Security Engineering

Lesson 6
Key Length and Key Management

Spring 2010
Dr. Marenglen Biba

Outline

Key Length

- 7.1 Symmetric Key Length
- 7.2 Public-Key Key Length
- 7.3 Comparing Symmetric and Public-Key Key Length
- 7.4 Birthday Attacks against One-Way Hash Functions
- 7.5 How Long Should a Key Be?

Security Strength

- The security of a symmetric cryptosystem is a function of two things:
  - the strength of the algorithm and
  - the length of the key.
- The former is more important, but the latter is easier to demonstrate.
- Assume that the strength of the algorithm is perfect.
- By perfect, we mean that there is no better way to break the cryptosystem other than trying every possible key in a brute-force attack.

Calculating complexity

- Calculating the complexity of a brute-force attack is easy.
- If the key is 8 bits long, there are 2^8, or 256, possible keys.
  - Therefore, it will take 256 attempts to find the correct key, with a 50 percent chance of finding the key after half of the attempts.
- If the key is 56 bits long, then there are 2^{56} possible keys. Assuming a supercomputer can try a million keys a second, it will take 2285 years to find the correct key.
- If the key is 64 bits long, then it will take the same supercomputer about 585,000 years to find the correct key among the 2^{64} possible keys.
- If the key is 128 bits long, it will take 1025 years. The universe is only 1010 years old, so 1025 years is a long time. With a 2048-bit key, a million million-attempts-per-second computers working in parallel will spend 10597 years finding the key.
- By that time the universe will have long collapsed or expanded into nothingness ☺

Key importance

- The security of a cryptosystem should rest in the key, not in the details of the algorithm.
  - Assume that any cryptanalyst has access to all the details of your algorithm.
  - Assume he has access to as much ciphertext as he wants and can mount an intensive ciphertext-only attack.
  - Assume that he can mount a plaintext attack with as much data as he needs.
  - Even assume that he can mount a chosen-plaintext attack.
- If your cryptosystem can remain secure, even in the face of all that knowledge, then you've got something.

Time and Cost Estimates for Brute-Force Attack

- Two parameters determine the speed of a brute-force attack: the number of keys to be tested and the speed of each test.
  - Most symmetric algorithms accept any fixed-length bit pattern as the key. DES has a 56-bit key; it has 2^{56} possible keys.
  - Some algorithms have a 64-bit key; these have 2^{64} possible keys. Others have a 128-bit key.
- The speed at which each possible key can be tested is also a factor, but a less important one.
- For the purposes of our analysis, we will assume that each different algorithm can be tested in the same amount of time.
- The reality may be that one algorithm may be tested two, three, or even ten times faster than another.
- But since we are looking for key lengths that are millions of times more difficult to crack than would be feasible, small differences due to test speed are irrelevant.

Cracking DES

- Most of the debate in the cryptologic community about the efficiency of brute-force attacks has centered on the DES algorithm.
- In 1977, Whitfield Diffie and Martin Hellman postulated the existence of a special-purpose DES-cracking machine.
- This machine consisted of a million chips, each capable of testing a million keys per second. Such a machine could test 2^{56} keys in 20 hours.
- If built to attack an algorithm with a 64-bit key, it could test all 2^{64} keys in 214 days

Parallel processors for brute-force attacks

- A brute-force attack is tailor-made for parallel processors.
- Each processor can test a subset of the keyspace.
- The processors do not have to communicate among themselves; the only communication required at all is a single message signifying success.
- There are no shared memory requirements.
- It is easy to design a machine with a million parallel processors, each working independent of the others.

Price for cracking

- More recently, Michael Wiener decided to design a brute-force cracking machine. (He designed the machine for DES, but the analysis holds for most any algorithm.)
- He designed specialized chips, boards, and racks. He estimated prices. And he discovered that for $1 million, someone could build a machine that could crack a 56-bit DES key in an average of 3.5 hours (results guaranteed in 7 hours).
- The price/speed ratio is linear.
- Remember Moore's Law: Computing power doubles approximately every 18 months.
- Pipelined computers might do even better

Price for cracking

Average Time Estimates for a Hardware Brute-Force Attack in 1995

Length of Key in Bits						
Cost	**40**	**56**	**64**	**80**	**112**	**128**
$100 K	2 seconds	35 hours	1 year	70,000 years	10^{14} years	10^{19} years
$1 M	.2 seconds	3.5 hours	37 days	7000 years	10^{13} years	10^{18} years
$10 M	.02 seconds	21 minutes	4 days	700 years	10^{12} years	10^{17} years
$100 M	2 milliseconds	2 minutes	9 hours	70 years	10^{11} years	10^{16} years
$1 G	.2 milliseconds	13 seconds	1 hour	7 years	10^{10} years	10^{15} years
$10 G	.02 milliseconds	1 second	5.4 minutes	245 days	10^{9} years	10^{14} years
$100 G	2 microseconds	.1 second	32 seconds	24 days	10^{8} years	10^{13} years
$1 T	.2 microseconds	.01 second	3 seconds	2.4 days	10^{7} years	10^{12} years
$10 T	.02 microseconds	1 millisecond	.3 second	6 hours	10^{6} years	10^{11} years

Software Crackers

- Without special-purpose hardware and massively parallel machines, brute-force attacks are significantly harder.
- A software attack is about a thousand times slower than a hardware attack.
- The real threat of a software-based brute-force attack is not that it is certain, but that it is "free."
- It costs nothing to set up a microcomputer to test possible keys whenever it is idle.
  - If it finds the correct key—great.
  - If it doesn't, then nothing is lost.
- It costs nothing to set up an entire microcomputer network to do that. A recent experiment with DES used the collective idle time of 40 workstations to test 2^{34} keys in a single day.
- At this speed, it will take four million days to test all keys, but if enough people try attacks like this, then someone somewhere will get lucky.

