

A Few Thoughts on Cryptographic Engineering

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

Matthew Green in messaging, Uncategorized © July 26, 2014 July 29, 2016 ☰ 2,914 Words

Noodling about IM protocols



(<https://matthewdgreen.files.wordpress.com/2014/07/cb4fb-stasicryptocat.png>)

The last couple of months have been a bit slow in the blogging department. It's hard to blog when there are exciting things going on. But also: I've been a bit blocked. I have two or three posts half-written, none of which I can *quite* get out the door.

Instead of writing and re-writing the same posts again, I figured I might break the impasse by changing the subject. Usually the easiest way to do this is to pick some random protocol and poke at it for a while to see what we learn.

The protocols I'm going to look at today aren't particularly 'random' — they're both popular encrypted instant messaging protocols. The first is [OTR](https://otr.cypherpunks.ca/) (Off the Record Messaging). The second is Cryptocat's group chat protocol. Each of these protocols has a similar end-goal, but they get there in slightly different ways.

I want to be clear from the start that this post has *absolutely no destination*. If you're looking for exciting vulnerabilities in protocols, go check out someone else's blog. This is pure noodling.

The OTR protocol

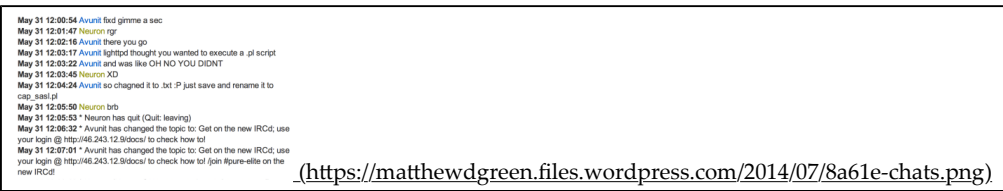
[OTR](https://otr.cypherpunks.ca/) is probably the most widely-used protocol for encrypting instant messages. If you use IM clients like [Adium](https://www.adium.im/) (<https://www.adium.im/>), [Pidgin](https://pidgin.im/) (<https://pidgin.im/>) or [ChatSecure](http://chrisballinger.info/apps/chatsecure/) (<http://chrisballinger.info/apps/chatsecure/>), you already have OTR support. You can enable it in some other clients through plugins and overlays.

OTR was originally developed by Borisov, Goldberg and Brewer and has rapidly come to dominate its niche. Mostly this is because Borisov *et al.* are smart researchers who know what they're doing. Also: they picked a cool name and released working code.

OTR works within the technical and usage constraints of your typical IM system. Roughly speaking, these are:

1. Messages must be ASCII-formatted and have some (short) maximum length.
2. Users won't bother to exchange keys, so authentication should be "lazy" (i.e., you can authenticate your partners after the fact).
3. Your chat partners are all **FBI informants** (http://en.wikipedia.org/wiki/Hector_Xavier_Monsegur) so your chat transcripts must be *plausibly deniable* — so as to keep them from being used as evidence against you in a court of law.

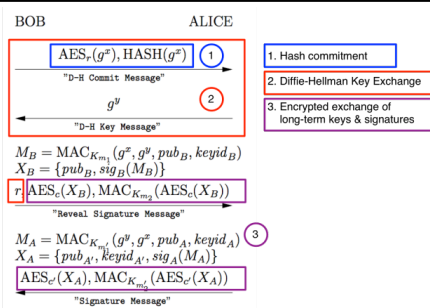
Coming to this problem fresh, you might find goal (3) a bit odd. In fact, to the best of my knowledge no court in the history of law has *ever* used a cryptographic transcript as evidence that a conversation occurred. However it must be noted that this requirement makes the problem a bit more sexy. So let's go with it!



(<https://matthewdgreen.files.wordpress.com/2014/07/8a61e-chats.png>)

"Dammit, they used a deniable key exchange protocol" said no Federal prosecutor ever.

