

# A Few Thoughts on Cryptographic Engineering

Some random thoughts about crypto. Notes from a course I teach. Pictures of my dachshunds.

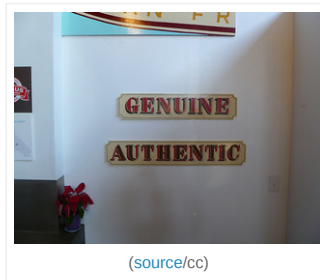
Showing posts sorted by relevance for query **authenticated**. [Sort by date](#) [Show all posts](#)

Saturday, May 19, 2012

## How to choose an Authenticated Encryption mode

If you've hung around this blog for a while, you probably know how much I like to complain. (Really, quite a lot.) You might even be familiar with one of my favorite complaints: *dumb crypto standards*. More specifically: dumb standards promulgated by smart people.

The people in question almost always have justifications for whatever earth-shakingly stupid decision they're about to take. Usually it's something like *'doing it right would be hard'*, or *'implementers wouldn't be happy if we did it right'*. Sometimes it's *'well, we give the option to do it right'*. In the worst case they'll tell you: *'if it bothers you so much, why don't you join the committee and suggest that idea yourself, Mr. Smartypants'*.



(source/cc)

Well, first of all, it's *Dr. Smartypants*. And moreover, I've tried. It doesn't work.

Case in point: I happen to be lurking on the mailing list of a standards committee that recently decided to allow unauthenticated **CBC mode encryption** as an option in their new web encryption standard. When I pointed out that *the exact same decision* led to the failure of a **previous standard** -- ironically, one that this new standard will probably replace -- I was told, politely, that:

1. Mandating authenticated encryption would be hard.
2. Real implementers don't know how to implement it.
3. We already offer the *option* to use authenticated encryption.
4. Stop telling us things we already know.

The worst part: they really *did* know. The committee included some smart, smart people. People who know that this is a bad idea, and who have decided either to just *go with it*, or else have convinced themselves that implementers won't *(a)* pick the easy, insecure option, and then *(b)* screw it up completely. I have news for these people: *Yes, they will*. This is why we write standards.

After all this build-up, it may surprise you that this is not a post about standards committees. It's not even a post about smart people screwing things up. What I'm here to talk about today is Authenticated Encryption, what the hell it is, why *you* need it. And finally, (assuming you're good with all that) *which* of the many, many AE schemes should you consider for your application.

First, some background.

### What's Authenticated Encryption and why should I care?

For those of you who *don't* know what AE is, I first need to explain one basic fact that *isn't* well explained elsewhere:

*Nearly all of the symmetric encryption modes you learned about in school, textbooks, and Wikipedia are (potentially) insecure.*

### About Me



#### Matthew Green

I'm a cryptographer and research professor at Johns Hopkins University. I've designed and analyzed cryptographic systems used

in wireless networks, payment systems and digital content protection platforms. In my research I look at the various ways cryptography can be used to promote user privacy.

My website

[My twitter feed](#)

Useful crypto resources

[RSS](#)

[Bitcoin tipjar](#)

[Matasano challenges](#)

[Journal of Cryptographic Engineering \(not related to this blog\)](#)

[View my complete profile](#)

### Popular Posts



#### On the NSA

Let me tell you the story of my tiny brush with the biggest crypto story of the year. A few weeks ago I received a call from a

reporter a...



#### What's the matter with PGP?

Image source: @bcrypt. Last Thursday, Yahoo announced their plans to support end-to-end encryption using a fork of Google's end-to-...



#### Here come the encryption apps!

It seems like these days I can't eat breakfast without reading about some new encryption app that will (supposedly) revolutionize our c...



#### Attack of the week: FREAK (or 'factoring the NSA for fun and profit')

Cryptography used to be considered 'munitions'. This is the story of how a handful of cryptographers 'hacked' the NSA....

ents), managed books, i  
th keys protected by th  
hmentis), Contacts, Rem  
vent Protected Until Fir  
opt-in to a specific Data  
rication by default.

#### Why can't Apple decrypt your iPhone?

Last week I wrote about Apple's new default encryption policy for iOS 8. Since that piece was intended for general audiences I mostly ...

This covers things like AES when used in standard modes of operation like [CBC](#) and [CTR](#). It also applies to stream ciphers like RC4. Unfortunately, the list of potentially insecure primitives includes many of the common symmetric encryption schemes that we use in practice.

Now, I want to be clear. These schemes are *not* insecure because they leak plaintext information to someone who just intercepts a ciphertext. In fact, most modern schemes hold up amazingly well under that scenario, assuming you choose your keys properly and aren't an idiot.

The problem occurs when you use encryption in *online* applications, where an adversary can intercept, tamper with, and submit ciphertexts to the receiver. If the attacker can launch such attacks, many implementations can fail catastrophically, allowing the attacker to **completely decrypt messages**.

Sometimes these attacks requires the attacker to see only an error message from the receiver. In other cases all he needs to do is measure *time* it takes for the receiver to acknowledge the submission. This type of attack is known as a [chosen ciphertext attack](#), and by far the most common embodiment is the 'padding oracle attack' discovered in 2002 by Serge Vaudenay. But there are others.

The simplest way to protect yourself against these attacks is to simply [MAC](#) your ciphertexts with a secure Message Authentication Code such as [HMAC-SHA](#). If you prefer this route, there are two essential rules:

1. Always compute the MACs on the ciphertext, never on the plaintext.
2. Use two different keys, one for encryption and one for the MAC.

Rule (1) prevents chosen-ciphertext attacks on block cipher modes such as CBC, since your decryption process can reject those attacker-tampered ciphertexts before they're even decrypted. Rule (2) deals with the possibility that your MAC and cipher will interact in some unpleasant way. It can also help protect you against side-channel attacks.

This approach -- encrypting something, then MACing it -- is not only secure, it's *provably* secure as long as your encryption scheme and MAC have certain properties. Properties that most common schemes do seem to possess.\*

### Dedicated AE(AD) modes

Unfortunately, the 'generic composition' approach above is not the right answer for everyone. For one thing, it can be a little bit complicated. Moreover, it requires you to implement two different primitives (say, a block cipher and a hash function for HMAC), which can be a hassle. Last, but *not* least, it isn't necessarily the fastest way to get your messages encrypted.

The efficiency issue is particularly important if you're either (a) working on a constrained device like an embedded system, or (b) you're working on a fast device, but you just need to encrypt lots of data. This is the case for network encryptors, which have to process data at line speeds -- typically many gigabytes per second!

For all of these reasons, we have specialized block cipher modes of operation called Authenticated Encryption (AE) modes, or sometimes Authenticated Encryption with Associated Data (AEAD). These modes handle both the encryption and the authentication in one go, usually with a single key.

AE(AD) modes were developed as a way to make the problem of authentication 'easy' for implementers. Moreover, some of these modes are lightning fast, or at least allow you to take advantage of *parallelization* to speed things up.

Unfortunately, adoption of AE modes has been a lot slower than one would have hoped for, for a variety of reasons. One of which is: it's hard to find good implementations, and another is that there are tons and tons of AE(AD) schemes.

### So, which AE mode should I choose?

And now we get down to brass tacks. There are a plethora of wonderful AE(AD) modes out there, but which one should you use? There are many things to consider. For example:

- How fast is encryption and decryption?
- How complicated is the implementation?
- Are there free implementations out there?

### Attack of the week: OpenSSL Heartbleed

Ouch. (Logo from heartbleed.com ) I start every lecture in my security class by asking the students to give us any interesting security ...

### Dear Apple: Please set iMessage free

Normally I avoid complaining about Apple because (a) there are plenty of other people carrying that flag, and (b) I honestly like Apple ...

### RSA warns developers not to use RSA products

In today's news of the weird, RSA (a division of EMC) has recommended that developers desist from using the (allegedly) 'backdoore...'



### Let's audit Truecrypt!

[ source ] A few weeks ago, after learning about the NSA's efforts to undermine encryption software , I wrote a long post urging d...



### Attack of the week: RC4 is kind of broken in TLS

Update: I've added a link to a page at Royal Holloway describing the new attack. Listen, if you're using RC4

as your primary c...

### My Blog List

**Schneier on Security**  
How the CIA Might Target Apple's XCode  
2 hours ago

**Bristol Cryptography Blog**  
2 days ago

**Bentham's Gaze | Information Security Research, University College London**  
A Digital Magna Carta?  
2 days ago

**Shtetl-Optimized**  
The ultimate physical limits of privacy  
4 days ago

**ellipticnews**  
Soliloquy, ideal lattices, and algorithms  
3 weeks ago

**root labs rdist**  
Was the past better than now?  
3 months ago

**Cryptanalysis**  
SSLv3 considered to be insecure -- How the POODLE attack works in detail  
5 months ago

**The MPC Lounge**  
5th Bar-Ilan Winter School 2015: Advances in Practical Multiparty Computation  
5 months ago

**Chargen**

### Subscribe

Posts

All Comments

### Followers



- Is it widely used?
- Can I parallelize it?
- Is it 'on-line', i.e., do I need to know the message length before I start encrypting?
- Is it patented?
- Does it allow me to include Associated Data (like a cleartext header)?
- What does Matt Green think about it?

