

Learning OpenPGP by Example

(Stupid Crypto Tricks with GPG?)

Seth Hardy

`shardy@aculei.net`

28 December 2004

Part I

Introduction

The Point of All This?

What will I be doing with the next hour of your time?

- Demonstrating the internals of the OpenPGP standard by means of example.
- ...showing off some tricks and some code I'm probably going to regret releasing.

Disclaimer

Do you have:

- A strong background in cryptography?
- A very good understanding of RFC 2440?

If so, you probably won't be as entertained as I'd like you to be.

Outline

Three main areas we'll cover:

- Higher level (keys, signatures, Perl Crypt::OpenPGP)
- Lower level (C source code to GnuPG)
- “Attacks” and (obnoxious) tricks

Should be enough to get a feel for how to go about things, or at least annoy a whole lot of GnuPG/PGP users.

Part II

Higher Level

Crypt::OpenPGP

Perl makes all this easy: Crypt::OpenPGP module, written by Benjamin Trott.

- Pure Perl implementation
- Compatibility with GnuPG, PGP 2.x, PGP 5.x
- Code is open / reusable.

subdsakey.pl: subliminal key reconstructor

Where this all started, in three “simple” steps:

- Patch/recompile the victim's crypto program.
- Have the victim make a signature on any data.
- Reconstruct their private signing key from the signature.

How is this done? A subliminal channel, which is a “feature” of DSA.

Signature and Key Versions

Two versions being used for signatures and keys: v3 and v4

- v3 keys are RSA only
- v3 keys deprecated for a number of reasons
- Still a lot of people holding on to PGP 2.x days, though...

Getting Packeted

- Everything is done in packets: key data, signatures, encryption, user ids, etc.
- Subpackets, too! Signature subpackets for everything including timestamps, trust levels, expiration, issuing key, preferences, policy statements, revocation reason, etc.
- RFC 2440 has many lists of octet flags for packet/subpacket types and values; I won't bore you with lists of numbers.

v3 Keys

Structure of a v3 key packet:

- 1 octet : Version Number (3)
- 4 octets: Timestamp
- 2 octets: Expiration Date (in days from creation)
- 1 octet : Public Key Algorithm (RSA)
- ? octets: Public Key Values n, e

KeyID: Low 64 bits of public modulus (n).

Fingerprint: MD5 hash of key material (but not length).

v4 Keys

Structure of a v4 key packet:

- 1 octet: Version Number (4)
- 4 octets: Timestamp
- 1 octet: Public Key Algorithm
- ? octets: Public Key Values ($n, e; p, q, g, y; p, g, y$)

KeyID: Low 64 bits of fingerprint.

Fingerprint: SHA-1 hash of [0x99, length, key data].

pgpdump

Easy way of looking at the packets of data:

```
0ld: Public Key Packet(tag 6)(418 bytes)
    Ver 4 - new
    Public key creation time - Thu Jan 24 02:01:50 EST 2002
    Pub alg - DSA Digital Signature Algorithm(pub 17)
    DSA p(1024 bits) - ...
    DSA q(160 bits) - ...
    DSA g(1024 bits) - ...
    DSA y(1024 bits) - ...
0ld: User ID Packet(tag 13)(30 bytes)
    User ID - Seth Hardy <shardy@aculei.net>
```

Observations

- v3 keyid low bits of public modulus: relatively easy to make keys with duplicate keyids.
- Length of v3 key not taken into account for hash.
- v4 keys are a lot more flexible (same with v4 signatures).

More Observations...

What about the keyids and fingerprints?

- Changing the timestamp will change the fp, possibly keyid.
- Algorithm parameters (p, g) can be reused, but will change the keyid/fingerprint.

This might seem trivial, but what can we do with it?

Vanity Key Generation!

Using little tricks like this we can increase the speed of generating keys of a specific type.

- A slightly different timestamp isn't such a bad thing, and is faster than generating new parameters or key values.
- Keeping parameters the same allows for faster generation of key values.

The goal? Partial (32-bit) hash collision, but easily distributable.
How badly do you want that vanity key?

