigated a harsh crackdown on student protesters after thousands demonstrated against the government Tuesday."

Surveillance of Internet use, social media, and email is a standard way to track protesters. This book presents a roadblock for surveillance of ideas. Here is a tool for privacy for your plans and calls for action. Here is a tool for freedom. There is free software online at Marpx.com. Learn how it works. Better yet, understand how to use computer smarts to strengthen privacy in the future.

Computer Science students

TABLE OF CONTENTS

SWAP, SCATTER

WHAT ABOUT KEYS?

ENCRYPTION

DECRYPTION

Invert half of the key
Assignment:

CHAPTER 15 Walk the encryption back

Unscatter bits
Reverse the swap
Repeat, repeat, repeat
Source code almost identical
If compressed, decompress
Verify the plain text cyclic check bytes
Assignment: Reverse the encryption you did earlier

IT'S ABOUT PEOPLE

CHAPTER 16 The players

The little guy
The little guy's programming buddy
The confidant
The team
The designer
The licensor
The adversary

CHAPTER 17 The little guy

Use an existing program
User interface options
    Programming buddy can rework
User preferences
User manual
User ID
    ensure the right of access
    Under the hood -- just like a pseudo-code
    Too short, too vulnerable
    Too long, too complex
Two factor User ID vs offline operation
The file selection and offset method
    Transmit a complex code with simplicity
        File or book, starting point

Make your small group's own key expanders

UP FRONT GLOSSARY

Selected terms are used repeatedly in this book; some of them are particularly important to help you understand the technology. Here is an alphabetic list. As this book develops, links to detailed explanations may be added.

Bit scattering

One way to destroy patterns in electronic content is to move each bit far away from its neighbors, as unpredictably as possible ... the further from the bit next to it, the better. Do this to randomly long sections of input at a time. Scatter bits far and wide. Use extended random sequences to guide the dispersal.

There is an unlimited number of ways to scatter bits. Ideally, use a different method each time.

Bit swapping

Another way to destroy patterns in electronic content is to substitute six or more bits at a time, again guided by extended random sequences. The computer app that you study here, *A Little Privacy,* alternates bit swapping and scattering.

Computational infeasibility

You can challenge hackers with so many alternatives that there is no chance they can try even a small portion of them in a reasonable length of time. Get them dealing with impossibly large counts. The count of particles in the entire universe is a number estimated to be 81 digits long. Try to think of that as a small number.

Confidant

A confidant is a person with whom you wish to exchange computer files and messages confidentially. It's common for a user of *A Little Privacy* to have dozens of confidants.

Confidant code

A confidant code is a long sequence of letters, digits, and other characters. Two confidants or a team of confidants each share (whether they know it or not) a sequence that is used in the encryption of a random file. This code is kept hidden from everyone else. In licensed versions of the app, it is hidden even from the confidants.

Entropy

In colloquial terms, an encrypted computer file has high entropy if it looks like the dog's breakfast or your teenager's bedroom. Entropy is more commonly understood in terms of the Second Law of Thermodynamics, to the effect that the amount of disorder in a closed system cannot decrease. Think of disorder, randomness, and lack of predictability.

In encrypted content, high entropy is a good thing. What is high? Unfortunately, there is no one-accepted-standard-measure of randomness or disorder in a file. But the more thoroughly patterns are destroyed, the more grief we give to hackers.

Key

In general, a key is information which controls the process of encrypting plain text and of reconstituting plain text from an encrypted version. Key lengths are measured in bits, and normally longer keys are stronger than shorter keys -- strong in the sense of yielding encryptions that are less vulnerable to attack. For our purposes, a key is a carefully selected subset of a key expander.

Key expander

A key expander consists of many random arrangements (or permutations) of the integers zero to 2^N - 1. If N were 4, then one possible permutation of the numbers 0 to 15 might be 14 3 11 9 2 5 15 4 0 8 12 13 6 1 7 10. Key expanders are used to strengthen privacy. Each single character in a pseudo-key is replaced by one of these much longer permutations. The purpose of key expanders is to convert relatively short pseudo-keys into keys that are far longer and as unpredictable as possible.

Pseudo-key

Pseudo-keys, key expanders, and keys are all invisible in *A Little Privacy* apps. Nonetheless they do exist. A pseudo-key is some combination of digits, capital letters, lower case letters, spaces, and periods. Here is an example of a 32 character pseudo-key: "eT8.3jLD8vDSp k1l1iubRm6 49iEAWd". There is a different permutation of integers to replace each of the 32 characters depending on both the character and its position in the pseudo-key. Pseudo-keys are selected by a highly random process; each substitution of a character is by a very unpredictable random permutation. The result is real keys that are long and next to impossible to predict.

Random numbers

Numbers are selected randomly to the extent that the choice of numbers to date has no bearing on what number might be selected next. Random numbers are used in this technology to create private key expanders, confidant codes, first-time user identifiers, etc. Therefore there must be the least possible bias in sequences of random numbers that underlie much of this technology. Unpredictability for hackers and quality of privacy for users all depend on high quality random number generation.

Roster / Confidant list

Roster is a common word for a list of people. Here we use it for a list of people with whom a user wishes to exchange files and messages confidentially, that is, a confidant list. In one version of *A Little Privacy*, users manage these lists for themselves. In other versions, a separate license is provided for each confidant relationship.

Shared versus private key expanders

There is a key expander built directly into the app. This *shared key expander* equips you to exchange private files and messages with anybody with whom you have a confidant relationship. In other words, there is a confidant code in place. either in a line in your roster or within a license that has been provided to you.

Private key expanders work privacy magic. If your confidant and you work securely, you are the only ones in the world who have that particular key

expander. The count of possible key expanders is a number currently 624 digits long (and likely bigger in the future). The hacker's problem: No access to the key expander, no decryption. That's not bad privacy protection. (Incidentally, for the present, private key expanders are available only in the United States.)

Symmetric key encryption

Symmetric means that the person encrypting and the person decrypting each must have access to the same key. Historically, the management of symmetric keys has been a *huge* problem. That problem has been solved. When you select a confidant, the *Little Privacy* app behind the scenes uses the confidant code to *encrypt* a file of random bytes. This encrypted file is treated as if it were a bank of pseudo-keys, different for every two second interval within a three month period. Of all the people in the world, the confidants alone have the ability to set up and use this pseudo-key bank.

A different random key file is provided for each calendar quarter.

Caution: A license is needed for commercial / government use of this technology from Marpex Inc.'s U.S. Patent No. 10,637,837.

User ID

In any program dealing with security, the user must identify himself / herself adequately. Many acceptable methods can be found. In this book, we touch on some ways this can be done to achieve both simplicity and security for the user. Security matters. Simplicity *really* matters. The trick is to do both.

PROBLEM, SOLUTION, SOURCE CODE

CHAPTER 1 The Problem

Here is a one chapter non-technical breather before we get into the bits, bytes, and C++ source code. Why this book? Why this technology?

1.1 Invasion of privacy

In a world of social media and Internet communications, we have access to neat tools, all for "free". We give up little fragments of information about ourselves, but we get in exchange so much more ... easy sharing with friends, instantaneous communication to anywhere in the world, a voice to broadcast our views.

The cost? Privacy.

We are told to get used to the loss of privacy, that it is a fair trade-off. "You get the benefit of more targeted advertising. What's not to like?" We were a little surprised to learn that our "little fragments of information" could be turned into political campaign tools featuring "5000 data points on every U.S. voter", but our reaction was to shrug and get on with day to day life.

Still, there is an uneasy feeling as more and more organizations get in on the act, asking for personal information from us. Columbia Gas provides the natural gas to heat my home. They want my social security number so they can provide me the convenience of paperless billing. Nineteen phone calls a week come from robo callers who want our information or our money (and preferably both).

You can work your way around the little invasions of privacy without too much difficulty. It's easy to ignore firms that demand too much information. Route the phone calls you don't recognize to the message machine: "Please say who you are. If we are here, and this is not a robo call, we *may* pick up the phone. Otherwise leave your name and phone number, followed by a brief message."

Constitutions of over eighty countries give lip service to a right to privacy. The Supreme Court of the United States in its Griswold v. Connecticut decision in 1965 chose to find such a right, hidden in the shadows and penumbras of the U.S. constitution. So politicians can rhapsodize in their speeches about how essential and good privacy is for good guys, while they quietly legislate how privacy is bad for bad guys.

There is lots of positive talk about privacy. But does privacy *really matter* in the Internet age?

That depends on who is invading privacy and why.

1.2 Transfer of power without consent

Knowledge is power. If you do not believe that, watch the behavior of bureaucracies and regulators. They want your information. They are often given power to enforce their demands, so that compliance becomes a major economic activity. Does that mean that bureaucracies and regulations are necessarily bad? Not at all. But agencies can justify budgets and grow in proportion to the wealth of data that they hold. That's a standard feature of organizational life.

There is no real problem with power, provided there is consent. In democracies, consent of the governed flows through representatives elected by the people.

You have attended wedding ceremonies. Have you ever paused to figure out what is really going on? The core matter is consent. "Will you have this man to be your lawful wedded husband?" "Is this what you really want? Really? For sure?" So long as the two persons affirm "I do", they are hooked. But without consent. freely given, there can be no marriage.

Consent really matters.

Let's look at what happens when there is no informed consent, freely given.

1.3 Lost privacy -- attempts to influence your behavior

Marketing has been unapologetic in its relentless collection of data points on individuals as it seeks to understand consumer behavior. [Disclosure: Your not-so-humble author is a retired Marketing professor.] You have a need for goods and services to support your ongoing existence. Marketers and advertisers have a thousand-times greater need to rent some real estate in your brain, so that you will be aware of their particular product or service. The Holy Grail of marketing is brand insistence, a commitment on your part that you *must* have *their* product and will be satisfied by no other.

Are marketers trying to influence you? You bet. Is this process efficient? Not really. Is there a cost to your privacy? Yes. Do you enjoy the constant bombardment of messages? You answer that one for yourself.

1.4 Lost privacy -- becoming a means to other people's ends

In 1960 a philosophy professor named Karol Wojtyla wrote a book entitled *Love and Responsibility*. It's not an easy read. His first chapter focuses on using other people as objects to fulfill personal ends. He addressed the question: *Is it permissible to regard a person as a means to an end and to use a person in that capacity?* Wojtyla's argument: People are subjects in their own right, not objects for the benefit of others. Using people, whether for fun, profit, or power, degrades not only the object individuals, but also society as a whole. Wojtyla had a deep personal knowledge of régimes that sought to degrade people ... the Nazis in World War II, the Communist rulers of Poland in the decades that followed. Both sets of rulers did their lethal best to rob the Polish people of their humanity, their privacy, their thought life, their cultural values. Why? Because it makes people easier to dominate and to rule.

Love and Responsibility was translated into English in 1981 and its readership grew significantly since as of 1978 Wojtyla had become more widely known as Pope John Paul II. His argument of people as subjects in their own right had a bracing effect on the citizens of Poland. Many attribute the ultimate downfall of the Communist régime to the power and authenticity of his understanding of what it means to be a human person

The dignity of the human person really matters. Privacy and a sense of personal worth are essential elements of that dignity.

1.5 Lost privacy -- arbiters of your values

The process of influence gets really messy when people out there want to determine your values for you. The ultimate goal is total control, not only of what you do, but of what you are. This is privacy violation at its worst.

The poster child for this overreach is the Communist Party of China (CCP). They currently are using their power to create the ultimate in surveillance of every aspect of life, not only in China itself, but also beyond. Facial recognition of way over a billion citizens? The CCP has made that a priority, and frankly they are getting good at it. The objective is to apply a point system that rewards those who conform to the Party's expectations, and makes life miserable for those who do not.