To answer these questions (and particularly the most important final one), let's take a look at a few of the common AE modes that are out there. All of these modes support Associated Data, which means that you can pre-pend an *unencrypted* header to your encrypted message if you want. They all take a single key and some form of *Initialization Vector* (nonce). Beyond that, they're quite different inside.

**GCM.** [Galois Counter Mode](#) has quietly become the most popular AE(AD) mode in the field today, despite the fact that everyone hates it. The popularity is due in part to the fact that GCM is extremely fast, but mostly it's because the mode is patent-free. GCM is 'on-line' and can be parallelized, and (best): recent versions of OpenSSL and Crypto++ provide good implementations, mostly because it's now supported as a [TLS ciphersuite](#). As a side benefit, GCM will occasionally visit your house and fix broken appliances.

Given all these great features, you might ask: why does everyone hate GCM? In truth, the only people who hate GCM are those who've had to *implement* it. You see, GCM is CTR mode encryption with the addition of a Carter-Wegman MAC set in a Galois field. If you just went 'sfjshuh?', you now understand what I'm talking about. Implementing GCM is a hassle in a way that most other AEADs are *not*. But if you have someone else's implementation -- say OpenSSL's -- it's a perfectly lovely mode.

**OCB.** In performance terms [Offset Codebook Mode](#) blows the pants off of all the other modes I mention in this post. It's 'on-line' and doesn't require any real understanding of Galois fields to implement\*\* -- you can implement the whole thing with a block cipher, some bit manipulation and XOR. If OCB was your kid, he'd play three sports and be on his way to Harvard. You'd brag about him to all your friends.

Unfortunately OCB is *not* your kid. It belongs to [Philip Rogaway](#), who also happens to hold a patent on it. This is no problem if you're developing GPL software ([it's free for you](#)), but if you want to use it in a commercial product -- or even license under Apache -- you'll probably have to pay up. As a consequence OCB is used in approximately no industry standards, though you might find it in some commercial products.

**EAX.** Unlike the other modes in this section, [EAX mode](#) doesn't even bother to *stand* for anything. We can guess that E is Encryption and A is Authentication, but X? I'm absolutely convinced that EAX is secure, but I cannot possibly get behind a mode of operation that doesn't have a meaningful acronym.

EAX is a two-pass scheme, which means that encryption and authentication are done in separate operations. This makes it much slower than GCM or OCB, though (unlike CCM) it is 'on-line'. Still, EAX has three things going for it: first, it's patent-free. Second, it's pretty easy to implement. Third, it uses only the Encipher direction of the block cipher, meaning that you could technically fit it into an implementation with a very constrained code size, if that sort of thing floats your boat. I'm sure there are EAX implementations out there; I just don't know of any to recommend.

Whatever you do, be sure not to confuse EAX mode with its dull cousin [EAX\(prime\)](#), which ANSI developed only so it could later be [embarrassingly broken](#).

**CCM.** [Counter Mode with CBC MAC](#) is the 1989 Volvo station wagon of AEAD modes. It'll get you to your destination reliably, just not in a hurry. Like EAX, CCM is also a two-pass scheme. Unfortunately, CCM is *not* 'on-line', which means you have to know the size of your message before you start encrypting it. The redeeming feature of CCM is that it's patent-free. In fact, it was developed and implemented in the 802.11i standard (*instead* of OCB) solely because of IP concerns. You can find an implementation in [Crypto++](#).

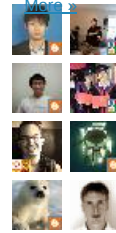
**The rest.** There are a few more modes that almost nobody uses. These include [XCBC](#), [IAPM](#) and [CWC](#). I have no idea why the first two haven't taken off, or if they're even secure. CWC is basically a much slower version of GCM mode, so there's no real reason to use it. And of course, there are probably plenty more that I haven't listed. In general: you should use those at your own risk.

Summing up



Join this site  
with Google Friend  
Connect

Members (188)



Already a member?  
[Sign in](#)

#### Blog Archive

- 2015 (5)
- 2014 (13)
- 2013 (23)
- ▼ 2012 (48)
  - December (1)
  - November (1)
  - October (4)
  - September (3)
  - August (4)
  - July (2)
  - June (3)
  - ▼ May (5)
  - TACK
  - If wishes were horses then beggars would ride... a...
  - How to choose an Authenticated Encryption mode
  - A tale of two patches
  - The future of electronic currency
- April (6)
- March (4)
- February (7)
- January (8)
- 2011 (39)

So where are we?

In general, the decision of which cipher mode to use is *not* something most people make every day, but when you *do* make that decision, you need to make the right one. Having read back through the post, I'm pretty sure that the 'right' answer for most people is to use GCM mode and rely on a ~~trusted~~ free implementation, like the one you can get from OpenSSL.

But there are subcases. If you're developing a commercial product, don't care about cross-compatibility, and don't mind paying 'a [small one-time fee](#)', OCB is also a pretty good option. Remember: even cryptographers need to eat.

Finally, if you're in the position of developing your own implementation from scratch (not recommended!) and you really don't feel confident with the more complicated schemes, you should seriously consider EAX or CCM. Alternatively, just use HMAC on your ciphertexts. All of these things are relatively simple to deal with, though they certainly don't set the world on fire in terms of performance.

The one thing you should not do is say '*gosh this is complicated, I'll just use CBC mode and hope nobody attacks it*', at least not if you're building something that will potentially (someday) be online and subject to active attacks like the ones I described above. There's already enough stupid on the Internet, please don't add more.

Notes:

\* Specifically, your encryption scheme must be IND-CPA secure, which would apply to CBC, CTR, CFB and OFB modes implemented with a secure block cipher. Your MAC must be existentially unforgeable under chosen message attack (EU-CMA), a property that's (believed) to be satisfied by most reasonable instantiations of [HMAC](#).

\*\* An earlier version of this post claimed that OCB didn't use Galois field arithmetic. This [commenter on Reddit](#) correctly points out that I'm an idiot. It does indeed do so. I stand by my point that the implementation is dramatically simpler than GCM.

Posted by [Matthew Green](#) at 6:21 PM 16 comments:

 +4 Recommend this on Google

Thursday, April 24, 2014

## Attack of the Week: Triple Handshakes (3Shake)

The other day Apple released a [major security update](#) that fixes a number of terrifying things that can happen to your OS/X and iOS devices. You should install it. Not only does this fix a possible remote code execution vulnerability in the JPEG parser (!), it also patches a TLS/SSL protocol bug known as the "[Triple Handshake](#)" vulnerability. And this is great timing, since Triple Handshakes are something I've been meaning (and failing) to write about for over a month now.



But before we get there: a few points of order.

First, if Heartbleed taught us one thing, it's that when it comes to TLS vulnerabilities, *branding is key*. Henceforth, and with apologies to Bhargavan, Delignat-Lavaud, Pironti, Fournet and Strub (who actually [discovered the attack](#)), for the rest of this post I will be referring to the vulnerability simply as "3Shake". I've also taken the liberty of commissioning a logo. I hope you like it.

On a more serious note, 3Shake is not Heartbleed. That's both good and bad. It's good because Heartbleed was nasty and 3Shake really isn't anywhere near as dangerous. It's *bad* since, awful as it was, Heartbleed was only an implementation vulnerability -- and one in a single TLS library to boot. 3Shake represents a novel and [fundamental bug](#) in the TLS protocol.

The final thing you should know about 3Shake is that, according to the cryptographic literature, it shouldn't exist.

You see, in the last few years there have been at least ~~three~~ [four major crypto papers](#) purporting to prove the TLS protocol secure. The existence of 3Shake doesn't make those results wrong. It may, however, indicate that cryptographers need to think a bit more about what 'secure' and 'TLS' actually mean. For me, that's the most fascinating implication of this new attack.

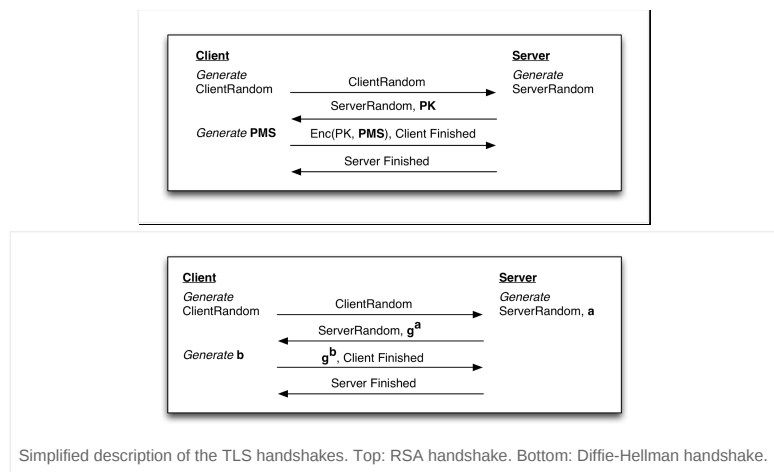
I'll proceed with the usual 'fun' question-and-answer format I save for this sort of thing.

