Abstract

This document is a reproduction of a rant that Peter Gutmann had in late 1997 regarding PFX/PKCS #12. Please note that it was meant as *satire* (smileys don't reproduce well in LaTeX), so no one should take it personally. So there.

PFX - How Not to Design a Crypto Protocol/Standard

Peter Gutmann
pgut001@cs.auckland.ac.nz

Late 1997 Sometime

1 Introduction

This document was originally intended to be a companion to my X.509 style guide, containing various hints and tips on how best to implement PFX/PKCS #12. However after trying to read it several times over, I've come to the conclusion that if this came from anyone but Microsoft, it would probably be regarded as some kind of deliberate sabotage attempt on crypto PDU design. After a week or so of not being able to bring myself to touch it I'd think "It can't be that bad, it just can't be that bad", and then go back and start reading again and find that it really *was* that bad.

As it turns out, because PFX is so comprehensively broken it's far easier to take the style guides "try and do this to demonstrate good style" and turn it around into PFX's "do this to demonstrate bad style". As a result, I've decided to do a rant instead of a proper discussion like the style guide. Rants are far more fun to write anyway.

So, here's the PFX anti-style guide, or "How not to design a crypto protocol/standard".

2 Confusion at a High Level

Your first task is to ensure that things are as confused as possible at the highest, most abstract level. To create this effect, try any or all of the following:

- Make sure you use gratuitously nonstandard terminology for everything. For example, everyone knows what a "message digest" is, and

there are well-defined standards and formats for it. Invent a new term like "thumbprint" and use that instead. The term "PDU" is well-defined and (at least within ISO-related work) widely used. Ignore it and call it a "bag" instead, or, better yet, mix the two up so that sometimes you use "bag" and sometimes you use "PDU", with no clear distinction. Create a list of definitions which defines "PDU" (which everyone knows anyway) but not "bag".

- Mix the 1988 and 1994 variants of ASN.1 and use elements of the 1988 version which were dropped in the 1994 version. This means that neither a 1988 nor a 1994 ASN.1-conformant parser can process the syntax (in fact if you're clever enough with the PDU design you can come up with something which forces the use of a hand-coded parser, as I'll show further on).

- Design the format in such a way that there is a considerable amount of confusion about what it's actually intended to do. For example what 99.9% of users need is a simple format to all them to securely stash their private key, and nothing more. Give them this, but also include the ability to store certificates, CRL's, keys, names, cookie recipes, dirty laundry, spare lightbulbs, and the kitchen sink. Noone will know what use any of this junk is, but everyone will have to add support for it in case someone else implements and (shudder) uses it (see also the comment in the "Confusion in the Details" section).

3 Confusion through Misuse of Standards

Your next avenue for mis-designing a crypto standard is to abuse the standards on which your one is based. For example:

- By far the most widely-used character set, and the one which is easiest to work with for most implementors, is ASCII (IA5String) or latin-1 (T61String). Coming in a rather distant second is Unicode (BMPString), which has somewhat patchy support on most systems and is often difficult to convert into anything useful. Allow only Unicode strings wherever text strings are used. This has the added advantage that no 1988-level ASN.1 compiler can handle the ASN.1.

- Extend this by including the null terminator as part of the Unicode string, in violation of the ASN.1 standard. This will make it impossible

for anyone to create an implementation which interoperates with yours without reverse-engineering your implementation.

- Crypto protocols typically require the use of a large number of byte (or octet) strings. Encode these as bit strings.

4 Confusion in the Details

Now that you've done all this, you can get down to the low-level details which make a protocol difficult or impossible to implement. Among the tricks you can use are:

- When you have an optional sequence, don't make the sequence itself optional, but instead make all its elements optional. So instead of:

```
        ...
        foo     SEQUENCE OPTIONAL,
        ...
```

  you'd use:

```
        ...
        foo     SEQUENCE,
        ...

foo ::= SEQUENCE {
        bar     OPTIONAL,
        baz     OPTIONAL,
        blem    OPTIONAL,
        blort   OPTIONAL,
        blarm   OPTIONAL
        }
```

- Confuse the OIDs so that they are mis-assigned, ambiguous, or multiply assigned. If you're really clever you can ensure that even within the US the only interoperable security level anyone can achieve is 40-bit RC2.

