**How (not) to use symmetric encryption**

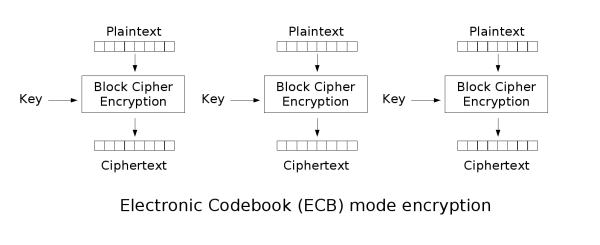
“How (not) to use symmetric encryption” http://blog.cryptographyengineering.com/2011/11/how-not-to-use-symmetric-encryption.html

[](http://cache.blippitt.com/wp-content/uploads/2009/12/pepsi-machine-fail.jpg)This is supposed to be a blog about cryptographic engineering. That means crypto, but it also means software engineering, protocol design, HCI and hardware engineering (fair warning: my hardware experience mostly consists of solder burns).  
  
And yet, in describing the attacks of the past few weeks, I've mostly been talking about basic *symmetric encryption*. This seems to be an area that people get wrong, no matter how straightforward it sounds.  
  
So at the risk of boring my readership to tears -- I'm going to spend the next two posts talking about the*dos and don'ts*of symmetric encryption, particularly symmetric encryption with [block ciphers](http://en.wikipedia.org/wiki/Block_cipher). I may also branch out a little into key derivation and management, but I know you'll be understanding.  
  
I realize that for some of you this may not make for scintillating reading, but here's my thinking: we do it once now, we'll never have to discuss it again. Right?  
  
Excellent. In this first post I'm going to start with the *don'ts.*

**Symmetric Encryption Don't #1: Don't encrypt with ECB mode**

Block ciphers are designed to process discrete chunks of data. For example, AES works on 128-bit blocks. To encrypt longer messages with them, we use one of several "[modes of operation](http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation)". These modes are not all created equal.

|  |
| --- |
| <http://upload.wikimedia.org/wikipedia/commons/f/f0/Tux_ecb.jpg> |
| Tux image: [Larry Ewing](mailto:lewing@isc.tamu.edu). (I will *not* thank the GIMP, no matter what his license says.) |

[](http://upload.wikimedia.org/wikipedia/commons/c/c4/Ecb_encryption.png)

ECB (Electronic Codebook) mode is by far the stupidest. Essentially you're just chopping the message up into blocks, then using the raw block cipher to encipher each block individually. There are two problems with ECB mode:

1. It's not randomized. This means that anytime you encrypt a given message under a key, you'll get exactly the same ciphertext. This goes for *substrings* of the message as well.
2. It treats each block of the message independently. As a result, some of the *structure* of the message can leak through. This includes things like padding, which will produce predictable patterns in the ciphertext.

The first point can be a problem in some circumstances. Imagine, for example, that you're sending a relatively small number of messages (*e.g.,*commands for a remote system). Every time you send a given command, you're sending exactly the same ciphertext. This gets obvious pretty fast.

I would say that the *second* problem is the more serious one. Perhaps you've seen Wikipedia's [classic image](http://upload.wikimedia.org/wikipedia/commons/f/f0/Tux_ecb.jpg) of an ECB-mode encrypted TIFF file (right). But probably this seemed a little contrived to you -- after all, who uses TIFF files anymore?

So allow me to give *my* favorite example of why ECB mode is problematic. This image comes from a software packaging system that used ECB mode to encrypt executable files. All I've done is open one of those encrypted files as a raw bitmap image. You'll have to squint a little.

|  |
| --- |
| [http://3.bp.blogspot.com/-nEkXO89eR-o/TtG2tNoWcJI/AAAAAAAAAHY/utKegIv-wGI/s400/software_ecb.png](http://3.bp.blogspot.com/-nEkXO89eR-o/TtG2tNoWcJI/AAAAAAAAAHY/utKegIv-wGI/s1600/software_ecb.png) |
| An executable file encrypted using ECB mode. Click to see a larger version. |

This doesn't give away the contents of the executable, but it gives a pretty good picture of where different segments are. Just look for the funny patterns and tire tracks. Just having this little bit of information might give you a nice head start on finding those segments when they're in memory, which is potentially what you're going to do next.

**Symmetric Encryption Don't #2: Don't re-use your IVs**

Every block cipher mode of operation except for ECB (which you shouldn't use!) employs a special per-message nonce called an [Initialization Vector](http://en.wikipedia.org/wiki/Initialization_vector), or IV. The basic purpose of an IV is to ensure that the encryption function works differently every time; it adds an element of randomness, or at least*unpredictability*to your ciphertexts.