Part III

Lower Level: GnuPG Source

Vanity keys as a new key type?

cipher/pubkey.c:

```
struct pubkey_table_s
  const char *name;
  int algo;
  int npkey;
  int nskey;
  int nenc;
  int nsig;
  int use;
  int (*generate)( int algo, unsigned nbits, MPI *skey, MPI **retfactors );
  int (*check_secret_key)( int algo, MPI *skey );
  int (*encrypt)( int algo, MPI *resarr, MPI data, MPI *pkey );
  int (*decrypt)( int algo, MPI *result, MPI *data, MPI *skey );
  int (*sign)( int algo, MPI *resarr, MPI data, MPI *skey );
  int (*verify)( int algo, MPI hash, MPI *data, MPI *pkey,
    int (*cmp)(void *, MPI), void *opaquev );
  unsigned (*get_nbits)( int algo, MPI *pkey );
```

Structure of Key Generation

g10/keygen.c:

do_generate_keypair

→ do_create

→ gen_dsa,rsa,elg

→ cipher table

The problem: reusing parameters and passing the desired keyid are very difficult (but not impossible?) without changing this entire part of the program.

Other Algorithms?

What about other algorithms:

Algorithm ID 18: Reserved for ECC

Algorithm ID 19: Reserved for ECDSA

- Add algorithm to cipher table in `pubkey.c`
- Add the standard functions and provide function pointers

Should be “really simple,” right?

It's been 10 years, and no standard for the parameters has been chosen. (So you'll get to use mine when I'm done with it.)

Part IV

“Attacks” and (Obnoxious) Tricks

Keyservers are a great target.

Ever lost a key?

- How many people have forgotten a passphrase, lost a secret key, or otherwise had a key become unusable?
- How many people have not had a valid revocation certificate (or lost that, too) to submit to a keyserver?

What happens to your key?

Key Generation...

Vanity key generation requires creating *lots* of keys.

How fast is key generation if...

- ...you reuse parameters?
- ...you put in bad data?
- ...you otherwise fuzz values?

Can you see where this is going?

A Simple Keyserver DoS: floodwot.pl

How big is one key?

```
shardy@fillerbunny $ gpg --export shardy > shardy.gpg
shardy@fillerbunny $ ls -l shardy.gpg
-rw-r--r-- 1 shardy users 1722 Dec 27 12:55 shardy.gpg
```

Only self-signed: under 2KiB.

SKS keyserver dump: around 80GiB (last I checked).

But it can get more complicated...

There are many ways to improve this, to speed it up or get around flood checking:

- Generate a lot of keys and cross-sign each other
- Have valid email addresses in the keys
- Do they keys even need to have good data in them?

Cross-signing: fakewot.pl

By generating a lot of keys and cross-signing them, we can get even more obnoxious!

- Step 1: Pick out a chunk of the web of trust.
- Step 2: Generate keys, copy user IDs.
- Step 3: Cross-sign keys according to the real keys.
- Step 4: ...
- Step 5: Profit!

Maybe no profit involved, but since people don't check fingerprints well we're encouraging a *lot* more bad data in the web of trust.

Signatures, too.

Signatures don't leave the keyserver network, either! Easy way to make someone else's key HUGE.

Or, what may or may not be worse (and you don't need a special perl script for it!):

```
shardy@fillerbunny $ gpg --list-sigs demo
pub 1024D/894B7FFF 2004-12-27 Demo Key (21C3 Presentation) <foo@bar.com>
sig 3 894B7FFF 2004-12-27 Demo Key (21C3 Presentation) <foo@bar.com>
sig 3 DFB7E505 2004-12-27 alQaeda Master Signing Key (Do not distribute!) <alqaeda@aol.com>
sub 2048g/7336A431 2004-12-27
sig 894B7FFF 2004-12-27 Demo Key (21C3 Presentation) <foo@bar.com>
```

Etiquette of Key Signing

Lots of supposed “etiquette rules” about signing keys:

- Don't upload signatures right to a keyserver.
- Don't sign someone's key unless you have their permission.
- Key signature escrow services exist!

What's the point?

Solutions?

These aren't new problems, and (while being simple) seem to be almost completely overlooked. Solutions?

- Validate email address before allowing a key submission.
- Limit number of keys submitted (e.g. hashcash).
- *Check the damn fingerprint before signing a key.*

Part V

Summary

Extensions to OpenPGP Programs

Simple tricks and knowledge about keys and signatures allow for additional functionality with little programming time or overhead:

- Proof of concept for sneaky protocol attacks.
- Generation of vanity keys.
- Adding additional signing/encryption algorithms.

And the stupid annoying tricks...

Code is out there for:

- Flooding keyserver networks.
- Copying chunks of the web of trust to encourage bad data.
- Enlarging a key through signatures.

Conclusions

So, what can we get from all of this? Stupid implementation tricks are out there now, but...

- This is just a starting point, much more is possible.
- We need to start thinking about other potential problems.
- This is the first step towards a key “fuzzer” (in progress).

Questions?