Designing Encryption Algorithms for Real People

Bruce Schneier

Counterpane Systems
schneier@chinet.com

Abstract:

There is a wide disparity between cryptographic algorithms as specified by researchers and cryptographic algorithms as implemented in software applications. Programmers are prone to implement poor key management, make mistakes coding the algorithm, and use the algorithm in ways and for periods of time not originally intended. I propose design heuristics for designing algorithms in the "hostile" implementation environment of real-world software development.

In the real world, it is easy to choose a secure encryption algorithm. There are several, all designed by respected cryptographers, all described in the open literature, and all implemented in public-domain software. It is much harder to implement the algorithm properly in a software application.

As a result, implementations of the algorithm are often far less secure in practice than its creators envisioned them. While it is not strictly the job of an algorithm designer to concern himself with implementation details, it is important to realize how algorithms are used in the real world. Armed with this knowledge, a cryptographer can make his algorithms resilient to the kinds of abuses that they will most likely face in the hands of naive programmers.

This paper concerns itself with software implementations of cryptographic algorithms. Traditionally strong encryption was almost exclusively found in special-purpose and embedded hardware. Since these hardware devices were self-contained, it was easier to design them securely and force secure implementations. Today, the increased demand for cheap encryption combined with the increased power of personal computers has made software encryption more ubiquitous. Mistakes are far more common in software cryptographic systems, because programmers have far more control over the details of a software system. They have more opportunities to make mistakes in programming, implement bad key management processes, ignore memory management issues, and cut corners to improve performance.

PROBLEMS AND CORRECTIONS

There are several real-world problems that a cryptographer should take in account and, if possible, protect against when designing an algorithm. These problems range from the entrenchment of standards, the ignorance of users, and the fallibility of implementers.

Algorithm Entrenchment:

Encryption algorithms are used for a long time, generally far longer than any original requirements specified. DES was originally approved as a five-year standard to be reviewed every five years [9]. Fifteen years later, it was approved for another five years [4, 10], despite its increased vulnerability to cryptanalytic attack. The entrenchment of DES within communities such as banking will make it difficult to switch to a different encryption algorithm, even if NIST does not recertify DES in the future.

Any algorithm developed today will not see widespread use for at least five years. If it becomes part of a standard, it will probably still be in use 25 years later, encrypting data that might need to remain secure for another 50 years.

Therefore, any algorithm developed today should be designed to be secure against cryptanalytic attacks mounted in the year 2075.

Since it is impossible to estimate computing power that far in the future, theoretical arguments have been used to justify security lengths. The Chinese lottery, the whimsical DESosaur, and genetically-engineered algae have all been proposed as brute-force cracking machines against cryptographic algorithms [12]. Results from these thought experiments indicate that an algorithm should be, at the very least, secure against an attack whose complexity is on the order of 2^{128}.

Bad Key Management:

Although most encryption algorithms are designed to accept a random binary key, certain key management practices can make this difficult or impossible. Many software implementations of DES use ASCII input from the user directly as the key [14]. Instead of a 56-bit random key, the actual key is limited to 8 printable ASCII characters. One implementation on MS-DOS computers--Symantec's Norton Discreet version 8.0--limits the key choice even further, allowing converting lower-case letters to upper-case and ignoring the low-order bit of each byte. The result is a keyspace of only 40 bits.

One way to correct this problem is to give users the option of using long easy-to-remember keys: pass phrases. It is possible to hash a long password, using a one-way hash function like MD5 [13] or SHA [11], into a shorter encryption key. This technique allows end-users to choose easy-to-remember passwords without succumbing as easily to dictionary attacks, and ensures that the encryption key has the maximum possible entropy. Assuming that the entropy of English is 1.3 bits per character [16,3], 43 characters of English text are required to generate a random 56-bit key, 49 characters are required to generate a random 64-bit key, and 98 characters are required to generate a 128-bit key.

Most applications do not bother to implement a one-way hash function in conjunction with an encryption algorithm, and it is unwise to assume that they will do so in the future. A better solution would be to design this hashing process into the encryption algorithm. In addition to accepting a binary key, an encryption algorithm should just as easily accept a longer natural-language key. Blowfish [15] accepts these sorts of keys.