### What is TLS and why should you care?

Since you're reading this blog, you probably already know something about TLS. You might even realize how much of our infrastructure is protected by this crazy protocol.

In case you don't: TLS is a secure transport protocol that's designed to establish communications between two parties, who we'll refer to as the Client and the Server. The protocol consists of two sub-protocols called the *handshake protocol* and the *record protocol*. The handshake is intended to authenticate the two communicating parties and establish shared encryption keys between them. The record protocol uses those keys to exchange data securely.

For the purposes of this blog post, we're going to focus primarily on the handshake protocol, which has (at least) two major variants: the RSA handshake and the Diffie-Hellman handshake (ECDHE/DHE). These are illustrated below.



As much as I love TLS, the protocol is a hot mess. For one thing, it inherits a lot of awful cryptography from its ancient predecessors (SSLv1-3). For another, it's only really beginning to be subjected to [rigorous, formal analysis](#).

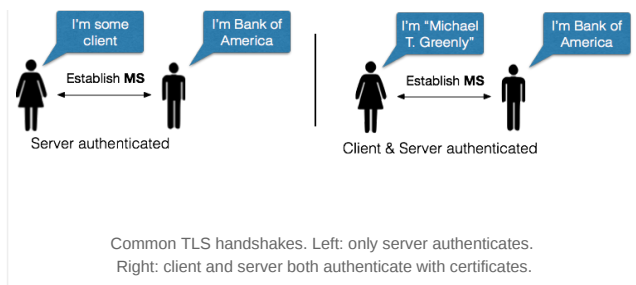
All this means we're just now starting to uncover some of the bugs that have been present in the protocol since it was first designed. And we're likely to discover more! That's partly because this analysis is at a very early stage. It's also partly because, from an analysts' point of view, *we're still trying to figure out exactly what the TLS handshake is supposed to do*.

### Well, what *is* the TLS handshake supposed to do?

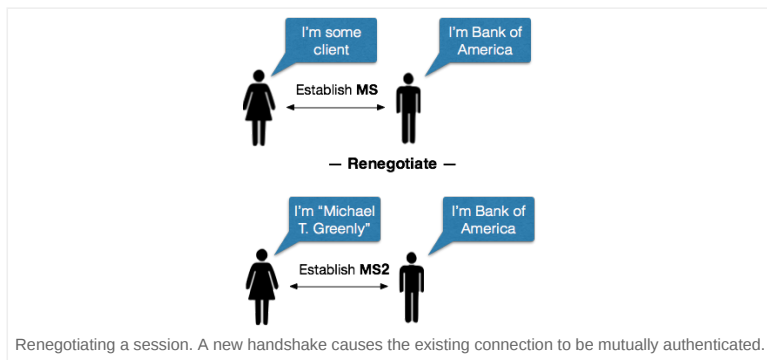
Up until this result, we thought we had a reasonable understanding of the purpose of the TLS handshake. It was intended to authenticate one or both sides of the connection, then establish a shared cryptographic secret (called the Master Secret) that could be used to derive cryptographic keys for encrypting application data.

The first problem with this understanding is that it's a bit too simple. There isn't just *one* TLS handshake, there are several variants of it. Worse, multiple different handshake types can be used within a single connection.

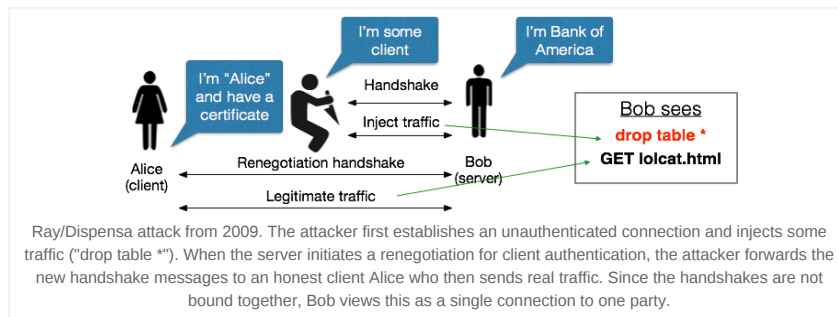
The standard handshake flow is illustrated -- without crypto -- in the diagram below. In virtually every TLS connection, the server authenticates to the client by sending a public key embedded in a certificate. The client, for its part, can *optionally* authenticate itself by sending a corresponding certificate and proving it has the signing key. However this client authentication is by no means common. Many TLS connections are authenticated only in one direction.



TLS also supports a "renegotiation" handshake that can switch an open connection from one mode to the other. This is typically used to change a connection that was authenticated only in one direction (Server->Client) into a connection that's authenticated in both directions. The server usually initiates renegotiation when the client has e.g., asked for a protected resource.



Renegotiation has had problems before. Back in 2009, [Ray and Dispensa showed](#) that a man-in-the-middle attacker could actually establish a (non-authenticated) connection with some server; inject some data; and when the server asks for authentication, the attacker could then "splice" on a real connection with an authorized client by simply forwarding the new handshake messages to the legitimate client. From the server's point of view, both communications would seem to be coming from the same (now authenticated) person:



To fix this, a "secure renegotiation" band-aid to TLS was proposed. The rough idea of this extension was to 'bind' the renegotiation handshake to the previous handshake, by having the client present the "Finished" message of the previous handshake. Since the Finished value is (essentially) a hash of the Master Secret and the (hash of) the previous handshake messages, this allows the client to prove that it -- not an attacker -- truly negotiated the previous connection.

All of this brings us back to the question of *what the TLS handshake is supposed to do*.

You see, the renegotiation band-aid now adds some pretty interesting new requirements to the TLS handshake. For one thing, the security of this extension depends on the idea that (1) no two distinct handshakes will happen to use the same Master Secret, and (2) that no two handshakes will have the same handshake messages, ergo (3) no two handshakes will have the same Finished message.

Intuitively, this seemed like a pretty safe thing to assume -- and indeed, many other systems that do 'channel binding' on TLS connections also make this assumption. The 3Shake attack shows that this is not safe to assume at all.

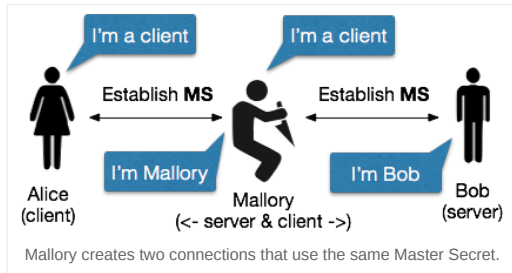
#### So what's the problem here?

It turns out that TLS does a pretty good job of establishing keys with people you've authenticated. Unfortunately there's a caveat. It doesn't truly guarantee the established

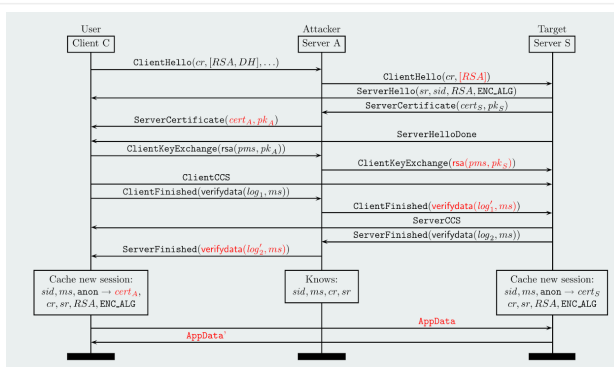


key *will be unique to your connection*. This is a pretty big violation of the assumptions that underlie the "secure renegotiation" fix described above.

For example: imagine that Alice is (knowingly) establishing a TLS connection to a server Mallory. It turns out that Mallory can simultaneously -- and unknown to Alice -- establish a different connection to a second server Bob. Moreover, if Mallory is clever, she can force *both* connections to use the same "Master Secret" (MS).



The first observation of the 3Shake attack is that this trick can be played if Alice supports either of the or RSA and DHE handshakes -- or both (it does not seem to work on ECDHE). Here's the RSA version:\*\*



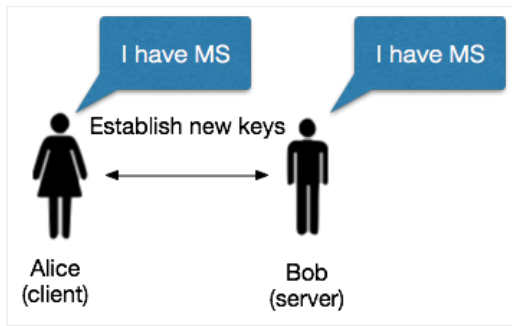
RSA protocol flow from the triple handshake attack ([source](#)). The attacker is in the middle, while the client and server are on the left/right respectively. **MS** is computed as a function of (pms, cr, sr) which are identical in both handshakes.

