PGP *and* You

+

PGP and You

Caleb Thompson

October 13, 2014

Contents

Introduction

Pretty Good Privacy (OpenPGP) is a standard that enables encrypted communication between individuals.

You may have received an email with a “NONAME” or “signature.asc” attachment and wondered what it was, or you might have seen an “armored” email wrapped in something like:

```
-----BEGIN PGP SIGNED MESSAGE-----
[Plaintext message]
-----BEGIN PGP SIGNATURE-----
[Jumble of confusing characters]
-----END PGP SIGNATURE-----
```

By the end of this post, you’ll understand what these attachments and gibberish mean, why they’re important, and how you can have your very own.

You’ll also have the superpowers to strongly encrypt messages, verify the source of tagged Git releases, and sign your emails with something stronger than your name in ASCII.

Part I

Getting Started

Getting Started

Getting started with PGP can be a little daunting, but I hope to make things simpler with this guide.

The very first thing you'll need to do is to install GNU Privacy Guard, or `gpg`, which is a FOSS implementation of PGP.

```
brew install gpg2
```

If you're on another *NIX system, it's likely that you already have `gpg`/`gnupg` installed. If not, it should be easy to find - you're looking for at least version 2.

Generating a public/private key pair

To get anywhere with PGP, you'll need to have what we refer to as a keypair.

A keypair is composed of two parts. A public key, which you'll publish, allows others to encrypt messages to you and verify messages from you. A private key, which you'll keep secret and safe, allows you to decrypt the messages encrypted to your public key and to sign messages so that others can verify they are from you.

```
$ gpg --gen-key
Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
```

```
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection? 1
```

Most of the defaults are good choices for this process, so let's start by accepting `(1) RSA and RSA (default)`.

```
RSA keys may be between 1024 and 8192 bits long.
What keysize do you want? (2048)
```

Again, 2048 isn't a bad option here. The size of the key determines how long it takes to brute-force a key as well as how long it takes to use the public key to encrypt messages and the private key to decrypt them.

Longer is stronger, but you'll also slow down anything using your key.

2048 is a good lower bound, but feel free to go higher. I'd personally recommend 4096 as a good middle-ground.

```
Requested keysize is 2048 bits
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
```

This step allows you to limit the length of time your keys will be valid for.

It's possible to refresh the key before it expires so you're not completely losing a key if it runs out of time.

I didn't want to bother with refreshing my key every so often so mine never expires. Several others at thoughtbot have an expiration for 1 year after the key was generated.

Any choice here is fine, just remember to refresh your key at whatever interval you define.

```
GnuPG needs to construct a user ID to identify your key.

Real name: Caleb Thompson
Email address: caleb@thoughtbot.com
Comment:
You selected this USER-ID:
    "Caleb Thompson <caleb@thoughtbot.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit?
```

This step is fairly self-explanatory. Comments are optional, and as you can see I don't usually add them.

```
You need a Passphrase to protect your secret key.

Enter passphrase:
```

This step is pretty important, which is why the prompt is phrased the way it is. It's possible, but highly discouraged, not to have a passphrase.

Remember that a passphrase is inherently longer than a password which makes it harder, but still possible, to brute-force.

Use standard password best practices to build out a memorable and strong phrase.

```
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
[…]
gpg: key A0ACE70A marked as ultimately trusted
public and secret key created and signed.

pub   2048R/A0ACE70A 2013-08-12
      Key fingerprint = B432 C068 2FD1 C2D0 6A8B  3951 1621 ADC2 A0AC E70A
uid                  Caleb Thompson <caleb@thoughtbot.com>
sub   2048R/545CA4DF 2013-08-12
```

Great, we're set up locally with a keypair!

Adding identities

You needn't go through this whole process for every email address you have.

