**What’s the matter with PGP?**

Last Thursday, Yahoo announced their plans to support end-to-end encryption using a fork of Google’s [end-to-end email extension](https://code.google.com/p/end-to-end/). This is a Big Deal. With providers like Google and Yahoo onboard, email encryption is bound to get a big kick in the ass. This is something [email badly needs](http://www.newyorker.com/tech/elements/the-daunting-challenge-of-secure-e-mail).

So great work by Google and Yahoo! Which is why following complaint is going to seem awfully ungrateful. I realize this and I couldn’t feel worse about it.

As transparent and user-friendly as the new email extensions are, they’re fundamentally just re-implementations of [OpenPGP](http://en.wikipedia.org/wiki/Pretty_Good_Privacy) — and non-legacy-compatible ones, too. The problem with this is that, for all the good PGP has done in the past, it’s a model of email encryption that’s fundamentally broken.

It’s time for PGP to die.

In the remainder of this post I’m going to explain why this is so, what it means for the future of email encryption, and some of the things we should do about it. Nothing I’m going to say here will surprise anyone who’s familiar with the technology — in fact, this will barely be a technical post. That’s because, fundamentally, most of the problems with email encryption aren’t hyper-technical problems. They’re still baked into the cake.

**Background: PGP**

Back in the late 1980s a few visionaries realized that this new ‘e-mail’ thing was awfully convenient and would likely be the future — but that Internet mail protocols made virtually no effort to protect the content of transmitted messages. In those days (and still in [these days](https://www.eff.org/deeplinks/2014/06/new-gmail-data-shows-rise-backbone-email-encryption)) email transited the Internet in cleartext, often coming to rest in poorly-secured mailspools.

This inspired folks like Phil Zimmermann to create tools to deal with the problem. Zimmermann’s PGP was a revolution. It gave users access to efficient [public-key cryptography](http://en.wikipedia.org/wiki/Public-key_cryptography) and fast symmetric ciphers in package you could install on a standard PC. Even better, PGP was compatible with legacy email systems: it would convert your ciphertext into a convenient [ASCII armored](http://en.wikipedia.org/wiki/Binary-to-text_encoding) format that could be easily pasted into the sophisticated email clients of the day — things like “mail”, “pine” or “the Compuserve e-mail client”.

It’s hard to explain what a big deal PGP was. Sure, it sucked badly to use. But in those days, *everything* sucked badly to use. Possession of a PGP key was a badge of technical merit. Folks held [key signing parties](http://en.wikipedia.org/wiki/Key_signing_party). If you were a geek and wanted to discreetly share this fact with other geeks, there was no better time to be alive.

We’ve come a long way since the 1990s, but PGP mostly hasn’t. While the protocol has evolved technically — [IDEA](http://en.wikipedia.org/wiki/International_Data_Encryption_Algorithm) replaced [BassOMatic](http://en.wikipedia.org/wiki/BassOmatic), and was in turn replaced by better ciphers — the fundamental concepts of PGP remain depressingly similar to what Zimmermann offered us in 1991. This has become a problem, and sadly one that’s difficult to change.

Let’s get specific.

**PGP keys suck**

Before we can communicate via PGP, we first need to exchange keys. PGP makes this downright unpleasant. In some cases, dangerously so.

Part of the problem lies in the nature of PGP public keys themselves. For historical reasons they tend to be large and contain lots of extraneous information, which it difficult to print them a business card or manually compare. You can write this off to a quirk of older technology, but even modern [elliptic curve implementations](http://tools.ietf.org/html/rfc6637) still produce surprisingly large keys.

|  |
| --- |
|  |
| |  | | --- | | Three public keys offering roughly the same security level. From top-left: (1) Base58-encoded Curve25519 public key used in [miniLock](https://minilock.io/). (2) OpenPGP 256-bit elliptic curve public key format. (3a) GnuPG 3,072 bit RSA key and (3b) key fingerprint. | |

Since PGP keys aren’t designed for humans, you need to move them electronically. But of course humans still need to verify the *authenticity* of received keys, as accepting an attacker-provided public key can be [catastrophic](http://en.wikipedia.org/wiki/Man-in-the-middle_attack).