So already we have a flaw in the logic undergirding secure renegotiation. The Master Secret (MS) values are *not necessarily distinct* between different handshakes.

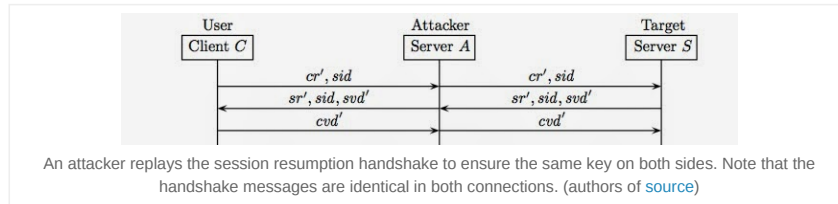
Fortunately, the above attack does not let us resurrect the Ray/Dispensa injection attack. While the attacker has tricked the client into using a specific **MS** value, the handshake Finished messages -- which the client will attach to the renegotiation handshake -- will *not* be the same in both handshakes. That's because (among other things) the certificates sent on each connection were very different, hence the handshake hashes are not identical. In theory we're safe.

But here is where TLS gets awesome.

You see, there is *yet another* handshake I haven't told you about. It's called the "session resumption handshake", and it allows two parties who've previously established a master secret ([and still remember it](#)) to resume their session with new encryption keys. The advantage of resumption is that it uses *no public-key cryptography or certificates at all*, which is supposed to make it faster.

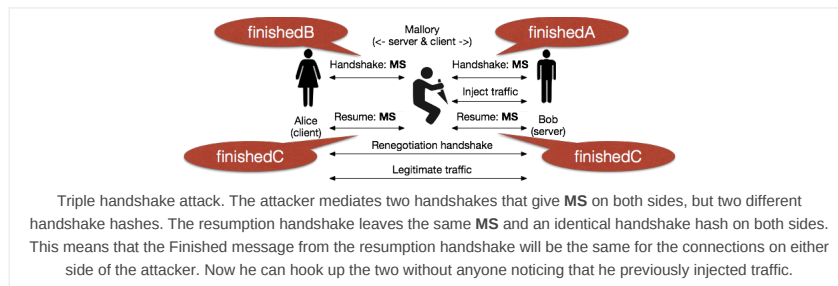


It turns out that if an attacker knows the previous **MS** and has caused it to be the same on both sides, it can now wait until the client initiates a session resumption. Then it can replay messages between the client and server in order to update both connections with new keys:



Which brings us to the neat thing about this handshake. Not only is the **MS** the same on both connections, but both connections now see exactly the same (resumption) handshake messages. Hence the hash of *these* handshakes will be identical, which means in turn that their "Finished" message will be identical.

By combining all of these tricks, a clever attacker can pull off the following -- and absolutely insane -- "triple handshake" injection attack:



In the above scenario, an attacker first runs a (standard) handshake to force both sides of the connection to use the same **MS**. It then causes both sides to perform session resumption, which results in both sides using the same **MS** and having the same handshake hash and Finished messages on both sides. When the server initiates renegotiation, the attacker can forward the *third* (renegotiation) handshake on to the legitimate client as in the Ray/Dispensa attack -- secure in the knowledge that both client and server will expect the same Finished token.

And that's the ballgame.

#### What's the fix?

There are several, and you can read about them [here](#).

One [proposed fix](#) is to change the derivation of the Master Secret such that it includes the handshake hash. This should wipe out most of the attacks above. Another fix is to bind the "session resumption" handshake to the original handshake that led to it.

#### Wait, why should I care about injection attacks?

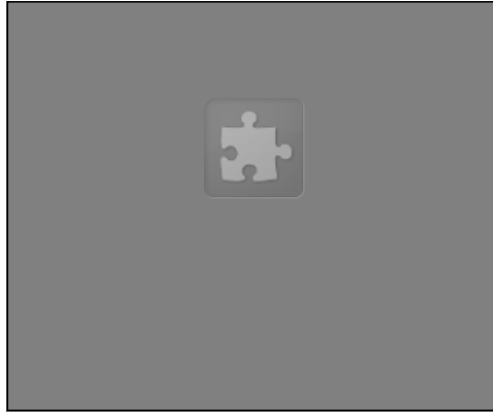
You probably don't, unless you happen to be one of the critical applications that relies on the client authentication and renegotiation features of TLS. In that case, like most applications, you probably assumed that a TLS connection opened with a remote user was *actually from that user the whole time*, and not from two different users.

If you -- like most applications -- made that assumption, you might also forget to treat the early part of the connection (prior to client authentication) as a completely untrusted bunch of



crap. And then you'd be in a world of hurt.

But don't take my word for it. There's video! ([See here for the source, background and details](#)):



#### What does this have to do with the provable security of TLS?

Of all the questions 3Shake raises, this one is the most interesting. As I mentioned earlier, there have been several recent works that purport to prove things about the security of TLS. They're all quite good, so don't take any of this as criticism.

However, they (with [one exception](#), the [miTLS project](#)) didn't find this attack. Why is that?

The first reason is simple: many of these works analyze only the basic TLS handshake, or they omit at least one of the possible handshakes (e.g., [resumption](#)). This means they don't catch the subtle interactions between the resumption handshake, the renegotiation handshake, and extensions -- all of which are the exact ingredients that make most TLS attacks possible.

The second problem is that we don't quite know *what* standard we're holding TLS to. For example, the common definition of security for TLS is called "Authenticated and Confidential Channel Establishment" (ACCE). Roughly speaking this ensures that two parties can establish a channel and that nobody will be able to determine what data is being sent over said channel.

The problem with ACCE is that it's a definition that was developed *specifically so that TLS could satisfy it*. As a result, it's necessarily weak. For example, ACCE does not actually require that each handshake produces a unique Master Secret -- one of the flaws that enables this attack -- because such a definition was not possible to achieve with the existing TLS protocol. In general this is what happens when you design a protocol first and prove things about it later.

#### What's the future for TLS? Can't we throw the whole thing out and start over again?

Sure, go ahead and make TLS Rev 2. It can strip out all of this nonsense and start fresh.

But before you get cocky, remember -- all these crazy features in TLS were put there for a reason. Someone wanted and demanded them. And sadly, this is the difference between a successful, widely-used protocol and *your* protocol.

Your new replacement for TLS might be simple and wonderful today, but that's only because nobody uses it. Get it out into the wild and before long it too will be every bit as crazy as TLS.

Notes:

\* An earlier version of this post incorrectly identified the researchers who discovered the attack.

\*\* The Diffie-Hellman (DHE) version is somewhat more clever. It relies on the attacker manipulating the D-H parameters such that they will force the client to use a particular key. Since DHE parameters sent down from the server are usually 'trusted' by TLS implementations, this trick is relatively easy to pull off.

Posted by [Matthew Green](#) at 12:13 PM 14 comments:

 +120 Recommend this on Google

Thursday, February 2, 2012

## Multiple encryption

While browsing some community websites, I noticed a few people talking about the security of *double* (or more generally, *multiple*) encryption. Multiple encryption addresses the following problem: you have two (or more) encryption schemes, and you're worried that one of them might get compromised. Surely if you encrypt with *both* at the same time you'll buy yourself an added safety margin.



Not everything combines well.

Let me preface this by saying that multiple encryption addresses a problem that *mostly* doesn't exist. Modern ciphers rarely get broken -- at least, not in the [Swordfish](#) sense. You're far more likely to get hit by malware or an implementation bug than you are to suffer from a catastrophic attack on AES.\*

That said, you really *are* likely to get hit by malware or an implementation bug. And that's at least one argument for multiple encryption -- if you're willing to encrypt on separate, heterogenous devices.\*\* There's also the future to think about. We feel good about AES today, but how will we feel in 2040?

I note that these are problems for the extremely paranoid -- governments, mostly -- *not* for the typical developer. The majority of us should work on getting *single encryption* right. But this kind of thing isn't ridiculous -- the [NESSIE standards](#) even recommend it. Moreover, my experience is that when people start *asking* questions about the security of *X*, it means that they're already *doing* *X*, and have been for some time.

So for all that, it's worth answering some of these questions. And roughly speaking, the questions are:

1. Am I better off encrypting with two or more encryption schemes (or keys?)
2. Could I be *worse off*?
3. If I have to do it, how should I do it securely?

Given how little sleep I've gotten recently I don't promise to answer these *fully*, or in any particular order. But I do hope I can provide a little bit of insight around the edges.

### Preliminaries

There are many ways to double encrypt, but for most people 'double encryption' means this:

$$\text{SuperDuperEncrypt}(KA, KB, M) = \text{EncryptA}(KA, \text{EncryptB}(KB, M))$$

This construction is called a *cascade*. Sometimes *EncryptA* and *EncryptB* are different algorithms, but that's not really critical. What does matter for our purposes is that the *keys* *KA* and *KB* are independently-generated.\*\*\* (To make life easier, we'll also assume that the algorithms are published.)