Viruses

- The greatest difficulty in getting millions of computers to work on a brute-force attack is convincing millions of computer owners to participate.
  - You could ask politely, but that's time-consuming and they might say no.
  - You could try breaking into their machines, but that's even more time-consuming and you might get arrested.
  - You could also use a computer virus to spread the cracking program more efficiently over as many computers as possible. This is a particularly insidious idea.
- The attacker writes and lets loose a computer virus. This virus doesn't reformat the hard drive or delete files; it works on a brute-force cryptanalysis problem whenever the computer is idle.
- Various studies have shown that microcomputers are idle between 70 percent and 90 percent of the time, so the virus shouldn't have any trouble finding time to work on its task.

The Chinese Lottery

- The Chinese Lottery is a suggestion for a massively parallel cryptanalysis machine.
- Imagine that a brute-force, million-test-per-second cracking chip was built into every radio and television sold.
- Each chip is programmed to test a different set of keys automatically upon receiving a plaintext/ciphertext pair over the airwaves.
- Every time the Chinese government wants to break a key, it broadcasts the data. All the radios and televisions in the country start chugging away.
- Eventually, the correct key will appear on someone's display, somewhere in the country. The Chinese government pays a prize to that person; this makes sure that the result is reported promptly and properly, and also helps the sale of radios and televisions with the cracking chips.

Brute-force with Chinese Lottery

Brute-Force Cracking Estimates for Chinese Lottery

Country	Population	# of Televisions/Radios	Time to Break	
			56-bit	64-bit
China	1,190,431,000	257,000,000	280 seconds	20 hours
U.S.	260,714,000	739,000,000	97 seconds	6.9 hours
Iraq	19,890,000	4,730,000	4.2 hours	44 days
Israel	5,051,000	3,640,000	5.5 hours	58 days
Wyoming	470,000	1,330,000	15 hours	160 days
Winnemucca, NV	6,100	17,300	48 days	34 years

Outline

Key Length

- 7.1 Symmetric Key Length
- 7.2 Public-Key Key Length
- 7.3 Comparing Symmetric and Public-Key Key Length
- 7.4 Birthday Attacks against One-Way Hash Functions
- 7.5 How Long Should a Key Be?

Prime Numbers

- A **prime** number is an integer greater than 1 whose only factors are 1 and itself: No other number evenly divides it. Two is a prime number. So are 73, 2521, 2365347734339, and 2756839 - 1. There are an infinite number of primes.
- Cryptography, especially public-key cryptography, uses large primes.

Prime numbers

- Multiplying two large primes is a one-way function; it's easy to multiply the numbers to get a product but hard to factor the product and recover the two large primes.
- Public-key cryptography uses this idea and today's dominant public-key encryption algorithms are based on the difficulty of factoring large numbers that are the product of two large primes.
- These algorithms are also susceptible to a brute-force attack, but of a different type.
- Breaking these algorithms does not involve trying every possible key; but involves trying to factor the large number.
- If the number is too small, you have no security.
- If the number is large enough, you have security against all the computing power in the world working from now until the sun goes nova—given today's understanding of the mathematics.

Factoring

- Factoring a number means finding its prime factors.
  - 10 = 2*5
  - 60 = 2*2*3*5
  - 252601 = 41*61*101
  - 2113 - 1 = 3391*23279*65993*1868569*1066818132868207
- The factoring problem is one of the oldest in number theory.
- It's simple to factor a number, but it's time-consuming. This is still true, but there have been some major advances in the state of the art.

Factoring Algorithms

- **Number field sieve (NFS)** The *general number field sieve* is the fastest-known factoring algorithm for numbers larger than 110 digits or so.
  - An early version was used to factor the ninth Fermat number: $2^{512} + 1$.

- Factoring is attempting to use computer networks.
  - In factoring a 116-digit number, Arjen Lenstra and Mark Manasse used 400 mips-years — the spare time on an array of computers around the world for a few months.

Computing Power

- Computing power is generally measured in mips-years: a one-million-instruction-per-second (mips) computer running for one year, or about $3*10^{13}$ instructions.
- By convention, a 1-mips machine is equivalent to the DEC VAX 11/780. Hence, a mips-year is a VAX 11/780 running for a year, or the equivalent. (A 100 MHz Pentium is about a 50 mips machine; a 1800-node Intel Paragon is about 50,000.)

- In 1977 Ron Rivest said that factoring a 125-digit number would take 40 quadrillion years. In 1994 a 129-digit number was factored!!!
  - If there is any lesson in all this, it is that making predictions is foolish.

Trend

- The 1983 factorization of a 71-digit number required 0.1 mips-years; the 1994 factorization of a 129-digit number required 5000.
- This dramatic increase in computing power resulted largely from the introduction of distributed computing, using the idle time on a network of workstations.
- This trend was started by Bob Silverman and fully developed by Arjen Lenstra and Mark Manasse.
- The 1983 factorization used 9.5 CPU hours on a single Cray X-MP; the 1994 factorization took 5000 mips-years and used the idle time on 1600 computers around the world for about eight months.
- Modern factoring methods lend themselves to this kind of distributed implementation.

Trend

Factoring Using the Quadratic Sieve		
Year	**# of decimal digits factored**	**How many times harder to factor a 512-bit number**
1983	71	>20 million
1985	80	>2 million
1988	90	250,000
1989	100	30,000
1993	120	500
1994	129	100

Factoring effort

Factoring Using the General Number Field Sieve	
# of bits	**Mips-years required to factor**
512	30,000
768	$2*10^{8}$
1024	$3*10^{11}$
1280	$1*10^{14}$
1536	$3*10^{16}$
2048	$3*10^{20}$

- Today you need a 1024-bit number to get the level of security you got from a 512-bit number in the early 1980s. If you want your keys to remain secure for 20 years, 1024 bits is likely too short.

Factoring today

- When the numbers are very large, no efficient integer factorization algorithm is publicly known;

- An effort concluded in 2009 by several researchers factored a 232-digit number (**RSA-768**) utilizing hundreds of machines over a span of 2 years.

RSA-768

- RSA-768 has 232 decimal digits and has been factored on December 12, 2009 by Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Pierrick Gaudry, Alexander Kruppa, Peter Montgomery, Joppe W. Bos, Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmermann.