Setting the Party up as the final authority on thought and belief now extends to religion. This headline is from a July 2020 report in the U.K. Daily Mail: "China orders Christians to destroy crosses on their churches and take down images of Jesus in intensifying crackdown on religion". Why should the leadership concern itself about religion? That's because religion (literally "re-" meaning again, and

"ligare" to bind) has something to do with tying things back together, or to get at meaning. The Party insists on its right to establish meaning.

The Communist Party of China sees itself as all-powerful, and now seeks to be all-knowing. Privacy? In China? The CCP says "forget it".

Is the People's Republic of China prepared to enforce its party-centered ideology? Ask its Uyghur population. Ask the young people of Hong Kong who insisted that a fifty year "one country, two systems" agreement should last longer than twenty-three years.

1.6 Lost privacy -- its high social cost

Our appetite for advertising is less than advertisers' willingness to feed their messages to us. Too often, their messages interrupt or distract. In an ideal world, we might enjoy a method that enables us to say when we are interested in what products and services, and let the marketers strut their stuff before us then. The present method seems a poor bargain, in which we are deluged and at the same time asked to sacrifice our privacy and personal information for the privilege of ads that might, maybe, perhaps, be more relevant to us.

Enter the politicians, trying to get attention in every news cycle, while behind the scenes polling our views and gathering information on us at every turn.

Add to that the drift toward the regulatory state, whether based in Brussels, Ottawa, or the District of Columbia. Are we persons or are we merely cogs in administrative machines? Feelings of alienation, of being dehumanized, show up increasingly under these pressures.

Up to a point, we can handle these annoyances. Things get really serious when the actor is a malevolent authoritarian government, as in China, Syria, North Korea, and other nations. This book addresses this last problem in particular.

1.7 Strike a blow for freedom

Let's use technology to raise surveillance costs to authoritarian régimes. Our contribution is limited to surveillance of ideas ... privacy for messages and computer files. If enough people exchange anonymous heaps of what look like random letters and digits, governments will have to focus their efforts. We wish them no luck in their attempts.

How strong is encryption by Marpex Inc.? We don't know. Nobody knows. Privacy technique takes time to evaluate, perhaps years. This book explains in

detail the logic and why we think it is fairly good. You kick the tires, experiment, write code, and make your own judgment. The one thing for sure: If you and others participate, we can together make this privacy protection even better. If it enhances the dignity of young people and protects them from unworthy rulers in Hong Kong or Belarus or the most recent hotspot, *A Little Privacy for the Little Guy* will have accomplished its purpose.

Democracies struggle for an ongoing balance between citizens' needs and national security. About eighty countries have personal privacy spelled out as a right of its citizens. That's not so in the United States; *however*, the Supreme Court ruled in Griswold v. Connecticut, 1965, that privacy is a constitutional right, to be found "in the shadows and penumbras" of the U.S. constitution.

Attitudes on privacy evolve. On the day after September 11, 2001 the United States was primed for intense focus on security as a national priority. Compare that with September 2, 2020 when the 9th Circuit Court of Appeals ruled that the National Security Agency's program of sweeping up metadata of American's phone calls was illegal and possibly unconstitutional. *There is simply no broad public support for mass surveillance in the United States.*

A personal note: On November 19, 2004 three of us from Franciscan University met at the National Security Agency's headquarters with their Technical Leader for Information Assurance Evaluations and with NIST's Manager, Security Technology, Computer Security Division. The topic was the strength of Pryvit encryption (my U.S. Patent No. 6,757,699). Their stance: NSA then had thirty percent of the world's computing power, and was not concerned. I wondered whether they were overconfident and therefore put Pryvit on a back shelf.

Might the technologies in *A Little Privacy* raise costs now for law enforcement? Might it force greater reliance on non-electronic policing methods? If the methods in this book are strengthened to the point of giving headaches to China's PLA cyber surveillance teams, then they will have an effect in democracies as well. Security versus privacy is inevitably a trade-off. The point: Totally sacrificing the dignity and privacy of individuals in order to achieve security (in China, stability) is a terrible error. Power gravitates to the government. The state becomes everything. The person becomes a mere cog. That is totalitarianism. It could happen, even here.

Let's have a little privacy for the little guy.

CHAPTER 2 What's new in this solution?

2.1 The solution, in under thirty words

- Give control to the little guy.
- Eliminate the need for managing keys.
- Make keys secure, invisible, super-long, and unpredictable;
- During encryption, destroy patterns by repeatedly swapping and scattering bits.

2.2 Control for the little guy

Encryption can be complicated.

The U.S. National Institute of Standards and Technology is preparing for an onslaught of cyber-invasion attempts, as the new quantum computing technology gets up to speed. You can guess what country is investing heavily and aims to be the world leader in quantum computers. It's China, of course.

Mathematics is not a favorite subject in our schools. If you take a look online at NIST's Post-Quantum Cryptography Program, you will see mathematics that would bend the average person's head in a hurry. It's no accident that the field of cryptography is left for the most part to experts. That's a problem. The need for privacy can be very personal.

There is lots of talk these days about the importance of privacy. If some firm wants to encrypt your stuff to "guard your privacy", the question to ask is: "Who has the keys?" In other words, who is in control? There are organizations out there that will gladly encrypt your content for you, but some of them will data-mine it first, and make money using your data.

Your privacy should be under your control. One way to keep control is to apply your own little bit of privacy first before handing it to the experts. The idea is to make it expensive for others to find meaning in your files and messages. I made available online in 2019 a program named *Marpx Privacy* that does just that. It makes your stuff private, even from me. The program does a reasonable job. If you and/or others with programming skills will learn from the chapters that follow, we can make this "little bit of privacy" a lot more private. The objectives: Your dignity as a human being. Your control of your privacy.

2.3 Look, Ma, no keys!

Coming up with a new key to encrypt every file or message that you create is a bother. None of us are good at making up unpredictable sequences to guide encryption. Add to that these challenges: record the key (or pseudo-key), keep it safe, securely transmit it to other persons who are intended recipients, and assure that their use is correct and safe. These combine to make "symmetric key management" so annoying that people in cybersecurity are hesitant to impose it upon lay people.

When you first try out the Windows program available at https://Marpx.com, you are given the opportunity to enter a code that will uniquely identify you in the future. Thereafter, you can click on the *Keyless Option*. From that point onward, the one thing you never see is an encryption key. The same will be true if you get the licensed versions or participate in bringing the free online program up to current specifications. There are no keys, not even any pseudo-keys, anywhere in sight.

Are there keys? Yes. They are invisible. In the updated version they change every two seconds and are never repeated. They are long. In the current upgrade, we are aiming for 32 byte pseudo-keys which map into key extenders that make each actual key 12,288 bits long. They are random and unpredictable. If you follow best practices, they will remain secure.

One neat aspect of keyless encryption is that it can be applied within almost any encryption method that can use symmetric keys. This book and the Creative Commons Attribution Share-Alike Noncommercial 4.0 International License give you the freedom to experiment, learn, and use this technology for yourself. Firms and governments are invited to license keyless encryption technology to build into their software as well.

Learn more in chapters xx, xx, ...

2.4 Re- re- re- re-shred down to electronic dust

Keys that are super-long and unpredictable make possible many passes during encryption and decryption. The metaphor of shredding paper fits. Shred paper once and it is in little pieces, perhaps 3/4 by 1/4 inch. Shred it again and the pieces are smaller. Shred it enough times, and you have paper dust.

If the process is well designed, electronic shredding has the advantage that it can be reversed. You can un-shred. Feed the program the encrypted file or message. So long as you are an authorized recipient, you can get back the original, identical byte for byte to the original input.

The way it is done is to alternate between two straight-forward processes. First, replace every so many (currently six) bits with an unpredictable set of the same number of bits. Do that across the entire input, in batches if the input is especially large -- for example, a movie. On a second pass, spread out the bits so that each is as far from its former neighbor as possible. "Send this next bit to Birmingham" -- not quite that, but as far as possible in a seemingly random way. On a third pass, swap bits again, this time following a very different random rule for the substitutions. On a fourth pass, scatter bits again, preferably over different distances and definitely following a different randomizing guide. Do enough passes, and you have the equivalent of electronic dust, with no patterns to cue the People's Liberation Army, the Communist Party of China, or even Facebook. Is encryption like that impossible to break? Nobody knows. Would attempts at breaking into your privacy require really high paid hackers and very expensive computers? Yes.

Learn more in chapters xx, xx, ...

2.5 Privacy that can be scaled up indefinitely

"Scale me up, Scotty, hackers are closing in." It might some day happen that hackers, especially those financed by rogue nation states, come up with ways to break through *A Little Privacy*. We will teach you ways to deal with that problem.

Learn more in chapters xx, xx, ...

2.6 Precision search while encrypted

This is fun. One of my patents, U.S. No. 7,433,893, is for a search engine with two very special properties. It filters out most of the poor search results, so that you don't have to wade through garbage to find what you really want. Second, the searchable data file does not contain the text. It helps if you keep the data file offline or encrypted when you are not using it. But you have the ability to search your stuff while it is encrypted, and the program tells you which file(s) to call down from the cloud or wherever so that you can get at your stuff, while the hackers can't.

Learn more in chapters xx, xx, ...

2.7 You are invited

This book is available to you online while it is being written. Its content is the work of many years, and it's time I got the technology from inside my head out to

others so that it is available in the years ahead. If in the process it makes life miserable for totalitarian governments, that would be a plus.

Please provide feedback via Marpx.com so that the book may achieve the highest standard of communication -- so that the reader cannot misunderstand. If a part is presented poorly, say which sentence or paragraph, tell what you do and don't get out of it. Your constructive feedback will really help.

More than that: Send in your ideas that can make the system provide even better privacy and control for the common man. You are invited to take part. This could become a tool for freedom for people living under rogue régimes. You can have a part.

CHAPTER 3 Source code

3.1 A progression of programs

The first in the following series of programs with source code produces a scrambled alphabet and digit cipher. You are invited to review the documentation for each program, then compile and walk through the program in debug mode so that you may see what happens at each stage. Look over the program output; if it is a form of disguise or encryption, consider how you might go about deciphering the output back into plain text.

Suggested assignments are intended to increase your mastery of the material.

The up front glossary provides a brief overview of concepts in U.S. Patents No. 10,505,715 and 10,637,837. These concepts will become clearer as you walk through programs that demonstrate how they work, one new concept at a time. Outputs of the first few programs are not particularly useful, but you will arrive at a point where you suddenly have a full bore traditional symmetric key encryption in hand, one that offers practical protection of privacy. Later in the series you will have a working keyless encryption program, a Do-It-Yourself tool in which you manage your own roster of confidants.

All going well, you will learn the basics of a licensing system. You will be encouraged to generate your own licensing method, and build it into a full service keyless encryption program.

The programs will be supplemented with executables that enable precision search of encrypted text files and messages.

3.2 Source code ... in C++ with C language overtones

These programs were created in Microsoft Visual Studio 2015. You are free to adapt them to whatever IDE (Integrated Development Environment) you wish. For that matter, you can use the C++ as a guide to express the same techniques in some other computer language.

C language overtones? Yes, some fairly standard methods of C++ are bypassed, for example, "try" and "private" segments within classes. Leftovers from the C language (for example, structs) show up occasionally.

Other C++ compilers may respond with error messages to Microsoft security variants -- fopen_s instead of fopen, strcpy_s instead of strcpy, and so on for

other standard functions . One way around having to change every such line in all the programs in other IDEs is to build your own security variant. Take the standard function, embed it in memory overflow protection, and add the _s to your new function's name.