We can edit a key to add another user id:

```
$ gpg --edit-key caleb@thoughtbot.com
Secret key is available.

pub  2048R/A0ACE70A  created: 2013-08-12  expires: never       usage: SC
                     trust: ultimate      validity: ultimate
sub  2048R/545CA4DF  created: 2013-08-12  expires: never       usage: E
[ultimate] (1). Caleb Thompson <caleb@thoughtbot.com>

gpg>
```

Typing `help` from this prompt informs us that the `adduid` command lets us "add a user ID". That sounds right.

```
gpg> adduid
Real name: Caleb Thompson
Email address: caleb@calebthompson.io
Comment:
You selected this USER-ID:
    "Caleb Thompson <caleb@calebthompson.io>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? o

You need a passphrase to unlock the secret key for
user: "Caleb Thompson <caleb@thoughtbot.com>"
2048-bit RSA key, ID A0ACE70A, created 2013-08-12


pub  2048R/A0ACE70A  created: 2013-08-12  expires: never       usage: SC
                     trust: ultimate      validity: ultimate
sub  2048R/545CA4DF  created: 2013-08-12  expires: never       usage: E
[ultimate] (1)  Caleb Thompson <caleb@thoughtbot.com>
```

```
[ unknown] (2). Caleb Thompson <caleb@calebthompson.io>

gpg> save
```

After saving, my key has two UIDs attached, representing my work and personal email addresses.

```
$ gpg --list-keys A0ACE70A
pub   2048R/A0ACE70A 2013-08-12
uid                  Caleb Thompson <caleb@thoughtbot.com>
uid                  Caleb Thompson <caleb@calebthompson.io>
sub   2048R/545CA4DF 2013-08-12
```

Generating a revocation certificate

It might be seem strange to revoke your key as soon as you've created it, but what you're really doing here is creating an insurance policy in case we lose access to a key.

There are some who would suggest writing down the (relatively short) revocation certificate and storing it in a safety deposit box, but I would say that a flash drive or a known location on multiple computers are more pragmatic options, if ultimately less secure.

To generate a revocation certificate:

```
$ gpg --output revoke.asc --gen-revoke caleb@thoughtbot.com

sec  2048R/A0ACE70A 2013-08-12 Caleb Thompson <caleb@thoughtbot.com>

Create a revocation certificate for this key? (y/N) y
Please select the reason for the revocation:
  0 = No reason specified
  1 = Key has been compromised
  2 = Key is superseded
  3 = Key is no longer used
```

```
  Q = Cancel
(Probably you want to select 1 here)
Your decision?
```

Since you're making this revocation in advance, you can't really specify the reason, so select 0.

For the same reason, skip the description.

```
Your decision? 0
Enter an optional description; end it with an empty line:
>
Reason for revocation: No reason specified
(No description given)
Is this okay? (y/N) y

You need a passphrase to unlock the secret key for
user: "Caleb Thompson <caleb@thoughtbot.com>"
2048-bit RSA key, ID A0ACE70A, created 2013-08-12

ASCII armored output forced.
Revocation certificate created.

Please move it to a medium which you can hide away; if Mallory gets
access to this certificate he can use it to make your key unusable.
It is smart to print this certificate and store it away, just in case
your media become unreadable.  But have some caution:  The print system of
your machine might store the data and make it available to others!
```

Great, make sure you don't lose that.

Sharing your public key to keyservers

By their nature, public keys are meant to be easily available.

The primary method of sharing keys is to publish them to a keyserver. The `gpg` program includes functionality to push keys. From `gpg2(1)`:

> –send-keys key IDs Similar to –export but sends the keys to a keyserver. [...] If no key IDs are given, gpg does nothing.

To publish to a keyserver, pass the key id from the output above to `--send-keys`:

```
$ gpg --send-keys A0ACE70A
gpg: sending key A0ACE70A to hkp server keys.gnupg.net
```

The `gpg` program comes preconfigured with the `keys.gnupg.net` keyserver. Keyservers propagate keys to each other, so this is a fine default unless you have a reason to use another server.

You can also publish your key in other places, e.g., my key is hosted on my server at `http://calebthompson.io/pubkey.asc.txt`.

The more places your key is, the easier it will be for others to find.

Protecting your private key

There are many extremes you could go to to keep your private key safe.

A good, pragmatic, option is to store it (possibly along with the revocation certificate and public key) on a flash drive which you keep in a safe, secure place that only you have access to.

Part II

Exchanging Public Keys

Exchanging Public Keys

The first step to exchanging keys is to actually get the key somehow.

The simplest way to add a key to your keyring is `--search-keys` for an email address or name which gives you an interactive program to select keys.

