Chapter 8: Cryptographic Foibles

Foible: a weakness or idiosyncrasy

1. Crypto can be used to help secure data from threats – but it does not prevent coding errors
    a. Provides data privacy and integrity
    b. Facilitates strong authentication
2. Common mistakes
    a. Use of poor random numbers
        i. Do not use rand or any other linear congruential function (they are predictable)
        ii. Use CryptGenRandom in Win32 (#include<wincrypt.h>)
            1. draws on numerous hardware sources
            2. hashes the resulting byte stream with SHA-1 to produce a 20-byte seed value to generate random numbers using FIPS 186-2 appendix 3.1: http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf (page 16 of .pdf)
            3. FIPS 140-1: provides a means to validate vendor's crypto products (http://www.itl.nist.gov/fipspubs/fip140-1.htm)
        iii. Linux: get_random_bytes from kernel interface (/dev/random and /dev/urandom) – discussion of generation in Linux (http://eprint.iacr.org/2006/086.pdf)
        iv. System.Security.Cryptography in .NET
        v.
    b. Using passwords to derive cryptographic keys
        i. Good keys are hard to guess and long
        ii. Humans like easy and short!
        iii. Passwords with large effective bit length and entropy (which results in random distribution of bits) should be used
        iv. Table

Scenario	**Available chars**	**Length for 56-bit key**	**Length for 128-bit key**
Numeric pin	0-9	17	40
Case insensitive alpha	A-Z or a-z	12	28
Case sensitive alpha	A-Z and a-z	10	23
Case sensitive alpha and numerica	0-9 and A-Z and a-z	10	22
" " and punctuation	" " and punctuation for a total of 93	9	20

        v. The Memorability and Security of Passwords: http://www.ftp.cl.cam.ac.uk/ftp/users/rja14/tr500.pdf

c. Poor key management techniques
    i. Considered the weakest link of crypto apps
    ii. Using crypto is easy; securely storing, exchanging, and using keys is hard (DVD encryption cracked due to this problem)
    iii. No passwords in code (of any form) – random passwords stick out like a sore thumb (due to entropy) in binaries (and there are tools to look for just such things)
    iv. Key types
        1. short term (ephemeral): used by networking protocols (IPSec, SSL/TLS, RPC, DCOM) – key gen is process is hidden from app and user
        2. long term: used for authentication, integrity, non-repudiation, protecting persistent data in databases and files
    v. Use appropriate key lengths
        1. the longer the better
            a. attacking symmetric ciphers (DES and RC4) requires brut force try at every key
            b. attacking asymmetric (RSA) requires random value generation using public and private keys (factoring)
            c. because of type of attack, it is "easier" to factor 512-bit RSA key than brute force 112-bit 3DES key)
            d. can protect symmetric keys using asymmetric keys
        2. Learn your ciphers and their strengths and weaknesses before using
    vi. Keep keys close to source: this prevents them from being passed around from entity to entity (increasing chance of discovery)
        1. send key in encrypted form as much as possible
        2. do not provide direct access to memory location of key if possible
    vii. Key exchange issues
        1. don't exchange private keys (they're private for a reason)
        2. do not embed key in code
        3. use sneakernet if possible (watch for social engineering)
        4. use a protocol that does the key exchange (securely) for you (SSL/TLS, IPSec)
        5. use tried and true Diffie-Hellman (http://en.wikipedia.org/wiki/Diffie-Hellman) or RSA key exchange (http://library.thinkquest.org/27158/concept2_4.html)

d. Creation of own crypto routines ("Our crypto rocks!")
    i. Don't do it

e. Steam cipher: a cipher that encrypts data one unit at a time (usually 1 byte)
    i. RC4 is an example of this
    ii. how it works:

            1. an encryption key is provided to a keystream generator algorithm
            2. stream of key bits is produced from this
            3. key bits are XORed with plain text to produce cipher text bits
            4. decrypting requires reversing the process (XOR the key stream with the cipher text to produce plain text)
        iii. symmetric cipher: same key is used to encrypt and decrypt (DES, 3DES, AES, IDEA, RC2 – all these are block ciphers – encrypt/decrypt done a block at a time – usually 64 or 128 bits at a time)
        iv. asymmetric cipher: two different, but related keys used to encrypt and decrypt
        v. stream ciphers are used because they don't use a lot of memory (result is about same size as original) and they can be very fast
        vi. if a key is discovered by an attacker, it can be re-used many times on the rest of the streams being sent (don't want different keys for each stream due to management issues)
        vii. key can be salted for protection (add additional bits – make sure the bits are random)
        viii. Bit flipping attacks: if general form of a message is known, can flip a single bit and corrupt the data – no one will know
            1. use a digital signature or keyed hash
            2. hashing is weak: attacker can change data then recalculate hash and you won't know data was modified
        ix. keyed hash: includes secret data only known to alice and bob
            1. created by hashing plaintext concatenated to a secret key
            2. need to know secret key to properly calculate keyed hash
            3. a keyed hash is a MAC (Message Authentication Code) or HMAC: http://en.wikipedia.org/wiki/Message_authentication_code
            4. process: plaintext + MAC key → MAC; plaintext + encryption key and function → ciphertext; ciphertext & MAC → message
            5. Ghastly code for creation in C++ Win32:

```
#include "stdafx.h"
DWORD HMACStuff(void *szKey, DWORD cbKey,
                void *pbData, DWORD cbData,
                LPBYTE *pbHMAC, LPDWORD pcbHMAC) {

    DWORD dwErr = 0;
    HCRYPTPROV hProv;
    HCRYPTKEY hKey;
    HCRYPTHASH hHash, hKeyHash;

    try {
        if (!CryptAcquireContext(&hProv, 0, 0,
            PROV_RSA_FULL, CRYPT_VERIFYCONTEXT))
            throw;

        // Derive the hash key.
        if (!CryptCreateHash(hProv, CALG_SHA1, 0, 0,
&hKeyHash))
            throw;

        if (!CryptHashData(hKeyHash, (LPBYTE)szKey,
cbKey, 0))
            throw;

        if (!CryptDeriveKey(hProv, CALG_DES,
            hKeyHash, 0, &hKey))
            throw;

        // Create a hash object.
        if(!CryptCreateHash(hProv, CALG_HMAC, hKey, 0,
&hHash))
            throw;

        HMAC_INFO hmacInfo;
        ZeroMemory(&hmacInfo, sizeof(HMAC_INFO));
        hmacInfo.HashAlgid = CALG_SHA1;

        if(!CryptSetHashParam(hHash, HP_HMAC_INFO,
                             (LPBYTE)&hmacInfo,
                              0))
            throw;

        // Compute the HMAC for the data.
        if(!CryptHashData(hHash, (LPBYTE)pbData, cbData,
0))
            throw;

        // Allocate memory, and get the HMAC.
        DWORD cbHMAC = 0;
        if(!CryptGetHashParam(hHash, HP_HASHVAL, NULL,
&cbHMAC, 0))
            throw;

        // Retrieve the size of the hash.
        *pcbHMAC = cbHMAC;
        *pbHMAC = new BYTE[cbHMAC];
        if (NULL == *pbHMAC)
            throw;
        if(!CryptGetHashParam(hHash, HP_HASHVAL, *pbHMAC,
&cbHMAC, 0))
            throw;

    } catch(...) {
        printf("Error - %d", GetLastError());
        dwErr = GetLastError();
    }

    if (hProv)      CryptReleaseContext(hProv, 0);
    if (hKeyHash)   CryptDestroyKey(hKeyHash);
    if (hKey)       CryptDestroyKey(hKey);
    if (hHash)      CryptDestroyHash(hHash);

    return dwErr;
}

void main() {
    // Key comes from the user.
    char *szKey = GetKeyFromUser();
    DWORD cbKey = lstrlen(szKey);
    if (cbKey == 0) {
        printf("Error - you did not provide a key.\n");
        return -1;
    }

    char *szData="In a hole in the ground...";
    DWORD cbData = lstrlen(szData);

    // pbHMAC will contain the HMAC.
    // The HMAC is cbHMAC bytes in length.
    LPBYTE pbHMAC = NULL;
    DWORD cbHMAC = 0;
    DWORD dwErr = HMACStuff(szKey, cbKey,
                            szData, cbData,
                            &pbHMAC, &cbHMAC);

    // Do something with pbHMAC.

    delete [] pbHMAC;
}
```
→Use OS or .NET libraries if possible (it's MUCH easier)

x. Creating a digital signature
   1. encrypt a hash with a private key
   2. process: plaintext sent to hash function → hash; hash + private key sent to digital signature function → digital

signature; plaintext sent to encryption function → ciphertext; ciphertext & digital signature → message

xi. Use MACs or digital signatures to verify integrity of data

xii. Be careful of plaintext in memory…

f. Threat mitigation using crypto

Threat	**Mitigation technique**	**Algorithms to use**
Information disclosure	Data encryption using a symmetric cipher	RC2, RC4, 3DES, AES
Tampering	Data and message integrity using hash functions, MACs, digital signatures	SHA-1, SHA-256, SHA-384, SHA-512, MD5, HMAC, RSA digital signatures, DSS, XML DSig
Spoofing	Authenticate data is from sender	Public key certificates and digital signatures