PGP addresses this with a hodgepodge of [key servers](http://en.wikipedia.org/wiki/Key_server_(cryptographic)) and [public key fingerprints](http://en.wikipedia.org/wiki/Public_key_fingerprint). These components respectively provide (untrustworthy) data transfer and a short token that human beings can manually verify. While in theory this is sound, in practice it adds complexity, which is always the enemy of security.

Now you may think this is purely academic. It’s not. It can bite you in the ass.

Imagine, for example, you’re a source looking to send secure email to a reporter at the Washington Post. This reporter publishes his fingerprint [via Twitter](https://twitter.com/bartongellman/status/412298520935153664), which means most obvious (and [recommended](https://twitter.com/ioerror/status/496369562276069378)) approach is to ask your PGP client to retrieve the key by fingerprint from a PGP key server. On the GnuPG command line can be done as follows:

[](https://matthewdgreen.files.wordpress.com/2014/08/58cb1-recvkeysworks.png)

Now let’s ignore the fact that you’ve just leaked your key request to an untrusted server via HTTP. At the end of this process you should have the right key with high reliability. Right?

Except maybe not: if you happen to do this with GnuPG 2.0.18 — one version off from the very latest GnuPG — the client [*won’t actually bother to check the fingerprint of the received key*](http://bugs.gnupg.org/gnupg/issue1579). A malicious server (or HTTP attacker) can ship you back the wrong key and you’ll get no warning. This is fixed in the very latest versions of GPG but… Oy Vey.

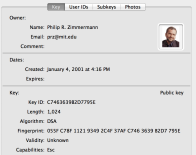
|  |
| --- |
|  |
| PGP Key IDs are also pretty terrible, due to the short length and continued support for the broken V3 key format. |

You can say that it’s unfair to pick on all of PGP over an implementation flaw in GnuPG, but I would argue it speaks to a fundamental issue with the PGP design. PGP assumes keys are too big and complicated to be managed by mortals, but then in practice it practically begs users to handle them anyway. This means we manage them through a layer of machinery, and it happens that our machinery is far from infallible.

Which raises the question: why are we bothering with all this crap infrastructure in the first place. If we *must* exchange things via Twitter, why not simply exchange *keys?* Modern EC public keys are tiny. You could easily fit three or four of them in the space of this paragraph. If we must use an infrastructure layer, let’s just use it to shunt all the key *metadata* around.

**PGP key management sucks**

Manual key management is a mug’s game. Transparent (or at least *translucent*) key management is the hallmark of every [successful](http://techcrunch.com/2014/02/27/apple-explains-exactly-how-secure-imessage-really-is/) [end-to-end secure](http://www.theverge.com/2014/7/29/5945547/signal-brings-painless-encrypted-calling-whisper-systems-moxie-marlinspike) encryption system.

If you can’t trust Phil, who can you trust?

Now often this does involve some tradeoffs — e.g.,, the need to trust a central authority to distribute keys — but even this level of security would be lightyears better than the current situation with webmail.

To their credit, both Google and Yahoo have the opportunity to build their own key management solutions (at least, for those who trust Google and Yahoo), and they [may *still* do so in the future](https://twitter.com/alexstamos/status/497591770645942275). But today’s solutions don’t offer any of this, and it’s not clear when they will. Key management, not pretty web interfaces, is the real weakness holding back widespread secure email.

ZRTP authentication string, as used in [Signal](https://whispersystems.org/).

For the record, classic PGP does have a solution to the problem. It’s called the “[web of trust](http://en.wikipedia.org/wiki/Web_of_trust)“, and it involves individuals signing each others’ keys. I refuse to go into the problems with WoT because, frankly, [life](https://lists.torproject.org/pipermail/tor-talk/2013-September/030235.html) is too short. The TL;DR is that ‘trust’ means different things to you than it does to me. Most OpenPGP implementations do a lousy job of presenting any of this data to their users anyway.