The search is pretty smart, so if you searched for me as “Caleb Thompson thoughtbot”, you’d get:

```
$ gpg --search-keys "Caleb Thompson thoughtbot"
gpg: searching for "Caleb Thompson thoughtbot" from hkp server keys.gnupg.net
(1)     Caleb Thompson <caleb@thoughtbot.com>
        Caleb Thompson <caleb@calebthompson.io>
          2048 bit RSA key A0ACE70A, created: 2013-08-12
Keys 1-1 of 1 for "Caleb Thompson thoughtbot".  Enter number(s), N)ext, or Q)uit > 1
gpg: requesting key A0ACE70A from hkp server keys.gnupg.net
gpg: key A0ACE70A : public key "Caleb Thompson <caleb@thoughtbot.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
```

Alternatively, if you were sent an email with a public key attachment or otherwise acquired a file containing a signature, you can import the file with `gpg`:

```
$ gpg --import pubkey.asc
gpg: key A0ACE70A: "Caleb Thompson <caleb@thoughtbot.com>" not changed
gpg: Total number processed: 1
gpg:              unchanged: 1
```

Finding a public key for another person isn't all you need to do to trust their identity and share with others that you do.

Now that my public key is in your keyring you can encrypt and sign messages to that key and verify that messages signed or encrypted with that key are valid. You can't, however, actually trust that the person who controls that key is me.

The reasons the key might not belong to me vary, and include Man-in-the-middle (MITM) attacks or even someone else deciding to make a key that appears to belong to me.

To do that, you'd need to verify the key's fingerprint by somehow having me tell it to you in a venue where you can be sure that you are talking to me.

Ideally this would be in person with me reading my fingerprint to you aloud while you verify the fingerprint on your own computer. This isn't always practical but you definitely want to be absolutely sure the person telling you that the key is mine is me. A phone or video call (if you would recognize my face or voice), verifying on social media, instant messaging, checking the signatures of others who have signed my key (and who you trust), or preferably some combination of these are all potential avenues for verification.

To display a key's fingerprint:

```
$ gpg --fingerprint "Caleb Thompson" # or A0ACE70A
pub   2048R/A0ACE70A 2013-08-12
      Key fingerprint = B432 C068 2FD1 C2D0 6A8B  3951 1621 ADC2 A0AC E70A
uid                  Caleb Thompson <caleb@thoughtbot.com>
uid                  Caleb Thompson <caleb@calebthompson.io>
sub   2048R/545CA4DF 2013-08-12
```

Trusting a key

Much of PGP's strength comes from its Web of Trust concept.

To paraphrase Phil Zimmermann, PGP's inventor, as you import keys from people you interact with you may find that you trust some people to verify others' identities.

PGP allows you to note the extent to which you trust someone's veracity at verifying the identity of keyholders.

First of all, this isn't something you would do for every key. Trust is personal and will never be shared. You can feel free to be truthful about your trust that this person is properly verifying keys - they won't ever know what level you give unless you tell them. You can also feel free not to trust a signature at all.

I do trust Mike, so I'll go ahead and trust.

Assuming I already have Mike's key and know its id:

```
$ gpg --edit-key 2846B014

pub  4096R/2846B014  created: 2013-11-27  expires: 2018-11-26  usage: SC
                     trust: unknown       validity: unknown
sub  2048R/14A5A932  created: 2013-11-27  expires: 2018-11-26  usage: S
[ unknown] (1). Michael John Burns <mike@mike-burns.com>
[ unknown] (2)  Michael John Burns <mburns@thoughtbot.com>
gpg> fpr
pub   4096R/2846B014 2013-11-27 Michael John Burns <mike@mike-burns.com>
 Primary key fingerprint: 5FD8 2CE6 A646 3285 538F  C3A5 3E67 61F7 2846 B014
```

Once we've verified the fingerprint as discussed above, we can assign trust.

```
gpg> trust
pub  4096R/2846B014  created: 2013-11-27  expires: 2018-11-26  usage: SC
                     trust: unknown       validity: unknown
sub  2048R/14A5A932  created: 2013-11-27  expires: 2018-11-26  usage: S
[ unknown] (1). Michael John Burns <mike@mike-burns.com>
[ unknown] (2)  Michael John Burns <mburns@thoughtbot.com>

Please decide how far you trust this user to correctly verify other users' keys
(by looking at passports, checking fingerprints from different sources, etc.)

  1 = I don't know or won't say
  2 = I do NOT trust
  3 = I trust marginally
  4 = I trust fully
```

```
  5 = I trust ultimately
  m = back to the main menu

Your decision? 4

pub  4096R/2846B014  created: 2013-11-27  expires: 2018-11-26  usage: SC
                     trust: full          validity: unknown
[ unknown] (1). Michael John Burns <mike@mike-burns.com>
[ unknown] (2)  Michael John Burns <mburns@thoughtbot.com>
Please note that the shown key validity is not necessarily correct
unless you restart the program.

gpg> save
```

This prompt is fairly self-explanatory, but it's worth pointing out that you should never trust anyone other than yourself ultimately.

