Proceedings of the 11th USENIX Security Symposium

San Francisco, California, USA
August 5-9, 2002

Lessons Learned in Implementing and Deploying Crypto Software

Peter Gutmann
University of Auckland

Abstract

Although the basic building blocks for working with strong encryption have become fairly widespread in the last few years, experience has shown that implementers frequently misuse them in a manner which voids their security properties. At least some of the blame lies with the tools themselves, which often make it unnecessarily easy to get things wrong. Just as no chainsaw manufacturer would think of producing a model without a finger-guard and cutoff mechanism, so security software designers need to consider safety features which will keep users from injuring themselves or others. This paper examines some of the more common problem areas which exist in crypto security software, and provides a series of design guidelines which can help minimise damage due to (mis-)use by inexperienced users. These issues are taken from extensive real-world experience with users of security software, and represent areas which frequently cause problems when the software is employed in practice.

1. Introduction

In the last five years or so the basic tools for strong encryption have become fairly widespread, gradually displacing the snake oil products which they had shared the environment with until then. As a result, it's now fairly easy to obtain software which contains well-established, strong algorithms such as triple DES and RSA instead of pseudo one-time-pads. Unfortunately, this hasn't solved the snake oil problem, but has merely relocated it elsewhere.

> *The determined programmer can produce snake oil using any crypto tools.*

What makes the new generation of dubious crypto products more problematic than their predecessors is that the obvious danger signs which allowed bad crypto to be quickly weeded out are no longer present. A proprietary, patent-pending, military-strength, million-bit-key, one-time pad built from encrypted prime cycle wheels is a sure warning sign to stay well clear, but a file encryptor which uses Blowfish with a 128-bit key seems perfectly safe until further analysis reveals that the key is obtained from an MD5 hash of an uppercase-only 8-character ASCII password. This type of second-generation snake oil crypto, which looks like the real thing but isn't, could be referred to as naugahyde crypto, with an appropriately similar type of relationship to the real thing.

Most crypto software is written with the assumption that the user knows what they're doing, and will choose the most appropriate algorithm and mode of operation, carefully manage key generation and secure key storage, employ the crypto in a suitably safe manner, and do a great many other things which require fairly detailed crypto knowledge. However, since most implementers are everyday programmers whose motivation for working with crypto is defined by "the boss said do it", the inevitable result is the creation of products with genuine naugahyde crypto. Sometimes this is discovered (for example when encryption keys are generated from the process ID and time, or when the RC4 keystream is re-used multiple times so the plaintext can be recovered with a simple XOR), but more frequently it isn't, so that products providing only illusory security may be deployed and used for years without anyone being any the wiser.

This paper looks at some of the ways in which crypto software developers and providers can work to avoid creating and deploying software which can be used to create naugahyde crypto. Much of the experience presented here comes from developing and supporting the open-source cryptlib toolkit [1][2], which has provided the author with a wealth of information on the ways in which crypto software is typically misused, and the principal areas in which users experience problems. Additional feedback was provided from users and developers involved with other open-source crypto efforts.

All of the events reported here are from real experiences with users, although the details have been obscured and anonymised, particularly where the users in question have more lawyers than the author's University has staff. In addition a few of the more interesting stories were excluded, but are referred to indirectly in the text (although no-one would have been able to identify the organisations involved, it was felt that having the event officially documented rather than existing only in the memory of a few implementers was too much of a legal liability). Although there are less references to sources than the author usually includes in his work, the reader should rest assured that all of the events mentioned here are real, and it's almost certain that they have either used, or been a part of the use of, one or more of the products which are not quite referred to.

2. Existing Work

There exists very little published research on the topic of proactively ensuring that crypto is used in a secure manner, as opposed to patching programs up after a problem is found. Most authors are content to present the algorithms and mechanisms and leave the rest to the implementer. An earlier work on why cryptosystems fail concentrated mostly on banking security [3][4], but did make the prophetic prediction that as implementers working with crypto products "lack skills at security integration and management, they will go on to build systems with holes".

Another paper examined user interface problems in encryption software [5], an area which badly needs further work by HCI researchers. Finally, the author of a widely-used book on crypto went on to write a followup work designed to address the problem that "the world was full of bad security systems designed by people who read [his first book]" [6]. Like the author of this paper, he found that "the weak points had nothing to do with mathematics [...] Beautiful pieces of mathematics were made irrelevant through bad programming". The followup work examines security in a very general-purpose manner as a process rather than a product, while this paper limits itself to trying to address the most commonly-made errors which occur when non-cryptographers (mis-)apply crypto.

In addition to these works there exist a number of general-purpose references covering security issues which can occur during application design and implementation [7][8][9]. These are targeted at application developers and are intended to cover (and hopefully eliminate) common application security problems such as buffer overflows, race conditions, elevation of privileges, access control issues, and so on. This work in contrast looks specifically at problems which occur when end users (mis-)use security software, and suggests design guidelines which can help combat such misuse.

3. Crypto Software Problems and Solutions

There are many ways in which crypto and security software can be misused. The main body of this paper covers some of the more common problem areas, providing examples of misuse and suggesting (if possible) solutions which may be adopted by developers of the security software to help minimise the potential for problems. While there is no universal fix for all problems (and indeed some of them have a social or economic basis which can't be easily solved through the application of technology), it is hoped that the guidelines presented here will both alert developers to the existence of certain problem areas and provide some assistance in combating them.

