You Don't Want XTS

30 April 2014

This piece is written for software designers, not end-users. If you're an end-user looking for crypto advice: use Truecrypt, use Filevault, use dm-crypt. Also, use PGP, and Tarsnap. Read on only if you're interested in crypto nerdery.

XTS is the de-facto standard disk encryption mode.

Because it's relatively new and high-profile, XTS looks like a desirable general-purpose mode. It isn't. Be wary of applications that claim to use it for anything other than disk encryption.

To see why, you need to understand what disk encryption is, why disk encryption sucks, and how XTS evolved.

A note on terminology:

Disk devices are made of blocks. Block ciphers work on blocks. Unfortunately, they're two different kinds of blocks (a "wide" disk block, and a "narrow" cipher block), and while it's possible to design a wide-block cipher that directly works on disk blocks, nobody does that.

So for clarity, I'll oversimplify: a "sector" is a disk block, and a "block" is a ciphertext block.

Disks Are The Last Thing You Want To Encrypt

Full disk encryption (FDE) is a last-ditch defense.

The idea is, you've accidentally left your (idling!) laptop on a park bench, or had your (idling!) home computer seized by the FBI. Data on the (inert!) encrypted drive is inaccessible to anyone without your key.

This sounds like a powerful capability. It isn't.

Your encrypted drives are either unlocked & usable, or locked & unusable. Locked & unusable isn't a very useful state. You're carrying that disk around, probably because you use it. But while you're using it, FDE can't really protect you. If your laptop is stolen or seized *while you're using it* , your secrets are exposed.

(or maybe even if it's simply powered on)

Absolutely do turn on FDE.

Someday you'll leave a laptop in the passenger seat of your parked car and lose it when someone cinderblocks the window. When that happens, you'll be glad for

the failsafe of locked-at-wakeup.

Just be aware of the limitations.

It's Hard To Encrypt A Disk

Think of sector-level FDE as "simulated hardware encryption". It's an obnoxious engineering problem. It must be transparent to end-users. Transparent to applications. Transparent even to the OS. So:

- FDE is oblivious to content and format. Filesystems pick unintuitive spots on the disk to stash file chunks. FDE doesn't see files , just a disconnected series of fixed-sized chunks of files.

  It's a a cryptographic version of the movie Memento.

- You can't authenticate the data. Authenticating every sector is too expensive. Try to (painfully) authenticate the whole disk and any disk error trashes the disk. Maybe your users enjoy exciting games of chance? Authenticate arbitrary groups of sectors and you can play Russian Roulette with your files.

  In reality, disk encryption schemes simply punt on authentication.

  Encryption without authentication is problematic, not just because attackers can rewrite /bin/ls into a bindshell, but because unauthenticated ciphertext allows attackers to launch chosen-ciphertext attacks.

- The filesystem might at any moment ask for any sector on the device, so we're required to support random access, not just for reading but for modification.

- The physical device imposes rigid format constraints. We get fixed-sized sectors to work with. We could steal some for metadata, but all we can track are sectors, and there are lots of them.

Attackers will of course play every conceivable game with sectors, blocks, ciphertext/block/sector correspondences, cutting and pasting, and leaked information. We'll see shortly how naive encryption schemes fall short against sophisticated attackers.

All this machinery has to work so fast that users perceive no lag.

Modern FDE schemes look sophisticated. But what they really are is a litany of difficult tradeoffs and messy compromises.

Simplistic Ways To Encrypt Sectors

Consider the "big three" block cipher modes: the default (ECB), CBC, and CTR. All three could form the basis of an FDE system.

The Default Mode

Though extremely simple and fast , any cryptographer knows that ECB can't safely encrypt a disk. Any penguins on the disk would be visible in the ciphertext. Attackers could cut and paste blocks and sectors, rewriting /bin/ls into a bindshell. Nothing would bind an encrypted sector to an actual location on the disk.

(chop a sector into blocks, jam each through the cipher core; done!)

CBC

CBC is used in some popular disk encryption software. CBC comes closer to a workable solution, but it too has problems:

(take a sector, figure out an IV, chop it into blocks, chain the

- Secure CBC wants random IVs, but the device offers no place to store them explicitly.

- You can use sector numbers as IVs. That binds ciphertexts to device locations. But then the IVs are predictable; attackers can generate plaintexts that cancel them out.

- If an attacker can stomach 16 bytes of leading garbage (they usually can!), sectors can still be cut and pasted.

- CBC's chaining property gives attackers a surgical modification capability. An attacker could bitflip a JE instruction into a JNE in /usr/bin/sshd.

blocks together)

(To this, Wikipedia adds a "watermarking vulnerability". Rogaway is dismissive. Who am I to question? There's an aftermarket bolt-on for CBC called ESSIV that tries to address this watermarking problem. It's unclear that it's really possible to avoid it; it's a covert channel problem.)