- RSA-768 = 123018668453011775513049495838496272077285356959533479219732245215172640050726365751874520219978646938995647494277406384592519255732630345373154826850791702612214291346167042921431160222124047927473779408066535141959745985 6902143413

- RSA-768 = 33478071698956898786044169848212690817704794983713768568912431388982883793 878002287614711652531743087737814467999489 × 36746043666799590428244633799627952632279158164343087642676032283815739666 5112792333734171433968102700927987363089 17

Recommended Public-key Key Lengths

- Assume a dedicated cryptanalyst can get his hands on 10,000 mips-years, a large corporation can get 10^7 mips-years, and that a large government can get 10^9 mips-years.
- Also assume that computing power will increase by a factor of 10 every five years.
- And finally, assume that advances in factoring mathematics allow us to factor general numbers at the speeds of the special number field sieve. (This isn't possible yet, but the breakthrough could occur at any time.)
- What is the key length appropriate for security during different years?

Recommended Public-key Key Lengths

Recommended Public-key Key Lengths (in bits)			
Year	**vs. Individual**	**vs. Corporation**	**vs. Government**
1995	768	1280	1536
2000	1024	1280	1536
2005	1280	1536	2048
2010	1280	1536	2048
2015	1536	2048	2048

Prediction by Bruce Schneier in 1995.

Long-range Factoring Predictions

- Not everyone will agree with these recommendations. The NSA has mandated 512-bit to 1024-bit keys for their Digital Signature Standard (see Section 20.1)—far less than what Scheiner recommends for long-term security.
- Pretty Good Privacy has a maximum RSA key length of 2047 bits.
- Arjen Lenstra, the world's most successful factorer, refuses to make predictions past 10 years.

Long-range Factoring Predictions

Year	Key Length (in bits)
1995	1024
2005	2048
2015	4096
2025	8192
2035	16,384
2045	32,768

Long-range Factoring Predictions

- Ron Rivest's key-length recommendations, originally made in 1990.
- While his analysis looks fine on paper, recent history illustrates that surprises regularly happen.

Rivest's Optimistic Key-length Recommendations (in bits)

Year	Low	Average	High
1990	398	515	1289
1995	405	542	1399
2000	422	572	1512
2005	439	602	1628
2010	455	631	1754
2015	472	661	1884
2020	489	677	2017

Key Length

- Why not use 10,000-bit keys?

- You can, but remember that you pay a price in computation time as your keys get longer.

- You want a key long enough to be secure, but short enough to be computationally usable.

Outline

Key Length

- 7.1 Symmetric Key Length
- 7.2 Public-Key Key Length
- 7.3 Comparing Symmetric and Public-Key Key Length
- 7.4 Birthday Attacks against One-Way Hash Functions
- 7.5 How Long Should a Key Be?

Comparing Symmetric and Public-Key Key Length

- A system is going to be attacked at its weakest point.

- If you are designing a system that uses both symmetric and public-key cryptography, the key lengths for each type of cryptography should be chosen so that it is equally difficult to attack the system via each mechanism.

- It makes no sense to use a symmetric algorithm with a 128-bit key together with a public-key algorithm with a 386-bit key, just as it makes no sense to use a symmetric algorithm with a 56-bit key together with a public-key algorithm with a 1024-bit key.

Comparing Symmetric and Public-Key Key Length

Symmetric and Public-key Key Lengths with Similar Resistances to Brute-Force Attacks

Symmetric Key Length	Public-key Key Length
56 bits	384 bits
64 bits	512 bits
80 bits	768 bits
112 bits	1792 bits
128 bits	2304 bits

Outline

Key Length

- 7.1 Symmetric Key Length
- 7.2 Public-Key Key Length
- 7.3 Comparing Symmetric and Public-Key Key Length
- 7.4 Birthday Attacks against One-Way Hash Functions
- 7.5 How Long Should a Key Be?

Brute-force attacks against a one-way hash function

- There are two brute-force attacks against a one-way hash function.
- The first is the most obvious: Given the hash of message, $H(M)$, an adversary would like to be able to create another document, M', such that $H(M) = H(M')$.
- The second attack is more subtle: An adversary would like to find two random messages, M, and M', such that $H(M) = H(M')$.
- This is called a **collision,** and it is a far easier attack than the first one.

Birthday paradox

- The birthday paradox is a standard statistics problem.
- How many people must be in a room for the chance that one of them shares your birthday?
  - The answer is 253.
- Now, how many people must there be for the chance that at least two of them will share the same birthday?
  - The answer is surprisingly low: 23. With only 23 people in the room, there are still 253 different *pairs* of people in the room.
- Finding someone with a specific birthday (yours) is analogous to the first attack;
- Finding two people with the same random birthday is analogous to the second attack.
- The second attack is commonly known as a **birthday attack**.

Birthday Attack

- Assume that a one-way hash function is secure and the best way to attack it is by using brute force.
- It produces an *m*-bit output. Finding a message that hashes to a given hash value would require hashing 2^m random messages.
- Finding two messages that hash to the same value would only require hashing $2^{m/2}$ random messages.
- A machine that hashes a million messages per second would take 600,000 years to find a second message that matched a given 64-bit hash.
  - The same machine could find a pair of messages that hashed to the same value in about an hour.
- This means that if you are worried about a birthday attack, you should choose a hash-value twice as long as you otherwise might think you need. For example, if you want to drop the odds of someone breaking your system to less than 1 in 280, use a 160-bit one-way hash function.

Outline

Key Length

- 7.1 Symmetric Key Length
- 7.2 Public-Key Key Length
- 7.3 Comparing Symmetric and Public-Key Key Length
- 7.4 Birthday Attacks against One-Way Hash Functions
- 7.5 How Long Should a Key Be?

How Long Should a Key Be?

- There's no single answer to this question; it depends on the situation.
- To determine how much security you need, you must ask yourself some questions.
  - How much is your data worth?
  - How long does it need to be secure?
  - What are your adversaries' resources?
- A customer list might be worth $1000. Financial data for an acrimonious divorce case might be worth $10,000. Advertising and marketing data for a large corporation might be worth $1 million. The master keys for a digital cash system might be worth billions.
- In the world of commodities trading, secrets only need to be kept for minutes. In the newspaper business, today's secrets are tomorrow's headlines. Product development information might need to remain secret for a year or two. U.S. Census data are required by law to remain secret for 100 years.