The OTR (version 2/3) handshake is based on the [SIGMA key exchange protocol](http://webee.technion.ac.il/~hugo/sigma-pdf.pdf) (<http://webee.technion.ac.il/~hugo/sigma-pdf.pdf>). Briefly, it assumes that both parties generate long-term DSA public keys which we'll denote by (**pubA**, **pubB**). Next the parties interact as follows:



(<https://matthewdgreen.files.wordpress.com/2014/07/f65f8-otrake.png>)

The OTRv2/v3 AKE. Diagram by [Bonneau and Morrison](http://www.jbonneau.com/doc/BM06-OTR_v2_analysis.pdf) (http://www.jbonneau.com/doc/BM06-OTR_v2_analysis.pdf), all colorful stuff added. There's also an OTRv1 protocol that's too horrible to talk about here.

There are four elements to this protocol:

- 1. Hash commitment.** First, Bob commits to his share of a Diffie-Hellman key exchange (g^x) by encrypting it under a random AES key r and sending the ciphertext and a hash of g^x over to Alice.
- 2. Diffie-Hellman Key Exchange.** Next, Alice sends her half of the key exchange protocol (g^y). Bob can now 'open' his share to Alice by sending the AES key r that he used to encrypt it in the previous step. Alice can decrypt this value and check that it matches the hash Bob sent in the first message. Now that both sides have the shares (g^x , g^y) they each use their secrets to compute a shared secret g^{xy} and hash the value several ways to establish shared encryption keys (c' , $Km2$, $Km'2$) for subsequent messages. In addition, each party hashes g^{xy} to obtain a short "session ID". The sole purpose of the commitment phase (step 1) is to prevent either Alice or Bob from *controlling* the value of the shared secret g^{xy} . Since the session ID value is derived by hashing the Diffie-Hellman shared secret, it's possible to use a relatively short session ID value to authenticate the channel, since neither Alice nor Bob will be able to force this ID to a specific value.
- 3. Exchange of long-term keys and signatures.** So far Alice and Bob have not actually authenticated that they're talking to each other, hence their Diffie-Hellman exchange could have been intercepted by a man-in-the-middle attacker. Using the encrypted channel they've previously established, they now set about to fix this. Alice and Bob each send their long-term DSA public key (**pubA**, **pubB**) and key identifiers, as well as a signature on (a MAC of) the specific elements of the Diffie-Hellman message (g^x , g^y) and their view of which party they're communicating with. They can each verify these signatures and abort the connection if something's amiss.**
- 4. Revealing MAC keys.** After sending a MAC, each party waits for an authenticated response from its partner. It then reveals the MAC keys for the previous messages.
- 5. Lazy authentication.** Of course if Alice and Bob never exchange public keys, this whole protocol execution is still vulnerable to a man-in-the-middle (http://en.wikipedia.org/wiki/Man-in-the-middle_attack) (MITM) attack. To verify that nothing's amiss, both Alice and Bob should eventually authenticate each other. OTR provides three mechanisms for doing this: parties may exchange *fingerprints* (essentially hashes) of (**pubA**, **pubB**) via a second channel. Alternatively, they can exchange the "session ID" calculated in the second phase of the protocol. A final approach is to use the Socialist Millionaires' Problem (http://en.wikipedia.org/wiki/Socialist_millionaire) to prove that both parties share the same secret.

The OTR key exchange provides the following properties:

Protecting user identities. No user-identifying information (e.g., long-term public keys) is sent until the parties have first established a secure channel using Diffie-Hellman. The upshot is that a purely *passive* attacker doesn't learn the identity of the communicating partners — beyond what's revealed by the higher-level IM transport protocol.*

Unfortunately this protection fails against an *active* attacker, who can easily smash an existing OTR connection to force a new key agreement and run an MITM on the Diffie-Hellman used during the next key agreement. This does not allow the attacker to intercept actual message content — she'll get caught when the signatures don't check out — but she can view the public keys being exchanged. From the client point of view the likely symptoms are a mysterious OTR error, followed immediately by a successful handshake.