Unfortunately, developers seem genuinely stumped by IVs. Maybe this isn't their fault. Every mode of operation has slightly different rules about how to pick IVs, and a slightly different set of consequences for when you screw it up.

So let's start with something simple. No matter what mode you use, this kind of thing is *never* ok:

|  |
| --- |
| [http://4.bp.blogspot.com/-dfE60W6CQI0/TtJ6onM_jII/AAAAAAAAAHg/gkQCrDks2XY/s400/ReusingIVs.png](http://4.bp.blogspot.com/-dfE60W6CQI0/TtJ6onM_jII/AAAAAAAAAHg/gkQCrDks2XY/s1600/ReusingIVs.png) |

This chunk of bad advice comes from an ancient (and hopefully obsolete) version of the [AACS](http://www.aacsla.com/home)specification. But it's hardly the exception. Grep for "IV" in the source repositories of just about any major software house, and I guarantee you'll find plenty of stuff like this.

Why is this a problem? Let me count the ways:

1. At a minimum, it eliminates any random behavior in the encryption scheme. With a fixed IV, a given message will *always* encrypt to the same ciphertext (if you're using the same key). This goes for two messages that are the same *up to a certain point.*See my discussion of ECB mode above for why this can be a problem.
2. If you're using a stream cipher mode of operation like [CTR](http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation#Counter_.28CTR.29) or [OFB](http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation#Output_feedback_.28OFB.29), it's a disaster. If you encrypt two different messages with the same key, *and* the IV is fixed, then an attacker can XOR two ciphertexts together. This will give them the XOR of the two underlying plaintexts. (Think this will be useless? I doubt it, especially if they're [clever](http://www.cs.jhu.edu/~jason/papers/mason+al.ccs06.pdf).)  
     
   By the way, this kind of thing also happens when people forget to change the IV when [encrypting](http://eprint.iacr.org/2005/007.pdf) multiple *versions* of a file. Don't do that either.
3. If you're using a chaining mode like CBC, use of a fixed IV can still lead to plaintext recovery. See, for example, [this chosen plaintext attack](http://blog.cryptographyengineering.com/2011/09/brief-diversion-beast-attack-on-tlsssl.html) on TLS, which only requires the adversary *know* which IV is being used. This type of attack is pretty tricky to implement, but it's definitely possible.
4. It will make you look stupid and embarrass you when a professional audits your code.

Clearly some of these issues are application-specific. Maybe you don't think anyone will be able to leverage them. You might even be right -- *in this version of the application.* But sooner or later, you or a future developer will bolt on new features, deploy it in the cloud, or make it into a browser applet. When they do, all of these issues will magically go from theoretically vulnerable to stupidly vulnerable.

And people will blame you.

So how do you use IVs correctly? I'll talk about this more in my next post. But if you're really chomping at the bit, my advice is to take a look at the [FIPS specification](http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf) for block cipher modes. (I must warn you, however: please don't operate heavy machinery while reading these documents.)

**Symmetric Encryption Don't #3: Don't encrypt your IVs**

So you've generated your IV correctly, you've *used* it correctly, but now you're hung up on a final question: *what do I do with this darn thing?*  
  
As I've said, IVs make people nervous. People know they're not keys, but they're not ciphertexts either. They wonder: is this an important value? Should I just send it over the wire as it is? Hmm, just to be safe, I'd better encrypt it. Even if I'm wrong, it can't *hurt*.

As this [Reddit commenter](http://www.reddit.com/r/crypto/comments/mzr81/how_not_to_use_symmetric_encryption/c355izg) can attest, what you don't know *can* hurt you.

Here's a simple rule of thumb. IVs are not keys. They're *not* secret. If you've chosen the IV correctly, you can send it along with your ciphertext *in the clear.*You should *authenticate*it (see below), but you should never *encrypt*it.  
  
The worst thing you can do is encrypt your IVs using the same key that you're using to encrypt messages. The absolute worst example is when you're using CTR mode encryption, and you make the mistake of encrypting your IV using ECB mode. When you do this, anyone can XOR the first block of ciphertext with the encrypted IV, and obtain the plaintext of that block.  
  
These problems aren't limited to CTR. My advice: have *faith* in your IVs, and they'll have faith in you.

