

*Applied Cryptography?*  
Oh, I read through that once.

Seth Hardy

shardy@aculei.net  
<http://www.aculei.net/~shardy>

ShmooCon  
February 5, 2005

# Disclaimer

*Never offend someone with style,  
when you can offend them with substance.*

– Sam Brown

# Encryption, it's what you don't know.

*[Program] is an advanced, self-programmed PHP Distributed Encryption web application under the GNU GPL. [Program] premieres at [con] in conjunction with a self-developed, new theory of encryption: geometric transformation. [Program] is a customizable, easily-maintained PHP Distributed Encryption web application including verified installation, maintenance and a powerful user interface. [Program] allows anyone to run their own GNU GPL encryption and fingerprinting server. We'll discuss general encryption, the functionality of [program], demonstrate a sample implementation and discuss future development. [Program], it's who you don't know.*

## Some More Details...

Some other details of the project:

- Four digit PINs as passwords (they're easier to remember).
- All encryption is done server-side.
- Anyone can run a server (and people are encouraged to).
- No trust model at all between servers.

# After the Presentation

Afterwards, I spoke with the presenter...

- *Applied Cryptography?*
- No crypto background, he's a PHP guy.
- We had a nice discussion about how to possibly fix the system.

The end result?

The project was scrapped and the website taken down.

# Talk Objectives

Some of the objectives of this talk are:

- Identify common errors in crypto implementations.
- Discuss methods of avoiding these errors.
- Show that even good systems have flaws.
- Not piss off too many people in the process.

# New Algorithms!

There are a lot of new crypto algorithms out there!

- “Geometric transformation”  
What does this even mean?
- Virtual Matrix Encryption  
Gives you million-bit keys!
- Home-grown methods  
There are always more of these...
- Etc.

# The Problem With New Algorithms

This one's simple, or at least it should be.

*Are you a professional cryptographer with a focus in either the design of symmetric [block or stream] ciphers, or public key algorithms?*

# Algorithms Aren't Trusted Overnight

Proper analysis of an algorithm can (does) take years...

- Elliptic curves are only recently trusted/accepted.
- Rijndael was accepted as AES after *extensive* analysis.
- Anyone heard of nTRU?

## Stating the obvious.

- If you don't know what you're doing, then chances are it's not going to work properly.
- Just because nobody has broken your algorithm doesn't mean it's safe.
- Corollary: Spamming `sci.crypt` or elsewhere saying "hey try to break my algorithm" and not getting a response doesn't mean it's safe. (It generally means that nobody can be bothered to look at your crap.)

## An example: DEFCON 12 CTF

I got (briefly) involved with CTF at DEFCON 12...

- The game involved depositing and reclaiming tokens.
- While everyone else was busy hax0ring, I tried to reverse-engineer the protocol and issue forged tokens.
- Tokens were a bitstring concatenated with a digital signature. (So much for forged tokens...)
- How did I figure this out? Client-side token verification, all the code was in Python.

# Token Verifier

This is my sort of thing, so I did a code audit to see if I could find any weaknesses in their implementation...

- Strong (at least, at the time) algorithms were used:
  - DSA was used for digital signatures.
  - MD5 was used for hashing.
- Unknown whether they were selecting the token identifying strings randomly and just checking against the signature or if there was a list.

Do we see any problems here?

# DSA Signing

A DSA signature on a message  $m$  using private key  $x$  is computed as follows:

- Select a random  $k$  s.t.  $0 < k < q$ .
- Compute  $r = (\alpha^k \bmod p) \bmod q$ .
- Compute  $s = k^{-1}(h + xr) \pmod{q}$ , where  $h$  is the SHA-1 hash of  $m$ .
- The signature for  $m$  is  $(r, s)$ .

# Breaking Spec

What's the problem with using one hash algorithm instead of another?

- MD5 is a 128-bit algorithm.
- SHA-1 is a 160-bit algorithm.
- The DSA signature is computed using  $h = \text{SHA-1}(m)$  as an intermediate value.

The algorithm is weakened, and it can't even be called DSA (because it breaks spec).

## Reporting the Bug

I spoke with the author of the program, to let him know about the bug:

Me: "Hey, I found a bug in your code."

Him: "Oh? Which one?"

Me: "You're using MD5 in DSA, which breaks spec and means it's not even really DSA."

Him: "...actually, I ripped that part of the code straight from pyCrypto."

Oops.