3.3 Source code ... in an unusual style

Imagine the sense of power for a young kid, working all alone with an IBM 1440 (a cut down version of the IBM 1401 mainframe) and watching from the 19th floor as the sun rose over downtown Toronto. It was the summer of 1964. The language was AutoCoder, a near relative of Assembler and machine language. There was a disk pack for input and output. The power? ... a full 4096 bytes of RAM to hold both the current program *and* the data.

4k of RAM induces a certain frugality in the mind of a young programmer. Keep things small. Make every byte and every bit count. Twenty years later when I invented my first search engine, *FindIt*, that frugality really paid off. Search is not about finding where text is. Search is about determining with utmost efficiency where the text *isn't*. A single bit near the top of a super-compressed bit tree can tell you a billion places where a word or phrase does not appear.

The preoccupation with efficiency continues to this day. Notice what is *not* in the programs presented here. There are no CStrings. Walking in debug through the support functions for CStrings, one can recoil in horror at the amount of wasted machine effort and human time. Neither is there any Unicode; html ampersand codes are used instead of 16 bit characters. And few libraries are added; the main exception is zlib which provides compression when a sample of target text lends itself to efficient compression. (That speeds up encryption and reduces patterns from the outset.)

A major advantage of this relatively simple style of C++ is portability. In an ideal world, the *A Little Privacy* technology would become available on a wide variety of operating systems and devices.

3.4 Interfaces

Many of the programs offered here are command line / console programs. There is no interface to intrude on the learning what really matters in any one concept. In Windows, click *Start*, enter *cmd.exe* in the box, and the DOS command line opens for you. To see a synopsis of what any Marpex Inc. DOS version expects, enter the name followed by a space and a question mark. Example: ALPriv01 ?.

For the major takeaway programs, a Microsoft CHtmlView interface is offered. These are cumbersome since they import much of the logic from Microsoft's current browser. But the user experience is very much like that experienced with web pages.

Want to substitute your own interface? Go for it!

3.5 Learning method -- your role

A most excellent way to grasp the concepts in a program is to proceed in debug mode. Skip by segments that look totally familiar, but stop long enough to see the results of each segment in data memory. Where it is new to you, walk a step at a time. Think like a computer! Stop at a line. Given the content of this line, what should happen? Step through that line, observe the result.

After a satisfactory first pass thorough the whole program, try little experiments. Make temporary changes in the code. What happens at a point that you changed?

Then attempt the assignments. If you get bogged down, can you find others who are working their way through these same programs? Socialize and learn.

If you are really ambitious, port the program to another operating system, device, or language.

Where the program results in disguise or encryption, consider how you might decipher the result. That's a major bit of learning that will help you evaluate the strength of privacy offered by the major take-away programs in this series.

In early stages while this book and its programs are still a work in process, your constructive feedback would be helpful. The goal is to make the technology impossible to misunderstand. Does the writing and do the programs live up to that standard?

3.6 Legal

United States law comes into play in this project. The legal basis for publishing is that the U.S. Patent and Trademark Office has already released the patents. Further, for off-the-shelf same-for-everybody encryption programs that are freely available, the U.S. Department of Commerce through its Bureau of Industry and Security has disavowed any jurisdiction. Licenses for computer programs are inevitably different for each user. Caution: The topic of licenses, especially multiple licenses as in the confidant model, has not been resolved.

The legal basis for your use is currently the Creative Commons Attribute Share-Alike Non-Commercial International 4.0 License. For personal use, go ahead, have a ball. You have no further need for licenses from Marpex Inc. unless you plan to sell products based on these technologies within the United States and its possessions.

The legalities may evolve, depending on how much interest is shown in the *A Little Privacy* project. If there is serious interest, and if we can effectively connect with potential users (the common man) within authoritarian régimes, we will give serious consideration to putting Marpex Inc.'s encryption patent 10,505,715 into the public domain with open source code. That is NOT the case now.

The solution to the symmetric key management problem, U.S. Patent No. 10,637,837, will continue to require a license for anything other than personal use.

SWAP, SCATTER

CHAPTER 4 A simple disguise for text files

Program ALPriv01

4.1 Scrambled alphabet and digit cipher

A Little Privacy program 01 is the first in a series of programs. The overall objective of this series is to make mass surveillance of ideas prohibitively expensive for authoritarian regimes. Another way of putting it: The series intends a little privacy for the little guy.

The technique in this introductory program is a simple swap of characters within a text file. The method pre-dates computers by centuries. The immediate goal is to familiarize programmers with a style of software presentation. The style and content will be adapted in programs in the following chapters.

Be sure to get the source code and relevant files at Marpx.com. ALPriv01 is a console program that takes as its one argument the name of a control file:

```
ALPriv01  IniFile.txt
```

Consider these 64 characters -- digits 0 to 9, capital letters A to Z, lower case letters a to z, and two punctuation characters, period and underscore. They can be arranged in forward order:

```
0123456789ABCDEF
GHIJKLMNOPQRSTUV
WXYZabcdefghijkl
mnopqrstuvwxyz._
```

This forward order is built directly into the source code for ALPriv01.

The same 64 characters could be re-arranged like this:

```
3sH9vDlPEmwMK2IR
rugb0i8nyfaJpC.U
dQqONY7FL6VSx15j
kZ4TGcBA_ezWothX
```

Question: How many different arrangements are possible for 64 characters in which each appears exactly once? Hint: The answer can be reported in two words. A second hint: You can choose the first character in 64 ways. There are 63 that remain. The second can be chosen in 63 ways.

Each possible arrangement is known as a permutation.

4.2 The role of control files

A control file is a commented and self-documenting text file. It provides to a specific program as many data elements as are needed to run the program. The comments explain what is needed. Comments across lines are embraced by a beginning forward slash asterisk /* and an ending asterisk forward slash */. Also within lines any portion that starts with a double forward slash // is considered a comment through to the end of the line. Inputs may include discrete data items as well as names of input or output files. The only command line argument is the name of the control file. This method is particularly helpful in programs that are to be run repetitively, which is the likely case in a learning situation.

Any program driven by such a control file starts by reading in the entire file and removing all comments. What remains are INI style lines each consisting of a key word within brace brackets, a space, and a data element. The program validates the entries before continuing.

Control files are readily edited in a word processor in order to try alternative inputs. IMPORTANT: When saving edited text files, it is very important that you refuse attempts by your word processor to inject formatting. If formatting gets into a control file, the sky will fall, the wrong people will win elections, and the control file will not be readable within the program.

4.3 The control file for ALPriv01

The following paragraphs are drawn from IniFile.txt, the control file for the ALPriv01 example.

At the bottom of IniFile.txt there is a line "{Permutation} Sample64.txt". This key word in brace brackets is followed by a space. The 64 character re-arrangement of 0-9 A-Z a-z . _ (starting 3sH, cited above) has been made into a file Sample64.txt so that it may be used as one of the inputs for ALPriv01.

The name of a second file follows the key word {Target}, again in brace brackets and followed by a space, then a file name. You can make up your own Target files to be disguised. For now, we have included a line "{Target} gettysbr.txt".

Name a file in which ALPriv01 may place the results. Use key word {Result}.

A file Report.txt is opened to handle error messages and warnings. Its name is hard coded within the program since it has to be able to report problems with the control file. Report.txt is deleted at program end if there is nothing to report.

The {Permutation}, {Target}, and {Result} lines provide the inputs needed so that you can examine the logic of ALPriv01 and to see how a very primitive method of disguise can force the would-be reader or hacker to do at least some work to discover meaning.

4.4 The header file ALPriv01.h

This header file follows a straight forward format that is followed through most of the programs in this series. There are under a dozen variables included in a single class. This reflects the simplicity of this first program in the series.

Notice the unsigned char *pTarg and *pCtl, in conjunction with the long values LenTarg and LenCtl. These are used in conjunction with the LoadFileInRam function. The boolean variable IsOpen tracks which dynamic memory is in play at any point in time. The boolean FpInUse in turn keeps account of which file pointers are active. These are all features that you will find standardized across all the source code provided.

4.5 The main file ALPriv01.cpp

In all the programs of this series, the main files are heavy on comments. Each main file starts with a quick overview of the program and how to run it. This introduction is repeated at the bottom of the file in a Usage_() function. The introduction is spread on the screen in response to typing on the command line the name of the program followed by a space and a question mark.

It's helpful to keep all lines throughout a program under 80 bytes in length.

For a person reading the source code, the remaining commentary in the main file provides a linear / time sequence overview of the program. Here are the comment lines from the body of ALPriv01 main:

> Initializing a class saves grief later. memset and memcpy are
> convenient, but they are risky if you are not always careful in
> their use.
>
> SetUp opens the report, gathers information from the control file,
> and the output file for writing.

Create and output the disguise.

Report status.

4.6 The Usage_ function

Build the Usage_ function late in the program, since the description for the user inevitably evolves while coding. To create Usage_, copy and paste the lines of introduction from the top of the main file to a Usage_ section at the bottom. Add line feeds (backslash, n, backslash), doubling where paragraphs are separated. Remove the starting double slash from the beginning of each line, insert backslashes in front of quotation marks. That's usually enough to offer a readable intro to a program.

Do you want to see the Usage_ function in action? Go through the preliminaries to compiling the ALPriv01 program:

- Choose your platform toolset. I use Visual Studio 2015 - Windows XP (v140_xp) to ensure compatibility with even outdated PCs.
- Choice of the Multi-Byte Character Set provokes warnings from Microsoft, but it keeps Unicode at bay, which is important to maintain direct control when manipulating text.
- Let the source code area be your working directory.
- Input the one argument, a question mark.
- Compile the program in your IDE's equivalent of Win32 Debug.

Step into the program and stop at this line near the top of main:

```
if( argc != 2 || argv[1][0] == '?' )
```

You are taken to the next line:

```
Usage_() ;
```

Step into Usage_(). (In Visual Studio, F11 takes you into a function.) Step through the fprintf line and you should see something like:

```
ALPriv01  IniFile.txt

A Little Privacy program # 01 demonstrates an elementary method
of disguising text. It also familiarizes a programmer with a
setup that supports walking through programs in debug mode. See
IniFile.txt which is included with the source code, and also
ALittlePrivacy.pdf, available at https://marpx.com.
```

```
The IniFile.txt input is self-documenting. The program ignores
all but lines that begin with a key word within brace brackets.
```

Break out of the program. Remember to switch the command line argument back to IniFile.txt.

4.7 The SetUp function

SetUp is called once the console arguments have been counted and any classes have been initialized. SetUp receives the argument(s) and their count. A brief overview is presented in comment lines at the top:

1. Open the report file for writing.
2. Load the control file into RAM.
3. Squeeze out comments in control file, squeeze out leading & trailing blanks in lines, count the remaining lines, check that they all begin with '{'.
4. Gather data on every remaining control file line.

SetUp does a lot of work, all of it preliminary to executing the objective, in this case, to disguise a text file.

In debug mode, stop at various points to see what is going on in memory. For example, at

```
IsOpen[CTRL_OPEN] = true;
```

pCtl shows up in memory starting with something like

```
0x00E06750
2f 2a 20 45 3a 5c 41 4c 50 5c 41 4c 50 72 69 76  /* E:\ALP\ALPriv
30 31 5c 49 6e 69 46 69 6c 65 2e 74 78 74 20 61  01\IniFile.txt a
73 20 6f 66 20 53 65 70 74 20 31 35 2c 20 32 30  s of Sept 15, 20
32 30 2e 0d 0a 0d 0a 41 20 4c 69 74 74 6c 65 20  20.....A Little
50 72 69 76 61 63 79 20 70 72 6f 67 72 61 6d 20  Privacy program
30 31 20 69 73 20 74 68 65 20 66 69 72 73 74 20  01 is the first
```

That runs of for over nine thousand bytes. Break again at roughly one hundred lines later,

```
LenCtl = ptNew;
```

and pCtl is now something like these 71 bytes:

```
{Permutation} Sample64.txt
{Target} gettysbr.txt
{Result} Disguised.txt
```

In other words, the comments have been nullified. If you walk the intervening code, you will see the task has been done rather efficiently. In pCtl you now see the handful of lines that will call for action in later stages of the program.