How Long Should a Key Be?

- You can even specify security requirements in these terms. For example:

- The key length must be such that there is a probability of no more than 1 in 232 that an attacker with $100 million to spend could break the system within one year, even assuming technology advances at a rate of 30 percent per annum over the period.

How Long Should a Key Be?

- Future computing power is harder to estimate, but here is a reasonable rule of thumb: The efficiency of computing equipment divided by price doubles every 18 months and increases by a factor of 10 every five years.
- Thus, in 50 years the fastest computers will be 10 billion times faster than today's! Remember, too, that these numbers only relate to general-purpose computers; who knows what kind of specialized cryptosystem-breaking equipment will be developed in the next 50 years?
- Assuming that a cryptographic algorithm will be in use for 30 years, you can get some idea how secure it must be.
  - An algorithm designed today probably will not see general use until 2000, and will still be used in 2025 to encrypt messages that must remain secret until 2075 or later.

Security Requirements

Security Requirements for Different Information

Type of Traffic	Lifetime	Minimum Key Length
Tactical military information	minutes/hours	56–64 bits
Product announcements, mergers, interest rates	days/weeks	64 bits
Long-term business plans	years	64 bits
Trade secrets (e.g., recipe for Coca-Cola)	decades	112 bits
H-bomb secrets	>40 years	128 bits
Identities of spies	>50 years	128 bits
Personal affairs	>50 years	128 bits
Diplomatic embarrassments	>65 years	at least 128 bits
U.S. census data	100 years	at least 128 bits

RSA predictions

- As of 2003 RSA Security claims that 1024-bit RSA keys are equivalent in strength to 80-bit symmetric keys, 2048-bit RSA keys to 112-bit symmetric keys and 3072-bit RSA keys to 128-bit symmetric keys.
- RSA claims that 1024-bit keys are likely to become crackable some time between 2006 and 2010 and that 2048-bit keys are sufficient until 2030. An RSA key length of 3072 bits should be used if security is required beyond 2030.
- NIST key management guidelines further suggest that 15360-bit RSA keys are equivalent in strength to 256-bit symmetric keys.

Break

RSA Factoring Challenge

RSA Number	Decimal digits	Binary digits	Cash prize offered	Factored on	Factored by
RSA-100	100	330	$1,000 USD	April 1, 1991	Arjen K. Lenstra
RSA-110	110	364	$4,429 USD	April 14, 1992	Arjen K. Lenstra and M.S. Manasse
RSA-120	120	397	$5,898	June 9, 1993	T. Denny et al.
RSA-129	129	426	$100 USD[4]	April 26, 1994	Arjen K. Lenstra et al.
RSA-130	130	430	$14,527 USD	April 10, 1996	Arjen K. Lenstra et al.
RSA-140	140	463	$17,226 USD	February 2, 1999	Herman te Riele et al.
RSA-150[5]	150	496		April 16, 2004	Kazumaro Aoki et al.
RSA-155	155	512	$9,383	August 22, 1999	Herman te Riele et al.
RSA-160	160	530		April 1, 2003	Jens Franke et al., University of Bonn
RSA-170	170	563		December 29, 2009	D. Bonenberger and M. Krone
RSA-576	174	576	$10,000 USD	December 3, 2003	Jens Franke et al., University of Bonn
RSA-180	180	596		*inactive*	
RSA-190	190	629		*inactive*	
RSA-640	193	640	$20,000 USD	November 2, 2005	Jens Franke et al., University of Bonn
RSA-200	200	663		May 9, 2005	Jens Franke et al., University of Bonn
RSA-210	210	696		*inactive*	
RSA-704	212	704	$30,000 USD	*inactive, prize retracted*	
RSA-220	220	729		*inactive*	
RSA-230	230	762		*inactive*	
RSA-232	232	768		*inactive*	
RSA-768	232	768	$50,000 USD	December 12, 2009	Thorsten Kleinjung et.al (*prize retracted*)

Break: GSM cracking

- http://www.thetechherald.com/article.php/200953/5009/GSM-cracking-project-moves-forward
- http://www.cs.virginia.edu/~kn5f/

Key Management

Outline

Key Management

- 8.1 Generating Keys
- 8.2 Nonlinear Keyspaces
- 8.3 Transferring Keys
- 8.4 Verifying Keys
- 8.5 Using Keys
- 8.6 Updating Keys
- 8.7 Storing Keys
- 8.8 Backup Keys
- 8.9 Compromised Keys
- 8.10 Lifetime of Keys
- 8.11 Destroying Keys
- 8.12 Public-Key Key Management

Real-world key management

- In the real world, key management is the hardest part of cryptography.
- Designing secure cryptographic algorithms and protocols isn't easy, but you can rely on a large body of academic research. Keeping the keys secret is much harder.
- Cryptanalysts often attack both symmetric and public-key cryptosystems through their key management.
- Why should Eve bother going through all the trouble of trying to break the cryptographic algorithm if she can recover the key because of sloppy key storage procedures? Why should she spend $10 million building a cryptanalysis machine if she can spend $1000 bribing a clerk?

Example of key management

- For example, the DiskLock program for Macintosh, sold at most software stores, claims the security of DES encryption.
- It encrypts files using DES. Its implementation of the DES algorithm is correct.
- However, DiskLock stores the DES key with the encrypted file.
- If you know where to look for the key, and want to read a file encrypted with DiskLock's DES, recover the key from the encrypted file and then decrypt the file.
- It doesn't matter that this program uses DES encryption—the implementation is completely insecure.

Outline

Key Management

- 8.1 Generating Keys
- 8.2 Nonlinear Keyspaces
- 8.3 Transferring Keys
- 8.4 Verifying Keys
- 8.5 Using Keys
- 8.6 Updating Keys
- 8.7 Storing Keys
- 8.8 Backup Keys
- 8.9 Compromised Keys
- 8.10 Lifetime of Keys
- 8.11 Destroying Keys
- 8.12 Public-Key Key Management

Reduced Keyspaces

- DES has a 56-bit key. Implemented properly, any 56-bit string can be the key; there are 2^{56} (10^{16}) possible keys.
- Norton Discreet for MS-DOS (versions 8.0 and earlier) only allows ASCII keys, forcing the high-order bit of each byte to be zero.
- The program also converts lowercase letters to uppercase (so the fifth bit of each byte is always the opposite of the sixth bit) and ignores the low-order bit of each byte, resulting in only 2^{40} possible keys.
- These poor key generation procedures have made its DES ten thousand times easier to break than a proper implementation.