3.1 Private Keys Aren't

One of the principal design features of cryptlib is that it never exposes private keys to outside access. The single most frequently-asked cryptlib question is therefore "How do I export private keys in plaintext form?". The reasons given for this are many and varied, and range from the logical ("I want to generate a test key for use with XYZ") to the dubious ("We want to share the same private key across all of our servers") through to the bizarre ("I don't know, I just want to do it").

In some cases the need to spread private keys around is motivated by financial concerns. If a company has spent $495 on a Verisign certificate which was downloaded to a Windows machine then they won't spend that much again for exactly the same thing in a different format. As a result, the private key is exported from the Windows key store (from which any Windows application can utilise it) into Netscape. And OpenSSL. And BSAFE. And cryptlib (although cryptlib deliberately makes it rather difficult to poke keys of unknown provenance into it). Eventually, every encryption-enabled application on the system has a copy of the key, and for good measure it may be spread across a number of systems for use by different developers or sysadmins. Saving CA fees by re-using a single private key for everything seems to be very popular, particularly among Windows users.

The amount of sharing of private keys across applications and machines is truly frightening. Mostly this appears to occur because users don't understand the value of the private key data, treating it as just another piece of information which can be copied across to wherever it's convenient. For example a few years ago a company had developed a PGP-based encrypted file transfer system for a large customer. The system used a 2048-bit private key which was stored on disk in plaintext form, since the software was run as a batch process and couldn't halt waiting for a password to be entered. One day the customer called to say that they'd lost the private key file, and could the company's programmers please reconstruct it for them. This caused some consternation at the company, until one of the developers pointed out that there were copies of the private key stored on a file server along with the source code, and in other locations with the application binaries. Further investigation revealed that the developers had also copied it to their own machines during the development process for testing purposes. Some of these machines had later been passed on to new employees, with their original contents intact. The file server on which the development work was stored had had its hard drives upgraded some time earlier, and the old drives (with the key on them) had been put on a nearby shelf in case they were needed later. The server was backed up regularly, with three staff members taking it in turns to take the day's tapes home with them for off-site storage (the standard practice was to drop them in the back seat of the car until they were re-used later on). In short, the only way to securely delete the encryption key being used to protect large amounts of long-term sensitive data would have been to carpet-bomb the city, and even then it's not certain that copies wouldn't have survived somewhere. While this represents a marvellous backup strategy, it's probably not what's required for protecting private keys.

If your product allows the export of private keys in plaintext form or some other widely-readable format, you should assume that your keys will end up in every other application on the system, and occasionally spread across other systems as well.

At least some of the problem arises from the fact the much current software makes it unnecessarily easy to move private keys around (see also section 3.2 for a variation of this problem). For example CAs frequently use PKCS #12 files to send a "certificate" to a new user because it makes things simpler than going through the multi-stage process in which the browser generates the private key itself. These files are invariably sent in plain text email, often with the password included. Alternatively, when the password is sent by out-of-band means, the PKCS #12 decryption key is generated directly from a hash of the uppercase-only ASCII password, despite warnings about the insecurity of this approach being well publicised several years ago [19]. Once such file, provided as a sample to the author, would have authorised access to third-party financial records in a European country. This method of key handling was standard practice for the CA involved.

Another CA took this process a step further when they attempted to solve the problem of not having their root certificate trusted by various browsers and mail programs by distributing a PKCS #12 file containing the CA root key and certificate to all relying parties. The thinking was that once the CA's private key was installed on their system, the user's PKI software would regard the corresponding certificate as being trusted (it still didn't quite fix the problem, but it was a start). This "solution" is in fact so common that the OpenSSL FAQ contains an entry specifically warning against it [55]. Incredibly, despite the strong warning in the FAQ that "this command will give away your CA's private key and reduces its security to zero", security books have appeared which give clear, step-by-step instructions on how to distribute the CA's private key "to all your user's web browsers" [10].

Making it more difficult to do this sort of thing might help alleviate some of the problems. Certainly in the case of cryptlib when users are informed that what they're asking for isn't possible, they find a means of working within those constraints (or maybe they quietly switch to CryptoAPI, which allows private keys to be sprayed around freely). However the real problem is a social and financial one, and is examined in more detail in section 3.2.

3.2 Everything is a Certificate

In 1996 Microsoft introduced a new storage format for private keys and certificates to replace the collection of ad hoc (and insecure) formats which had been in use before then [11][12]. Initially called PFX (Personal Information Exchange) [13][14][15][16], it was later re-released in a cleaned-up form as PKCS #12 [17]. One of the main motivations for its introduction was for use in Internet kiosks in which users carried their personal data around on a floppy disk for use wherever they needed it. In practice this would have been a bad idea since Internet Explorer retains copies of the key data so that the next user who came along could obtain the previous user's keys by exporting them back onto a floppy. Internet kiosks never eventuated, but the PKCS #12 format has remained with us.