4.8 The LoadFileInRam function

LoadFileInRam is an old warhorse. It can reliably suck a hundred megabytes into RAM. Work within RAM can be far and away the most efficient way to handle text data sets that will either retain their size or grow smaller. Consider this snippet of code from SetUp():

```
else if (!_strnicmp((const char *)(pCtl + pt), "{Target} ", 9))
{
    pt += 9;
    strcpy_s(Path, 256, (const char *)(pCtl + pt));
    pTarg = LoadFileInRam(Path, &LenTarg);
    if (pTarg == NULL || LenTarg == 0)
    {
        fprintf(fpR, "\n Unable to load file to be disguised %s\n\n",
            Path);
        Errors = true;
        return(false);
    }
    IsOpen[TARG_OPEN] = true;
}
```

The {Target} file is later reworked in place by ALPriv01. LoadFileInRam is always called with a file path, and invites by argument return of the file length. The unsigned char dynamic memory is made the appropriate length to just hold the file, and the file is returned in that memory.

Assigned dynamic memory requires a mechanism to ensure that it is deleted when no longer needed. That is assured by the ReclaimMemory function.

4.9 The ValidatePerm function

A permutation of 64 ASCII characters was presented above :

3sH9vDlPEmwMK2IR
rugb0i8nyfaJpC.U
dQqONY7FL6VSx15j
kZ4TGcBA_ezWothX

These were placed in an unformatted text file, Sample64.txt. In ValidatePerm this file is first read into RAM, then copied into a 65 byte buffer "Perm" to eliminate line ends and to align them so that the location of each character associates it

with an integer 0 through 63. '3' is in Perm[0], 's' is in Perm[1], 'H' is in Perm[2], ... and 'X' is in Perm[63].

"Anything that can go wrong will go wrong." It was a human, not an angel, that created the Sample64.txt file. Anyone familiar with Murphy's Law will want to assure that there are exactly 64 characters, that they are in the expected group (0 to 9, A to Z, a to z, period, and underscore).and that no character is repeated. In other words, ValidatePerm assures that the input is a valid permutation.

4.10 The Output function

Every character eligible for substitution has a numeric value, 0 to 9 for digits, 10 to 35 for upper case, 36 to 61 for lower case, 62 for period, and 63 for underscore. The characters to be substituted each have a position value 0 to 63 in the permutation buffer.

The work of substitution is done during the process of writing the input in modified form. If a character is eligible for substitution, its numeric equivalent is used to fetch the byte in the corresponding position in the permutation buffer.

Notice what happens to many punctuation characters. When there is no substitute, the character is output as is. See for example the hyphens in the extract from the result below.

4.11 The ReclaimMemory function

Some compilers offer to take care of garbage collection automatically. Neat features and "helps for tired programmers" usually come at a cost, in loss of both control and efficiency. The cost of a do-it-yourself approach is to maintain boolean values to track which dynamic memory remains undeleted and which file pointers are still active. You need only assure that each successful "new" and each call to LoadFileInRam is noted with an IsOpen[] set true, and that each fopen_s is matched by an FpInUse[] being set.

It is trivially easy to adapt this method, the #define statements in the header, and the ReclaimMemory function to each new program that you write.

4.12 Examine the results

R4_c B74cL NZF BLeLZ oLNcB NV4, 4_c 6NASLcB Yc4_VSA 64cAS _T4Z ASxB 74ZAxZLZA N ZLz ZNAx4Z: 74Z7LxeLF xZ jxYLcAo, NZF FLFx7NALF A4 ASL Tc4T4BxAx4Z ASNA Njj kLZ NcL 7cLNALF LG_Njh

n4z zL NcL LZVNVLF xZ N VcLNA 7xexj zNc -- ALBAxZV zSLASLc ASNA ZNAx4Z, 4c NZo ZNAx4Z B4 74Z7LxeLF NZF B4 FLFx7NALF -- 7NZ j4ZV LZF_cLh dL NcL kLA 4Z N VcLNA YNAAjL6xLjF 46 ASNA zNch

dL SNeL 74kL A4 FLFx7NAL N T4cAx4Z 46 ASNA 6xLjF NB N 6xZNj cLBAxZV TjN7L 64c AS4BL zS4 SLcL VNeL ASLxc jxeLB ASNA ASxB ZNAx4Z kxVSA jxeLh gA xB NjA4VLASLc 6xAAxZV NZF Tc4TLc ASNA zL BS4_jF F4 ASxBh

Don't you wish you had said that? Actually, President Abraham Lincoln beat you to it on November 19, 1863 in his Gettysburg Address. Notice the alignment of the words, "R4_c B74cL NZF BLeLZ" with "Four score and seven". Notice the lone letter 'N' in the second line -- very likely the word "a". Notice the letter 'h' at the end of each paragraph -- very likely a period.

What does it take to decipher the result if you did not know the input? Not much. It is a cryptogram, extended a little bit by including digits and two punctuation characters. Cryptograms, usually of famous quotations, used to appear alongside crossword puzzles in papers and magazines. Deciphering was a brief evening's relaxation. Letter frequency is a good signal. See the two files with suffix .byt included with the ALPriv01 files. They show the byte distributions of input and output. With disguised.byt in hand, it's easy to work out that the input is English text and that the normal English frequency patterns apply. "E T A O I N S H R L D U" closely fits the reducing frequencies of letters.

We are starting easy. Ask yourself at each instance later on what it would take to decipher the latest disguise or encryption produced by one of these *A Little Privacy* programs.

4.13 Chapter review

Source code has been offered to you with the intention of familiarizing you with a style of C+ programming. You have been provided tips on how to compile the code and to observe its functioning by walking it in debug mode. This first program is a variation of a *monoalphabetic substitution cipher* [Ed. note: big words makes it sound more impressive.] By changing the input file, you may disguise just about anything that is in standard ASCII text. But the disguise is weak. But it serves as a foundation for what will follow.

4.14 Assignment: A true cryptogram puzzle generator

Adapt ALPriv01 so that it only substitutes alphabetic letters for alphabetic letters. Digits, periods, and underscores should remain unchanged. Do that, and contact your local newspaper editor to offer an unending supply of cryptogram puzzles.

4.15 Thought experiment: A binary file disguise

64 is a lovely number to work with in "computational linguistics", since most meaning in text is derived from the 26 capital letters, the 26 lower case letters and the ten digits. The punctuation characters are at best assistants to meaning.

That's of no help if there are accented letters, special symbols, Greek or Cyrillic or binary characters.

How might ALPriv01 be adapted to cover all 256 characters? How might that improve file disguises? What does that do to the complexity of the program?

Incidentally, 64 X 63 X 62 X ... X 2 X 1, or *64 factorial* is a number 90 digits long. That's a lot of permutations of 0 to 9, A to Z, a to z, period, and underscore. 256 factorial is a bit bigger ... a number 507 digits long. Factorial numbers will come into play later as we attempt to present computational infeasibility to hackers and to the People's Liberation Army Unit 61398 (China).

CHAPTER 5 Simple bit substitution disguise for any file

Program ALPriv02

5.1 From byte to bit substitution

A Little Privacy program 02 builds upon and expands its predecessor, ALPriv01. Both programs produce substitution ciphers, that is, disguises. Be familiar with ALPriv01 before you proceed to this program. We saw in ALPriv01 that byte substitution works only for text files, and that the logic provides a thin disguise at best. ALPriv01 does nothing for binary files.

The two programs ALPriv01 and ALPriv02 share a similar structure and much of the same C++ code. This chapter focuses on their differences. The major change is that ALPriv02 swaps six bits at a time instead of a full eight bit byte. Six bits can hold 64 different values, binary 000000 through 111111, that is, decimal 0 through 63. Another big change: a file management function is added to enable processing of many target and result files in a single run. You will also learn how Base64 may optionally wrap binary so that it appears entirely as printable ASCII characters.

ALPriv02 provides an opportunity to increase your familiarity with hexadecimal notation of numeric values. Familiarity makes the debug process much easier to follow since many IDEs default to hex in debug display of memory.

5.2 The control file for ALPPriv02

To the three key word lines {Permutation}, {Target}, and {Result} is added a fourth key word, {Base64}. If {Base64} is followed by Yes, following result files will be wrapped in printable ASCII characters. {Base64} is a toggle line, that is, it may appear as often as desired in the control file, alternating between "Yes" and "No", that is, between wrapping turned on and wrapping turned off.

Indeed, all of the four key word lines may appear multiple times. A {Permutation} line must appear before the first {Target} line, so that the program knows the substitutions to be made within the target. Another logical requirement: {Target} and {Result} lines can only appear in pairs, target file name first and result file name following.

The permutation file takes the form of integers separated by spaces in an ASCII text file. Each integer between 0 and 63 must appear exactly once. The file is

identified to the program in the same way, the key word {Permutation} followed by a space and the file name.

To simplify text wrapping, the control file uses the same "0 to 9, A to Z, a to z, period, underscore" numbering sequence as in ALPriv01's disguise base. The two uses are different; both use the same numbering sequence because it happens to be convenient for coding in C++ or other languages.

5.3 Unchanged functions

LoadFileInRam is typically unchanged from one program to the next.

The wording in **Usage_** is different for each program, but this function works in exactly the same manner to splash an explanation on the output screen. The wording for ALPriv02:

```
ALPriv02  IniFile02.txt

A Little Privacy program # 02 substitutes for each six bits in input
an alternative value in the range 0 to 63. This binary substitution
cipher follows up on and expands the capabilities of program ALPriv01.
A permutation of integers 0 to 63 guides successive six bit swaps. A
Base64 option is added; when toggled on, the output file shows ASCII
characters to represent the binary six bit segments.

Where possible, processing in ALPriv02 follows the sequence and logic
of ALPriv01.
```

ReclaimMemory is adapted for each program to resolve the dynamic memory allocated and the file pointers that remain active.

5.4 The header file ALPPriv02.h

Change the number following #define MAX_CTL_LINES in the header if you wish to increase the potential number of key word lines that the program can handle in one run.

Notice the struct that precedes the one class in the header:

```
struct CtlLine
{
    long
        Offset;     // of data start within a keyword line within
                    // the comment-reduced version of the control file.
    int
        LnType;     // One of the four CTL_ types defined above
};
```

The four line types are CTL_LN_PERMUT, CTL_LN_TARGET, CTL_LN_RESULT, and CTL_LN_BASE64. The line type and the offset within the control file are all that is needed for the FileMgt function to validate the sequence of key word lines, to handle the three file types, and to toggle the Base64 setting.

Other significant changes in the header's single class are addition of a count of key word lines and a Base64 boolean.

5.5 The main file ALPPriv02.cpp

A significant part of the length in all the programs in this series is the overview of usage at the top, repeated in the Usage_ function at the bottom.

The only structural difference from main in ALPriv01 is that SetUp here gathers somewhat different data, then calls the FileMgt function to iterate through the key word lines. FileMgt in turn calls Output once for each result file.

5.6 The SetUp function

The first three tasks in SetUp are as before:

1. Open the report file for writing.
2. Load the control file into RAM.
3. Squeeze out comments in control file, squeeze out leading & trailing blanks in lines, count the remaining lines, check that they all begin with '{'.