When you've answered, you'll be back at the prompt.

```
gpg> save
```

You need to save the key for the trust change to persist, and then you can quit.

Signing a key

Signing a key marks an implicit trust. This means that you have done some amount of work to verify the identity of the keyholder.

The level of trust can be specified, and ranges from "I didn't actually check anything" to "I have verified this key's fingerprint and this person's identity in person using legal documents".

A conceptual similarity to signing a key at the highest level would be to vouch in court that the person the key represents is the person you met and that they are the person they claim to be.

Ideally, this would mean that you had personally met the person and verified their fingerprint.

Signing a key is useful because it informs others that they can trust the signature to the extent that they trust you to verify signatures.

If you do not personally know the person you are exchanging keys with, actual verification of legal identity (photo id) and of the ownership of the email address are important steps. PGP Key Signing has a good overview of how these verifications could be accomplished.

Just like trusting, we need to edit the key (these steps can actually be combined if you want to do both of them at once).

Let's look at two scenarios: Mike Burns has been my coworker as long as I've been at thoughtbot. I'm completely confident in his identity.

I met Mike in person at our annual gathering and we exchanged key signatures at a "key signing party". At this party, I searched for his key on a public keyserver and he verified his fingerprint by reading it out loud from his computer while I read the fingerprint of the downloaded key on my computer.

Since the whole key matched, and I know Mike personally, I signed his key at level 3 (described as "I have done very careful checking.")

As you saw in the example above, I also trusted that Mike's signature on another person's key would have been checked as extensively as I checked his, so I trusted him "fully".

George is also a longtime coworker. Although he lives in Stockholm, I've met him before and am confident in his identity.

However, I can't verify his key's fingerprint in person because his voice doesn't carry across the Atlantic, the East Coast, the South, and through Texas to me in Austin.

There's still a way I can sign George's key. I'll start by retrieving it based on the key id he told me in Slack:

```
$ gpg --recv-keys B51FFCFB
gpg: requesting key B51FFCFB from hkp server keys.gnupg.net
gpg: key B51FFCFB: public key "George Richard John Brocklehurst (thoughtbot) <george@thought
[…]
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
```

I can open the key to inspect and edit it:

```
$ gpg --ask-cert-level --edit-key george@thoughtbot.com

pub  2048R/B51FFCFB  created: 2013-11-04  expires: never       usage: SC
                     trust: unknown       validity: full
sub  2048R/E666A729  created: 2013-11-04  expires: never       usage: E
[  full  ] (1). George Richard John Brocklehurst (thoughtbot) <george@thoughtbot.com>
[  full  ] (2)  George Richard John Brocklehurst <george@georgebrock.com>
[  full  ] (3)  [jpeg image of size 8985]

gpg>
```

We're shown the key with all identities (user ids) as well as the trust level we have for each of them.

I have "full" trust in George's keys, thanks to the Web of Trust, because someone I've trusted has signed George's key:

```
gpg> check
uid  George Richard John Brocklehurst (thoughtbot) <george@thoughtbot.com>
sig!         AFEB61EC 2013-11-04  Michael John Burns (Mike) <mike@mike-burns.com
sig!3        B51FFCFB 2013-11-04  [self-signature]
sig!         2846B014 2014-01-13  Michael John Burns <mike@mike-burns.com>
uid  George Richard John Brocklehurst <george@georgebrock.com>
sig!         AFEB61EC 2013-11-04  Michael John Burns (Mike) <mike@mike-burns.com
sig!3        B51FFCFB 2013-11-04  [self-signature]
sig!         2846B014 2014-01-13  Michael John Burns <mike@mike-burns.com>
uid  [jpeg image of size 8985]
sig!         AFEB61EC 2013-11-06  Michael John Burns (Mike) <mike@mike-burns.com
sig!3        B51FFCFB 2013-11-05  [self-signature]
sig!         2846B014 2014-01-13  Michael John Burns <mike@mike-burns.com>
21 signatures not checked due to missing keys
```