Since PKCS #12 stores both keys and certificates, and (at least under Windows) the resulting files behave exactly like certificates, many users are unable to distinguish certificates from PKCS #12 objects. In the same way that "I'm sending you a document" typically heralds the arrival of a Microsoft Word file, so "I'm sending you a my certificate" is frequently accompanied by a PKCS #12 file. This problem isn't helped by the fact that the Windows "Certificate Export Wizard" actually creates PKCS #12 files as output, defaulting to exporting the private key alongside the certificate. The situation is further confused by some of the accompanying documentation, which refers to the PKCS #12 data as a "digital ID" (rather than "certificate" or "private key"), with the implication that it's just a certificate which happens to require a password when exported. The practice of mixing public and private keys in this manner, and referring to the process of and making the behaviour of the result identical to the behaviour of a plain certificate, are akin to pouring weedkiller into a fruit juice bottle and storing it on an easily accessible shelf in the kitchen cupboard.

The author, being a known open-source crypto developer, is occasionally asked for help with certificate-management code, and has over the years accumulated a small collection of users' private keys and certificates, ranging from disposable email certificates through to relatively expensive higher-assurance certificates (the users were notified and the keys deleted where requested). The current record for a key obtained in this manner (reported by another open-source crypto developer in a similar situation) is the key for a Verisign Class 3 code-signing certificate, the highest-level certificate provided by Verisign which requires notarisation, background investigations, and fairly extensive background checking [18].

Once the PKCS #12 file is obtained, the contents can generally be recovered, either by recovering the password [19][20][21] or by taking advantage of the fact that the Certificate Export Wizard will export keys without any password if the user just keeps clicking 'Next' in standard Wizard fashion (they are in fact encrypted with a password consisting of two null characters, a Microsoft implementation bug which was reverse-engineered back into PKCS #12).

In contrast, PGP has no such problems. PGP physically separates the public and private portion of the key into two files, and makes it quite clear that the private-key file should never be distributed to anyone: "keep your secret key file to yourself [...] Never give your secret key to anyone else [...] Always keep physical control of your secret key, and don't risk exposing it by storing it on a remote timesharing computer. Keep it on your own personal computer" [22]. When distributing keys to other users, PGP only extracts the public components, even if the user explicitly forces PGP to read from the private key file (the default is to use the public key file). Even if the user never bothers to read the documentation which warns about private key security, PGP's safe-by-default key handling ensures that they can't accidentally compromise the key.

Make very clear to users the difference between public and private keys, either in the documentation/user interface or, better, by physically separating the two.

The single biggest reason for the re-use of a single key wherever possible is, as already mentioned in section 3.1, the cost of the associated certificate. A secondary reason is the complexity involved in obtaining the certificate, even if it is otherwise free. Examples of the latter include no-assurance email certificates, sometimes known as "clown-suit certificates" because of the level of identity assurance they provide [23]. Generating a new key rather than re-using the current one is therefore expensive enough and cumbersome enough that users are given the incentive to put up with considerable inconvenience in order to re-use private keys. Users have even tried to construct ways of sharing smart cards across multiple machines in order to solve the annoying problem that they can't export the private key from the card. Another approach, which only works with some cards, is to generate the key externally and load it onto the card, leaving a copy of the original in software to be used from various applications and/or machines (the fact that people were doing this was discovered because some cards or card drivers handle external key loads in a peculiar manner, leading to requests for help from users).

PGP on the other hand, with its easily-generated, self-signed keys and certificates, suffers from no such problem, and real-world experience indicates that users are quite happy to switch to new keys and discard their old ones whenever they feel the need.

In order to solve this problem, it is necessary to remove the strong incentive provided by current X.509-style certificate management to re-use private keys. One solution to this problem would be for users to be issued key-signing certificates which they could use to create their own certificates when and as needed. This represents a somewhat awkward workaround for the fact that X.509 doesn't allow multiple signatures binding an indentity to a certificate, so that it's not possible to generate a self-signed certificate which is then endorsed through further, external signatures. In any case since this solution would deprive CAs of revenue, it's unlikely to ever be implemented. As a result, even if private key sharing is made as difficult as possible, sufficiently motivated users will still find ways to spread them around. It is, unfortunately, very difficult to fix social/economic issues using technology.

3.3 Making Key Management Easy

One popular solution for key management, which has been around since the technology was still referred to as dinosaur oil, is the use of fixed, shared keys. Despite the availability of public-key encryption technology, the use of this type of key management is still popular, particularly in sectors such as banking which have a great deal of experience in working with confidential information. Portions of the process have now been overtaken by technology, with the fax machine replacing trusted couriers for key exchange.

Another solution which is popular in EDI applications is to transmit the key in another message segment in the transaction. If XML is being used, the encryption key is placed in a field carefully tagged as <password> or <key>. Yet another solution, popularised in WEP, is to use a single fixed key throughout an organisation [24].

Even when public-key encryption is being used, users often design their own key-management schemes to go with it. One (geographically distributed) organisation solved the key management problem by using the same private key on all of their systems. This allowed them to deploy public-key encryption throughout the organisation while at the same time eliminating any key management problems, since it was no longer necessary to track a confusing collection of individual keys.

Straight Diffie-Hellman requires no key management. This is always better than other no-key-management alternatives which users will create.