Once the control file is reduced to just its key word lines, SetUp undertakes its fourth task, one that is simpler than in ALPriv01's SetUp. Here, SetUp fills a struct with line types and offsets, in preparation for processing in FileMgt.

5.7 The ValidatePerm function

There is here the same concern as before that each value in the permutation be unique and of the expected type. ValidatePerm handles these concerns in much the same way. The difference: The permutation is of integers 0 through 63 rather than of ASCII printable characters.

5.8 The FileMgt function

Some of the tasks in ALPriv01's SetUp are moved here into a file management function. These tasks include calling ValidatePerm and opening the result file.

Again, the purpose of FileMgt is to handle a significant number of key word lines and disguises of many files.

The new task of toggling the Base64 setting requires only three lines.

Processing the result lines is largely error message handling. The critical point is the call to the Output function, once for each different result file that is requested.

5.9 The Output function

Output starts with its own overview: Disguise the file six bits at a time. Four iterations of six bits occupy three bytes, so processing is actually three bytes at a time. Notice the end condition: If the target is not a multiple of three bytes in length, either one or two nulls must be appended.

Bit operations play a large role in this and in many of the programs that follow. Bit swaps and bit shifts play a critical role in *A Little Privacy*. In C++ and C language, bit operations are fast. The program can perform great numbers of bit operations without slowing down significantly.

The code to rework 24 bits (three bytes) into four six bit intervals has to be constructed carefully. Take the top six bits of the first byte and shift them to the right two bits:

```
SixBits[0] =  (EightBits[0] >> 2);
```

The next six bits are composed of the last two bits of the first byte, shifted to the left four bits, followed by the top four bits of the second byte, shifted right four bits:

```
SixBits[1] = ((EightBits[0] & 0x03) << 4) | (EightBits[1] >> 4);
```

The third set of six bits is made up of the last four bits of the second byte, shifted to the left two bits, followed by the top two bits of the third byte:

```
SixBits[2] = ((EightBits[1] & 0x0f) << 2) | (EightBits[2] >> 6);
```

The fourth six bits are simply the last six bits of the third byte:

```
SixBits[3] =  (EightBits[2] & 0x3F);
```

Once you have this pattern working correctly, the code remains stable -- the same logic works for any division of three bytes into four consecutive six bit segments. Good news: That's as close as you will get anywhere in this book to rocket science complexity.

Applying the disguise to each six bits requires only two lines of code:

```
for( Iter = 0 ; Iter < 4 ; Iter++)
    SixBits[Iter] = Perm[SixBits[Iter]];
```

The Base64 setting comes into play during output. Base64 dates back to 1987 as a method to convert binary content such as images to printable text characters... digits, capital letters, lower case letters, and two punctuation characters to bring the total to 64. Base64 is used heavily; for example, it is found in email files (.eml) to encode images. In A Little Privacy, Base64 text wrapping is recommended for email attachments or whenever disguised files or messages are likely to be transmitted by software that is prone to making unwelcome changes to non-text files. (Yes, that does happen ... too often. Sigh!)

If Base64 is toggled OFF, the disguised six bit segments above have to be recombined into binary, exactly reversing the logic above:

```
EightBits[0] = (SixBits[0] << 2) | (SixBits[1] >> 4);
EightBits[1] = (SixBits[1] << 4) | (SixBits[2] >> 2);
EightBits[2] = (SixBits[2] << 6) |  SixBits[3];
```

If Base64 is toggled ON, we can replace each six bit segment directly with its equivalent Base64 character.

Since Base64 characters can be easily distinguished from blanks, line feeds, and carriage returns, it's okay to beautify the output by inserting occasional blanks and linefeeds. We will see that in the next section.

5.10 Examine the results

Each of three files are 1638 bytes long and reflect the same information: Gettysbr.txt, Disguised.txt (from ALPriv01), and Gettysbr_dsg.dat (from ALPriv02). Here is an 80 byte segment from bytes 141 through 220 of each:

Gettysbr.txt (the original plain text) --

```
141: 46 6f 75 72 20 73 63 6f 72 65 20 61 6e 64 20 73  Four score and s
157: 65 76 65 6e 20 79 65 61 72 73 20 61 67 6f 2c 20  even years ago,
173: 6f 75 72 20 66 61 74 68 65 72 73 20 62 72 6f 75  our fathers brou
189: 67 68 74 20 66 6f 72 74 68 20 75 70 6f 6e 20 74  ght forth upon t
205: 68 69 73 20 63 6f 6e 74 69 6e 65 6e 74 20 61 20  his continent a
```

Disguised.txt (text byte substitutions by ALPriv01) --

```
141: 52 34 5f 63 20 42 37 34 63 4c 20 4e 5a 46 20 42  R4_c B74cL NZF B
157: 4c 65 4c 5a 20 6f 4c 4e 63 42 20 4e 56 34 2c 20  LeLZ oLNcB NV4,
173: 34 5f 63 20 36 4e 41 53 4c 63 42 20 59 63 34 5f  4_c 6NASLcB Yc4_
189: 56 53 41 20 36 34 63 41 53 20 5f 54 34 5a 20 41  VSA 64cAS _T4Z A
205: 53 78 42 20 37 34 5a 41 78 5a 4c 5a 41 20 4e 20  SxB 74ZAxZLZA N
```

Gettysbr_dsg.dat (binary file formed through six bit substitutions by ALPriv02) --

```
141: e0 7d e6 cf 4d 9d f0 bd c4 a6 ad 9a 4c 7d 67 cc  .}..M.......L}g.
```

```
157: bb 0b a4 8f e7 f8 8b 1a cc f5 27 f0 83 2d 59 1d  ..........'..-Y.
173: ad 33 13 a7 a4 73 50 92 fb 04 cc 4d b4 cc 7d e6  .3...sP....M..}.
189: a4 b6 90 3a fa 6d cc fe 15 39 9b 2e 4c bf e7 32  ...:.m...9..L..2
205: f6 86 cc 4d 98 4c ba 10 90 8a 22 4c fd 67 f2 ad  ...M.L...."L.g..
```

These extracts were produced by the Dump function; that's one of the utility programs whose source code is included with this book. Notice that the main body of the dump consists of sixteen hexadecimal values. Two hex bytes plus a space neatly convey values from 0 to 255. Think of letters a through f as numbers 10 through 15. The left column in hex is single units. The next column to the left is multiplied by 16 to convert it to decimal. So hex cf (often shown in C++ with a leading 0x) or 0xcf is c X 16 + f X 1 = 12 X 16 + 15 X 1 = 192 + 15 + 207. Life's easier when you get to the stage in which you think in hex rather than convert to decimal. If that's a bit much, there are two utility programs for which source code has been included -- Hex_Dec and Dec_Hex.

In the Gettysburg Address example, even working by hand, the first disguise is pretty easy to convert back to plain text. ALPriv02 produces a greater challenge through substituting six bits at a time instead of entire bytes.

Gettysbr.txt was also disguised using Base64 output. It's difficult to match up the bytes, since it is fifty percent larger, 2458 instead of 1638 bytes. Here are the first few lines of Gettysbr_dsg.txt (not a dump, but the actual file):

```
RvdZRNCY CPuepBmc p7pdr8uN p7iTp4kd
yFNdr8m4 y8QQJgsK a8eOJBZ5 RHtEKb_2
HdGwRvd4 RNC6C7i5 EvejC7i1 y7i4E9ve
MH3ZI9yT Elt5EPuL fgsqynuG Jli7a8ix
```

The binary file Gettysbr_dsg.dat and the text file Gettysbr_dsg.txt each hold exactly the same information. The latter is the first plus Base64 text wrapping.

A binary file, CDC_Directive1.doc was disguised, applying Base64 wrap and the same six bit substitutions. The disguise, CDC_Directive1_dsg.txt, starts out:

```
GKou_wlZ l5v33333 33333333 33333333
Rd393RWX Hr3l3333 33333333 333s3333
Mk333333 3333v333 Kr3333v3 339hXXXX
33333H_3 339XXXXX XXXXXXXX XXXXXXXX
XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX
XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX
```

Whoops! Not nice. There are another 15 lines of all letter X, with all manner of repetitious content following that. Here is a learning that will really matter later on: Patterns may be reduced by testing whether input is compressible, and if so, applying a standard compression algorithm before disguising or encrypting it. That's a win-win tactic, with early reduction of patterns, shorter output, and better quality disguise or encryption. It also thwarts a hacking technique of applying an encryption method to a file composed entirely of a single letter.

5.11 Chapter review

Program ALPriv02 applies a six bit substitution scheme to disguise inputs. It can be applied to any file whatsoever, and not just to text. Bitwise substitution offers a better disguise. The program is structured so that it may disguise an extended series of files. These may be output in binary form; alternatively, the same data is conveyed by a longer version in which each six disguised bits is rendered as a Base64 text letter, digit, or punctuation character.

This chapter also touched briefly on hexadecimal numbers.

5.12 Assignment: Frequencies of six bit segments

Source code in the bytes utility program demonstrates a way of counting the frequency of every one of the 256 byte patterns that may occur in a file. Make a version of bytes that calculates the frequency of every one of the 64 six bit patterns in a file disguised by ALPriv02 and output in a binary file.

How might you use this information to help break a six bit substitution disguise?

5.13 Thought experiment: Reverse the substitution

How would you design a program to undo a six bit disguise? What might be similarities and differences with this new program and ALPriv02?

CHAPTER 6 Scatter bits to disguise any file

Program ALPriv03

6.1 Objective -- bits widely separated

There are two operations at the heart of *A Little Privacy* -- swapping bits and scattering bits. By alternating between the two operations, and applying a different random sequence to each operation, we reduce the incoming content to a near-patternless array of either binary bytes or Base64 characters. The power is in the back-and-forth combination of the two operations plus the massive import of randomness through very large keys (12,288 bit keys in the current software).

In the scattering process, each bit should wind up in a location widely separated from its neighbors. Scattering is carried out in blocks; the block sizes should vary and be unpredictable. Ideally, each scattering should be processed with differing algorithms, and/or differing parameters so that no aggregate block patterns emerge in large encrypted files.

This chapter provides simply one sample, one of probably hundreds of ways in which bits might be scattered in any one pass through the content. And, of course, each pass is guided by a different highly randomized permutation of integers.

6.2 The ALPriv03 bit scattering program

In order to learn this new bit scattering program, it will help if you have a good grasp of ALPriv02. The reason: All but two of the functions in ALPriv02 and ALPriv03 are virtually identical except for the date stamp and name of the included header file. The significant changes: FileMgt in ALPriv02 differs from FileMgt in ALPriv03 in that the latter calls a new function BitGames before calling Output, which in turn is much simplified in ALPriv03.

6.3 In praise of prime numbers

The BitGames function randomly selects a block size and a distance, each of which is a prime number, each different from the other. What's that all about?

A prime number cannot be factored into two numbers that multiply together, except itself and the number *one*. Even numbers above 2 all can be divided by 2,

so they are not prime. Numbers above 5 that end in 0 or 5 can all be divided by 5, so none of those are prime numbers. Those two facts ensure that all prime numbers greater than 10 end with 1, 3, 7, or 9. Prime numbers range upward 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, etc.

The first test below uses prime numbers to show what actually happens. A later test will show the effect of using numbers that are not prime and happen to have a common factor (for example, 10 and 15 both divided by 5).