Poorly-Chosen Keys:

As if poor key management techniques don't do enough damage, end users make things even worse. Users do not choose keys uniformly from the keyspace provided to them. They are far more likely to choose words or near-words as keys [5]. A brute-force keysearch machine that first tries words and near-words will have an excellent chance of finding encryption keys quickly.

Pass phrases, described above, help. Still, there is no way to prevent this problem. Many end users will choose weak keys as long as they are allowed to. However, the problem can be mitigated.

In some applications it may be wise to reserve 16 or more key bits for an application-specific or system-specific key. This is a random bit string appended to the user key, designed to make brute-force keysearch more difficult. Depending on the system, these key bits can be common among a group of users, or individual to each specific instance of encryption. The bits do not have to be secret; their security lies in the fact that an attacker cannot build a single dictionary that can attack all implementations of a given algorithm. If the encryption algorithm has more key bits than strictly required for security, this can easily be done.

Programming Errors:

While most commercial software implementations execute DES encryption in accordance with NIST specifications [9], some do not [14]. The two most common mistakes are to ignore the distinction between big-endian and little-endian computer architectures, and to incorrectly index the S-boxes. DES is susceptible to subtle changes in the design of the algorithm [2]: changing the data in the S-boxes, the manner in which they are accessed, or even their order greatly reduces the security of the algorithm.

This illustrates the problem with not completely specifying the algorithm. It is essential that any algorithm specification be completely unambiguous. It should include sample source code and, at the very least, test encryptions and decryptions.

Modes of Encryption:

Block encryption algorithms can be implemented in one of several modes. Most software implementations of block ciphers use Electronic Code Book mode, because it is the fastest and easiest. However, any of the feedback modes--CBC, OFB, or CFB--are more secure in that they prevent the compilation of a code book. For bulk encryption applications, it is prudent to design algorithms with built-in feedback mechanisms.

Stream ciphers can have similar problems. However, their bit-wise granularity makes them a poor choice for software encryption, and they are hardly ever used for those reasons.

Size of Secret Information:

Computers are not very good at keeping secrets. Deleted files remain on disks, temporary files litter disks, and whole chunks of RAM are occasionally saved to disk at the whim of the operating system. All of these are potential implementation problems. A good algorithm will limit the amount of secret information that the computer must deal with. DES, with 103 bytes of secret information (7-byte key plus sixteen 6-byte subkeys) is easier to implement than Blowfish [15], with 4224 bytes of secret information (56-byte key plus 4168 bytes of subkeys).

Another possible solution for bulk encryption is an algorithm where the key modifies itself during encryption and decryption. This would serve to erase secret information on the fly, and would limit the amount of information a cryptanalyst who breaks into a computer could get.

Variable Parameters:

Some algorithms have variable parameters. Khufu and Khafre have a variable number of iterations [8]. The HAVAL one-way hash function has a variable number of iterations and can produce variable-length hash value [17]. Variable parameters such as these invite mistakes. Often the algorithms are implemented by programmers who do not understand the cryptographic reasons behind these variations, and will invariably choose the parameters that make the algorithm run the fastest. An algorithm should be secure even if all variable parameters are set to their minimum. Better yet, an algorithm should not have any variable parameters.

Implementers will try to vary parameters anyway. DES has no variable parameters, but some encryption products offer an 8-round DES option [14]. One product has an option for single-round DES, even though this variant leaves half the plaintext unencrypted [14].

Key length is the only exception to the above. Implementers always have the option of using keys smaller then the algorithm is designed for. Users often choose keys of lengths shorter than the maximum allowed keylength. Cryptographers should assume that this kind of thing will happen, and ensure that using a short key does not affect the security of the algorithm to a greater degree than the reduction in key size.

DESIGN DIVERSITY

In today's cryptographic world, DES is by far the most common algorithm. This is good, because DES is the most cryptanalyzed of today's public algorithms. On the other hand, a single breakthrough in the cryptanalysis of DES will have broad ramifications for security; there are just too many eggs in a single cryptographic basket. Better would be for different software applications to use different encryption algorithms, or for software applications to give users a choice of encryption algorithms. This would minimize the catastrophe if a common algorithm is ever successfully cryptanalyzed.