Obviously this method of (non-)key management is still vulnerable to a man-in-the-middle (MITM) attack, however this requires an active attack at the time the connection is established. This type of attack is considerably more difficult than a passive attack performed an arbitrary amount of time later, as is possible with unprotected, widely-known, or poorly-chosen shared keys, or, worse yet, no protection at all because a general solution to the problem isn't available [25]. In situations like this the engineering approach (within ±10% of the target with reasonable effort) is often better than the mathematician's approach (100% accuracy with unreasonable effort, so that in practice nothing gets done).

3.4 What Time is it Anyway?

Many security protocols, and in particular almost all PKI protocols which deal with validity intervals and time periods, assume they're operating in the presence of precisely-synchronised clocks on all systems. The fact that this frequently isn't the case was recognised a decade ago both by security researchers (mostly as a result of Kerberos V4's use of timestamps) [26][27][28] and by implementers of post-Kerberos V4 protocols such as IBM's KryptoKnight, which replaced the time-stamps with nonces [29][30][31][32], Bell-Atlantic's Yaksha [33][34], and to some extent Kerberos V5, which allows for (but doesn't require) nonces [35]. More recently, one of the few published papers on PKI implementation experience pointed out the problems inherent in using timestamps for synchronisation in the CMP PKI protocol [36].

The author has seen Windows machines whose time was out by tens of minutes (incorrect settings or general clock drift), one or more hours (incorrect settings or incorrect time zone/daylight savings time adjustment), one or more days (incorrect settings or incorrect time zone, for example a machine in New Zealand set to GMT), and various larger units (weeks or months). In the most extreme case the time was out by several decades but wasn't noticed by the user until cryptlib complained about a time problem while processing certificates with a known validity period. In addition to the basic incorrect time problems, combinations such as an offset of one day + one hour + 15 minutes have been spotted.

In addition to problems due to incorrect settings, there are also potential implementation problems. One PKI pilot ran into difficulties because of differences in the calculation of offsets from GMT in different software packages [37]. Time zone issues are extremely problematic because some operating systems handle them in a haphazard manner or can be trivially mis-configured to get the offset wrong. Even when everything is set up correctly it can prove almost impossible to determine the time offset from a program in one time zone with daylight savings time adjustment and a second program in a different time zone without daylight savings time adjustment.

A further problem with a reliance on timestamps is the fact that it extends the security baseline to something which is not normally regarded as being security-relevant, and which therefore won't be handled as carefully as obviously-security-related items such as passwords and crypto tokens. To complicate things further, times are often deliberately set incorrectly to allow expired certificates to continue to be used without paying for a new one, a trick which shareware authors countered many years ago to prevent users from running trial versions of software indefinitely. For example Netscape's code signing software will blindly trust the date incorporated into a JAR file by the signer, allowing expired certificates to be rejuvenated by backdating the signature generation time. It would also be possible to resuscitate a revoked certificate using this trick, except that the software doesn't perform revocation checking so it's possible to use it anyway.

Don't incorporate the system clock (or the other parties' system clocks) in your security baseline. If you need synchro-nisation, use nonces.

If some sort of timeliness guarantees are required, this can still be achieved even in the presence of completely desynchronised clocks by using the clock as a means of measuring the passage of time rather than as an absolute indicator of time. For example a server can indicate to a client that the next update will take place 15 minutes after the current request was received, a quantity which can be measured accurately by both sides even if one side thinks it's currently September 1986. To perform this operation, the client would submit a request with a nonce, and the server would respond with a (signed or otherwise integrity-protected) reply containing a relative time to the next update. If the client doesn't receive the response within a given time, or the response doesn't contain the nonce they sent, then there's something suspicious going on. If everything is OK, they know the exact time (relative to their local clock) of the next update, or expiry, or revalidation. Although this measure is simple and obvious, the number of security standards which define mechanisms which assume the existence of perfectly synchronised clocks for all parties is somewhat worrying.

In the presence of arbitrary end user systems, relative time measures work. Absolute time measures don't.

For non-interactive protocols which can't use nonces the solution becomes slightly more complex, but can generally be implemented using techniques such as a one-off online query, or time-stamping [38].

3.5 RSA in CBC Mode

When the RSA algorithm is used for encryption, the operation is usually presented as "encrypting with RSA". The obvious consequence of this is that people try to perform bulk data encryption using pure RSA rather than using it purely as a key exchange mechanism for a fast symmetric cipher. In most cases this misunderstanding is quickly cleared up because the crypto toolkit API makes it obvious that RSA can't be used that way, however the JCE API, which attempts to provide a highly orthogonal interface to all ciphers even if the resulting operations don't make much sense, allows for bizarre combinations such as RSA in CBC mode with PKCS #5 padding alongside the more sensible DES alternative with the same mode and padding (CBC and PKCS #5 are mechanisms designed for use with block ciphers, not public-key algorithms). As a result, when a programmer is asked to implement RSA encryption of data, they implement the operation exactly as the API allows it. One of the most frequently-asked questions for one open-source Java crypto toolkit covers assorted variations on the use of bulk data encryption with RSA, usually relating to which (block cipher) padding or chaining mode to use, but eventually gravitating towards "Why is it so slow?" once the code nears completion and testing commences.