One consequence of this is that an attacker could conceivably determine which of several clients you're using to initiate a connection.

Weak deniability. The main goal of the OTR designers is *plausible deniability*. Roughly, this means that when you and I communicate there should be no binding evidence that we really had the conversation. This rules out obvious solutions like GPG-based chats, where individual messages would be digitally *signed*, making them non-repudiable.

Properly defining deniability is a bit complex. The standard approach is to show the existence of an efficient 'simulator' — in plain English, an algorithm for making *fake* transcripts. The theory is simple: if it's trivial to make fake transcripts, then a transcript can hardly be viewed as evidence that a conversation really occurred.

OTR's handshake doesn't quite achieve 'strong' deniability — meaning that anyone can fake a transcript between any two parties — mainly because it uses signatures. As signatures are non-repudiable (<http://en.wikipedia.org/wiki/Non-repudiation>), there's no way to fake one without actually knowing your public key. This reveals that we did, in fact, communicate at some point. Moreover, it's possible to create an evidence trail that I communicated with you, e.g., by encoding my identity into my Diffie-Hellman share (g^x). At very least I can show that at some point you were online and we did have contact.

But proving contact is not the same thing as proving that a specific conversation occurred. And this is what OTR works to prevent. The guarantee OTR provides is that *if the target was online at some point and you could contact them*, there's an algorithm that can fake just about any conversation with the individual. Since OTR clients are, by design, willing to initiate a key exchange with just about anyone, merely putting your client online makes it easy for people to fake such transcripts.***

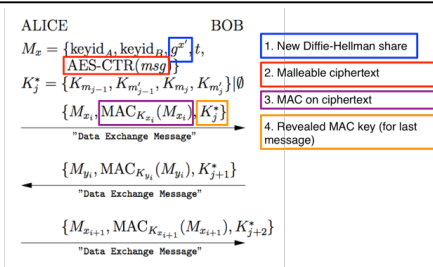
Towards strong deniability. The 'weak' deniability of OTR requires at least tacit participation of the user (Bob) for which we're faking the transcript. This isn't a bad property, but in practice it means that fake transcripts can only be produced by either *Bob himself*, or someone interacting online with Bob. This certainly cuts down on your degree of deniability.

A related concept is 'strong deniability', which ensures that *any party* can fake a transcript using only public information (e.g., your public keys).

OTR doesn't try to achieve strong deniability — but it does try for something in between. The OTR version of deniability holds that an attacker who *obtains the network traffic of a real conversation* — even if they aren't one of the participants — should be able to alter the conversation to say anything he wants. Sort of.

The rough outline of the OTR deniability process is to generate a new message authentication key for each message (using Diffie-Hellman) and then reveal those

keys once they've been used up. In theory, a third party can obtain the private key and — if they know the original message content — they can 'maul' the AES-CTR encrypted messages into messages of their choice, then they can forge their own MACs on the new messages.



(https://matthewdgreen.files.wordpress.com/2014/07/b6559-otr_message.png)

OTR message transport (source: [Bonneau and Morrison \(http://www.jbonneau.com/doc/BM06-OTR_v2_analysis.pdf\)](http://www.jbonneau.com/doc/BM06-OTR_v2_analysis.pdf), all colored stuff added).

Thus our hypothetical transcript forger can take an old transcript that says "would you like a Pizza" and turn it into a valid transcript that says, for example, "would you like to hack STRATFOR"... Except that they probably can't, since the first message is too short and... oh lord, this whole thing is a stupid idea — let's stop talking about it.

The *OTRv1 handshake*. Oh yes, there's also an OTRv1 protocol that has a few issues and isn't really deniable. Even better, an *MITM attacker* (http://en.wikipedia.org/wiki/Man-in-the-middle_attack) can force two clients to downgrade to it, provided both support that version. Yuck.