Recently, other algorithms have begun to see limited use. IDEA [6] has been used in several applications. Special workshops devoted to algorithm design have been organized [1].

CASCADING MULTIPLE ALGORITHMS

One way to increase an algorithm's resilience is to cascade multiple algorithms in sequence. A multi-algorithm cascade with independent keys has been proven to be at least as strong as the first or, in the case of stream ciphers, at least as secure as the best [7]. Even so, it does seem that a cascade of algorithms is better than individual algorithms, provided that the second and subsequent algorithms are secure against chosen ciphertext attacks, and provided all the algorithms' keys are independent.

The real benefit of cascading algorithms is to take advantage of design diversity; it makes the overall implementation less vulnerable to a programming error or new cryptanalytic attack. If there are two or even three algorithms in a cascade, each strong enough on its own to satisfy the security requirements of the entire system, than security holes in one or two will not breach overall security. Successful attacks against all three would be required to break the cascaded system.

CONCLUSIONS

I have shown how misguided programmers can inadvertently subvert the best intentions of cryptographers. While it is impossible to completely shield programmers from their own ignorance, it is possible to design encryption algorithms to be resilient against some of their most common mistakes.

ACKNOWLEDGMENTS

Steve Bellovin and Matt Blaze both provided numerous helpful comments on an earlier version of this paper. Any mistakes are wholly my own.

REFERENCES

1. R. Anderson, ed., *Fast Software Encryption, Lecture Notes in Computer Science 809*, Springer-Verlag, 1994.

2. E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard,* Springer-Verlag, 1993.

3. T.M. Cover & R.C. King, "A Convergent Gambling Estimate of the Entropy of English," *IEEE Transactions on Information Theory,* v. IT-24, n. 4, Jul 1978, pp. 413-421.

4. NBS FIPS PUB 46-2, "Data Encryption Standard," National Bureau of Standards, U.S. Department of Commerce, Jan 1988.

5. D.V. Klein, "'Foiling the Cracker': A Survey of, and Implications to, Password Security," *Proceedings of the USENIX UNIX Security Workshop,* Aug 1990, pp. 5- 14.

6. X. Lai, J. Massey, and S. Murphy, "Markov Ciphers and Differential Cryptanalysis," *Advances in Cryptology--EUROCRYPT '91 Proceedings,* Springer-Verlag, 1991, pp. 17-38.

7. U.M. Maurer and J.L. Massey, "Cascade Ciphers: The Importance of Being First," Journal of Cryptology, v. 6, 1993, pp. 55-61.

8. R. Merkle, "Fast Software Encryption Functions," *Advances in Cryptology—CRYPTO '90 Proceedings,* Springer-Verlag, 1991, pp. 476-501.

9. National Bureau of Standards, *Data Encryption Standard,* U.S. Department of Commerce, FIPS Publication 46, Jan 1977.

10. NIST FIPS PUB 46-2, "Data Encryption Standard," National Bureau of Standards, U.S. Department of Commerce, Dec 1993.

11. NIST FIPS PUB 180, "Secure Hash Standard," National Institute of Standard and Technology, U.S. Department of Commerce, Apr 1993.

12. J.-J. Quisquater and Y.G. Desmedt, "Chinese Lotto as an Exhaustive Code-Breaking Machine," *Computer,* v. 24, n. 11, Nov 1991, pp. 14-22.

13. R. Rivest, "The MD5 Message Digest Algorithm," RFC 1321, Apr 1992.

14. B. Schneier, "Data Guardians," *MacWorld,* Feb 93, 145-151.

15. B. Schneier, "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)," *Fast Software Encryption, Lecture Notes in Computer Science 809*, R. Anderson, ed., Springer-Verlag, 1994, pp. 191-204.

16. C.E. Shannon, "Prediction and entropy of printed English," *Bell Systems Technical Journal,* v. 30, n. 1, Jan 1951, pp. 50-64.

17. Y. Zheng, J. Piepryzk, and J. Siberry, "HAVAL—One-Way Hashing Algorithm with Variable Length of Output," *Advances in Cryptology—AUSCRYPT '92 Proceedings,* Springer-Verlag, 1994.