Number of possible keys

Number of Possible Keys of Various Keyspaces

	4-Byte	5-Byte	6-Byte	7-Byte	8-Byte
Lowercase letters (26):	460,000	$1.2*10^7$	$3.1*10^8$	$8.0*10^9$	$2.1*10^{11}$
Lowercase letters and digits (36):	1,700,000	$6.0*10^7$	$2.2*10^9$	$7.8*10^{10}$	$2.8*10^{12}$
Alphanumeric characters (62):	$1.5*10^7$	$9.2*10^8$	$5.7*10^{10}$	$3.5*10^{12}$	$2.2*10^{14}$
Printable characters (95):	$8.1*10^7$	$7.7*10^9$	$7.4*10^{11}$	$7.0*10^{13}$	$6.6*10^{15}$
ASCII characters (128):	$2.7*10^8$	$3.4*10^{10}$	$4.4*10^{12}$	$5.6*10^{14}$	$7.2*10^{16}$
8-bit ASCII characters (256):	$4.3*10^9$	$1.1*10^{12}$	$2.8*10^{14}$	$7.2*10^{16}$	$1.8*10^{19}$

- Number of possible keys with various constraints on the input strings

Search Time

Exhaustive Search of Various Keyspaces (assume one million attempts per second)

	4-Byte	5-Byte	6-Byte	7-Byte	8-Byte
Lowercase letters (26):	.5 seconds	12 seconds	5 minutes	2.2 hours	2.4 days
Lowercase letters and digits (36):	1.7 seconds	1 minute	36 minutes	22 hours	33 days
Alphanumeric characters (62):	15 seconds	15 minutes	16 hours	41 days	6.9 years
Printable characters (95):	1.4 minutes	2.1 hours	8.5 days	2.2 years	210 years
ASCII characters (128):	4.5 minutes	9.5 hours	51 days	18 years	2300 years
8-bit ASCII characters (256):	1.2 hours	13 days	8.9 years	2300 years	580,000 years

Poor Key Choices

- When people choose their own keys, they generally choose poor ones.
- They're far more likely to choose "Barney" than "*9 (hH/A." This is not always due to poor security practices; "Barney" is easier to remember than "*9 (hH/A."
- The world's most secure algorithm won't help much if the users habitually choose their spouse's names for keys or write their keys on little pieces of paper in their wallets.
- A smart brute-force attack doesn't try all possible keys in numerical order; it tries the obvious keys first. (**Dictionary attack)**

Random Keys

- Good keys are random-bit strings generated by some automatic process. If the key is 64 bits long, every possible 64-bit key must be equally likely.
- Generate the key bits from either a reliably random source or a cryptographically secure pseudo-random-bit generator
- If these automatic processes are unavailable, flip a coin or roll a die.

Weak Keys

- Some encryption algorithms have weak keys: specific keys that are less secure than the other keys.
- DES has only 16 weak keys out of 2^{56}, so the odds of generating any of these keys are incredibly small.
- It has been argued that a cryptanalyst would have no idea that a weak key is being used and therefore gains no advantage from their accidental use.
- It has also been argued that not using weak keys gives a cryptanalyst information.
- However, testing for the few weak keys is so easy that it seems imprudent not to do so.

Generating keys

- Generating keys for public-key cryptography systems is harder, because often the keys must have certain mathematical properties (they may have to be prime, be a quadratic residue, etc.).
- However, in Mathematics there are techniques for generating large random prime numbers.
- The important thing to remember from a key management point of view is that the *random seeds* for those generators must be just that: *random*.

Remembering the Key

- Generating a random key isn't always possible. Sometimes you need to remember your key. (See how long it takes you to remember 25e8 56f2 e8ba c820).
- If you have to generate an easy-to-remember key, make it obscure. The ideal would be something easy to remember, but difficult to guess. Here are some suggestions:
  - Word pairs separated by a punctuation character, for example "turtle*moose" or "zorch!splat"
  - Strings of letters that are an acronym of a longer phrase; for example, "Mein Luftkissenfahrzeug ist voller Aale!" generates the key "MLivA!"

Pass Phrases

- A better solution is to use an entire phrase instead of a word, and to convert that phrase into a key. These phrases are called **pass phrases**.
- A technique called **key crunching** converts the easy-to-remember phrases into random keys.
  - Use a one-way hash function to transform an arbitrary-length text string into a pseudo-random-bit string.
- For example, the easy-to-remember text string:

  My name is Ozymandias, king of kings. Look on my works, ye mighty, and despair.

might crunch into this 64-bit key:

e6c1 4398 5ae9 0a9b

X9.17 Key Generation

- The ANSI X9.17 standard specifies a method of key generation.
  - http://www.msen.com/fievel/mmill/X9.17.html
- This does not generate easy-to-remember keys; it is more suitable for generating session keys or pseudo-random numbers within a system.
- The cryptographic algorithm used to generate keys is triple-DES, but it could just as easily be any algorithm.
- Let $E_K(X)$ be triple-DES encryption of X with key K. This is a special key reserved for secret key generation. V_0 is a secret 64-bit seed. T is a timestamp. To generate the random key R_i, calculate:
  - $R_i = E_K(E_K(T_i)\ (+)\ V_i)$
- To generate V_{i+1}, calculate:
  - $V_{i+1} = E_K(E_K(T_i)\ (+)\ R_i)$
- To turn R_i into a DES key, simply adjust every eighth bit for parity.
- If you need a 64-bit key, use it as is. If you need a 128-bit key, generate a pair of keys and concatenate them together.