# pyCrypto

From Demo/secimp/sign.py:

```
data=input.read()
hash=MD5.new(data).digest()      # Compute hash of the code object
K = "random bytes"
signature=key.sign(hash, K)      # Sign the hash value
marshal.dump(signature, output)  # Save signature to the file
output.write(data)               # Copy code object to signed file
```

As of 3-Feb-05, this error is still there.

## The Problem...

The problem with writing your own implementation? It's difficult.

(And yes, I understand that pyCrypto is trying to do the first(?) comprehensive crypto implementation for Python, but if you're in that situation, you have to be extra vigilant for mistakes.)

## Stating the obvious, again.

If you're going to write your own crypto implementation...

- Peer code reviews are useful (if open source).
- Test against known examples.  
FIPS 186 has test vectors for DSA; obviously these weren't tested against in this case...
- Bug fixes. Immediately. Damnit.
- ...or just use an existing, trusted, library.

Which leads us into the next common mistake...

## How to keep an algorithm more secure?

There are many ways to keep an algorithm more secure:

- Don't tell anyone how it works, because then they might be able to find bugs!
- If you do find a bug, don't tell anyone, because then they'll be able to break your encryption!
- Or, don't tell anyone, because it'll hurt your reputation and thus make people think anything you write isn't secure!

...right?

# The Problem With Announcing Weaknesses

Trick slide. The problem is when you *don't* announce weaknesses.

- All weaknesses and bugs should be reported and fixed.
- Breaking crypto is *not* a personal attack.
  - It's actually common for people to report breaks (and fixes) to their own systems.
- Everyone loses when something stays broken.

The example from the beginning wasn't personal... at first.

## Stating the obvious, yet again.

How to handle bugs and algorithm attacks?

- Remember to not take attacks against crypto personally (and this goes for both sides).
- Research a weakness when it's reported; back it up with proof.
- Fix the weakness if possible.
- If not possible, announce that the algorithm is broken and should not be used.

Which is more important, how you look to some people on usenet/slashdot/whatever, or the security of your data?

## A Well Known Example

GPG/PGP? ...are good.

But does that mean that there's nothing to worry about?

## A Well Known Example

GPG/PGP? ...are good.

But does that mean that there's nothing to worry about?

## A Well Known Example

GPG/PGP? ...are good.

But does that mean that there's nothing to worry about?

## Some problems, are they just annoying and obnoxious?

Even with reputable programs like GPG/PGP, there are still “attacks” available...

- These attacks don't have to be full breaks.
- In most cases, people won't have to worry about these, right?
- You're only not concerned about something until it happens, and then it's too late...

## Subliminal Messages in Digital Signatures

Messages can be embedded undetectably (unless you know what you're looking for, that is) in digital signatures.

- In DSA, this is *really* easy.
- You can effectively make hidden “metadata” for any number of purposes.
- If someone else uses your malicious code, they can leak their own signing key with every (valid!) digital signature they make without realizing it.

This has been known about for many years; code now exists to have GPG leak DSA keys. (DEFCON 12, 5HOPE.)

# Keyserver Attacks

Attacks can be made against the web of trust via the keyserver network.

- Fast key generation (i.e. for generating vanity keys) + a keyserver network where nothing can be removed = denial of service.
- + fast signature generation = targeted denial of service.
- + human carelessness when it comes to signing keys = cascading trust attacks.

These attacks have been known for many years, too; again, code now exists to take advantage of these problems. (21C3)

# Can these attacks be prevented or fixed?

The simple answer: usually.

- Any attack involving flooding a keyserver can be prevented.
  - Identity checking upon keyserver submission.  
“PGP Global Network” does email verification.
  - Hashcash-enabled keyservers.
  - Stronger, more fine-grained control over key data on keyservers by the owner of the key.
- Trust attacks are entirely due to human error.
  - Too many people still don't verify fingerprints, for example.

# A false sense of security...

...is worse than knowing you're unsafe.

## Stating the obvious, one more time.

Cryptography, like any kind of security, requires constant vigilance. Don't assume you're safe.

- Don't lock your front door but keep the window open.
- Follow algorithms and protocols *exactly* as they are specified.
- Don't trust anything/anyone you don't need to.

# Summary

Four simple things that seem like obvious common sense would fix so many of the crypto problems today:

- Don't invent algorithms unless you know what you're doing.
- Use an implementation that you have reason to believe is safe.
- Fix your mistakes; don't take attacks against code personally.
- Don't assume you're safe; constantly improve your methods.

Remember that security is always dependent on the weakest link.

Questions?