**Symmetric Encryption Don't #4: Don't forget to authenticate your ciphertexts**

Once upon a time cryptographers looked at [encryption](http://en.wikipedia.org/wiki/Encryption) and [authentication](http://en.wikipedia.org/wiki/Message_authentication_code) as two unrelated operations. Encryption was for protecting the *confidentiality* of your data, and authentication was used to keep people from tampering with it.\*  
  
Nowadays we know that the two are much more tightly linked. You may not care about people tampering with your data, but your encryption scheme just well might. The problem is *active* *attacks.*See [here](http://www.usenix.org/event/woot10/tech/full_papers/Rizzo.pdf) and [here](http://blog.cryptographyengineering.com/2011/10/attack-of-week-xml-encryption.html) for just a couple of examples.  
  
To make a long story short, many of the clever attacks on symmetric encryption schemes these days require an attacker to tamper with ciphertexts, then submit them to be decrypted. Even if the decryptor leaks just a tiny bit of information (*e.g.,*is the padding correctly formatted, is the message properly formatted), that can be enough to gradually peel away the encryption and recover the plaintext.  
  
Obviously you don't want this.  
  
The very elegant solution is to authenticate your ciphertexts, and not just in any willy-nilly fashion. There are basically two approaches that won't lead to heartburn down the road:

1. Best: use an [authenticated mode of operation](http://en.wikipedia.org/wiki/AEAD_block_cipher_modes_of_operation), such as GCM, CCM, OCB or EAX. These modes handle encryption *and*authentication in one go (and they can even authenticate some optional *unencrypted*'associated' data for you). Best yet, they use a single key.
2. Almost as good: first encrypt your message using a secure mode of operation (say, CTR), then compute a [Message Authentication Code](http://en.wikipedia.org/wiki/Message_authentication_code) (*e.g.,*[HMAC-SHA1](http://en.wikipedia.org/wiki/HMAC)) on the resulting ciphertext and its IV. Use two totally different keys to do this. And please, don't forget to MAC the darn IV!

What you should not do is apply the MAC to the plaintext. First of all, this won't necessarily prevent active attacks. Padding oracle attacks, for example, can still be leveraged against a scheme that authenticates the message (but not the padding). Furthermore, even if you MAC the padding, there's still a slim possibility of [timing attacks](http://blog.cryptographyengineering.com/2011/11/bram-cohen-corrects-me.html)against your implementation.

**Symmetric Encryption Don't #5: Don't CBC-encrypt in real time**

Let me use this space to reiterate that there's nothing wrong with CBC mode, provided that you use it correctly*.*The unfortunate thing about CBC mode is that there are many ways to use it *incorrectly*.  
  
Knowing this, you shouldn't be surprised to hear that CBC is the most popular mode of operation.  
  
This 'don't' is really a variant of point #2 above. CBC mode can be *insecure* when an attacker has the ability to submit chosen plaintexts to be encrypted, *and*if the encryption is on a live stream of data where the adversary can see the ciphertext blocks immediately after they come out (this is called 'online' encryption). This is because the adversary may learn the encryption of the previous plaintext block he submitted, which can allow him to craft the next plaintext block in a useful way.  
  
If he can do this, he might be able to take *some other* ciphertext that he's intercepted, maul it, and feed it through the encryptor. This kind of attack is challenging, but given the right circumstances it's possible to decrypt the original message. This attack is called a [blockwise chosen plaintext attack](http://www.springerlink.com/content/7jla1bu02bqw458p/), and it's essentially what the [BEAST](http://blog.cryptographyengineering.com/2011/09/brief-diversion-beast-attack-on-tlsssl.html) attack does.

**Symmetric Encryption Don't #6: Don't share a single key across many devices**

A wise man once said that [a secret is something you tell one other person](http://www.threechordsandthetruth.net/u2lyrics/achtungbaby.php). I'm not sure he realized it, but what he was saying is this: *don't put the same symmetric key into a large number of devices*(or software instances, etc.) *if you want it to remain secret*.  
  
About the fastest way to lose security in a system is to spread a single key broadly and widely. It doesn't matter if you've embedded that key inside of a tamper-resistant chip, buried in a block of solid concrete, and/or placed it in a locked file cabinet with a sign saying '[beware of leopard](http://www.planetclaire.org/quotes/hitchhikers/)'.  
  
The probability of your system being compromised goes up exponentially with each additional copy of that key. If you're doing this in your current design, think hard about not doing it.

**Symmetric Encryption Don't #7: Don't pluralize your keys using XOR**

|  |
| --- |
|  |

This one is really a flavor of #5, but a more subtle and stupid one.