Since I don't have George here to tell me his fingerprint in person, I'll go through a few extra steps. I've already confirmed that my trusted introducer, Mike Burns, has signed George's key. Next, I'll ask George to give me his fingerprint in a

couple of different locations: an email from George's thoughtbot address, our company's Slack room, and his Twitter account. Once it's been confirmed in those locations, and given that I know George personally and he gave me the key id to download in the first place, I'm willing to sign his key and confirm that I believe it to be controlled by George.

```
gpg> sign
Really sign all user IDs? (y/N)
```

This prompt, and the default "N" choice, attempt to point me in the right direction about signing.

While I'm sure that George has a thoughtbot.com email address, I'm less positive that he owns the georgebrock.com email address. George also attached a photo, which I can open in a default photo viewer with:

```
gpg> showphoto
```

That's a picture of George, so I'll sign user id as well:

```
gpg> sign
Really sign all user IDs? (y/N) n
Hint: Select the user IDs to sign

gpg> 1

pub  2048R/B51FFCFB  created: 2013-11-04  expires: never       usage: SC
                     trust: unknown       validity: full
sub  2048R/E666A729  created: 2013-11-04  expires: never       usage: E
[  full  ] (1)* George Richard John Brocklehurst (thoughtbot) <george@thoughtbot.com>
[  full  ] (2)  George Richard John Brocklehurst <george@georgebrock.com>
[  full  ] (3)  [jpeg image of size 8985]

gpg> 3

pub  2048R/B51FFCFB  created: 2013-11-04  expires: never       usage: SC
                     trust: unknown       validity: full
```

```
sub  2048R/E666A729  created: 2013-11-04  expires: never       usage: E
[  full  ] (1)* George Richard John Brocklehurst (thoughtbot) <george@thoughtbot.com>
[  full  ] (2)  George Richard John Brocklehurst <george@georgebrock.com>
[  full  ] (3)* [jpeg image of size 8985]

gpg>
```

Now when I issue `sign`, the program knows to only sign identities 1 and 3.

```
gpg> sign

pub  2048R/B51FFCFB  created: 2013-11-04  expires: never       usage: SC
                     trust: unknown       validity: full
 Primary key fingerprint: 0750 F6BF 8064 E22C 68D2  3D90 0C64 3A97 B51F FCFB

     George Richard John Brocklehurst (thoughtbot) <george@thoughtbot.com>
     [jpeg image of size 8985]

How carefully have you verified the key you are about to sign actually belongs
to the person named above?  If you don't know what to answer, enter "0".

   (0) I will not answer. (default)
   (1) I have not checked at all.
   (2) I have done casual checking.
   (3) I have done very careful checking.

Your selection? (enter `?' for more information):
```

Unlike the situation with Mike where we were together in person, I haven't been able to fully validate George's key.

I've gone through what I'll consider "casual" checking: his email address, his Twitter account, his Slack account, and a signature from a trusted introducer (Mike).

```
Your selection? (enter `?' for more information): 2
Are you sure that you want to sign this key with your
```

```
key "Caleb Thompson <caleb@thoughtbot.com>" (A0ACE70A)

I have checked this key casually.

Really sign? (y/N) y

You need a passphrase to unlock the secret key for
user: "Caleb Thompson <caleb@thoughtbot.com>"
2048-bit RSA key, ID A0ACE70A, created 2013-08-12


gpg> save
```

Great, now I've signed my local copy of George's key, marking that I trust his identity and verified it casually.

As a final step validation step, rather than sending the signature myself as described later I will export and encrypt it to the key, and email it to George so that he can import it and publish, provided that he has access to the key.

```
$ gpg --armor --export george@thoughtbot.com > george_at_thoughtbot.asc
$ gpg --armor --encrypt \
  --recipient george@thoughtbot.com \
  --output george_at_thoughtbot_ENCRYPTED.asc \
  george_at_thoughtbot.asc
```

I'd send the encrypted file to George with instructions:

```
$ gpg --decrypt george_at_thoughtbot_ENCRYPTED.asc > george_at_thoughtbot.asc
$ gpg --import george_at_thoughtbot.asc
$ gpg --send-keys george@thoughtbot.com
```

Sending the key to George this way provides an additional layer of trust, that he has access to unlock and use the secret key to decrypt the signature.