Outline

Key Management

- 8.1 Generating Keys
- 8.2 Nonlinear Keyspaces
- 8.3 Transferring Keys
- 8.4 Verifying Keys
- 8.5 Using Keys
- 8.6 Updating Keys
- 8.7 Storing Keys
- 8.8 Backup Keys
- 8.9 Compromised Keys
- 8.10 Lifetime of Keys
- 8.11 Destroying Keys
- 8.12 Public-Key Key Management

Nonlinear Keyspaces

- Imagine that you are a military cryptography organization, building a piece of cryptography equipment for your troops.
- You want to use a secure algorithm, but you are worried about the equipment falling into enemy hands. The last thing you want is for your enemy to be able to use the equipment to protect *their* secrets.
- If you can put your algorithm in a tamperproof module, here's what you can do:
  - You can require keys of a special and secret form; all other keys will cause the module to encrypt and decrypt using a severely weakened algorithm.
  - You can make it so that the odds of someone, not knowing this special form but accidentally stumbling on a correct key, are vanishingly small.
- This is called a *nonlinear keyspace*, because all the keys are not equally strong: some are weak and some are strong!!!

Outline

Key Management

- 8.1 Generating Keys
- 8.2 Nonlinear Keyspaces
- 8.3 Transferring Keys
- 8.4 Verifying Keys
- 8.5 Using Keys
- 8.6 Updating Keys
- 8.7 Storing Keys
- 8.8 Backup Keys
- 8.9 Compromised Keys
- 8.10 Lifetime of Keys
- 8.11 Destroying Keys
- 8.12 Public-Key Key Management

Transfering the key

- Alice and Bob are going to use a symmetric cryptographic algorithm to communicate securely; they need the same key.
- Alice generates a key using a random-key generator.
- Now she has to give it to Bob—securely. If Alice can meet Bob somewhere (a back alley, a windowless room, etc), she can give him a copy of the key. Otherwise, they have a problem.
- Public-key cryptography solves the problem nicely and with a minimum of prearrangement, but these techniques are not always available

Key-Encryption Keys

- The X9.17 standard specifies two types of keys: key-encryption keys and data keys. **Key-Encryption** Keys encrypt other keys for distribution.

- **Data Keys** encrypt message traffic.

- These key-encrypting keys have to be distributed manually (although they can be secured in a tamperproof device, like a smart card), but only seldomly.

- Data keys are distributed more often.

Outline

Key Management

- 8.1 Generating Keys
- 8.2 Nonlinear Keyspaces
- 8.3 Transferring Keys
- 8.4 Verifying Keys
- 8.5 Using Keys
- 8.6 Updating Keys
- 8.7 Storing Keys
- 8.8 Backup Keys
- 8.9 Compromised Keys
- 8.10 Lifetime of Keys
- 8.11 Destroying Keys
- 8.12 Public-Key Key Management

Verifying Keys

- When Bob receives a key, how does he know it came from Alice and not from someone pretending to be Alice?
- If Alice gives it to him when they are face-to-face, it's easy.
- If Alice sends her key via a trusted courier, then Bob has to trust the courier.
- If the key is encrypted with a key-encryption key, then Bob has to trust the fact that only Alice has that key.
- If Alice uses a digital signature protocol to sign the key, Bob has to trust the public-key database when he verifies that signature. (He also has to trust that Alice has kept her key secure.)
- If a *Key Distribution Center (KDC)* signs Alice's public key, Bob has to trust that his copy of the KDC's public key has not been tampered with.

Verifying Keys

- Some people have used this argument to claim that public-key cryptography is useless. Since the only way for Alice and Bob to ensure that their keys have not been tampered with is to meet face-to-face, public-key cryptography doesn't enhance security at all.
  - This view is naïve. It is theoretically true, but reality is far more complicated.
- Public-key cryptography, used with digital signatures and trusted KDCs, makes it much more difficult to substitute one key for another.
- Bob can never be absolutely certain that Mallory isn't controlling his entire reality, but Bob can be confident that doing so requires more resources than most real-world Mallorys have access to.

Error Detection during Key Transmission

- Sometimes keys get garbled in transmission. Since a garbled key can mean megabytes of undecryptable ciphertext, this is a problem.
- All keys should be transmitted with some kind of error detection and correction bits. This way errors in transmission can be easily detected and, if required, the key can be resent.
- One of the most widely used methods is to encrypt a constant value with the key, and to send the first 2 to 4 bytes of that ciphertext along with the key.
  - At the receiving end, do the same thing. If the encrypted constants match, then the key has been transmitted without error. The chance of an undetected error ranges from one in 216 to one in 232.

Outline

Key Management

- 8.1 Generating Keys
- 8.2 Nonlinear Keyspaces
- 8.3 Transferring Keys
- 8.4 Verifying Keys
- 8.5 Using Keys
- 8.6 Updating Keys
- 8.7 Storing Keys
- 8.8 Backup Keys
- 8.9 Compromised Keys
- 8.10 Lifetime of Keys
- 8.11 Destroying Keys
- 8.12 Public-Key Key Management

Software Encryption

- You can't tell when the operating system will suspend the encryption application in progress, write everything to disk, and take care of some pressing task.
  - When the operating system finally gets back to encrypting whatever is being encrypted, everything will look just fine.
- No one will ever realize that the operating system wrote the encryption application to disk, and that it wrote the key along with it.
- The key will sit on the disk, unencrypted, until the computer writes over that area of memory again. It could be minutes or it could be months. It could even be never; the key could still be sitting there when an adversary goes over the hard drive with a fine-tooth comb.
- In a preemptive, multitasking environment, you can set your encryption operation to a high enough priority so it will not be interrupted.
  - This would mitigate the risk. Even so, the whole thing is not perfectly secure.

Hardware implementations and session keys

- Hardware implementations are safer. Many encryption devices are designed to erase the key if tampered with. For example, the IBM PS/2 encryption card has a unit containing the DES chip, battery, and memory. Of course, you have to trust the hardware manufacturer to implement the feature properly.

- Some communications applications, such as telephone encryptors, can use **session keys**. A session key is a key that is just used for one communications session—a single telephone conversation—and then discarded. There is no reason to store the key after it has been used.
  - And if you use some key-exchange protocol to transfer the key from one conversant to the other, the key doesn't have to be stored before it is used either. This makes it far less likely that the key might be compromised.