CTR

We can turn a block cipher into a stream cipher, and encrypt the whole disk a byte at a time. But:

- CTR has some of the same problems that CBC does with regards to explicit IVs (here, nonces).

- CTR falls apart if keys and counters repeat. But sectors can be modified in place. Attackers can record different ciphertexts for the same sector and mount statistical attacks.

- CTR is especially malleable. Attacker alterations are bit-granular, without garbling.

You're now up to speed on what standard "straight out of Schneier and Furguson" block crypto has to say about disk encryption.

Tweakable Ciphers

In a perfect world we'd encrypt every block of every sector with its own key:

- All ciphertext would be cryptographically bound to its location.

- If you tried to swap one sector for another, the decrypted result would be garbage.

- The same plaintext wouldn't reveal itself in repeated ciphertext blocks on different parts of the device.

But encrypting every block under its own key poses two problems:

- We need some way of safely generating large numbers of keys. You can't just take a static key and successively increment it.

- Most ciphers burn a bunch of cycles transforming a master key into round keys. We'd like to avoid running these "key schedules" repeatedly and achieve key agility.

Enter "tweakable" ciphers.

A typical cipher is a transformation function, *E(k,d)*, taking two arguments: a key and a block of data.

A tweakable cipher is *E(k,t,d)*, including a third "tweak" argument. The intent of the tweak is explicit variability, for every invocation of the block transform. Think of a tweak like the "salt" in a password hash.

You can derive a tweakable cipher from any block cipher with a variety of tweaking mechanisms (such as universal hashing, or encrypted nonces). A tweakable cipher is higher-level than a cipher core like AES, *but lower level than a block cipher mode*. You use a tweakable cipher to build tweakable modes of operation.

LRW is a good place to start reading about this idea.

Time To Nerd Out On XTS

XTS is a tweaked cipher mode that uses sector numbers and offsets into sectors as tweak inputs.

XTS is derived from Rogaway's XEX cipher. XEX stands for "XOR Encrypt XOR". The kernel of XEX (and thus XTS) is

$$E(X \oplus \Delta) \oplus \Delta$$

… where E() is (typically) AES, X is the data block, $\oplus$ is XOR, and Δ is the tweak. The Δ tweak is defined as:

$$E(i) \cdot \alpha^j$$

… where E() is again AES, *i* is the tweak input, α is a primitive polynomial , and *j* is an offset. For now, think of *i* as a "master tweak" that changes rarely and *j* as a "subtweak".

You don't need to understand the polynomial math. It takes place in a binary field and resembles a CRC, but is even simpler. Just assume you can encrypt a number and multiply it by some other number and you're fine.

That's XEX. Back to XTS:

XTS is a block mode built on top of the XEX cipher. NIST basically took XEX, added some fins to lower wind-resistance, and a pretty-sharp racing stripe.

XTS consumes "wide blocks" (sectors). Sectors are big. XEX works in terms of the "narrow blocks" of the block cipher core. A sector might be 32 or more cipher blocks. XTS is basically ECB mode for the XEX cipher.

So, to get your head around how this works, imagine we're encrypting a whole hard drive:

- We have two AES keys; the "data" AES key (k1) and the "tweak" AES key (k2).
- The disk is made of 512-byte sectors. We work sector by sector, starting at sector 0.
- Each has an XEX "master tweak" *i*, which will be the sector number.
- A sector is 32 16-byte AES-blocks long, and so we'll be invoking XEX 32 times, each with the same *i* but a different *j*.

So, starting at sector 0, we'll be doing:

1. c_0 = AES(k2, 0)

2. *XEX-AES(k1, sec-0-block-0, i=c_0, j=0)*
3. *XEX-AES(k1, sec-0-block-1, i=c_0, j=1)*
4. *XEX-AES(k1, sec-0-block-2, i=c_0, j=2)*
5. *and so on…*
6. *XEX-AES(k1, sec-0-block-31, i=c_0), j=31)*
7. *c_1 = AES(k2, 1)*
8. *XEX-AES(k1, sec-1-block-0, i=c_1, j=0)*
9. *XEX-AES(k1, sec-1-block-1, i=c_1, j=1)*
10. *and so on…*

Observe that *i* increments with the sector, and *j* with the block offset into the sector. *j* changes a lot, *i* changes rarely. The XEX tweak is designed to be particularly fast to compute in successive calls. It minimizes calls to the block cipher core, but still behaves as if every block has its own key.

Again: XTS works like ECB. It's deterministic. If you're looking for the penguins, they're there, but you have to look for them across time instead of space: successive writes to the same sector-block location will repeat, but encryptions of the same plaintext at different locations will be randomized.