[](http://farm3.staticflickr.com/2155/2750437716_22b6705181.jpg)Key 'pluralization' refers to a process where you obtain multiple distinct keys from a single master key, or 'seed'. Usually this is done using some strong cryptographic function, for example, a [pseudo-random function](http://en.wikipedia.org/wiki/Pseudorandom_function_family).  
  
This happens all over the place. For example: TLS does it to derive separate MAC and encryption keys from a master secret. But an extreme type of pluralization oftenoccurs in large-scale systems that provide unique keys to a large number of users.  
  
Think of a cellular provider distributing [SIM cards](http://en.wikipedia.org/wiki/Subscriber_Identity_Module), for example. A provider could generate millions of random authentication keys, distribute them to individual SIM cards, and then store the whole package in a back-end database. This works fine, but they'd have to store this database and do a lookup everytime someone contacts them to authorize a phone call.  
  
Alternatively, they could start with one short master seed, then *pluralize* to derive each of the SIM keys on demand. This process would take as input the seed plus some auxiliary value (like the subscriber ID), and would output a key for that user. The advantage is that you no longer need to keep a huge database around -- just the tiny initial seed.

This approach is so tempting that sometimes people forget about the 'strong cryptographic function' part, and they derive their keys using tools that *aren't* so secure. For example, they might just XOR the master seed with the subscriber or device ID.  
  
No, you say, nobody could be that dumb. And yet [KeeLoq](http://www.cosic.esat.kuleuven.be/keeloq/keeloq-rump.pdf) was. Millions of cars keys were provisioned with cryptographic keys that were generated this way. It turns out that if you can extract*any one* of those per-car keys, and if you know the device's serial number, you can easily recover the master key -- which means you can break into *any* *other* car.

**Symmetric Encryption Don't #8: Don't use DES or RC4 or @#(\*&@!**

Ok, I said this was mostly going to be about block ciphers. DES fits that category, and I hope you know why not to use it. But [RC4](http://en.wikipedia.org/wiki/RC4) also deserves a special mention just for being the world's most popular dubious stream cipher.

RC4 shows up everywhere. It shows up in [products](http://support.microsoft.com/kb/245030). It shows up in [malware](http://softwarefindings.wordpress.com/2011/01/14/was-rsa-or-rc4-broken-to-decipher-conficker/). Basically, it shows up anywhere someone needed crypto, but was too lazy to download a copy of AES. Hell, I've used it myself -- um, for personal reasons, not for work, mind you.

If you use RC4 correctly, it's probably ok. For now. The problem is twofold. First of all, cryptanalysts are slowly chipping away at it -- sooner or later they're going to make some serious progress.

The second problem is that it's not always used securely. Why not? You might as well ask why meth labs explode at a disproportionate rate. My guess is that the set of people who use caution when mixing volatile chemicals simply doesn't *overlap* well with the set of people who cook methamphetamine. Ditto RC4 and proper usage.

I could waste a lot of time going on about all of this, but instead I'm just going to quote [Thomas Ptacek](http://chargen.matasano.com/chargen/2009/8/27/a-working-theory-about-rc4.html):

*if you see a bespoke crypto design, and it dates from after 2000, and it uses RC4, that’s an audit flag.*

Now if Thomas says this about RC4, what do you think he's going to say about your homebrew protocol based on the Russian [GOST cipher](http://en.wikipedia.org/wiki/GOST_(block_cipher))? That's right: nothing nice. Don't let it happen to you.  
  
**In Summary**  
  
So this is where I leave you. I doubt this list is complete -- I'll try to update it if I think of anything else. At the very least, if we could fix these issues it would knock out a healthy chunk of the silly crypto issues I see on a day to day basis.  
  
Oh, [and a pony too](http://examinedlife.typepad.com/johnbelle/2004/03/if_wishes_were_.html).  
  
Ok, so, I'm a little skeptical that all of these problems will go away that easily, but I'd be content with even just one or two of the points. So if you're designing a new crypto product and could spare a minute just to glance at the above, you would certainly make my day.  
  
*Notes:*  
  
\* Honestly, there was even a certain amount of confusion on this point. If you look at old protocols like  Needham-Schroeder, you'll see that they basically treat encryption as authentication. Don't do this. Most common modes of operation are malleable, meaning that you can mess with the ciphertext, cut and paste different ones, etc.

Posted by [Matthew Green](http://www.blogger.com/profile/05041984203678598124)at [2:04 PM](http://blog.cryptographyengineering.com/2011/11/how-not-to-use-symmetric-encryption.html)