I'll delete the key locally, and redownload it using `--recv-keys` or `import` so that I don't accidentally push up the signature myself.

```
$ gpg --delete-key B51FFCFB

pub  2048R/B51FFCFB 2013-11-04 George Richard John Brocklehurst (thoughtbot) <george@thought

Delete this key from the keyring? (y/N) y
```

Now that you know I'd sign a key in this way, you may want to take that into account when assigning a level of trust to me. For example, a person I knew went through a process like this I wouldn't trust beyond "marginal", because I'd not want to trust this process by another person "fully".

Revoking a key signature

A major part of why I'd be willing to sign George's key "casually" is that key signatures can be revoked.

There are various reasons for doing this, including that a person lost access to their key (hopefully they'd also have revoked it), you made a mistake signing it, the person no longer controls an email address (possibly they left a job), or you realize that the person wasn't actually who you thought.

```
$ gpg --edit-key george@thoughtbot.com

pub  2048R/B51FFCFB  created: 2013-11-04  expires: never       usage: SC
                     trust: unknown       validity: full
sub  2048R/E666A729  created: 2013-11-04  expires: never       usage: E
[  full  ] (1). George Richard John Brocklehurst (thoughtbot) <george@thoughtbot.com>
[  full  ] (2)  George Richard John Brocklehurst <george@georgebrock.com>
[  full  ] (3)  [jpeg image of size 8985]
```

The `revsig` command allows us to create a revocation certificate for the signature.

```
gpg> revsig
You have signed these user IDs on key B51FFCFB:
     George Richard John Brocklehurst (thoughtbot) <george@thoughtbot.com>
```

```
   signed by your key A0ACE70A on 2014-09-03
     George Richard John Brocklehurst <george@georgebrock.com>
     [jpeg image of size 8985]
   signed by your key A0ACE70A on 2014-09-03
```

We are prompted to revoke each signed user id, and I'll answer yes to both:

```
user ID: "George Richard John Brocklehurst (thoughtbot) <george@thoughtbot.com>"
signed by your key A0ACE70A on 2014-09-03
Create a revocation certificate for this signature? (y/N) y
user ID: "[jpeg image of size 8985]"
signed by your key A0ACE70A on 2014-09-03
Create a revocation certificate for this signature? (y/N) y
```

We get a prompt asking why we are revoking:

```
You are about to revoke these signatures:
     George Richard John Brocklehurst (thoughtbot) <george@thoughtbot.com>
   signed by your key A0ACE70A on 2014-09-03
     [jpeg image of size 8985]
   signed by your key A0ACE70A on 2014-09-03
Really create the revocation certificates? (y/N) y
Please select the reason for the revocation:
  0 = No reason specified
  4 = User ID is no longer valid
  Q = Cancel
```

Entering 4 will let us leave a description.

```
Your decision? 4
```

We enter a description, owning up to the reason your signature isn't valid anymore.

```
Enter an optional description; end it with an empty line:
> Incorrectly verified. This is not George's key.
```

```
>
Reason for revocation: User ID is no longer valid
Incorrectly verified. This is not George's key.
Is this okay? (y/N) y

You need a passphrase to unlock the secret key for
user: "Caleb Thompson <caleb@thoughtbot.com>"
2048-bit RSA key, ID A0ACE70A, created 2013-08-12


pub  2048R/B51FFCFB  created: 2013-11-04  expires: never       usage: SC
                     trust: unknown       validity: full
sub  2048R/E666A729  created: 2013-11-04  expires: never       usage: E
[  full  ] (1). George Richard John Brocklehurst (thoughtbot) <george@thoughtbot.com>
[  full  ] (2)  George Richard John Brocklehurst <george@georgebrock.com>
[  full  ] (3)  [jpeg image of size 8985]

gpg> save
```

You can now publish the revocation certificate, and the key signature would no longer be valid.

Of course, I didn't actually revoke this key signature.

Publishing your changes

Just like when I created my original key, I can push up changes such as signing with the `--send-keys` command and key ids:

```
$ gpg --send-keys B51FFCFB 2846B014
```

Part III

For Fun and Profit

For Fun and Profit

So you've followed this guide, have a key and have potentially signed or trusted other keys.