The lack of transparent key management in PGP isn’t unfixable. For those who don’t trust Google or Yahoo, there are experimental systems like [Keybase.io](https://keybase.io/) that attempt to tie keys to user identities. In theory we could even exchange our offline encryption keys through voice-authenticated channels using apps like OpenWhisperSystems’ [Signal](https://itunes.apple.com/us/app/signal-private-messenger/id874139669?mt=8). So far, nobody’s bothered to do this — all of these modern encryption tools are islands with no connection to the mainland. Connecting them together represents one of the real challenges facing widespread encrypted communications.

**No forward secrecy**

Try something: go delete some mail from your Gmail account. You’ve hit the archive button. Presumably you’ve also permanently wiped your Deleted Items folder. Now make sure you wipe your browser cache and the mailbox files for any IMAP clients you might be running (e.g., on your phone). Do any of your devices use [SSD drives](http://arstechnica.com/security/2011/03/ask-ars-how-can-i-safely-erase-the-data-from-my-ssd-drive/)? Probably a safe bet to securely wipe those devices entirely. And at the end of this Google may *still* have a copy which could be vulnerable to law enforcement request or civil subpoena.

(Let’s not get into the NSA’s [collect-it-all policy for encrypted messages](http://www.theguardian.com/world/2013/jun/20/fisa-court-nsa-without-warrant). If the NSA is your adversary just forget about PGP.)

[Forward secrecy](http://en.wikipedia.org/wiki/Forward_secrecy) (usually misnamed “*perfect* forward secrecy”) ensures that if you can’t destroy the ciphertexts, you can at least dispose of keys when you’re done with them. Many online messaging systems like [off-the-record](https://otr.cypherpunks.ca/) messaging use PFS by default, essentially deriving a new key with each message volley sent. Newer ‘ratcheting’ systems like Trevor Perrin’s [Axolotl](https://github.com/trevp/axolotl/wiki) (used by [TextSecure](https://play.google.com/store/apps/details?id=org.thoughtcrime.securesms&hl=en)) have also begun to address the offline case.

Adding forward secrecy to asynchronous offline email is a much bigger challenge, but fundamentally it’s at least *possible* to some degree. While securing the initial ‘introduction’ message between two participants may be challenging\*, each subsequent reply can carry a new ephemeral key to be used in future communications. However this requires breaking changes to the PGP protocol and to clients — changes that aren’t likely to happen in a world where webmail providers have doubled down on the PGP model.

**The OpenPGP format and defaults suck**

Poking through a modern OpenPGP [implementation](https://www.gnupg.org/) is like visiting a museum of 1990s crypto. For legacy compatibility reasons, many clients use old ciphers like [CAST5](http://en.wikipedia.org/wiki/CAST-128) (a cipher that predates the [AES competition](http://competitions.cr.yp.to/aes.html)). RSA encryption uses padding that looks disturbingly like [PKCS#1v1.5](http://tools.ietf.org/html/rfc2313) — a format that’s been [relentlessly exploited in the past](https://blog.cryptographyengineering.com/2012/06/bad-couple-of-years-for-cryptographic.html). Key size defaults don’t reach the 128-bit security level. [MACs](http://en.wikipedia.org/wiki/Message_authentication_code) are optional. Compression is often on by default. Elliptic curve crypto is (still!) [barely supported](https://code.google.com/p/gnupg-ecc/).

If Will Smith looked like this when your cryptography was current, you need better cryptography.

Most of these issues are *not* exploitable unless you use PGP in a non-standard way, e.g., for [instant messaging](http://box.matto.nl/gnupgjabber.html) or online applications. And some people do use PGP this way.

But even if you’re just using PGP just to send one-off emails to your grandmother, these bad defaults are *pointless and unnecessary.*It’s one thing to provide optional backwards compatibility for that one friend who runs PGP on his Amiga. But few of *my* contacts do — and moreover,*client versions are clearly indicated in public keys.\*\**Even if these archaic ciphers and formats aren’t exploitable today, the current trajectory guarantees we’ll still be using them a decade from now. Then all bets are off.

On the bright side, both Google and Yahoo seem to be pushing towards modern implementations that break compatibility with the old. Which raises a different question. If you’re going to break compatibility with most PGP implementations, why bother with PGP at all?