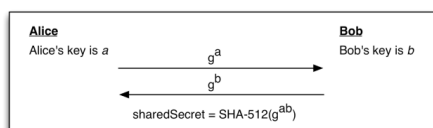
So that's the OTR protocol. While I've pointed out a few minor issues above, the truth is that the protocol is generally an excellent way to communicate. In fact it's such a good idea that if you really care about secrecy it's probably one of the best options out there.

Cryptocat

Since we're looking at IM protocols I thought it might be nice to contrast with another fairly popular chat protocol: *Cryptocat* (<https://crypto.cat/>)'s group chat. Cryptocat is a web-based encrypted chat app that now runs on iOS (and also in Thomas Ptacek's darkest nightmares).

Cryptocat implements OTR for 'private' two-party conversations. However OTR is *not* the default. If you use Cryptocat in its default configuration, you'll be using its *hand-rolled protocol* (<https://blog.cryptographyengineering.com/2013/03/here-come-encryption-apps.html>) for group chats.

The Cryptocat group chat specification can be found here (<https://github.com/cryptocat/cryptocat/wiki/Multiparty-Protocol-Specification>), and it's remarkably simple. There are no "long-term" keys in Cryptocat. Diffie-Hellman keys are generated at the beginning of each session and re-used for all conversations until the app quits. Here's the handshake between two parties:



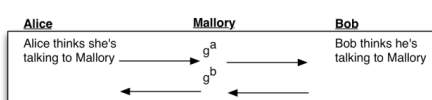
(<https://matthewdgreen.files.wordpress.com/2014/07/a74bc-cryptocathandshake.png>)

Cryptocat group chat handshake (current revision). Setting is *Curve25519* (<http://cr.yo.jp/to/ecdh.html>). Keys are generated when the application launches, and re-used through the session.

If multiple people join the room, every pair of users repeats this handshake to derive a shared secret between every pair of users. Individuals are expected to verify each others' keys by checking fingerprints and/or running the *Socialist Millionaire protocol* (http://en.wikipedia.org/wiki/Socialist_millionaire).

Unlike OTR, the Cryptocat handshake includes no key confirmation messages, nor does it attempt to bind users to their identity or chat room. One implication of this is that I can transmit someone else's public key as if it were my own — and the recipients of this transmission will believe that the person is actually part of the chat.

Moreover, since public keys aren't bound to the user's identity or the chat room, you could potentially route messages between a different user (even a user in a different chat room) while making it look like they're talking to *you*. Since Cryptocat is a *group* chat protocol, there might be some interesting things you could do to manipulate the conversation in this setting.****



(<https://matthewdgreen.files.wordpress.com/2014/07/42453-cryptocatmitm.png>)

Does any of this matter? Probably not *that* much, but it would be relatively easy (and good) to fix these issues.

Message transmission and consistency. The next interesting aspect of Cryptocat is the way it transmits encrypted chat messages. One of the core goals of Cryptocat is to

ensure that messages are *consistent* to individual users. This means that every user should be able to verify that the other user is receiving the same data as it is.

Cryptocat uses a slightly complex mechanism to achieve this. For each pair of users in the chat, Cryptocat derives an AES key and an MAC key from the Diffie-Hellman shared secret. To send a message, the client:

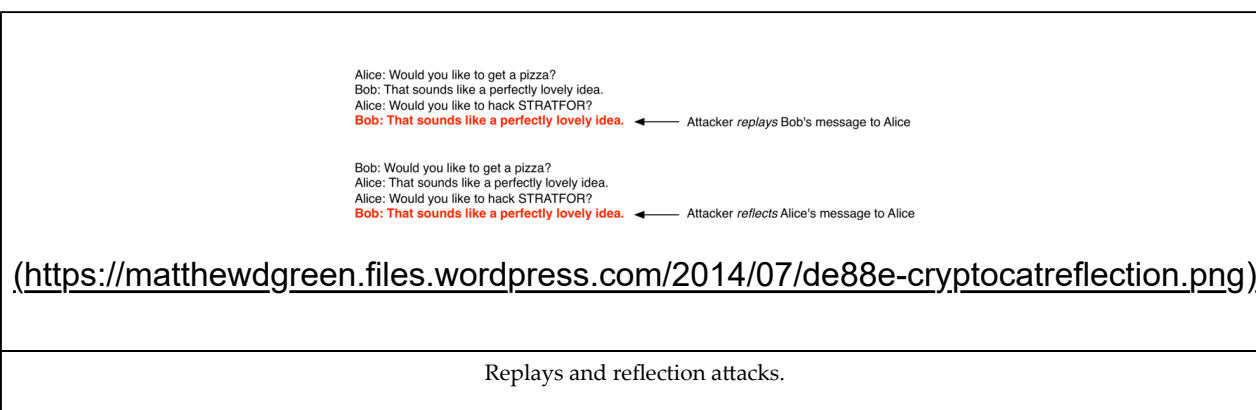
1. Pads the message by appending 64 bytes of random padding.
2. Generates a random 12-byte Initialization Vector for *each of the N users* in the chat.
3. Encrypts the message using AES-CTR under the shared encryption key for each user.
4. Concatenates all of the N resulting ciphertexts/IVs and computes an HMAC of the whole blob under each recipient's key.
5. Calculates a 'tag' for the message by hashing the following data:
padded plaintext || HMAC-SHA512alice || HMAC-SHA512bob || HMAC-SHA512carol || ...
6. Broadcasts the N ciphertexts, IVs, MACs and the single 'tag' value to all users in the conversation.

When a recipient receives a message from another user, it verifies that:

1. The message contains a valid HMAC under its shared key.
2. This IV has not been received before from this sender.
3. The decrypted plaintext is consistent with the 'tag'.

Roughly speaking, the idea here is to make sure that every user receives the same message. The use of a hashed plaintext is a bit ugly, but the argument here is that the random padding protects the message from guessing attacks. Make what you will of this.

Anti-replay. Cryptocat also seeks to prevent replay attacks, e.g., where an attacker manipulates a conversation by simply replaying (or reflecting) messages between users, so that users appear to be repeating statements. For example, consider the following chat transcripts:



Alice: Would you like to get a pizza?
Bob: That sounds like a perfectly lovely idea.
Alice: Would you like to hack STRATFOR?
Bob: That sounds like a perfectly lovely idea. ← Attacker replays Bob's message to Alice

Bob: Would you like to get a pizza?
Alice: That sounds like a perfectly lovely idea.
Alice: Would you like to hack STRATFOR?
Bob: That sounds like a perfectly lovely idea. ← Attacker reflects Alice's message to Alice

(<https://matthewdgreen.files.wordpress.com/2014/07/de88e-cryptocatreflection.png>)

Replays and reflection attacks.

Replay attacks are prevented through the use of a global 'IV array' that stores all previously received and sent IVs to/from all users. If a duplicate IV arrives, Cryptocat will reject the message. This is unwieldy but it generally seems ok to prevent replays and reflection.

A limitation of this approach is that the IV array does not live forever. In fact, from time to time Cryptocat will reset the IV array *without* regenerating the client key. This means that if Alice and Bob both stay online, they can repeat the key exchange and wind up using the same key again — which makes them both vulnerable to subsequent replays and reflections. (**Update:** This issue has since been fixed).

In general the solution to these issues is threefold:

1. Keys shouldn't be long-term, but should be regenerated using new random components for each key exchange.
2. Different keys should be derived for the Alice->Bob and Bob->Alice direction
3. It would be more elegant to use a message counter than to use this big, unwieldy key array.

The good news is that the Cryptocat developers are working on a totally new version (<https://blog.cryptocat.com/2014/01/mpotr-project-plan/>) of the multi-party chat protocol that should be enormously better.