6.4 Choose a block size and distance

A *block size* is the number of bytes that are to be reworked in one segment of one pass through content that is in process of encryption. A block size is chosen based on a segment within the permutation that guides the current round of bit swapping. A *distance* (how far in bits a particular bit is to be moved from its home) is chosen based on a different segment of the same permutation. In debug mode, run ALPriv03 and stop it near the top of the BitGames function. The input in pTarg starts with:

```
3c 48 31 3e 47 65 74 74 79 73 62 75 72 67 20 41  <H1>Gettysburg A
64 64 72 65 73 73 2c 20 62 79 20 41 62 72 61 68  ddress, by Abrah
61 6d 20 4c 69 6e 63 6f 6c 6e 3c 2f 48 31 3e 0d  am Lincoln</H1>.
```

The 64 values of Perm in hexadecimal are:

```
03 36 11 09 39 0d 2f 19 0e 30 3a 16 14 02 12 1b
35 38 2a 25 00 2c 08 31 3c 29 24 13 33 0c 3e 1e
27 1a 34 18 17 22 07 0f 15 06 1f 1c 3b 01 05 2d
2e 23 04 1d 10 26 0b 0a 3f 28 3d 20 32 37 2b 21
```

Notice the integer declaration for Prime[8] in BitGames. It ensures that any numbers picked for block size or distance will be prime numbers. Those happen to have been chosen for this BitGames function. A different set should probably be chosen for any other bit shifting functions in your version of *A Little Privacy* encryption.

```
Prime[8] = { 7, 19, 23, 31, 41, 53, 59, 61 }, // Byte block lengths
      // and bit shift distances are selected from these
      // prime numbers.
```

Near the top of BitGames, this source code does the actual selection:

```
 // Set parameters according to next random 0 to 63 in
 // Perm. Ignore instances in which CtBytes and CtBits
 // are the same, since only specific bits are hit, repeatedly.
 CtBytes = CtBits = 0;
 while (CtBytes == CtBits || 8 * CtBytes < CtBits)
 {
```

```
    CodeNo++;
    if (CodeNo == 64)
        CodeNo = 0;
    Rand63 = Perm[CodeNo];
    CtBytes = Prime[Rand63 >> 3];
    CtBits = Prime[Rand63 & 7];
}
```

The first value of Perm, 0x03 comes up with a CtBytes (block size) of 7 and a CtBits (distance) of 31. The logic breaks out of the above *while* loop, drops past an end condition test, sets up a chunk of input, and nullifies a "Replace" buffer.

6.5 Scatter those bits

Here is the code that performs the bit shifting for the current small block. Notice the RBitInChunk variable which starts at 31 in this example:

```
// Now the bit manipulation ... again, a suggested method only.

RBitInChunk = 0;  // Will range from 0 up to BitsInChunk - 1.
for (ByteNo = 0; ByteNo < CtBytes; ByteNo++)
{
    for (BitNo = 0; BitNo < 8; BitNo++)
    {
        RBitInChunk += CtBits;
        if (RBitInChunk >= BitsInChunk)
            RBitInChunk -= BitsInChunk;
        if ((LocalIn[ByteNo] & BitPos[BitNo]) == 0)
            continue;
        RByte = (RBitInChunk >> 3);
        RBit = (RBitInChunk & 7);
        Replace[RByte] |= BitPos[RBit];
    }
}
```

Each time that a bit is empty, there is no point in moving it. It would simply overwrite the null bit that is already there in the (nullified) Replace buffer.

It's worthwhile to follow this segment of code in debug. Watch how each "on" bit is copied to a new location in the Replace buffer. The four bits that are turned on in the first byte '<' (0x3c) in pTarg are moved to scattered locations in Replace.

When the above loop is done, the Replace buffer looks like this:

```
0a b4 ee 16 be 68 24
```

These seven bytes overwrite the original seven bytes

```
3c 48 31 3e 47 65 74  <H1>Get
```

People who are comfortable with hexadecimal can readily count the bits in each byte. 0x0a in bit form is 00001010 which has two bits, 0xb4 is 11010100 which has four bits, etc. The bit count for the seven bytes in Replace is 26. The bit count for the seven bytes from the target is 26. The counts will always be identical if the C++ is coded accurately. It's the same bits. They have been scattered.

6.6 What if not prime numbers?

What would happen if the two number, block size and distance, were not primes and if they had a common factor? Let's try it and find out. Suppose block size of 10 and distance of 15 were chosen. Insert these two *temporary* lines after the selection loop two sections back:

```
CtBytes = 10;   // ZZZZZZZ
CtBits = 15;    // ZZZZZZZ
```

A comment of ZZZ or longer is my signal for "get these lines out of here" when in production mode.

The first ten bytes from above are:

```
3c 48 31 3e 47 65 74 74 79 73  <H1>Gettys   [40 bits in all]
```

The Replace buffer contains only 14 bits. Because of the common factor between the two numbers, some bits cannot be reached, and others have bits moved into them multiple time over.

```
20 84 10 42 08 20 84 10 42 08
```

That's a consistent pattern: If the two numbers are not prime, bit shifting cannot be done accurately with this algorithm.

6.7 Watch the end conditions

What if not enough is left at the end of a pass through the file? Here is how end conditions are handled:

```
// Near the end of the input target file, cut the length of the
// last chunk. Make CtBits the first prime number in the above
// list higher than CtBytes. This forces CtBytes and CtBits to
// differ, and enhances end condition reliability.

if (CtBytes > LenTarg - pt)
{
    CtBytes = LenTarg - pt;
```

```
        CtBits = -1;
        for (i = 0; i < 8; i++)
        {
            if (Prime[i] > CtBytes)
            {
                CtBits = Prime[i];
                break;
            }
        }
        if (CtBits == -1)
            CtBits = 67; // The next available prime.
    }
```

If you break at the "CtBytes =" line above for a first pass through the Gettysburg sample, you will find LenTarg = 1638, pt = 1598, so that CtBytes is changed to 40. The following logic changes CtBits to 41, just greater than 40 and with no common factors. If none can be found, CtBits goes to a higher prime number. The replacement that follows is successful.

6.8 The cumulative block size problem and solution

These little blocks do not offer patterns to hackers. But there is some risk that repeated trips through the same permutation may introduce detectable patterns. The solution is to use, not the same bit scattering function (for example, this BitGames function) over and over again, but instead have multiple functions with widely different ranges, particularly in block sizes.

6.9 Assignment: Algorithm # 2 to algorithm # N

Devise at least one other algorithm that scatters bits effectively, speedily, and accurately. Think outside the box. Here is one example: There are many permutations within a key expander. At the end of using the first permutation, use the permutation that follows instead of the same one again.

The number of possibilities is endless. Use the Feedback button at Marpx.com to send in your suggestions and/or your C++ source code. Thank you in advance!

CHAPTER 7 Orderly thoughts about randomness

Program ALPriv04

7.1 What is entropy? Is it good?

Objective in this chapter: To offer orderly thoughts about randomness.

Our daughter Sue has random thoughts about orderliness. Occasionally. Not too often. Nothing to learn about cybersecurity there, unless disorder is a good thing.

Leave things alone and things go downhill. Don't do routine maintenance, and life will eventually bite back. Leave the universe as it is for a few tens of billions of years, and watch the second law of thermodynamics take over and reduce its beauty to blah.

But the moment we focus on cybersecurity, the discussion changes. Entropy, in the form of seeming randomness in encrypted content ... that's good. The goal in this context is high entropy, the destruction of all patterns that might provide clues to adversaries on how to decipher the encryption. Disorder, randomness, lack of predictability are all good in this context. Unfortunately, there is no one accepted-standard-measure of randomness or disorder in a file. But the more thoroughly patterns are destroyed, the more grief we give to hackers.

7.2 Computers don't produce entropy

Look back at the three programs we have considered so far. Unless you change an input, if you run any one of those programs more than once, the same output results, over and over again.

What's going on? Think of your computer as a dull-witted clerk that happens to be *very* fast. A program is a set of instructions. Those instructions are followed, with great precision and great speed, but with no imagination, flair, inventiveness. Those qualities have to come from the programmer. In Artificial Intelligence systems, the computer appears to learn, but step back and see where the instructions that drive the "learning" originate. In a programmer. Automated code generation? That too traces back to human input.

The point: Computers are deterministic devices. Given a particular set of inputs, the output is fully predictable. And in cybersecurity, the last thing we want is predictable results.

Randomness has to be imported for a computer to encrypt content.

7.3 Import entropy

A line "{Permutation} Sample64.txt" was an input to program ALPriv01.txt. That file consisted of 64 digits, letters, and punctuation characters in no evident order. A line "{Permutation} integers64.txt" was an input in each of programs ALPriv02 and ALPriv03. It held the numbers 0 through 63, in random order. In each case, that {Permutation} line was the source of randomness that enabled us to disguise other inputs.

> A momentary intrusion: To exclusive OR two bytes manually, put their bit layouts one above the other:
>
> 0xB9 = 10111001
> 0x8D = 10001101
>
> When both have a 1, write a zero in the result.
> When both have a 0, write a zero in the result.
> Where there is a 1 and a zero, write a one in the result.
> Result for the pair: 00110100 = 0x34

The ultimate in randomness is a one time pad -- a file of random binary bytes at least as large as what is being encrypted. If there are no human handling errors, exclusive OR-ing each byte of input with the corresponding byte in the one time pad leads to encryption that is inviolable. Nobody can beat it. Unfortunately, one time pads are clumsy to share, human errors are common, and results are insecure the moment a one time pad becomes a two time pad.

Randomness needs to be imported for computer-based encryption. But that needs to be done efficiently and in ways that minimize opportunity for human error. *A Little Privacy* imports randomness through key expanders and key banks, both of which will be explained in the KEYS, PLEASE! section below.

7.4 Random number generator

Some of the programs ahead in this book require random inputs. You will need a source of supply. The problem: Many random number generators fail to produce cryptographically secure results. For learning purposes, it is probably okay to use a free library function, downloadable program, or low cost hardware device. For serious production work, you should research carefully and expect to spend some money. The underlying question in randomness is whether two successive numbers are truly independent of one another. The answers have been shifting in recent decades, and one method after another of generating random numbers has fallen into disrepute. Be sure to include recent inputs in your research.

One simple point, no matter what method of random number generation you may choose: Do not recycle any previously used random sequence. That's why in ALPriv04 the program reports the offset of the next unused byte within a file of random numbers. Use that offset as the starting point in a subsequent run of the program.

7.5 Random permutations of integers

A Little Privacy program 04 demonstrates a method to create sets of random permutations of integers. It's not just any set of integers; it is a set of all the integers from zero to one less than some power of two -- in ALPriv04, from zero to 63.

Why this preoccupation with special sets of integers? They are selected with a view to maximizing entropy (disorder, randomness) in encrypted content. Every element within the sequence is guaranteed to be different from every other element. A sequence guides one iteration of either swapping or scattering bits, and the lack of duplication raises the probability of a more random output from that single pass through the data.

At this stage, and through much of this book, the *bit basis* is six, that is, six bits are swapped at a time, and the permutations are of two to the power six integers, with each integer occurring only once. Toward the conclusion, we will examine what happens if the bit basis is larger, and the effect that would have on the resistance of encryption to cryptanalysis and hacking.

7.6 Factorial random permutations of integers

How many different arrangements of integers zero through 63 are possible, with each integer appearing exactly once? The first integer in a permutation may be chosen from among the 64 integers. Suppose the first is the integer 18. That integer is now out of the running and not eligible for reassignment later in the permutation. 63 integers are left. The second integer is selected from the 63. Say it is 55. Now two integers are disqualified from later assignment, so 62 integers remain. The third is selected from those 62 ... and so on down to the last integer which may be selected in only one way, the lone integer that remains.