This can lead to a variety of interesting debates. Typically a customer asks for "RSA encryption of data", and the implementers deliver exactly that. The customer claims that no-one with an ounce of crypto knowledge[1] would ever perform bulk data encryption with RSA and the implementers should have known better, and the implementers claim that they're delivering exactly what the customer asked for. Eventually the customer threatens to withhold payment until the code is fixed, and the implementers sneak the changes in under "Misc.Exp." at five times the original price.

Don't include insecure or illogical security mechanisms in your crypto tools.

3.6 Left as an Exercise for the User

Crypto toolkits sometimes leave problems which the toolkit developers couldn't solve themselves as an exercise for the user. For example the gathering of entropy data for key generation is often expected to be performed by user-supplied code outside the toolkit. Experience with users has shown that they will typically go to any lengths to avoid having to provide useful entropy to a random number generator which relies on this type of user seeding. The first widely-known case where this occurred was with the Netscape generator, whose functioning with inadequate input required the disabling of safety checks which were designed to prevent this problem from occurring [39]. A more recent example of this phenomenon was provided by an update to the SSLeay/OpenSSL generator, which in version 0.9.5 had a simple check added to the code to test whether any entropy had been added to the generator (earlier versions would run the pseudo-random number generator (PRNG) with little or no real entropy). This change lead to a flood of error reports to OpenSSL developers, as well as helpful suggestions on how to solve the problem, including seeding the generator with a constant text string [40][41][42], seeding it with DSA public key components (whose components look random enough to fool entropy checks) before using it to generate the corresponding private key [43], seeding it with consecutive output byes from `rand()` [44], using the executable image [45], using /etc/passwd [46], using /var/log/syslog [47], using a hash of the files in the current directory [48], creating a dummy random data file and using it to fool the generator [49], downgrading to an older version such as 0.9.4 which doesn't check for correct seeding [50], using the output of the unseeded generator to seed the generator (by the same person who had originally solved the problem by downgrading to 0.9.4, after it was pointed out that this was a bad idea) [51], and using the string "0123456789ABCDEF0" [52]. Another alternative, suggested in a Usenet news posting, was to patch the code to disable the entropy check and allow the generator to run on empty (this magical fix has since been independently rediscovered by others [53]). In

[1] Equivalent to 31 grams of crypto knowledge, being worth its weight in gold.

later versions of the code which used /dev/random if it was present on the system, another possible fix was to open a random disk file and let the code read from that thinking it was reading the randomness device [54]. It is likely that considerably more effort and ingenuity has been expended towards seeding the generator incorrectly than ever went into doing it right.

The problem of inadequate seeding of the generator became so common that a special entry was added to the OpenSSL frequently-asked-questions (FAQ) list telling users what to do when their previously-fine application stopped working when they upgraded to version 0.9.5 [55], and since this still didn't appear to be sufficient later versions of the code were changed to display the FAQ's URL in the error message which was printed when the PRNG wasn't seeded. Based on comments on the OpenSSL developers list, quite a number of third-party applications which used the code were experiencing problems with the improved random number handling code in the new release, indicating that they were working with low-security cryptovariables and probably had been doing so for years. Because of this problem, a good basis for an attack on an application based on a version of SSLeay/OpenSSL before 0.9.5 is to assume the PRNG was never seeded, and for versions after 0.9.5 to assume it was seeded with the string "string to make the random number generator think it has entropy", a value which appeared in one of the test programs included with the code and which appears to be a favourite of users trying to make the generator "work".

The fact that this section has concentrated on SSLeay/OpenSSL seeding is not meant as a criticism of the software, the change in 0.9.5 merely served to provide a useful indication of how widespread the problem of inadequate initialisation really is. Helpful advice on bypassing the seeding of other generators (for example the one in the Java JCE) has appeared on other mailing lists. The practical experience provided by cases such as the ones given above shows how dangerous it is to rely on users to correctly initialise a generator — not only will they not perform it correctly, they'll go out of their way to do it wrong. Although there is nothing much wrong with the SSLeay/OpenSSL generator itself, the fact that its design assumes that users will initialise it correctly means that it (and many other user-seeded generators) will in many cases not function as required.

If a security-related problem is difficult for a crypto developer to solve, there is no way a non-crypto user can be expected to solve it. Don't leave hard problems as an exercise for the user.

In the above case the generator should handle not only the PRNG step but also the entropy-gathering step itself, while still providing a means of accepting user optional entropy data for those users who do bother to initialise the generator correctly. As a generalisation, crypto software should not leave difficult problems to the user in the hope that they can somehow miraculously come up with a solution where the crypto developer has failed.

3.7 This Function can Never Fail

A few years ago a product was developed which employed the standard technique of using RSA to wrap a symmetric encryption key such as a triple DES key which was then used to encrypt the messages being exchanged (compare this with the RSA usage described in section 3.5). The output was examined during the pre-release testing and was found to be in the correct format, with the data payload appropriately encrypted. Then one day one of the testers noticed that a few bytes of the RSA-wrapped key data were the same in each message. A bit of digging revealed that the key parameters being passed to the RSA encryption code were slightly wrong, and the function was failing with an error code indicating what the problem was. Since this was a function which couldn't fail, the programmer hadn't checked the return code but had simply passed the (random-looking but unencrypted) result on to the next piece of code. At the receiving end, the same thing occurred, with the unencrypted symmetric key being left untouched by the RSA decryption code. Everything appeared to work fine, the data was encrypted and decrypted by the sender and receiver, and it was only the eagle eyes of the tester which noticed that the key being used to perform the encryption was sitting in plain sight near the start of each message.