In conclusion

I said this would be a post that goes nowhere, and I delivered! But I have to admit, it helps to push it out of my system.

None of the issues I note above are the biggest deal in the world. They're all subtle issues, which illustrates two things: first, that crypto is hard to get right. But also: that crypto rarely fails catastrophically. The exciting crypto bugs that cause you real pain are still few and far between.

Notes:

* In practice, you might argue that the higher-level IM protocol already leaks user identities (e.g., Jabber nicknames). However this is very much an implementation choice. Moreover, even when using Jabber with known nicknames, you might access the Jabber server using one of several different clients (your computer, phone, etc.). Assuming you use Tor, the only indication of this might be the public key you use during OTR. So there's certainly useful information in this protocol.

** Notice that OTR signs a MAC instead of a hash of the user identity information. This happens to be a safe choice given that the MAC used is based on HMAC-SHA2, but it's not *generally* a safe choice. Swapping the HMAC out for a different MAC function (e.g., CBC-MAC) would be catastrophic.

*** To get specific, imagine I wanted to produce a simulated transcript for some conversation with Bob. Provided that Bob's client is online, I can send Bob any g^x value I want. It doesn't matter if he really wants to talk to me — by default, his client will cheerfully send me back his own g^y and a signature on $(g^x, g^y, \text{pub}_B, \text{keyid}_B)$ which, notably, does not include my identity. From this point on all future authentication is performed using MACs and encrypted under keys that are known to both of us. There's nothing stopping me from faking the rest of the conversation.

Incidentally, a similar problem exists in the OTRv1 protocol.

15 thoughts on “Noodling about IM protocols”

Anonymous says:

July 26, 2014 at 5:42 pm

Nice post! It would be interesting to hear your thoughts on TextSecure/Axolotl.

newlog says:

July 26, 2014 at 7:15 pm

+1 for TextSecure

kevin says:

July 26, 2014 at 8:52 pm

SCIMP strikes me as pretty interesting. Have you written about it before? <https://silentcircle.com/scimp-protocol>

Anonymous says:

July 26, 2014 at 11:43 pm

What is note **** meant to refer to?

Anonymous says:

July 27, 2014 at 6:57 am

+1 for TextSecure

Perseids says:

July 27, 2014 at 7:22 am

> What is note **** meant to refer to?

I believe the second *** in the text is supposed to refer to the last footnote.

I'd also be interested in TextSecure, especially in comparison against OTR.

Anonymous says:

July 28, 2014 at 6:52 am

“Swapping the HMAC out for a different MAC function (e.g., CBC-MAC) would be catastrophic.”

Why?

Perseids says:

July 28, 2014 at 10:32 am

> “Swapping the HMAC out for a different MAC function (e.g., CBC-MAC) would be catastrophic.”

> Why?

Because HMAC is collision resistant and CBC-MAC is not. If you control the message and have access to the key you can change the CBC-MAC to any value you desire with the last block of the message. With HMAC the signature X_B extends to the values contained in M_B . With CBC-MAC a MITM attacker can exchange the Diffie-Hellman values in M_B without any party knowing and thus remain undetected.

Anonymous says:

July 28, 2014 at 6:07 pm

OTR uses 1536-bit MODP Group which has $< 2^{80}$ security. Isn't that a problem?

Anonymous says:

July 29, 2014 at 9:20 pm

What if your names aren't Alice and Bob. Does this still work?

Anonymous says:

July 29, 2014 at 10:51 pm

Citing from <https://otr.cypherpunks.ca/otr-codecon.pdf>, page #4: Alice uses her public key to sign a message

Citing from http://www.jbonneau.com/doc/BM06-OTR_v2_analysis.pdf, page #4: If she forwards Alice's "Signature" message to Bob, Bob will know it was not signed with Mallory's public key.