A [lot](#) has been written about cascade encryption, some good and some bad. The answer to the question largely depends on whether the algorithms are simply block ciphers, or if they're true *encryption* algorithms (e.g., a mode of operation using a block cipher). It also depends on what security definition you're trying to achieve.

### The good

Let's consider the positive results first. If either *EncryptA* or *EncryptB* is 'semantically secure', i.e., indistinguishable under [chosen-plaintext attack](#), then so is the [cascade of the two](#). This may seem wonky, but it's actually very handy -- since many common cryptosystems are specifically analyzed under (at least) this level of security. For example, in the symmetric setting, both [CBC](#) and [CTR](#) modes of operation [can both be shown](#) to achieve this security level, provided that they're implemented with a secure block cipher.

So how do we know the combined construction is secure? A formal proof can be found in this

[2002 paper](#) by Herzberg, but the intuition is pretty simple. If there's an attack algorithm that 'breaks' the combined construction, then we can use that algorithm to attack either of the two underlying algorithms by simply *picking our own key* for the other algorithm and simulating the double encryption on its ciphertexts.

This means that an attack on the combination is an attack on the underlying schemes. So if one is secure, you're in good shape.

### The not-so-good

Interestingly, Herzberg [also shows](#) that the above result does *not* apply for all definitions of security, particularly strong definitions such as [adaptive-chosen ciphertext security](#). In the symmetric world, we usually achieve this level of security using [authenticated encryption](#).

To give a concrete (symmetric encryption) example, imagine that the inner layer of encryption (EncryptB) is [authenticated](#), as is the case in [GCM-mode](#). Authenticated encryption provides both *confidentiality* (attackers can't read your message) and *authenticity* (attackers can't tamper with your message -- or change the ciphertext in any way.)

Now imagine that the outer scheme (EncryptA) *doesn't* provide this guarantee. For a simple example, consider CBC-mode encryption with padding at the end. CBC-mode is well known for its [malleability](#); attackers can flip bits in a ciphertext, which causes predictable changes to the underlying plaintext.

The combined scheme still provides some authenticity protections -- if the attacker's tampering affects the inner (GCM) ciphertext, then his changes should be detected (and rejected) upon combined decryption. But if his modifications only change the CBC-mode *padding*, then the combined ciphertext could be accepted as valid. Hence the combined scheme is 'benignly' malleable, making it technically *weaker* than the inner layer of encryption.

Do you care about this? Maybe, maybe not. Some protocols really *do* require a completely non-malleable ciphertext -- for example, to prevent [replay attacks](#) -- but in most applications these attacks aren't world-shattering. If you do care, you can find some alternative constructions [here](#).

### The ugly

Of course, so far all I've discussed is whether the combined encryption scheme is *at least* as secure as either underlying algorithm. But some people want more than 'at least as'. More importantly, I've been talking about entire encryption algorithms (e.g., modes of operation), not raw ciphers.

So let's address the first question. Is a combined encryption scheme significantly more secure than either algorithm on its own? Unfortunately the answer is: *not necessarily*. There are at least a couple of counterexamples here:

1. *The encryption scheme is a group.* Imagine that EncryptA and EncryptB are the same algorithm, with the following special property: when you encrypt sequentially with KA and KB you obtain a ciphertext that can be decrypted with some *third* key KC.\*\*\*\* In this case, the resulting ciphertext ought to be at least as vulnerable as a single-encrypted ciphertext. Hence double-encrypting gives you no additional security *at all*. Fortunately modern *block ciphers* [don't \(seem\) to have](#) this property -- in fact, cryptographers explicitly design against it, as it can make the cipher weaker. But some [number-theoretic schemes](#) do, hence it's worth looking out for.
2. *Meet-in-the-Middle Attacks.* MiTM attacks are the most common 'real-world' counterexample that come up in discussions of cascade encryption (really, cascade *encipherment*). This attack was first discovered by [Diffie and Hellman](#), and is a member of a class we call [time-space tradeoff attacks](#). It's useful in constructions that use a deterministic algorithm like a block cipher. For example:

```
DOUBLE_DES(KA, KB, M) = DES_ENCRYPT(KA, DES_ENCRYPT(KB, M))
```

On the face of it, you'd assume that this construction would be substantially stronger than a single layer of DES. If a brute-force attack on DES requires  $2^{56}$  operations (DES has a 56-bit key), you'd hope that attacking a construction with *two* DES keys would require on the order of  $2^{112}$  operations. But actually this hope is a false one -- *if the attacker has lots of storage*.

The attack works like this. First, obtain the encryption C of some *known* plaintext

$M$  under the two unknown secret keys  $K_A$  and  $K_B$ . Next, construct a huge table comprising the encipherment of  $M$  under every possible DES key. In our DES example there are  $2^{56}$  keys, this would take a corresponding amount of effort, and the resulting table will be astonishingly huge. But leave that aside for the moment.

Finally, try decrypting  $C$  with every possible DES key. For each result, check to see if it's in the table you just made. If you find a match, you've now got two keys:  $K_A'$  and  $K_B'$  that satisfy the encryption equation above.\*\*\*\*\*

If you ignore storage costs (ridiculously impractical, but which may also be traded for time), this attack will run you  $(2^{56})^2 = 2^{112}$  cipher operations. That's *much* less than the  $2^{112}$  we were hoping for. If you're willing to treat it as a *chosen plaintext attack* you can even re-use the table for many separate attacks.

3. *Plaintext distribution issues*. Maurer showed [one more interesting result](#), which is that in a cascade of *ciphers*, the entire construction is guaranteed to be as secure as the first cipher, but *not necessarily any stronger*. This is because the first cipher may introduce certain patterns into its output that can assist the attacker in breaking the second layer of encipherment. Maurer even provides a (very contrived) counterexample in which this happens.

I presume that this is the source of the following folklore construction, which is referenced in Applied Cryptography and other sources around the Internet:

```
UberSuperEncrypt(KA, KB, M) = EncryptA(KA, R⊕M) || EncryptB(KB, R)
```

Where  $||$  indicates concatenation, and  $R$  is a random string of the same length of the message. Since in this case both  $R$  and  $R \oplus M$  both have a random distribution, this tends to eliminate the issue that Maurer notes. At the cost of doubling the ciphertext size!

Now the good news is that multiple encipherment (done properly) can *probably* make things more secure. This is precisely what constructions like DESX and 3DES try to achieve (using a single cipher). If you make certain strong assumptions about the *strength* of the cipher, it is possible to show that these constructions are harder to attack than the underlying cipher itself (see this analysis of [DESX](#) and this one of [3DES](#)).

I warn you that these analyses use an unrealistic model for the security of the cipher, and they don't treat multiple *distinct* ciphers. Still, they're a useful guide -- assuming that your attacker does not have any special attack against (at least one) of the underlying schemes. Your mileage may vary, and I would generally advise against assembling this sort of thing yourself unless you really know what you're doing.

### In summary

I'm afraid this post will end with a whimper rather than a bang. It's entirely possible to combine encryption schemes in secure ways (many of which are *not* cascade constructions), but the *amount* of extra security you'll get is subject to some debate.

In fact, this entire idea has been studied for a quite a while under the heading of (*robust*) *combiners*. These deal with combining cryptosystems (encryption, as well as hashing, signing, protocols, etc.) in a secure way, such that the combination remains secure even if some of the underlying schemes are broken.

If you're interested, that's the place to start. But in general my advice is that this is not something that most people should spend a lot of time doing, outside of (perhaps) the government and the academic world. If you want to do this, you should familiarize yourself with some of the academic papers already mentioned. Otherwise, think hard about *why* you're doing it, and what it's going to buy you.

Notes:

\* And yes, I know about MD5 and the recent biclique attacks on AES. That *still* doesn't change my opinion.


\*\* Note that this is mostly something the government likes to think about, namely: how to use consumer off-the-shelf products together so as to achieve the same security as trusted, government-certified hardware. I'm dubious about this strategy based on my suspicion that all consumer products will soon be manufactured by [Foxconn](#). Nonetheless I wish them luck.

\*\*\* This key independence is a big deal. If the keys are related (worst case:  $KA \text{ equals } KB$ ) then all guarantees are off. For example, consider a stream cipher like [CTR mode](#), where encryption and decryption are the same algorithm. If you use the same algorithm and key, you'd completely cancel out the encryption, *i.e.*:  $CTR\_ENC(K, IV, CTR\_ENC(K, IV, M)) = M$ .

\*\*\*\* Classical [substitution ciphers](#) (including the [Vigenere](#) cipher and Vernam One-Time Pad) have this structure.

\*\*\*\*\* The resulting  $KA'$  and  $KB'$  *aren't* necessarily the right keys, however, due to *false positives*: keys that (for a single message  $M$ ) satisfy  $DES(KA', DES(KB', M)) = DES(KA, DES(KB, M))$ . You can quickly eliminate the bad keys by obtaining the encryption of a second message  $M'$  and testing it against each of your candidate matches. The chance that a given false positive will work on two messages is usually quite low.