- As an obvious extensions of the above, provide a whole raft of semi-redundant OIDs all of which do more or less the same thing but

which are all subtly incompatible. By providing things like pkcs-12-PBEWithSha1And128BitRC4, pkcs-12-PBEWithSha1And40BitRC4, pkcs-12-PBEWithSha1AndRC4 (none of which are the same, duplicate this confusion for the other proprietary ciphers like RC2 but omit anything useful like IDEA or Blowfish), pkcs-12-PBEWithSomeHashFunctionOrOtherAndACiphe and assorted similar OIDs, you can ensure that the only level at which anyone can ever interoperate is good old 40-bit RC2, just like the US government would want.

- As a further extension to the above extension, hardcode the hash algorithm into the OIDs to ensure that if the hash is ever broken, every single implementation and piece of data "protected" by it will also be broken and in need of replacement. If you're lucky this won't happen until after the first 100 million or so copies have been deployed, allowing you to really maximise the amount of damage caused.

- Standards like PKCS #7 have been designed with a reasonable level of care to allow one-pass processing. For example hashed and signed data has the hash algorithm information before the data so you know you should hash the data as you process it, and the actual hash value at the end.

  To avoid this, provide the ability to tag nasty surprises onto the end of the data with no warning that they're going to be there, so that the implementation has to go back to the start of the data and process it a second time.

- To extend the comment about designing the format to allow a confusing number of PDU's and other crufties, set up the PDU's so that the individual elements haven't been so much assembled into a PDU as sprayed at random across the standard. Include lots of optional, oddball octet strings (all encoded as bit strings of course, see above) with ill-defined uses and scatter them throughout the standard, occasionally tagged with OIDs, some of which are also used to identify password-based stream ciphers described somewhere else in the standard. If you're really careful with this, you can ensure that it takes a dozen or more steps (where each step can involve a considerable amount of work) to do something as simple as recovering a private key.

- Add redundant ASN.1 tags all over the place. For example adding explicit [0] tags to elements in an ASN.1 sequence will add another

level of nesting and bloat up the encoding without actually providing any benefit. Since these elements are always tagged explicitly, you can also bloat up the definition itself by adding a redundant EXPLICIT keyword whenever one of these tags occurs.

- Finally, whatever you do, don't provide any test vectors. If it was difficult to create a mess on this scale, it should be difficult to implement as well, dammit! This way you're guaranteed to see a bare minimum of three incompatible implementations: Microsofts guess at what the format is, Netscapes guess at what the format is, and everyone elses guess at what the format is, with software which reads the PDU's needing to perform complex analyses of the encoding to try and figure out whose guess it's actually looking at.

5 Confusion via Corporate Politics

This one is rather tricky and will require some careful manoeuvering to get right, but the resulting effect on security and interoperability can be devastating:

- Convince one or more large corporations, especially ones notorious for not being able to implement even the simplest protocol (for example SMTP) correctly, to hack together an implementation in an afternoon or so and then start shipping it in their beta products. Once this buggy, incorrect implementation is in the hands of the developers, it'll be too late to change it, requiring anyone who wants to interoperate to create a complex chameleon of coding which changes its performance and behaviour depending on whose incorrect implementation it thinks it's up against.

- If you're one of the large corporations, don't perform any interoperability testing with anyone else. You're bigger than they are, and if noone can read what your software is spitting out, it's their problem, not yours. In your press releases, define yourself to be in full compliance with the standard.

There you have it, how not to design a crypto protocol. If you follow these guidelines to the letter, you can virtually guarantee an endless amount of confusion, implementation problems, and relying on the lowest common denominator level of (in-)security when your substandard is fielded.

Finally, a modest proposal: Rename this whole abortion to "PKCS #13" and get an IETF group to start again from scratch and do PKCS #12 properly. Failing that, an update of the (now rather dated and limited) PKCS #5 format would probably have more or less the same effect.