Outline

Key Management

- 8.1 Generating Keys
- 8.2 Nonlinear Keyspaces
- 8.3 Transferring Keys
- 8.4 Verifying Keys
- 8.5 Using Keys
- 8.6 Updating Keys
- 8.7 Storing Keys
- 8.8 Backup Keys
- 8.9 Compromised Keys
- 8.10 Lifetime of Keys
- 8.11 Destroying Keys
- 8.12 Public-Key Key Management

Key updating

- Sometimes it's a pain to distribute a new key every day. An easier solution is to generate a new key from the old key; this is sometimes called *key updating*.
- All it takes is a one-way function.
- If Alice and Bob share the same key and they both operate on it using the same one-way function, they will get the same result. Then they can take the bits they need from the results to create the new key.
- Key updating works, but remember that the new key is only as secure as the old key was.
  - If Eve managed to get her hands on the old key, she can perform the key updating function herself. However, if Eve doesn't have the old key and is trying a ciphertext-only attack on the encrypted traffic, this is a good way for Alice and Bob to protect themselves.

Outline

Key Management

- 8.1 Generating Keys
- 8.2 Nonlinear Keyspaces
- 8.3 Transferring Keys
- 8.4 Verifying Keys
- 8.5 Using Keys
- 8.6 Updating Keys
- 8.7 Storing Keys
- 8.8 Backup Keys
- 8.9 Compromised Keys
- 8.10 Lifetime of Keys
- 8.11 Destroying Keys
- 8.12 Public-Key Key Management

Single user case

- The least complex key storage problem is that of a single user, Alice, encrypting files for later use.
- Since she is the only person involved, she is the only person responsible for the key.
- Some systems take the easy approach: The key is stored in Alice's brain and never on the system.
- Alice is responsible for remembering the key and entering it every time she needs a file encrypted or decrypted.

Magnetic stripe card

- Another solution is to store the key in a magnetic stripe card, plastic key with an embedded ROM chip (called a *ROM key*), or smart card.
- A user could then enter his key into the system by inserting the physical token into a special reader in his encryption box or attached to his computer terminal.
- While the user can use the key, he does not know it and cannot compromise it. He can use it only in the way and for the purposes indicated by the control vector.

Magnetic stripe card

- This technique is made more secure by splitting the key into two halves, storing one half in the terminal and the other half in the ROM key.
- The U.S. government's STU-III secure telephone works this way.
- Losing the ROM key does not compromise the cryptographic key—change that key and everything is back to normal. The same is true with the loss of the terminal.
- This way, compromising either the ROM key or the system does not compromise the cryptographic key—an adversary must have both parts.

Outline

Key Management

- 8.1 Generating Keys
- 8.2 Nonlinear Keyspaces
- 8.3 Transferring Keys
- 8.4 Verifying Keys
- 8.5 Using Keys
- 8.6 Updating Keys
- 8.7 Storing Keys
- 8.8 Backup Keys
- 8.9 Compromised Keys
- 8.10 Lifetime of Keys
- 8.11 Destroying Keys
- 8.12 Public-Key Key Management

Key escrow

- Alice is the chief financial officer at Secrets, Ltd.—"We don't tell you our motto." Like any good corporate officer, she follows the company's security guidelines and encrypts all her data.
  - Unfortunately, she ignores the company's street-crossing guidelines and gets hit by a truck. What does the company's president, Bob, do?
- Unless Alice left a copy of her key, he's in deep trouble. The whole point of encryption is to make files unrecoverable without the key.
- Bob can avoid this in several ways. The simplest is sometimes called **key escrow:**
  - He requires all employees to write their keys on paper and give them to the company's security officer, who will lock them in a safe somewhere (or encrypt them all with a master key).

Secret-sharing protocol

- The problem with this key management system is that Bob has to trust his security officer not to misuse everyone's keys.
- Even more significantly, all the employees have to trust the security officer not to misuse their keys.

- A far better solution is to use a secret-sharing protocol.
  - When Alice generates a key, she also divides up that key into some number of pieces.
  - She then sends each piece—encrypted, of course—to a different company officer.
  - None of those pieces alone is the key, but someone can gather all the pieces together and reconstruct the key.

Outline

Key Management

- 8.1 Generating Keys
- 8.2 Nonlinear Keyspaces
- 8.3 Transferring Keys
- 8.4 Verifying Keys
- 8.5 Using Keys
- 8.6 Updating Keys
- 8.7 Storing Keys
- 8.8 Backup Keys
- 8.9 Compromised Keys
- 8.10 Lifetime of Keys
- 8.11 Destroying Keys
- 8.12 Public-Key Key Management

Loosing the key

- All of the protocols, techniques, and algorithms in this course are secure only if the key (the private key in a public-key system) remains secret.
- If Alice's key is lost, stolen, printed in the newspaper, or otherwise compromised, then all her security is gone.
- If the compromised key was for a symmetric cryptosystem, Alice has to change her key and hope the actual damage was minimal.
- If it was a private key, she has bigger problems; her public key is probably on servers all over the network.
  - And if Eve gets access to Alice's private key, she can impersonate her on the network: reading encrypted mail, signing correspondence, entering into contracts, and so forth. Eve can, effectively, become Alice.

Loosing the key

- It is vital that news of a private key's compromise propagate quickly throughout the network.
  - Any databases of public keys must immediately be notified that a particular private key has been compromised.
- One hopes Alice knows when her key was compromised. If a KDC is managing the keys, Alice should notify it that her key has been compromised.
- If there is no KDC, then she should notify all correspondents who might receive messages from her.
- Someone should publicize the fact that any message received after her key was lost is suspect, and that no one should send messages to Alice with the associated public key.
- The application should be using some sort of timestamp, and then users can determine which messages are legitimate and which are suspect.