Posted by [Matthew Green](#) at 1:07 PM 4 comments:

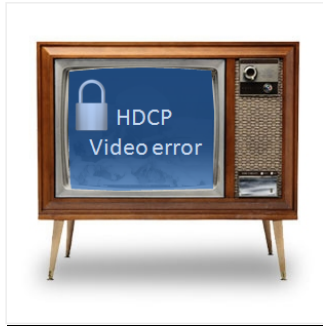
 +4 Recommend this on Google

Monday, August 27, 2012

## Reposted: A cryptanalysis of HDCP v2.1

**Update 8/27:** This post was originally published three weeks ago under a different title. I subsequently [took it down](#) to give affected vendors time to patch the bugs. As a result of the notification, Digital Content Protection LLC (DCP) has updated the spec to v2.2.

Contrary to my understanding when I wrote the original post, HDCP v2 actually is used by a number of devices. I would like to give credit to Alon Ziv at [Discretix](#), who had previously discovered the Locality Check issue, and to [Martin Kaiser](#) who experimentally verified the master secret issue on a European Samsung TV and a Galaxy S II.



Finally, I would like to thank Hanni Fakhoury and Marcia Hofmann at the [Electronic Frontier Foundation](#) for all of their helpful advice. The EFF is one of the only organizations that represents security researchers. Please consider [donating](#) so they can keep doing it!

Over the past couple of weeks I've mostly been blogging about [inconsequential things](#). Blame summer for this -- it's hard to be serious when it's 104 degrees out. But also, the world just hasn't been supplying much in the way of interesting stuff to write about.

Don't get me wrong, this is a good thing! But in a (very limited) way it's also too bad. One of the best ways to learn about security systems is to take them apart and see *how they fail*. While individual systems can be patched, the knowledge we collect from the process is invaluable.

Fortunately for us, we're not completely helpless. If we want to learn something about system analysis, there are plenty of opportunities right out there in the wild. The best place to start is by finding a public protocol that's been published, but not implemented yet. Download the spec and start poking!

This will be our task today. The system we'll be looking at is completely public, and (to the best of my knowledge) has not yet been deployed anywhere (**Update:** see *note above*). It's good practice for protocol cryptanalysis because it includes all kinds of complicated crypto that hasn't been seriously reviewed by anyone yet.

(Or at least, my Google searches aren't turning anything up. I'm very willing to be corrected.)

Best of all, I've never looked at this system before. So whatever we find (or don't find), we'll be doing it together.

A note: this obviously *isn't* going to be a short post. And the TL;DR is that there *is* no TL;DR. This post isn't about finding bugs (although we certainly will), it's about learning how the process works. And that's something you do for its own sake.

## HDCPv2

The protocol we'll be looking at today is the [High Bandwidth Digital Content Protection](#) (HDCP) protocol *version 2*. Before you get excited, let me sort out a bit of confusion. We are not going to talk about HDCP *version 1*, which is the famous protocol you probably have running in your TV right now.

HDCPv1 was [analyzed way back in 2001](#) and found to be wanting. Things got much worse in 2010 when someone [leaked the HDCPv1 master key](#) -- effectively killing the whole system.

What we'll be looking at today is the replacement: HDCP v2. This protocol is everything that its predecessor was not. For one thing, it uses standard encryption: RSA, AES and HMAC-SHA256. It employs a certificate model with a revocation list. It also adds exciting features like 'localization', which allows an HDCP transmitter to determine *how far away* a receiver is, and stop people from piping HDCP content over the Internet. (In case they actually wanted to do that.)

HDCPv2 has barely hit shelves yet (**Update:** though it was recently selected as the transport security for MiraCast). The Digital Content Protection licensing authority has been keeping a pretty up-to-date set of draft [protocol specifications on their site](#). The latest version at the time of this writing is [2.1](#), and it gives us a nice opportunity to see how industry 'does' protocols.

### An overview of the protocol

As cryptographic protocols go, HDCPv2 has a pretty simple set of requirements. It's designed to protect high-value content running over a wire (or wireless channel) between a transmitter (e.g., a DVD player) and a receiver (a TV). The protocol accomplishes the following operations:

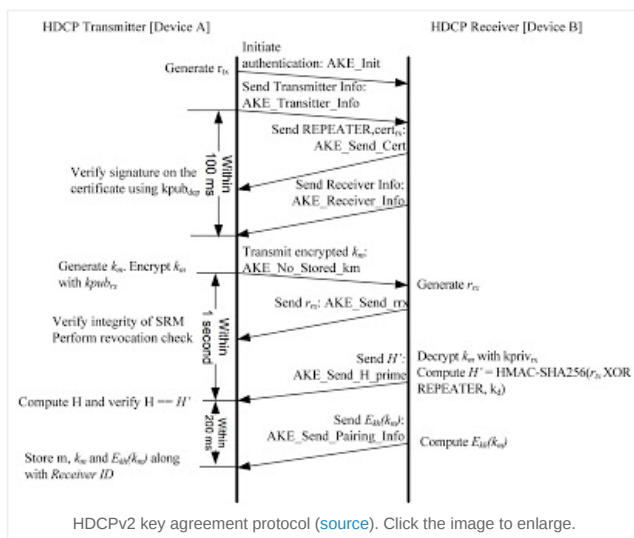
1. Exchanging and verifying public key certificates.
2. Establishing shared symmetric keys between the transmitter and receiver.
3. Caching shared keys for use in later sessions.
4. Verifying that a receiver is *local*, i.e., you're not trying to proxy the data to some remote party via the Internet.

These functions are accomplished via three (mostly) separate protocols: a public-key Authenticated Key Agreement (AKE) protocol, a pairing protocol, where the derived key is cached for later use, and a *locality check protocol* to ensure that the devices are physically close.

I'm going to take these protocols one at a time, since each one involves its own messages and assumptions.

### Phase (1): Authenticated Key Agreement (AKE)

The core of HDCPv2 is a custom key exchange protocol, which looks quite a bit like TLS. (In fact, the resemblance is so strong that you wonder why the designers didn't just use TLS and save a lot of effort.) It looks like this:





Now, there's lots going on here. But if we only look at the crypto, the summary is this:

The transmitter starts by sending 'AKE\_Init' along with a random 64-bit nonce  $R_{tx}$ . In response, the receiver sends back its certificate, which contains its RSA public key and device serial number, all signed by the HDCP licensing authority.

If the certificate checks out (and is not revoked), the transmitter generates a random 128-bit 'master secret'  $K_m$  and encrypts it under the receiver's public key. The result goes back to the receiver, which decrypts it. Now both sides share  $K_m$  and  $R_{tx}$ , and can *combine* them using a wacky custom [key derivation function](#). The result is a shared session key  $K_d$ .

The last step is to verify that both sides got the same  $K_d$ . The receiver computes a value  $H'$ , using [HMAC-SHA256](#) on inputs  $K_d$ ,  $R_{tx}$  and some other stuff. If the receiver's  $H'$  matches a similar value computed at the transmitter, the protocol succeeds.

Simple, right?

Note that I've ignored one last message in the protocol, which turns out to be *very important*. Before we go there, let's pause and take stock.

If you're paying close attention, you've noticed a couple of worrying things:

1. The transmitter doesn't authenticate itself *at all*. This means anyone can pretend to be a transmitter.
2. None of the handshake messages (e.g., AKE\_Transmitter\_Info) appear to be authenticated. An attacker can modify them as they transit the wire.
3. The session key  $K_d$  is based solely on the inputs supplied by the transmitter. The receiver *does* generate a nonce  $R_{rx}$ , but it isn't used in the above protocol.

None of these things *by themselves* are a problem, but they make me suspicious.

### Phase (2): Pairing

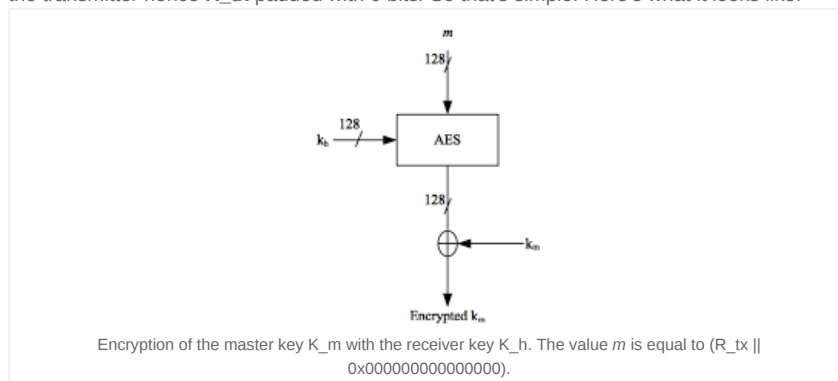
Public-key operations are *expensive*. And you only really need to do them once. The designers recognized this, and added a feature called 'pairing' to cache the derived  $K_m$  for use in later sessions. This is quite a bit like what TLS does for session resumption.