Citing from <https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html>: OTR public keys are used to generate signatures;

Am I missing something w.r.t. usage of DSA in OTR? That is, isn't signing done in a fashion similar to the description available here: http://en.wikipedia.org/wiki/Digital_Signature_Algorithm#Signing (that is, using the `_private_key`)

Anonymous says:

August 6, 2014 at 6:14 pm

Hi, why don't you compare and include the echo protocol e.g. in <http://goldbug.sf.net> to OTR and Cryptocat?

Bart says:

August 12, 2014 at 7:29 pm

A little off-topic – link to Matasano crypto challenges is no longer valid. I suggest to replace it with <http://cryptopals.com/>

Regards

Otr 214424 says:

Is Your Computer Sluggish or Plagued With a Virus? – If So you Need Online Tech Repairs

As a leader in online computer repair, Online Tech Repairs Inc has the experience to deliver professional system optimization and virus removal. Headquartered in Great Neck, New York our certified technicians have been providing online computer repair and virus removal for customers around the world since 2004. Our three step system is easy to use; and provides you a safe, unobtrusive, and cost effective alternative to your computer service needs. By using state-of-the-art technology our computer experts can diagnose, and repair your computer system through the internet, no matter where you are. Our technician will guide you through the installation of Online Tech Repair Inc secure software. This software allows your dedicated computer expert to see and operate your computer just as if he was in the room with you. That means you don't have to unplug everything and bring it to our shop, or have a stranger tramping through your home.

From our remote location the Online Tech Repairs.com expert can handle any computer issue you want addressed, like:

- – System Optimization
- – How it works Software Installations or Upgrades
- – How it works Virus Removal
- – How it works Home Network Set-ups

Just to name a few.

If you are unsure of what the problem may be, that is okay. We can run a complete diagnostic on your system and fix the problems we encounter. When we are done our software is removed; leaving you with a safe, secure and properly functioning system. The whole process usually takes less than an hour. You probably couldn't even get your computer to your local repair shop that fast!

Call us now for a FREE COMPUTER DIAGNOSTIC using DISCOUNT CODE (otr214424@gmail.com) on +1-914-613-3786 or chat with us on <http://www.onlinetechrepairs.com>.

Otr 214424 says:

August 22, 2014 at 3:49 am

Problem: HP Printer not connecting to my laptop.

I had an issue while connecting my 2 year old HP printer to my brother's laptop that I had borrowed for starting my own business. I used a quick google search to fix the problem but that did not help me.

I then decided to get professional help to solve my problem. After having received many quotations from various companies, i decided to go ahead with Online Tech Repair (www.onlinetechrepairs.com).

Reasons I chose them over the others:

- 1) They were extremely friendly and patient with me during my initial discussions and responded promptly to my request.
- 2) Their prices were extremely reasonable.
- 3) They were ready and willing to walk me through the entire process step by step and were on call with me till i got it fixed.

How did they do it

- 1) They first asked me to state my problem clearly and asked me a few questions. This was done to detect any physical connectivity issues with the printer.
- 2) After having answered this, they confirmed that the printer and the laptop were functioning correctly.
- 3) They then, asked me if they could access my laptop remotely to troubleshoot the problem and fix it. I agreed.
- 4) One of the tech support executives accessed my laptop and started troubleshooting.
- 5) I sat back and watched as the tech support executive was navigating my laptop to spot the issue. The issue was fixed.
- 6) I was told that it was due to an older version of the driver that had been installed.

My Experience

I loved the entire friendly conversation that took place with them. They understood my needs clearly and acted upon the solution immediately. Being a technical noob, i sometimes find it difficult to communicate with tech support teams. It was a very

different experience with the guys at Online Tech Repairs. You can check out their website <http://www.onlinetechrepairs.com> or call them on 1-914-613-3786.

Would definitely recommend this service to anyone who needs help fixing their computers.

Thanks a ton guys. Great Job....!!

Comments are closed.



Menu