There are some interesting things you can do with your new setup.

Signing Git commits and tags

A major benefit of using PGP is that you're able to sign your commits and tags, allowing others to verify that they have versions of software from a trusted source.

Why Sign Commits?

Just like emails and other messages, a commit signed by someone in your Web of Trust (WoT) can be confirmed to be from a trusted source. Even if the signer is not in your WoT, you can still fairly simply verify their signature is valid.

A strict policy of signing all commits, therefore, could prevent someone committing as you (perhaps with `GIT_COMMITTER_NAME` and `GIT_COMMITTER_EMAIL`) from fully blaming you for a change.

An even stricter policy of only allowing commits signed by those in a maintainer's Web of Trust could help to ensure that a codebase's quality, security, etc. were maintained.

Why Sign Tags?

Signing commits

You can tell `git-commit` to sign a commit by passing the `-S` or `--gpg-sign` argument with a key id (hex or name/email).

```
$ git commit --gpg-sign A0ACE70A
```

The argument is optional if you've configured a signing key.

Signing tags

git-bump

Configure signing key

Configuring your signing key is fairly simple.

```
$ git config --global user.signingkey "Caleb Thompson <caleb@thoughtbot.com>"
```

Configure Git to always sign commits

If you choose, you can also configure `git` to default to signing all commits:

```
$ git config --global commit.gpgsign true
```

It's worth noting that without using an agent to manage your key, this can result in needing to type your passphrase a lot during operations such as rebasing, interactive commits, or stashes.

Verifying signatures

Signing and encrypting emails

First, a word on etiquette:

It's important when you receive an encrypted message that you not respond in plaintext with the original encrypted message quoted.

Ideally, when you receive an encrypted message you should respond with an encrypted message.

Whenever messaging someone whose key I have or who sends me a message with a signature attached I send further emails encrypted.

Basically, this means that I use encryption whenever possible.

This isn't any more difficult for me than signing, and assuming the message recipient also has a reasonable setup, they'll only need to enter their passphrase to read the message.

Setting up Mutt for PGP

My mutt configuration represents a fairly normal mutt/PGP setup.

It includes signing emails by default as attachments, which to most people looks like "NONAME" or "signature.asc", but for others with a PGP setup is a lot more useful than your name at the end.

`muttrc(5)` and `gpg2(1)` have extensive documentation on these settings, and I'll leave that research as an exercise for the reader.

Encrypting emails

From the email compose view, press p to open the PGP menu:

```
PGP (e)ncrypt, (s)ign, sign (a)s, (b)oth, (i)nline format, or (c)lear?
```

By default, the configuration I linked earlier will `(s)ign` as the default configured key, and this menu allows you to change that behavior for a specific email.

Generally when encrypting, it makes sense to select `(b)oth` so that you have a signature added as well.

`(i)nline format` provides the ASCII-armored format I made fun of at the beginning of this post. There are one or two mail clients, such as K-9 mail, which only understand this format.

Setting up Mail.app for PGP

If you're planning to use PGP with Mail.app, GPGTools provide GPGMail for integration with your PGP key, as well as some other GPG tools that provide a GUI interface to OpenPGP.

Using an agent for managing your passphrase

Agents are programs which remember your GPG passphrase for a limited time, allowing you to enter it once and not need to re-enter it for some set amount of time, which is really nice during rebases or stashes in Git or when sending several emails in a short amount of time.

I use the `gpg-agent` program, which should be available in your favorite package manager.

Part IV

Bibliography and Further Reading

- The GNU Privacy Handbook
- Exchanging Keys
- PGP Setup
- PGP Trust
- Is it okay to sign a PGP key without an IRL meeting?
- How to revoke a GnuPG/PGP signature on a key

http://dev.mutt.org/trac/wiki/MuttGuide/UseGPG http://mikegerwitz.com/papers/git-horror-story http://git.kernel.org/cgit/git/git.git/tree/Documentation/SubmittingPatches?id=HEAD https://www.google.com/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=web%20of%20trust%20pgp

Part V

Thanks

Special thanks to Mike Burns and Pat Brisbin, who have been instrumental in my own PGP education, and to George Brocklehurst and Mike Burns for kindly allowing the use of their keys in my examples.