However, there's one catch, and it's where things get complicated: some receivers *don't have* a secure non-volatile storage area for caching keys. This didn't phase the designers, who came up with a 'clever' workaround for the problem: the receiver can simply ask the *transmitter* to store  $K_m$  for it.

To do this, the receiver *encrypts*  $K_m$  under a fixed internal AES key  $K_h$  (which is derived by hashing the receiver's RSA private key). In the last message of the AKE protocol the receiver now sends this ciphertext back to the transmitter for storage. This appears in the protocol diagram as the ciphertext  $E(K_h, K_m)$ .

The obvious intuition here is that  $K_m$  is securely encrypted. *What could possibly go wrong?* The answer is to ask *how*  $K_m$  is encrypted. And that's where things get worrying.

According to the spec,  $K_m$  is encrypted using AES in what amounts to [CTR mode](#), where the 'counter' value is defined as some value  $m$ . On closer inspection,  $m$  turns out to be just the transmitter nonce  $R_{tx}$  padded with 0 bits. So that's simple. Here's what it looks like:



Now, CTR is a perfectly lovely encryption mode provided that you obey one unbreakable rule: the counter value must never be re-used. Is that satisfied here? Recall that the counter  $m$  is actually *chosen* by another party -- the transmitter. This is worrying. If the transmitter wants, it

could certainly ask the receiver to encrypt anything it wants under the same counter.

Of course, an honest transmitter won't do this. But what about a *dishonest* transmitter? Remember that the transmitter is *not* authenticated by HDCP. The upshot is that an attacker can pretend to be a transmitter, and submit *her own*  $K_m$  values to be encrypted under  $K_h$  and  $m$ .

Even this might be survivable, if it weren't for one last fact: *in CTR mode, encryption and decryption are the same operation.*

All of this leads to the following attack:

1. Observe a legitimate communication between a transmitter and receiver. Capture the values  $R_{tx}$  and  $E(K_h, K_m)$  as they go over the wire.
2. Now: pretend to be a transmitter and initiate your *own* session with the receiver.
3. Replay the captured  $R_{tx}$  as your initial transmitter nonce. When you reach the point where you pick the master secret, *don't* use a random value for  $K_m$ . Instead, set  $K_m$  equal to the ciphertext  $E(K_h, K_m)$  that you captured earlier. Recall that this ciphertext has the form:

$$AES(K_h, R_{Tx} || 000...) \oplus K_m$$

Now encrypt this value under the receiver's public key and send it along.

4. Sooner or later the receiver will encrypt the 'master secret' you chose above under its internal key  $K_h$ . The resulting ciphertext can be expanded to:

$$AES(K_h, R_{Tx} || 000...) \oplus AES(K_h, R_{Tx} || 000...) \oplus K_m$$

Thanks to the beauty of XOR, the first two terms of this ciphertext simply cancel out. The result is the original  $K_m$  from the first session! *Yikes!*

This is a huge problem for two reasons. First,  $K_m$  is used to derive the session keys used to encrypt HDCP content, which means that you may now be able to decrypt any past HDCP content traces. And even worse, thanks to the 'pairing' process, you may be able to use this captured  $K_m$  to initiate or respond to further sessions involving this transmitter.

Did I mention that protocols are *hard*?

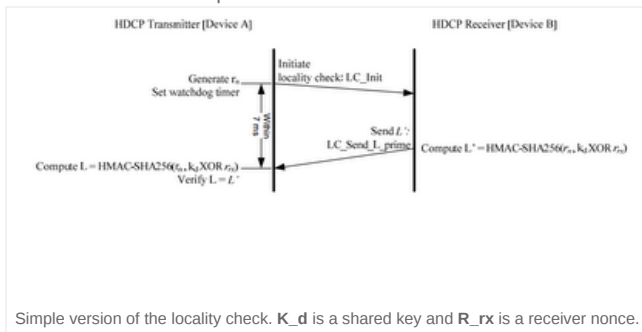
### Phase (3): The Locality Check

For all practical purposes, the attack above should be our stopping point. Once you have the stored  $K_m$  you can derive the session keys and basically do whatever you want. But just for fun, let's go on and see what else we can find.

At its heart, the locality check is a pretty simple thing. Let's assume the transmitter and receiver are both trusted, and have successfully established a session key  $K_d$  by running the AKE protocol above. The locality check is designed to ensure that the receiver is nearby - specifically, that it can provide a cryptographic response to a *challenge*, and can do it in < 7 milliseconds. This is a short enough time that it should prevent people from piping HDCP over a WAN connection.

(Why anyone would want to do this is a mystery to me.)

In principle the locality check should be simple. In practice, it turns out to be pretty complicated. Here's the 'standard' protocol:



Now this isn't too bad: in fact, it's about the simplest challenge-response protocol you can imagine. The transmitter generates a random nonce  $R_n$  and sends it to the receiver. The receiver has 7 milliseconds to kick back a response  $L'$ , which is computed as HMAC-SHA256

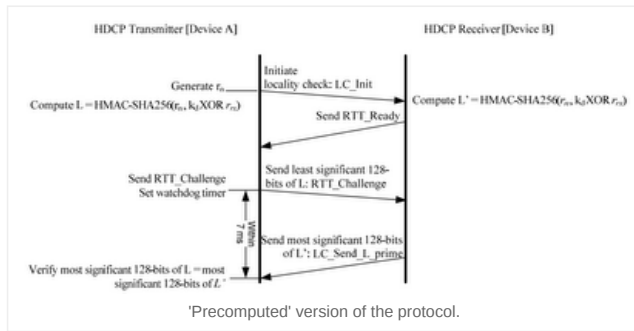
of {the session key  $K_d$ , challenge nonce  $R_n$ , and a 'receiver nonce'  $R_{rx}$ }. You may recall that the receiver nonce was chosen during the AKE.

So far this looks pretty hard to beat.

But here's a wrinkle: some devices are slow. Consider that the 7 milliseconds must be the round-trip communication time, *as well as the time required to compute the HMAC*. There is a very real possibility that some slower, embedded devices might be not be able to respond in time.

Will HDCP provide a second, optional protocol to deal with those devices? You bet it will.

The second protocol allows the receiver to *pre-compute* the HMAC response before the timer starts ticking. Here's what *it* looks like:



This is *nearly* the same protocol, with a few small differences. Notably, the transmitter gives the receiver all the time it wants to compute the HMAC. The timer doesn't start until the receiver says it's ready.

Of course, there has to be *something* keeping the RTT under 7ms. In this case the idea is to keep the receiver from speaking until it's received some authenticator from the transmitter. This consists of the *least significant* 128-bits of the expected HMAC result ( $L'$ ), which is computed in the same way as above. The receiver won't speak until it sees those bits. Then it'll kick back its own response, which consists of the *most-significant* 128 bits of the same value.

Ok, so here we have a protocol that's much more complicated. But considered its own, this one looks pretty ok by me.

But here's a funny question: *what if we try running both protocols at once?*

No, I'm not being ridiculous. You see, it turns out that the receiver and transmitter get to *negotiate* which protocol they support. By default they run the 'simple' protocol above. If both support the pre-computed version, they must indicate this in the AKE\_Transmitter\_Info and AKE\_Receiver\_Info messages sent during the handshake.

This leads to the following conjecture: what if, as a man-in-the-middle attacker, we can convince the transmitter to run the 'pre-computed' protocol. And at the same time, convince the receiver to run the 'simple' one? Recall that none of the protocol flags (transmitted during the AKE) are authenticated. We might be able to trick both sides into seeing a different view of the other's capabilities.

Here's the setup: we have a receiver running in China, and a transmitter located in New York. We're a man-in-the-middle sitting next to the transmitter. We want to convince the transmitter that the receiver is close -- close enough to be on a LAN, for example. Consider the following attack:

1. Modify the message flags so that the transmitter thinks we're running the pre-computed protocol. Since it thinks we're running the pre-computed protocol, it will hand us  $R_n$  and then give us *all the time in the world* to do our pre-computation.
2. Now convince the receiver to run the 'simple' protocol. Send  $R_n$  to it, and wait for the receiver to send back the HMAC result ( $L'$ ).
3. Take a long bath, mow the lawn. Watch Season 1 of Game of Thrones.
4. At your leisure, send the RTT\_READY message to the transmitter, which has been politely waiting for the receiver to finish the pre-computation

5. The transmitter will now send us some bits. *Immediately* send it back the most significant bits of the value  $L'$ , which we got in step (2).

6. Send video to China.

Now this attack may not always work -- it hinges on whether we can convince the two parties to run different protocols. Still, this is a great teaching example in that it illustrates a key problem in cryptographic protocol design: parties *may not share the same view of what's going on*.

A protocol designer's most important job is to ensure that such disagreements can never happen. The best way to do this is to ensure that there's only one view to be had -- in other words, dispense with all the options and write a single clear protocol. But if you must have options, make sure that the protocol only succeeds if both sides *agree* on what those options are. This is usually accomplished by authenticating the negotiation messages, but even this can be a hard, hard problem.