How many possibilities are there altogether? 64 times 63 times 62 times ... times 2 times 1. In short, there are 64 factorial ways. When we were taught factorials in high school, we made a game of it. How many sequential factorials could you say in a row working entirely in your mind (no paper, no calculators, no nothing). Try it: 1, 2, 6, 24, 120 (1 times 2 is 2, times 3 is 6, times 4 is 24, times 5 is 120, and so on to 720, 5040, 40320, 362880, 3638800. Most of us kids got shaky at calling

out the numbers when we reached times 8 or times 9. But you get the drift: The numbers in a factorial series start small, but they accelerate at an ever increasing rate.

How much is 64 factorial? The number is 90 digits long. That's longer that the count of all the particles in the universe. There are that many different combinations of two to the power six, factorial. Another way to put it:

$$((2^6)!) = 1.26886942 \text{ X } 10^{89}.$$

The fun with factorial permutations is that numbers grow beyond the stratosphere when there is one such permutation after another, that is, a round of bit swaps, a different round of bit scattering, another round (different yet again) of bit swaps, and so on.

Now, suppose there were 32 iterations of alternate swapping and scattering, each selected in $1.26886942 \text{ X } 10^{89}$ ways. There are $1.26886942 \text{ X } 10^{89}$ ways to select the first permutation, $1.26886942 \text{ X } 10^{89}$ ways to select the second permutation, and so on through 32 iterations. That works out to:

$$((2^6)!)^{32} = (1.26886942 \text{ X } 10^{89})^{32} = 2.040 \text{ X } 10^{2851}$$

Might numbers like this provide meaningful challenge to a skilled cryptanalyst? Might this provide a little privacy to the little guy, at least until we have to scale the method up to swap and scatter higher numbers of bits at a time?

7.7 Random permutations of Base64 characters

Walk the code in debug mode through ALPriv04's MakePermu and MakeCodes functions. Creating a permutation opens far more possibilities than choosing 32 Base64 characters for a string. In the latter, there are 64 choices for each character, since in this case repetitions are allowed. Choosing 32 characters can be done in 64 to the power 32 ways. The count of possibilities is a mere 57 characters long.

Here is a sample output of a permutation of integers, found in Result04.txt:

```
Permutation 1:
29, 59, 04, 09, 30, 58, 38, 07, 48, 24, 08, 15, 50, 46, 57, 28,
34, 36, 21, 05, 32, 44, 11, 45, 42, 53, 56, 27, 43, 41, 63, 61,
26, 14, 19, 06, 55, 10, 51, 47, 25, 00, 52, 12, 22, 18, 60, 49,
62, 20, 16, 39, 54, 01, 17, 23, 02, 40, 13, 33, 03, 31, 35, 37
```

Notice the lack of repetition and how the code satisfies that condition.

Here is a sample random output of Base64 characters:

Code 4: GSybwuJDBGPsm5TgAoflqFoOJoo6VGEF

This time there is considerable repetition -- four instances of 'o'. That's in keeping with the rules built into the source code,

7.8 Assignment: Squeeze a permutation to its minimum size

The sample permutation of integers above occupies 255 bytes -- 128 digits, 63 commas, etc. Experiment with the code in ALPriv04's MakePermu function. Into how few bytes can you squeeze any permutation of the values 0 through 63?

CHAPTER 8 Swap and scatter -- how often?

Program ByteSeq

8.1 Swap and scatter aim for entropy

Chapter 5 demonstrated that substitution for six bits at a time, when driven by some random sequence, reduces people's ability to find meaning in the input. Scattering bits under the guidance of a random sequence likewise disguises input, as in Chapter 6.

To achieve *A Little Privacy*, the swap and scatter functions alternate. Each pass through the input is driven by a different random sequence. Each random sequence is difficult to predict. Swaps always operate the same in the sense that, if the value in the input six bits is N (say 57), then the output for that swap is whatever is held in the Nth (in this example, 57th) group of six bits in the random sequence. Bit scattering, ideally, differs from one pass to the next in either parameters or algorithm, and in each random size block within the pass.

A single pass of swapping or scattering produced a moderate disguise. Now it's time to get serious. Swap, guided by some random sequence. Take the output, scatter its bits following a different random guide. Take that output, swap ... and continue back and forth for multiple passes through the input.

Visualize a paper shredder -- the cross-cut style that grinds paper into pieces about three quarters by five sixteenths of an inch. Imagine taking two cups of those pieces, and running them through the shredder again. The inevitable result -- smaller pieces. Run those smaller pieces through -- repeat, repeat, repeat. Sooner or later the result is largely dust.

The same metaphor applies here; let's call it electronic shredding. For the paper shredder, substitute a *Little Privacy* program. For paper, substitute computer files. For paper dust, substitute electronic entropy. That's the goal.

8.2 Entropy and pattern measurement

As mentioned earlier, there is no accepted standard for measuring entropy in electronic files.

You can get a quick impression whether a file has high entropy by compressing it. I downloaded a text copy of *A Tale of Two Cities* by Charles Dickens, book 98

from the Project Gutenberg web site. Using WinZip, its 766,622 bytes reduced to 270,194 bytes -- a 65 percent reduction. High compressibility goes hand in hand with lots of patterns within the file. Among the ALPriv04 files that come with this book is a random bytes file, Rand10k.dat. When compressed with WinZip, its 10,000 bytes *grew* to 10,161. Random byte files should have no discernible patterns; the higher the quality of randomness, the fewer patterns.

Again, compression can yield a very rough, but quick, impression of the extent of patterns in a file. High shrinkage means low entropy and many patterns. Very low shrinkage or even expansion suggests relatively high entropy and few patterns.

Let's try a more direct method.

8.3 A byte sequence utility program

This book is accompanied by C++ source code for a variety of programs, aimed either to teach some aspect of *A Little Privacy* or to serve as utility programs. My byte sequence program *ByteSeq* presents a way of detecting patterns in any file.

As with all of these files, compile them within your IDE (Integrated Development Environment). On a command line, input "ByteSeq ?" and you are shown what is expected:

> Usage: ByteSeq AnyFile Report.txt
>
> ByteSeq looks for every sequence of four bytes that occurs at least three times in any input file whatsoever. This program is useful to contrast patterns among unencrypted files, files encrypted by any encryption method, and "A Little Privacy" files.
>
> If the file contains text, note that \20 represents a space, while \0d and \0a together represent a line end.

In this and many utilities in this series, non-printing binary bytes are shown by a backslash and their two byte hexadecimal value.

I ran this command:

```
byteseq "E:\DataSets\Dickens\Tale2\a tale of two cities, by
charles dickens 98.txt" Cities2.txt
```

Notice the inclusion of a path before the file name. Since there are spaces in the first argument, it is enclosed in double quotes.

Useful trick in DOS: Rework the above command in a word processor so that the file path and name are correct for your computer. Highlight and paste (Ctrl-V) the

command into the clipboard. Then *right click* at the command prompt in the DOS window. A little menu appears. Click on the word *Paste* within the pop-up that is presented, and your command appears after the prompt.

The result of the above command for the text version of A Tale of Two Cities is a file Cities2.txt which is 19,671 bytes long. It starts like this:

> Four byte patterns that occur at least three times in file
> E:\DataSets\Dickens\Tale2\a tale of two cities, by charles dickens 98.txt
> (length 766622 bytes):
> 19663 patterns reported, total of 748537 instances.
>
> Patterns per million bytes = 25648.
> Instances per million bytes = 976409.
>
> 9433 of 20746865 = \20the
> 7418 of 74686520 = the\20
> 5064 of 616e6420 = and\20
> 4786 of 20616e64 = \20and
> 3959 of 206f6620 = \20of\20
> 3403 of 20746f20 = \20to\20
> 3321 of 0d0a0d0a = \0d\0a\0d\0a
> 3254 of 696e6720 = ing\20
> 2597 of 2c20616e = ,\20an
> 2408 of 68697320 = his\20
> 2407 of 20696e20 = \20in\20
> 2148 of 68617420 = hat\20
> 2036 of 0a0d0a22 = \0a\0d\0a"
> 2032 of 20746861 = \20tha
> 1950 of 20686973 = \20his
> 1855 of 6e207468 = n\20th
> 1834 of 74686174 = that
> 1781 of 20776173 = \20was
> 1729 of 220d0a0d = "\0d\0a\0d
> 1682 of 77617320 = was\20
> 1649 of 20796f75 = \20you
> 1632 of 64207468 = d\20th
> 1472 of 20776974 = \20wit
> 1448 of 74686572 = ther
> 1442 of 77697468 = with

With over nineteen thousand patterns each four bytes in length, all occurring at least three times, the conclusion is quickly reached: No entropy here.

VERY IMPORTANT: Run the byteseq program against *A Tale of Two Cities* after it has been encrypted using *MarpxPrivacy*, a weaker version of *A Little Privacy*, to the older specification. The command:

```
byteseq "E:\MyPrivacy\DBL_a tale of two cities, by charles
dickens 98.txt.enc" Cities3.txt
```

The complete result in Cities3.txt of the above command is as follows:

> Four byte patterns that occur at least three times in file E:\MyPrivacy\DBL_a tale of two cities, by charles dickens 98.txt.enc (length 283401 bytes):
>
> 0 patterns reported, total of 0 instances.
>
> Patterns per million bytes = 0.
> Instances per million bytes = 0.

There are zero patterns whatsoever. That suggests high entropy, in this case resulting from seven passes, alternately swapping and scattering bits.

There are only 256^4 = 4.2 billion four byte patterns possible. So if you test byte sequences in multi-megabyte files encrypted with some variation of *A Little Privacy*, a few four byte patterns may be repeated enough to be reported. That's by random chance.

8.4 Shred and scatter -- how often?

U.S. Patent No. 10,505,715 was written in terms of three iterations -- swap, scatter, swap again, and treat that as the result. It works. But a kindly cryptanalyst recommended that I boost the frequency. Therefore the 2019 version downloadable at Marpx.com was set to seven times.

Enter a cryptographer who claimed (without trying) to the effect: "Oh, I can break that." Hmmm. Current decision: Let the user select any one of four frequencies -- 8, 16, 24, or 32 passes of alternate swap and scatter.

If a file to be encrypted is quite long, it may take noticeable time to pass through the data 32 times. As of this writing (Autumn 2020), no-one has actually proven that they can decipher encrypted files that have gone through only seven or eight passes. Let's play it safe and offer electronic shredding to the 32nd power. If your files are vital to national security, then tolerate the few extra seconds that it takes. The message to your adversaries: Eat my electronic dust.

If you want simply a little privacy, settle for the fastest option of eight passes. If you are unsure, try 16 or 24 passes.

8.5 Import lots of randomness

Thirty-two passes, each based on a different permutation of integers 0 through 63, means a lot of randomness will be needed. The next section takes us into key expanders, pseudo-keys, and real keys. For what it's worth, we will need long, l__o__n__g keys to import the amount of randomness needed. A key based on the current specs turns out to be 12,288 bits long. During the 1990s, venturing beyond 64 bits was utterly suspect. Times have changed. And encryption has become much stronger.

Now even the little guy can enjoy a little privacy -- and, for the first time, totally under the little guy's control.

8.6 A thought experiment

How few passes would be needed to make *A Little Privacy* fast enough to support secure transmission of sound?

WHAT ABOUT KEYS?

CHAPTER 9 Farewell, symmetric keys. Hello, privacy.

Program MakePKEs

9.1 Where do keys come from?

Keys are delivered by storks.

> Let that little fib satisfy you
> until you are ready to deal
> with the birds and the bees
> of invisible keys.

In the next chapter you will learn how U.S. Patent No. 10,637,837 brings invisible keys to birth. This chapter focuses on ordinary keys, and methods to import boatloads of randomness into the encryption process.