**Terrible mail client implementations**  
  
This is by far the worst aspect of the PGP ecosystem, and also the one I’d like to spend the least time on. In part this is because UX isn’t technically PGP’s problem; in part because the experience is inconsistent between implementations, and in part because it’s inconsistent between *users:* one person’s ‘usable’ is another person’s technical nightmare.

But for what it’s worth, many PGP-enabled mail clients make it ridiculously easy to send confidential messages with encryption turned off, to send unimportant messages with encryption turned on, to accidentally send to the wrong person’s key (or the wrong subkey within a given person’s key). They demand you encrypt your key with a passphrase, but routinely bug you to enter that passphrase in order to sign outgoing mail — exposing your decryption keys in memory even when you’re not reading secure email.

Most of these problems stem from the fact that PGP was designed to retain compatibility with standard (non-encrypted) email. If there’s one lesson from the past ten years, it’s that people are comfortable moving *past* email. We now use [purpose](http://techcrunch.com/2014/02/27/apple-explains-exactly-how-secure-imessage-really-is/)–[built](https://www.facebook.com/mobile/messenger) [messaging](http://www.whatsapp.com/) systems on a day-to-day basis. The startup cost of a secure-by-default environment is, at this point, basically an app store download.

Incidentally, the new Google/Yahoo web-based end-to-end clients dodge this problem by providing essentially no user interface at all. You enter your message into a separate box, and then plop the resulting encrypted data into the Compose box. This avoids many of the nastier interface problems, but only by making encryption non-transparent. This may change; it’s too soon to know how.

**So what should we be doing?**

Quite a lot actually. The path to a proper encrypted email system isn’t that far off. At minimum, any real solution needs:

* **A proper approach to key management.** This could be anything from centralized key management as in Apple’s [iMessage](http://images.apple.com/iphone/business/docs/iOS_Security_Feb14.pdf) — which would still be better than nothing — to a decentralized (but still usable) approach like the one offered by Signal or OTR. Whatever the solution, in order to achieve mass deployment, keys need to be made much more manageable or else submerged from the user altogether.
* **Forward secrecy baked into the protocol.**This should be a pre-condition to any secure messaging system.
* **Cryptography that post-dates the Fresh Prince**. Enough said.
* **Screw backwards compatibility.**Securing both encrypted and unencrypted email is too hard. We need dedicated networks that handle this from the start.

A number of projects are already going in this direction. Aside above-mentioned projects like Axolotl and TextSecure — which pretend to be text messaging systems, but are really email in disguise — projects like [Mailpile](https://www.mailpile.is/) are trying to re-architect the client interface (though they’re sticking with the PGP paradigm). Projects like [SMIMP](https://github.com/smimp/smimp_spec/blob/master/smimp_specification.md) are trying to attack this at the protocol level.\*\*\* At least in theory projects like [DarkMail](https://www.darkmail.info/) are also trying to adapt [text messaging](https://silentcircle.com/scimp-protocol) protocols to the email case, though details remain few and far between.  
It also bears noting that many of the issues above could, in principle at least, be addressed within the confines of the OpenPGP format. Indeed, if you view ‘PGP’ to mean nothing more than the OpenPGP transport, a lot of the above seems easy to fix — with the exception of forward secrecy, which really does seem hard to add without some serious hacks. But in practice, this is rarely all that people mean when they implement ‘PGP’.

**Conclusion**

I realize I sound a bit cranky about this stuff. But as they say: a PGP critic is just a PGP user who’s actually *used* the software for a while. At this point so much potential in this area and so many opportunities to do better. It’s time for us to adopt those ideas and stop looking backwards.

*Notes:*  
 *\**Forward security even for introduction messagescan be implemented, though it either require additional offline key distribution (e.g., [TextSecure’s ‘pre-keys’](https://whispersystems.org/blog/asynchronous-security/)) or else the use of [advanced primitives](https://www.cs.umd.edu/~jkatz/papers/forward-enc-full.pdf). For the purposes of a better PGP, just handling the second message in a conversation would be sufficient.

\*\* Most PGP keys indicate the precise version of the client that generated them (which seems like a dumb thing to do). However if you want to add metadata to your key that indicates which ciphers you prefer, you have to use an optional command.

\*\*\* Thanks to Taylor Hornby for reminding me of this.