Outline

Key Management

- 8.1 Generating Keys
- 8.2 Nonlinear Keyspaces
- 8.3 Transferring Keys
- 8.4 Verifying Keys
- 8.5 Using Keys
- 8.6 Updating Keys
- 8.7 Storing Keys
- 8.8 Backup Keys
- 8.9 Compromised Keys
- 8.10 Lifetime of Keys
- 8.11 Destroying Keys
- 8.12 Public-Key Key Management

Lifetime of Keys

- No encryption key should be used for an indefinite period. It should expire automatically like passports and licenses. There are several reasons for this:
  - The longer a key is used, the greater the chance that it will be compromised. People write keys down; people lose them. Accidents happen. If you use the same key for a year, there's a far greater chance of compromise than if you use it for a day.
  - The longer a key is used, the greater the loss if the key is compromised. If a key is used only to encrypt a single budgetary document on a file server, then the loss of the key means only the compromise of that document. If the same key is used to encrypt all the budgetary information on the file server, then its loss is much more devastating.

Lifetime of Keys

- The longer a key is used, the greater the temptation for someone to spend the effort necessary to break it — even if that effort is a brute-force attack.
  - Breaking a key shared between two military units for a day would enable someone to read and fabricate messages between those units for that day.
  - Breaking a key shared by an entire military command structure for a year would enable that same person to read and fabricate messages throughout the world for a year.
- It is generally easier to do cryptanalysis with more ciphertext encrypted with the same key.

Lifetime of Keys

- Key-encryption keys don't have to be replaced as frequently. They are used only occasionally (roughly once per day) for key exchange.
  - This generates little ciphertext for a cryptanalyst to work with. However, if a key-encryption key is compromised, the potential loss is extreme
- Encryption keys used to encrypt data files for storage cannot be changed often. The files may sit encrypted on disk for months or years before someone needs them again. Decrypting them and re-encrypting them with a new key every day doesn't enhance security in any way; it just gives a cryptanalyst more to work with.
- Private keys for public-key cryptography applications have varying lifetimes, depending on the application.
  - Private keys used for digital signatures and proofs of identity may have to last years (even a lifetime).

Outline

Key Management

- 8.1 Generating Keys
- 8.2 Nonlinear Keyspaces
- 8.3 Transferring Keys
- 8.4 Verifying Keys
- 8.5 Using Keys
- 8.6 Updating Keys
- 8.7 Storing Keys
- 8.8 Backup Keys
- 8.9 Compromised Keys
- 8.10 Lifetime of Keys
- 8.11 Destroying Keys
- 8.12 Public-Key Key Management

Destroying keys

- Given that keys must be replaced regularly, old keys must be destroyed. Old keys are valuable, even if they are never used again. With them, an adversary can read old messages encrypted with those keys.
- Keys must be destroyed securely.
  - If the key is written on paper, the paper should be shredded or burned. Be careful to use a high-quality shredder; many lousy shredders are on the market.
- Algorithms in this course are secure against brute-force attacks costing millions of dollars and taking millions of years.
  - If an adversary can recover your key by taking a bag of shredded documents from your trash and paying 100 unemployed workers in some backwater country ten cents per hour for a year to piece the shredded pages together, that would be $26,000 well spent.

Destroying keys

- If the key is in a hardware EEPROM, the key should be overwritten multiple times. If the key is in a hardware EPROM or PROM, the chip should be smashed into tiny bits and scattered to the four winds.
- If the key is stored on a computer disk, the actual bits of the storage should be overwritten multiple times or the disk should be shredded.
- A potential problem is that, in a computer, keys can be easily copied and stored in multiple locations.
  - Any computer that does its own memory management, constantly swapping programs in and out of memory, exacerbates the problem.
  - There is no way to ensure that successful key erasure has taken place in the computer, especially if the computer's operating system controls the erasure process.

(The more paranoid person should consider writing a special erasure program that scans all disks looking for copies of the key's bit pattern on unused blocks and then erases those blocks. Also remember to erase the contents of any temporary, or "swap," files.)

Outline

Key Management

- 8.1 Generating Keys
- 8.2 Nonlinear Keyspaces
- 8.3 Transferring Keys
- 8.4 Verifying Keys
- 8.5 Using Keys
- 8.6 Updating Keys
- 8.7 Storing Keys
- 8.8 Backup Keys
- 8.9 Compromised Keys
- 8.10 Lifetime of Keys
- 8.11 Destroying Keys
- 8.12 Public-Key Key Management

Public-key key management

- Public-key cryptography makes key management easier, but it has its own unique problems.
- Each person has only one public key, regardless of the number of people on the network. If Alice wants to send a message to Bob, she has to get Bob's public key. She can go about this several ways:
  - She can get it from Bob.
  - She can get it from a centralized database.
  - She can get it from her own private database.
- There are a number of possible attacks against public-key cryptography, based on Mallory substituting his key for Bob's. The scenario is that Alice wants to send a message to Bob. She goes to the public-key database and gets Bob's public key. But Mallory, who is sneaky, has substituted his own key for Bob's.

Public-key Certificates

- A *public-key certificate* is someone's public key, signed by a trustworthy person. Certificates are used to defeat attempts to substitute one key for another.
- Bob's certificate, in the public-key database, contains a lot more than his public key. It contains information about Bob—his name, address, and so on—and it is signed by someone Alice trusts: Trent (usually known as a *certification authority*, or CA).
- By signing both the key and the information about Bob, Trent certifies that the information about Bob is correct and that the public key belongs to Bob.
  - Alice checks Trent's signature and then uses the public key, secure in the knowledge that it is Bob's and no one else's. Certificates play an important role in a number of public-key protocols such as PEM and X.509.

Distributed Key Management

- In some situations, this sort of centralized key management will not work. Perhaps there is no CA whom Alice and Bob both trust. Perhaps Alice and Bob trust only their friends. Perhaps Alice and Bob trust no one.
- Distributed key management, used in PGP, solves this problem with introducers.
  - Introducers are other users of the system who sign their friends' public keys.
- For example, when Bob generates his public key, he gives copies to his friends: Carol and Dave. They know Bob, so they each sign Bob's key and give Bob a copy of the signature.
- Now, when Bob presents his key to a stranger, Alice, he presents it with the signatures of these two introducers.
  - If Alice also knows and trusts Carol, she has reason to believe that Bob's key is valid.
  - If she knows and trusts Carol and Dave a little, she has reason to believe that Bob's key is valid.
  - If she doesn't know either Carol or Dave, she has no reason to trust Bob's key.

End of Lesson

- Readings
  - Applied Crypto: Chap. 7.1 and 7.2