Another example of this problem occurred in Microsoft Internet Information Server (IIS), which tends to fail in odd ways under load, a problem shared with MS Outlook, which will quietly disable virus scanning when the load becomes high enough so that as much as 90% of incoming mail is never scanned for viruses [56]. In this case the failure was caused by a race condition in which one thread received and decrypted data from the user while a second thread, which used the same buffer for its data, took the decrypted data and sent it back to the user. As a result, when under load IIS was sending user data submitted over an SSL connection back to the user unencrypted [57][58]. The fix was to use two buffers, one for plaintext and one for ciphertext, and zero out the ciphertext buffer between calls. As a result, when the problem occurred, the worst which could happen was that the other side was sent an all-zero buffer [9].

To avoid problems of this kind, implementations should be designed to fail safe even if the caller ignores return codes. A straightforward way to do this is to set output data to a non-value (for example fill buffers with zeroes and set numeric or boolean values to –1) as the first operation in the function being called, and to move the result data to the output as the last operation before returning to the caller on successful completion. In this way if the function returns at any earlier point with an error status, no sensitive data will leak back to the caller, and the fact that a failure has taken place will be obvious even if the function return code is ignored.

Make security-critical functions fail obviously even if the user ignores return codes.

Another possible solution is to require that functions use handles to state information (similar to file or BSD sockets handles) which record error state information and prevent any further operations from occurring until the error condition is explicitly cleared by the user. This error-state-propagation mechanism helps make the fact that an error has occurred more obvious to the user, even if they only check the return status at the end of a sequence of function calls, or at sporadic intervals.

3.8 Careful with that Axe, Eugene

The functionality provided by crypto libraries constitute a powerful tool. However, like other tools, the potential for misuse in inexperienced hands is always present. Crypto protocol design is a subtle art, and most users who cobble their own implementations together from a collection of RSA and 3DES code will get it wrong. In this case "wrong" doesn't refer to (for example) missing a subtle flaw in Needham-Schroeder key exchange, but to errors such as using ECB mode (which doesn't hide plaintext data patterns) instead of CBC (which does).

The use of ECB mode, which is simple and straightforward and doesn't require handling of initialisation vectors (IVs) and block chaining and synchronisation issues is depressingly widespread among users of basic collections of encryption routines, despite this being warned against in every crypto textbook. Confusion over block cipher chaining modes is a significant enough problem that several crypto libraries include FAQ entries explaining what to do if the first 8 bytes of decrypted data appear to be corrupted, an indication that the IV wasn't set up properly.

As if the use of ECB itself wasn't bad enough, users often compound the error with further implementation simplifications. For example one vendor chose to implement their VPN using triple DES in ECB mode, which they saw as the simplest to implement since it doesn't require any synchronisation management if packets are lost. Since ECB mode can only encrypt data in multiples of the cipher block size, they didn't encrypt any leftover bytes at the end of the packet. The interaction of this processing mechanism with interactive user logins, which frequently transmit the user name and password one character at a time, can be imagined by the reader.

The issue which needs to be addressed here is that the average user hasn't read any crypto books, or has at best had some brief exposure to portions of a popular text such as Applied Cryptography, and simply isn't able to operate complex (and potentially dangerous) crypto machinery without any real training. The solution to this problem is for developers of libraries to provide crypto functionality at the highest level possible, and to discourage the use of low-level routines by inexperienced users. The job of the crypto library should be to protect users from injuring themselves (and others) through the misuse of basic crypto routines.

Instead of "encrypt a series of data blocks using 3DES with a 192-bit key", users should be able to exercise functionality such as "encrypt a file with a password", which (apart from storing the key in plaintext in the Windows registry) is almost impossible to misuse. Although the function itself may use an iterated HMAC hash to turn the password into a key, compress and MAC the file for space-efficient storage and integrity-protection, and finally encrypt it using (correctly-implemented) 3DES-CBC, the user doesn't have to know (or care) about this.

Provide crypto functionality at the highest level possible in order to prevent users from injuring themselves and others through misuse of low-level crypto functions with properties they aren't aware of.

4. Conclusion

Although snake oil crypto is rapidly becoming a thing of the past, its position is being taken up by a new breed of snake oil, naugahyde crypto, which misuses good crypto in a manner which makes it little more effective than the more traditional snake oil. This paper has covered some of the more common ways in which crypto and security software can be misused by users. Each individual problem area is accompanied (where possible) by guidelines for measures which can help combat potential misuse, or at least warn developers that this is an area which is likely to cause problems with users. It is hoped that this work will help reduce the incidence of naugahyde crypto in use today.