> A (perhaps pertinent) memory: In the mid-1950s in Kingston, Ontario the main classroom for first year students at the Royal Military College sloped down toward the front. There was a wide aisle behind the back seats. Out of the professor's sight, that aisle was almost always filled with students, sound asleep, exhausted from the rigors of early morning inspections and military drills. Professors could destroy your life only once a semester with bad grades. The third and fourth year students, abetted by Regimental Sergeant Major J.E. Coggins, were there to ensure that life was hell for the recruits ten times every day.

The moral of that: Stay awake for this chapter and the next. Otherwise you are not going to understand how *A Little Privacy* works.

9.2 What does an encryption key do?

In a computer setting, *keys* are sequences of bytes / bits, supposedly unique and unpredictable, that are used in conjunction with sets of commands (algorithms) to scramble plain text and decipher encrypted text. Early development in this field owes much to Alan Turing and his colleagues at Bletchley Park in England during World War II. Their success in breaking the German Enigma machine coding

stems in part from German military culture which led people to do things that were anything but unique and unpredictable. For example, the German phrase equivalent to "weather survey 0600" was broadcast at precisely 6 a.m. each morning with often predictable wording. Bletchley Park listened and learned.

Keys are a means to bring about randomness in cipher text, a means and not an end.

Recall the Base64 grouping of digits 0 to 9, capital letters A to Z, lower case letters a to z, the period, and the underscore. Random sequences of these 64 bytes have potential as keys. The program ALPriv04 showed how to generate random sequences of these characters, 32 at a time. Source code was set so that characters might be repeated. It is unpredictable how the first character might be selected; there are 64 choices. The other characters can be chosen each in 64 ways. So a pair may be chosen in 64^2 ways, a triplet in 64^3 ways, and a 32 character Base64 string in 64^{32} ways. 64 to the power 32 is a number 57 digits long, so length 32 is probably overkill in terms of most people's privacy.

Notice that these strings of Base64 characters are referred to as pseudo-keys in the up-front glossary and frequently throughout this book. That's because they are used to stand in on behalf of much longer keys. Each byte in a pseudo-key maps to a 48 byte permutation within a key extender. Information on the design, creation, and use of key extenders follows later in this chapter.

9.3 Symmetric keys versus asymmetric keys

A key is symmetric if both the person encrypting and the person who decrypt the same content must have access to the same key.

Asymmetric keys have moved to the forefront since the 1970s. The letter 'a' at the front of a word often means "not". Asymmetric keys do *not* require access to the same key. Instead, phrases like "public key / private key" and "Diffie - Hellman" or acronyms like AES or RSA all denote situations where the secret at one end does not have to be communicated to the person at the other end. In that sense, asymmetric keys are simpler. However, the mathematics and their underlying algorithms are often quite complex. Asymmetric key encryption tends therefore to be left to the experts; there's little opportunity for control by the little guy.

9.4 Why are symmetric keys a pain?

None of us in the human race are good at devising unpredictable, unique, random keys. And that's only the first step if one wants to use symmetric keys for encryption. Other tasks: Record it in a way that others cannot get the key, type it

correctly when encrypting something, ensure that the key is not re-used. transmit it securely and in a timely way to (and only to) the intended recipient, who must in turn receive it securely, and use it securely. At an appropriate time, both parties need to dispose of the key, again securely.

This is called the symmetric key management problem. There is simply too much opportunity for human error. The cybersecurity community has little confidence that the little guy will use symmetric keys well.

Administration of symmetric keys require work. People are busy. They want a result, privacy, not a task of encryption and key management.

9.5 Goal: No symmetric keys in sight or mind

Here is a goal and a promise: By the end of the next chapter, we will have enough information to banish concern for any of the steps listed above, and to relieve ourselves entirely of any need whatsoever to think about symmetric keys.

Neither will *asymmetric* keys be in sight or on mind. There will not be any need whatsoever to think about keys -- period.

Does that mean there will be no keys? I didn't say that. Out of sight and out of mind does not exclude the possibility of invisible automated keys, below the surface, secure, unknown to you, unknown to adversaries, permanently.

9.6 How shall we import randomness?

Keys are a mechanism to import randomness for use in encryption and decryption.

Invisible automated keys are random in themselves. They support vastly stronger privacy if they are associated with key expanders, the technology introduced in the next section. People have a need to communicate privately, one on one, with assurance that unwanted persons do not intrude. The world population is 7.8 billion people. If everyone wanted to interact with everyone else privately, that would involve 7.8 billion times 7.8 billion communication pairs. That's not going to happen. There is a much better chance that a small subset, perhaps one billion people might like to exchange messages and files privately with five or ten others. Even that is a lot. That will call for boatloads of randomness,

It turns out that creating support for gadzillions of confidant relationships is not all that difficult.

9.7 Anatomy of a key expander

Key expanders are a brain child of U.S. Patent No. 10,505,715. [Thank you for that gracious round of applause.] Naming these blocks of computer bits is a challenge, particularly since we want to disavow the word "key". "Randomness" is too long a word to fit inside a name, and the word "entropy" is too puzzling to an outsider. Take comfort: By whatever name, key expanders will be almost as invisible as keys by the time we finish the next chapter.

A key expander is an array of permutations of integers 0 to (2^B - 1) followed by a few check bytes. 'B' is a bit basis, a count of bits that are swapped at a time. 'B' for much of this book is the value 6. Hence the focus here is on permutations of the integers 0 through 63.

The size of a permutation for bit basis 6 is 48 binary bytes. There are 64 integers, each of which fits in six bits. 64 X 6 = 384 bits = 48 bytes. When a permutation is in use, it is decompressed into 64 separate integers, but it's convenient in many ways to keep each in its more compressed 48 byte form for transmission and storage.

Let's arbitrarily decide to group permutations in batches of 64, and further, that there be 32 batches. With these numbers in mind, a key expander consists of 64 X 32 = 2048 permutations of the numbers 0 to 63, each occurring exactly once in each permutation. Taking into account the 48 byte compressed permutation size of 48 bytes, 2048 X 48 = 98,304 bytes. Add to that a one byte flag and four check bytes, and the size of a key expander is 98,309 bytes.

The flag and check bytes are set out in the program MakePKEs. The source code is available with this book. If you were to make your own implementation of *A Little Privacy*, you might wish to use different check bytes and/or add other material that you believe might be useful.

98,304 bytes of randomly generated parameters is a lot of randomness. Used well, the key expanders can strengthen privacy nicely.

9.8 Let's make one (or a thousand) key expanders

The program is named MakePKEs. The letter 'P' stands for Private. Here is the summary shown in response to "MakePKEs ?" at a command prompt:

```
Usage:  MakePKEs  IniPKEs.txt

MakePKEs generates Private Key Expander (PKE) files, each
98,309 bytes, in your current location. Input is a series
of random binary byte files, numbered upward sequentially
```

from the designated starter, as for example in Rand00053.dat (always ending in five digits and .dat). Do not reuse an input of random bytes that has been even partially used in a previous run of MakePKEs.

Outputs include the requested number of PKE files, a separate list of PKEs with their check sums, and (if errors encoutered) a report file.

Rarely used option: MakePKEs can output a single Shared Key Expander (SKE) in the form of a header file that may be built into the source code for an implementation of the Little Privacy program.

Operation of MakePKEs is guided by a self-documenting .INI style file.

9.9 Source code notes on the MakePKEs program

MakePKEs source code shared at the Marpx.com site is for the full operational program. It is geared to produce fully a thousand key expanders in a single pass. That in turn requires large quantities of random byte files, each four million bytes, and each with names ending in five digits followed by .dat, and consecutively numbered.

The SetUp function simply verifies that each argument is as expected and opens files. GetNewInput closes a random bytes file near its end and opens the next sequential random bytes file. As in most programs to date, LoadFileInRam puts entire files into RAM memory; be sure to delete the dynamic RAM allocated when finished.

DoCheckSum accumulates the sum of four arbitrary bytes in every one of the 2048 permutations in a nearly-finished key expander. The last byte of each of the four sums, together with a flag byte, is appended to the key expander. This data is used later on to ensure that the correct key expander has been used in each decryption process.

Most processing time is spent in the Output function. In program ALPriv04 we built single random permutations of integers 0 to 63. The Output function does the same, but 2048 of them at a time, each compressed into 48 bytes. Producing a thousand takes perhaps a minute or two of computer time, but the process is nicely automated.

Results of a run include a number of Private Key Expanders, a list of them with their check sums, and a report of random file inputs that were used. An optional output is one Shared Private Key Expander, suitable to drop into source code.

9.10 Shared versus private key expanders

Every implementation of the *Little Privacy* program includes a Shared Key Expander (SKE). It is built right into the source code through a header file created by the MakePKEs program. The SKE is needed in association with user ID information to decrypt one or more files. For example, the Do It Yourself version of the Marpx Privacy program requires a roster (list of confidants with their information) that must be kept secure from eavesdroppers.

The Shared Key Expander adds strength to an encryption that would depend otherwise on the randomness imported by the characters of the User ID along. It does not, however, add any unpredictability. The mapping of each individual character in the ID with a permutation in the SKE is quite open, at least to a person competent in reverse engineering. (We know that's illegal, but it is also reality.)

A private key expander (PKE) is equally effective with the SKE in adding strength to encryption. Its payoff: Only two persons in the world typically have the PKE in use -- the two confidants. The PKEs are routinely encrypted when not in use, and decrypted only in RAM temporarily. If the two confidants each protect their copy of their Private Key Expander, a hacker is *seriously* challenged to figure out the actual key. The hacker has one chance in a number 624 digits long of guessing the right key.

9.11 Let's make an invisible key

We treated the Base64 strings above as pseudo-keys, that is, as stand-ins for something much longer. To be exact, each character in the pseudo-key is replaced by a 48 byte permutation of the integers 0 through 63. In that sense, the pseudo-keys are shorthand.

Convert the first character in a pseudo-key to a numeric value, where '0' to '9' are values 0 to 9, 'A' to 'Z' are values 10 to 35, 'a' to 'z' are values 36 to 61, period is value 62, and underscore is value 63. Suppose the character is 'n'. The value then is 49. Within the first 64 permutations in the SKE or PKE currently in use, select the 49th permutation. That becomes the first segment of the key.

Forgive the repetition, but it's important you get this. Convert the *second* character in a pseudo-key to a numeric value, where '0' to '9' are values 0 to 9, 'A' to 'Z' are values 10 to 35, 'a' to 'z' are values 36 to 61, period is value 62, and underscore is value 63. Suppose the character is 'B'. The value then is 11. Within the *second* 64 permutations in the SKE or PKE currently in use, select the 11th permutation. That becomes the second segment of the key.

Repeat, repeat, repeat, for as many characters as you wish to use from the pseudo-key. Suppose you want the utmost privacy and use all 32 characters. Your key is not 32 bytes, but 32 segments, each of which is 48 bytes. Your actual key is 32 times 48 bytes. Convert that to bits and your key is 32 times 48 times 8 equals 12,288 bits long.

9.12 Assignment

To follow.

CHAPTER 10 Farewell, keys [period]. Hello, simplicity.

10.1 Confidants

10.1 Confidants
10.2 DIY confidant codes
    Interface
10.3 Confidant codes embedded in licenses
    Interface
10.4 Encrypt a random file, what do you get?
    A random bytes file for each time period
        four million for a calendar quarter
    pseudo-key banks
10.5 The symmetric key management solution
    Automated invisible keys
    pseudo-key + key expander = key
    SKE level
    PKE level
10.6 Reminder: Non-commercial use only
10.7 Proof of concept program
    only 54,000 bytes
    upgrade
10.8 The purpose of confidant codes
    Invisible keys
    Keys that change every two seconds
    Keys that never repeat