Compared to the importance of learning those lessons, actually breaking localization is pretty trivial. It's a stupid feature anyway.

### In Conclusion

This has been a long post. To the readers I have left at this point: thanks for sticking it out.

The only remaining thing I'd like to say is that this post is not intended to judge HDCPv2, or to make it look bad. It may or it may not be a good protocol, depending on whether I've understood the specification properly *and* depending on whether the above flaws make it into real devices. Which, hopefully they won't now.

What I've been trying to do is teach a basic lesson: *protocols are hard*. They can fail in ruinous, subtle, unexpected, exciting ways. The best cryptographers -- working with BAN logic analyzers and security proofs -- still make mistakes. If you don't have those tools, steer clear.

The best 'fix' for the problem is to recognize how dangerous protocols can be, and to avoid designing your own. If you absolutely must do so, please try to make yours *as simple as possible*. Too many people fail to grok this lesson, and the result is, well, HDCPv2.

===

**Update 8/27:** As I mentioned above, DCP has released a new version of the specification. Version 2.2 includes several updates: it changes the encryption of  $K_m$  to incorporate *both* the Transmitter and Receiver nonces. It also modifies the locality check to patch the bug described above. Both of these changes appear to mitigate the bugs above, at least in *new* devices.

Posted by [Matthew Green](#) at 1:05 PM 6 comments:

 +5 Recommend this on Google

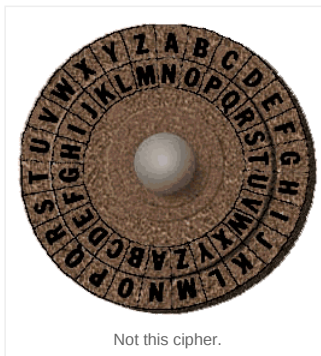
Tuesday, October 9, 2012

## So you want to use an alternative cipher...

Once in a while I run into discussions that hinge on the following dubious proposition: that AES is bad and we need to replace it. These discussions always make me leery, since they begin with facts not in evidence, and rarely inspire any confidence that the *solution* is going to be any better than the 'problem'.

In fact, this whole point of view is so rarified that I've debated whether to even write this post, since my opinion is that AES is the *last* place your system is going to break down -- and you should be focusing your attention on fixing all the other broken things first.

Moreover, the simple advice on AES mirrors the



ancient wisdom about IBM: nobody ever got fired for choosing it. Not only is AES the NIST standard (certified in [FIPS 140](#) and [NSA's Suite B](#)), but there are hundreds of solid implementations to choose from. If that's not enough for you, many processors now support AES operations natively, meaning that your application can now offload most of the work to hardware *without* the help of an expensive co-processor.

So why not just stick with AES? People who have these discussions generally give a variety of reasons, some of which are more valid than others. First, there's what I call the 'slight paranoia' viewpoint, which holds that AES has been around for a long time and could (soon/someday) fail. In just the last few years we've seen a few *impractical* attacks on the construction, which could be beginning of a trend. Or maybe not.

The second (less paranoid) objection is that AES is somewhat troublesome to implement in software. To make it run fast you have to expand the key and pre-compute a series of tables -- all of which increases key setup time and potentially makes you vulnerable to [cache timing attacks](#). Good implementations take this into account, but even the best ones aren't perfect. And in a few cases your performance constraints are so tight that AES just isn't fast enough.

Now I'm not saying that any of these (except possibly for the last reason) are good reasons to ditch AES. In fact, ditching AES would be the *opposite* of my recommendation. But let's say that you've already made the decision to explore more recent, modern alternatives. What are they? Should you trust them? And most importantly: what will they buy you?

### Salsa20

Based on an informal (and totally unscientific poll), the consensus among *advanced* AES-switchers is that [Salsa20](#) has a lot going for it. This is mostly due to Salsa20's performance characteristics, but also because people are growing increasingly confident in its security.

Now, just for the record, Salsa20 is a *stream cipher*, while AES is a block cipher. This distinction is important: stream ciphers produce a string of pseudo-random output bits which are XORed with the message to be encrypted. Block ciphers can be configured to do the same thing (e.g., by running them in CTR or OFB mode), but they can also process plaintext blocks in other ways.

One important difference -- and a reason implementers have historically preferred block ciphers -- is that many block cipher modes allow you to *randomly access* just a portion of an encrypted message, without wasting time decrypting the whole thing. A second advantage is that block ciphers can be used to construct both encryption *and* message authentication (MACs), which makes them a wonderful building block for constructing [authenticated encryption](#) modes.

Salsa20 takes care of the first issue by providing a means to randomly access any block of the generated keystream. Each invocation of the Salsa20 keystream generator takes a key, a nonce (serving as an IV), and a *block position* in the stream. It then outputs the 512-bit block corresponding to that position. This makes it easy to, for example, seek to the last block of a multi-gigabyte file.

It's also possible to use Salsa20 in an authenticated encryption mode -- but it's not trivial. And to do this the cipher must be composed with a polynomial-based MAC like Dan Bernstein's [poly1305](#). I won't lie and say that this usage is standardized and well-defined -- certainly not in the way that, say, [EAX](#) or [GCM](#) modes are with AES.

On the positive side, Salsa20 is *fast* in software. The key setup time is negligible and it has one of the lowest cycles-per-byte counts of any reputable ciphers. [Current figures](#) show Salsa20/12 to be 2-3x as fast as a heavily optimized AES-CTR, and maybe even faster for the implementation that you would actually use (of course, hardware implementations of AES could make up for a lot of this advantage).

The basic limitation a cipher like Salsa20 is the same as with *any* non-standard cipher -- no matter how good the design, you're using an alternative. Alternatives don't get the same attention that standards do. To its credit, Salsa20 has received a [decent amount of academic cryptanalysis](#), most of it positive, but still nothing compared to AES.

### Threefish

Threefish is another recent contribution to the 'alternative ciphers' canon, and hails from Schneier, Ferguson, Lucks, Whiting, Bellare, Kohno, Callas, and Walker. (With a list of authors this long, how could it not be excellent?)

Threefish's distinction is that it's one of a relatively small number of ciphers that recently passed through (most of) a NIST competition. The dubious aspect is that the competition *wasn't a competition for designing a cipher*. Rather, Threefish was submitted as a building block for the SHA3 candidate [Skein](#), which made it to the final round of the competition but was ultimately passed over in favor of [Keccak](#).

Threefish is a wide-block cipher that can be configured to operate on 256, 512 or 1024-bit blocks. Right off the bat this seems useful for applications like disk encryption, where ciphers are typically used to encrypt large blocks of material (sometimes in 'wide block cipher modes' like CMC or EME). But it seems like a nice choice for security and performance reasons.

While Threefish has seen some cryptanalysis, this is still relatively limited (a few major results). None of these results are 'successful', which is noteworthy and confidence-inspiring. But even with this work, it's hard to say where the cipher stands in relation to AES.

#### The AES could-have-beens: Twofish, Serpent, etc.

AES seems like it's always been AES, so it's easy to forget that just a few years ago it was called Rijndael and there were four other finalists that could just as easily have taken the title. Those finalists all received a lot of cryptanalytic attention, and none of them went away when AES was selected.

The two ciphers I occasionally run into are Bruce Schneier's Twofish and Anderson/Biham/Knudsen's Serpent. Both have decent performance in software, and both have stood up relatively well to cryptanalysis. On the flipside, neither of the two ciphers has received very much in the way of analysis since the AES competition ended (Twofish's most recent *significant* result was in 2000).

Any of these ciphers *could* be a worthy alternative to AES if you were desperate, but I wouldn't go out of my way to use one.

#### In conclusion

I realize none of the above actually tells you *which* AES alternative to use, and that's mostly because I don't want to legitimize the question. Unless your adversary is the NSA or you have some serious performance constraints that AES can't satisfy, my recommendation is to stick with AES -- it's the one standard cipher that nobody gets fired for using.

But if you are in the latter category (meaning, you have performance constraints) I'm pretty impressed by Salsa20/12's performance in software, *provided* you have a good strategy for providing authentication. Even better, while Salsa20 is not standardized by NIST, standardization efforts *are* ongoing in the [eCRYPT eStream](#) project. The result could be increasing adoption of this cipher.

If your concern is with the *security* of AES, I have less advice to give you. The beautiful thing about AES is that it's so widely studied and used that we'll almost certainly have plenty of notice should the cipher really start to fail. That is, provided the people attacking it are doing so in the public, academic literature. (If your enemy is the NSA I'm just not sure what to tell you. Just *run*.)

That still leaves a class of folks who worry about encrypting things for the long-haul. For these folks the best I can propose is to *securely combine* AES with another well-studied cipher like Salsa20, Threefish or one of the AES finalists. This will cost you -- and there's no guarantee that *both* ciphers will stand over the long term. But the probability of two significant breaks seems lower than the probability of one.

Posted by [Matthew Green](#) at 12:09 PM 8 comments:

 +5 Recommend this on Google

[Home](#)

[Next Posts](#)

Subscribe to: [Posts \(Atom\)](#)