Unfortunately the single largest class of problems, key management, can't be solved as easily as the other ones. Solving this extremely hard problem in a manner practical enough to ensure uses won't bypass it for ease-of-use or economic reasons will require a multi-faceted approach involving better key management user interfaces, user-certified or provided keys of the kind used by PGP, application-specific key management such as that used with ssh, and a variety of other approaches [59]. Until the key management task is made much more practical, "solutions" of the kind presented in this paper will continue to be widely employed.

5. References

[1] "The Design of a Cryptographic Security Architecture", Peter Gutmann, *Proceedings of the 1999 Usenix Security Symposium*, August 1999, p.153.

[2] "cryptlib Security Toolkit", `http://www.-cs.auckland.ac.nz/~pgut001/-cryptlib/`.

[3] "Why Cryptosystems Fail", Ross Anderson, *Proceedings of the ACM Conference on Computer and Communications Security*, 1993, p.215.

[4] "Why Cryptosystems Fail", Ross Anderson, *Communications of the ACM*, **Vol.37**, **No.11** (November 1994), p.32.

[5] "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0", Alma Whitten and J.D.Tygar, *Proceedings of the 1999 Usenix Security Symposium*, August 1999, p.169.

[6] "Secrets and Lies", Bruce Schneier, John Wiley and Sons, 2000.

[7] "Building Secure Software", John Viega and Gary McGraw, Addison-Wesley, 2001.

[8] "Security Engineering", Ross Anderson, John Wiley and Sons, 2001.

[9] "Writing Secure Code", Michael Howard and David LeBlanc, Microsoft Press, 2001.

[10] "Linux Security", Ramón Hontañón, Sybex, 2001.

[11] "How to break Netscape's server key encryption", Peter Gutmann, posting to the cypherpunks mailing list, message-ID `84366802803808@-cs26.cs.auckland.ac.nz`, 25 September 1996.

[12] "How to break Netscape's server key encryption - Followup", Peter Gutmann, posting to the cypherpunks mailing list, message-ID `84373168812186@cs26.cs.auckland.-ac.nz`, 26 September 1996.

[13] "PFX: Personal Information Exchange Syntax and Protocol Standard", version 0.019, Microsoft Corporation, 1 September 1996.

[14] "PFX: Personal Information Exchange APIs", version 0.019, Microsoft Corporation, 1 September 1996.

[15] ""PFX: Personal Information Exchange Syntax and Protocol Standard", version 0.020, Microsoft Corporation, 27 January 1997.

[16] "PFX — How Not to Design a Crypto Protocol/Standard", Peter Gutmann, `http://-www.cs.auckland.ac.nz/~pgut001/-pubs/pfx.html`.

[17] "Personal Information Exchange Syntax Standard", PKCS #12, RSA Laboratories, 24 June 1999.

[18] "VeriSign Certification Practice Statement (CPS), Version 2.0", Verisign, 31 August 2001.

[19] "How to recover private keys for Microsoft Internet Explorer, Internet Information Server, Outlook Express, and many others — or — Where do your encryption keys want to go today?", Peter Gutmann, `http://www.cs.-auckland.ac.nz/~pgut001/pubs/-breakms.txt`, 21 January 1998.

[20] "How to recover private keys for various Microsoft products", Peter Gutmann, posting to the cryptography@c2.net mailing list, message-ID `88531016604880@cs26.cs.auckland.-ac.nz`, 21 January 1998.

[21] "An update on MS private key (in)security issues", Peter Gutmann, posting to the cryptography@c2.net mailing list, message-ID `88650938015887@cs26.cs.auckland.-ac.nz`, 4 February 1998.

[22] "PGP User's Guide, Volume I: Essential Topics", Philip Zimmermann, 11 October 1994.

[23] "Re: Purpose of PEM string", Doug Porter, posting to pem-dev@tis.com mailing list, message-ID `93Aug16.003350pdt.13997-2@well.sf.ca.us`, 16 August 1993.

[24] "Intercepting Mobile Communications: The Insecurity of 802.11", Nikita Borisov, Ian Goldberg, and David Wagner, *Proceedings of the 7th Annual International Conference on Mobile*

Computing and Networking (Mobicom 2001), 2001.

[25] "ssmail: Opportunistic Encryption in sendmail", Damian Bentley, Greg Rose, and Tara Whalen, *Proceedings of the 13th Systems Administration Conference (LISA '99)*, November 1999, p.1.

[26] "A Security Risk of Depending on Synchronized Clocks", Li Gong, *Operating Systems Review*, **Vol.26**, **No.1** (January 1992), p.49.

[27] "A Note on the Use of Timestamps as Nonces", B.Clifford Neuman and Stuart Stubblebine, *Operating Systems Review*, **Vol.27**, **No.2** (April 1993), p.10.

[28] "Limitations of the Kerberos Authentication System", Steven Bellovin and Michael Merritt, *Proceedings of the Winter 1991 Usenix Conference*, 1991, p.253.

[29] "Systematic Design of Two-Party Authentication Protocols", Ray Bird, Inder Gopal, Amir Herzberg, Philippe Janson, Shay Kutten, Refik Molva, and Moti Yung, *Advances in Cryptology (Crypto'91)*, Springer-Verlag Lecture Notes in Computer Science No.576, August 1991, p.44.