There's one more complication to XTS. What happens to partial blocks? What if your sectors are 520 bytes long instead of 512? To handle that case, XTS employs a technique called "ciphertext stealing", which it borrows from CBC mode (where it's called CTS).

CTS is complicated, but more in the "hard to remember all the details" sense than in the "hard to get your head around it sense". You can implement it straight from Wikipedia. Rogaway does a good job explaining why it works.

What's Wrong With XTS?

Two things. Sector-level crypto is a painful problem you want to avoid. And XTS is an imperfect construction.

First, the fiddly problems. XTS is not beloved of cryptographers:

- It's complicated. It's a wide-block tweakable mode built out of a narrow-block tweakable mode, it uses two keys unnecessarily, and it uses ciphertext stealing to handle variable-length inputs. Another way to say "complicated" is "hard to prove correct".

- It's unclear what XTS's goals are. It's ECB-like. It can't do a perfect job of providing privacy. It's not authenticated. Attackers can rewrite plaintexts. It's hard for a cryptographer to know what they'd be trying to prove. "behavesLikeXTS"? OK, and?

- The wide-block-of-narrow-blocks property weakens XTS unnecessarily. As the disk rewrites a given sector, attackers can collect fine-grained (16 byte) ciphertexts. They get to manipulate ciphertexts surgically. It would have been possible to define a "native" wide-block tweakable cipher without that property, but that wouldn't have been as performant.

How much do these problems matter in practice? Probably not much. It's certainly better than ECB, CBC, and CTR for FDE. For the crappy job we ask it to do, XTS is probably up to the task.

But that's the big problem: sector-level encryption sucks. It's messy, provides fewer security guarantees than conventional message encryption, and makes tradeoffs tailored to the challenges of encrypting disk sectors.

Sector-level crypto is last-resort crypto.

That problem is significant enough to be dispositive. If you don't have the sector-level problem, don't use sector-level crypto. So:

- If you're encrypting files, and not disk sectors, don't use XTS. Use an AEAD scheme, like GCM or OCB or Salsa20+Poly1305. If you're working with files, you have format flexibility. Use it. Randomize your encryption. Authenticate it with a proper MAC tag.

- If you're encrypting a filesystem and not disk blocks, still don't use XTS! Filesystems have format-awareness and flexibility. Filesystems can do a much better job of encrypting a disk than simulated hardware encryption can.

- If you're encrypting a filesystem that you're going to host on a cloud service, consider trying to reframe your problem. Sectors or nothing? By all means, stick a Truecrypt volume on Dropbox. But you have better options.

  Remember that disk encryption is designed to counter an attacker with very limited capabilities. That's why it falls to "Evil Maids": the threat model doesn't really accommodate attackers with multiple bytes at the (physical) apple. But whatever margin of safety XTS gets you on physical media probably goes out the window when you stick a Truecrypt volume on Dropbox. From the vantage point of Dropbox, attackers have far more capabilities than the XTS designers planned for.

Simulated hardware encryption sucks. If you free yourself from the idea that you need to encrypt a whole disk, you win a bunch of things:

- Fully randomized encryption, of all your data (not just little chunks of it).

- Complete awareness of the application and format, and the ability to authenticate data in different places.

- Secure, flexible storage of arbitrary metadata.

Encrypt things at the highest layer you can.

Some application crypto problems look suspiciously similar to FDE. Take database encryption ; the persistent backing data for a database might be handled in terms of "pages". Tweaked modes have some promise there. But XTS is probably not a good answer; wide-block authenticated tweaked modes are a better plan.

There are other big problems with database encryption, like online key storage.

If you take away just two things from this post, let it be:

1. Sector-level simulated hardware crypto is a last-resort solution that protects you less than you think it does.

2. The "XTS" label on the tin doesn't signify powerful, advanced crypto. If you see it used for something other than simulated hardware crypto, watch out.

Further Nerdery

From what I can tell, Wikipedia does a pretty bad job of explaining disk crypto. Here's a decent reading list:

- The LRW paper does a great job of introducing tweaked ciphers and modes. Despite what Wikipedia says, LRW isn't really a disk encryption scheme. It's a tweaked cipher design, proposed as the basis for a variety of tweaked modes.
- Rogaway's XEX paper refines LRW with a tweaking construction that is fast in successive calls, which is why it forms the basis for XTS. Read the introduction, then skip to the "Intuition" paragraph.
- Rogaway's PMAC paper introduces some of the math used in the XEX tweak.
- The NIST public comments on XTS are interesting; particularly Ferguson's.
- Truecrypt is the best-known implementation of XTS, and is happily open-source.

Thanks to Matthew Green, Nate Lawson, Sean Devlin, and Tim Newsham for reading drafts, catching errors, and good ideas.