[30] "KryptoKnight Authentication and Key Distribution System", Refik Molva, Gene Tsudik, Els Van Herreweghen and Stefano Zatti, *Proceedings of the 1992 European Symposium on Research in Computer Security (ESORICS'92)*, Springer-Verlag Lecture Notes in Computer Science No.648, 1992, p.155.

[31] "The KryptoKnight Family of Light-Weight Protocols for Authentication and Key Distribution", Ray Bird, Inder Gopal, Amir Herzberg, Philippe Janson, Shay Kutten, Refik Molva, and Moti Yung, *IEEE/ACM Transactions on Networking*, **Vol.3**, **No.1** (February 1995), p.31.

[32] "Network Security", Charlie Kaufman, Radia Perlman, and Mike Speciner, Prentice-Hall, 1996.

[33] "Yaksha: Augmenting Kerberos with Public Key Cryptography", Ravi Ganesan, *Proceedings of the 1995 Symposium on Network and Distributed System Security (NDSS'95)*, February 1995, p.132.

[34] "The Yaksha Security System", Ravi Ganesan, *Communications of the ACM*, **Vol.39**, **No.3** (March 1996), p.55.

[35] "The Kerberos Network Authentication Service (V5)", RFC 1510, John Kohl and B.Clifford Neuman, September 1993.

[36] "Jonah: Experience Implementing PKIX Reference Freeware", Mary Ellen Zurko, John Wray, Ian Morrison, Mike Shanzer, Mike Crane, Pat Booth, Ellen McDermott, Warren Marcek, Ann Graham, Jim Wade, and Tom Sandlin, *Proceedings of the 1999 Usenix Security Symposium*, 1999, p.185.

[37] "Phase II Bridge Certification Authority Interoperability Demonstration Final Report", A&N Associates Inc, prepared for the Maryland Procurement Office, 9 November 2001.

[38] "Internet X.509 Public Key Infrastructure: Time-Stamp Protocol (TSP)", RFC 3161, Carlisle Adams, Pat Cain, Denis Pinkas, and Robert Zuccherato, August 2001.

[39] "Re: A history of Netscape/MSIE problems", Phillip Hallam-Baker, posting to the cypherpunks mailing list, message-ID `3238962F.1372@-ai.mit.edu`, 12 September 1996.

[40] "Re: Problem Compiling OpenSSL for RSA Support", David Hesprich, posting to the openssl-dev mailing list, 5 March 2000.

[41] "Re: "PRNG not seeded" in Window NT", Pablo Royo, posting to the openssl-dev mailing list, 4 April 2000.

[42] "Re: PRNG not seeded ERROR", Carl Douglas, posting to the openssl-users mailing list, 6 April 2001.

[43] "Bug in 0.9.5 + fix", Elias Papavassilopoulos, posting to the openssl-dev mailing list, 10 March 2000.

[44] "Re: setting random seed generator under Windows NT", Amit Chopra, posting to the openssl-users mailing list, 10 May 2000.

[45] "1 RAND question, and 1 crypto question", Brian Snyder, posting to the openssl-users mailing list, 21 April 2000.

[46] "Re: unable to load 'random state' (OpenSSL 0.9.5 on Solaris)", Theodore Hope, posting to the openssl-users mailing list, 9 March 2000.

[47] "RE: having trouble with RAND_egd()", Miha Wang, posting to the openssl-users mailing list, 22 August 2000.

[48] "Re: How to seed before generating key?", 'jas', posting to the openssl-users mailing list, 19 April 2000.

[49] "Re: "PRNG not seeded" in Windows NT", Ng Pheng Siong, posting to the openssl-dev mailing list, 6 April 2000.

[50] "Re: Bug relating to /dev/urandom and RAND_egd in libcrypto.a", Louis LeBlanc, posting to the openssl-dev mailing list, 30 June 2000.

[51] "Re: Bug relating to /dev/urandom and RAND_egd in libcrypto.a", Louis LeBlanc, posting to the openssl-dev mailing list, 30 June 2000.

[52] "Re: PRNG not seeded ERROR", Carl Douglas, posting to the openssl-users mailing list, 6 April 2001.

[53] "Error message: random number generator:SSLEAY_RAND_BYTES / possible solution", Michael Hynds, posting to the openssl-dev mailing list, 7 May 2000.

[54] "Re: Unable to load 'random state' when running CA.pl", Corrado Derenale, posting to the openssl-users mailing list, 2 November 2000.

[55] "OpenSSL Frequently Asked Questions", `http://www.openssl.org/support/-faq.html`.

[56] "Re: Announcing Public Availability of NoHTML for Outlook 2000/2002", Vesselin Bontchev, posting to the ntbugtraq@listserv.-ntbugtraq.com mailing list, message-ID `5.1.0.14.0.20011217140633.-035a3a60@127.0.0.1`, 17 December 2001.

[57] "IIS 4.0 SSL ISAPI Filter Can Leak Single Buffer of Plaintext (Q244613)", Microsoft Knowledge Base article Q244613, 17 April 2000.

[58] "Patch Available for Windows Multithreaded SSL ISAPI Filter Vulnerability ", Microsoft Security Bulletin MS99-053, 2 December 1999.

[59] "PKI: It's Not Dead, Just Resting", Peter Gutmann, to appear.