“God does not play dice”

Musings on Randomness, Hashes & Security

By Jonathan Levin

JL@Hisown.Com

(C) 2004 - JL@Hisown.com - All Rights Reserved

Random Values Wh*

What are Random values?

Any type of data that is generated "on the fly" by the programmer with the following requirements:

- Dynamic: Different instances of program generate different values
- Unbiased: distributed uniformly over sample space (a.k.a. "fair")
- Unpredictable: cannot be determined a priori to actual generation.

(C) 2004 - JL@Hisown.com - All Rights Reserved

Before we start talking about Random Values, (and their security implications), it's important to define a few basic terms, which we'll use frequently:

Random values, as noted above, are values which an application generates "on the fly". As opposed to static initialization values, which are always fixed, Random Data is **dynamic***:* The programmer has no idea what values will be generated. All the programmer may set is the **sample space**: the range of the values required. This may be ASCII data (0-255), a simulated coin toss or other Boolean value ("heads/tails",true/false, etc..), a roll of a die (1-6), etc.

Whatever the sample space, we'd usually want our values (which I will hence interchange with "numbers") to be unbiased. That is, distributed uniformly. For those of you who don't recall their statistics 101 – that means that any value in the sample space is as probable as any other (equal probability).

Another requirement, is **unpredictablity**: The random numbers should be obscure as possible, and it should not be possible to guess, or even "guesstimate" the actual values that will be generated in the process. This is an especially important requirement, as we will see shortly.

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Random Values Wh*

Where are Random values used?

Random values, once used only in statistical applications, are no longer as esoteric.

Nowadays, they are prevalent in applications of all types, often embedded deep in operating system functionality – you're likely to be using them, without even knowing it!

(C) 2004 - JL@Hisown.com - All Rights Reserved

Random numbers are an essential building block of many applications, including:

- Decision Making Algorithms: For "Coin Flipping" or other probabilistic tests
- Games: For simulating chance rolls. Especially in e-Casinos
- Security-specific purposes: Generating initial passwords and secret keys
- Web Applications: Generating Session IDs

But, if you've ever written any TCP or UDP/IP based application – you've used them without even being aware of that. The operating system uses random numbers for its TCP sequence numbers, as well as DNS queries.

Random Values Wh*

If they're so random, Why would anyone care about 'em?!

Because by predicting random numbers, you can…

- Spoof TCP (by guessing TCP Sequence #s)
- Make $$$ (e.g. Beating computer-based casinos)
- Make even MORE $$$ (by hijacking user sessions)

(C) 2004 - JL@Hisown.com - All Rights Reserved

So, like, "why should I care?"

Well, because Random numbers have become a major cornerstone of information security. If that corner-stone is shaken, down goes the entire structure.

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Random Values Wh*

Who creates those random values, or numbers?

Two sources of random numbers exist:

- The programming language libraries
- The operating system random number generator (e.g. /dev/random, /dev/urandom, CAPI)

(C) 2004 - JL@Hisown.com - All Rights Reserved

Since Random numbers have so many requirements, and statistical requirements are hard to satisfy with trivial algorithms, most random numbers are generated in standard places:

- The libraries of the programming language used
- The underlying operating system.

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Generating Random Numbers

(the wrong way)

.. is commonly performed in a library function

C/C++	VB	Java	VB.Net
#include <stdlib.h> long a = rand();	Dim a as long a = int(rnd *max) +min	Import java.lang.Math a = random()* max + min	RNG = new Random() Num = RNG.Next(min, max + 1)

Behind the scenes, the actual work is carried out by a Pseudo Random Number Generator (PRNG)

Ensuring samples are uniformly distributed, but - just about it.

(C) 2004 - JL@Hisown.com - All Rights Reserved

Usually, programmers would directly generate random numbers, by using the API Provided by the standard library of the language used. Behind the scenes, a **PseudoRandom Number Generator** (PRNG) does all the work. Notice the name – "Pseudo" implies that there is nothing random about the numbers being generated. In fact, they're fully deterministic!

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Generating Random Numbers

The PRNG ensures a (nearly) uniform distribution

```
int main ()
{
  int i;
  for (i=0; i<10; i++)
    {
      printf ("%d\n",rand());
    }
}
```

Output:
1804289383
846930886
1681692777
1714636915
1957747793
424238335
719885386
1649760492
596516649
1189641421

However… it is fully deterministic!
Identical calls will return the SAME sequence!

➔ We need some form of randomness to change in between calls.

This is what we call the random "seed".

(C) 2004 - JL@Hisown.com - All Rights Reserved

Every language has its own methods of creating random numbers, but they all generally involve the same mechanism. That is, given a ***seed*** value, the numbers generated are **pseudorandom** – uniformly distributed, and seemingly random, on the one hand, but in actuality – fully deterministic values, if the seed is known.

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Generating Random Numbers

(Still the wrong way, but somewhat wiser)

The language API would always allow you to set the seed:

C/C++	VB	Java	VB.Net

#include <stdlib.h>
srand(time());
long a = rand();
	Dim a as long
	Randomize Timer
	a = int(rnd *max) +min
		Import java.util.Random;
		RNG = new Random
		(System.currentTimeMillis())
		a = RNG.next;
			RNG = new Random(seed)
			Num = RNG.Next(min, max+1)

And, setting the Random Seed to the system time is somewhat standard practice.. BUT:

- The System time IS a 32-bit value
- It is predictable (just look at your watch)
- The seed value is applied only ONCE.

(C) 2004 - JL@Hisown.com - All Rights Reserved

Based on the seed, a linear congruential generator issues the random numbers. This opens up an avenue for attack. If the attacker can "tap onto" the random number stream, and identify the seed, the next number will be easily guessed. Note, also, that any linear transformation (multiplication, addition, etc..) performed on the random number stream will NOT offer any additional security, as the number can still be guessed easily.

Notice, also, that most implementations only seed once. This means, that the entire stream of random numbers is fixed. Should an attacker be able to obtain a few consecutive samples of random values, he or she may be able to deduce the seed value – and any number following thereafter.

Generating Random Numbers

(your options)

There are two basic types of generators:

- Linear Congruential Generators
  - Based on a simple equation: $Z_n+1 = aZ_n+b \bmod n$
  - Entropy is solely provided by the seed.
  - INSECURE

- Cryptographically Secure Random Number Generators
  - Based on hash functions
  - Gather Entropy from a variety of sources (!)
  - Standardized (FIPS-140)

(C) 2004 - JL@Hisown.com - All Rights Reserved

Since random number generation is fully deterministic, it is only as strong as the amount of "uncertainty" involved in it. Fortunately, there is a term to well-define and measure this "uncertainty":

Entropy, the term coined by Claude Shannon, describes the amount of actual data, in bits, present in a message, or information source. It is, in may respects, the amount of "uncertainty" in the data (hence the name, borrowed from the thermodynamic context).

The common type of generators, called Linear Congruential, are commonly employed, and are highly insecure.

A better standard of random number generation is emerging – and known as FIPS-140. This defines the cryptographic requirements for random numbers, and just how "random looking" and unpredictable they need be.

As examples of Cryptographically secure PRNGs, consider Blum-Blum-Shub (assuming factoring is hard, this is provably secure.. But SLOW!), Yarrow (Based on SHA-1/3DES) and Tiny (using AES). NIST approved (FIPS-140 compliant) generators are specified in DSS,

FIPS 186-2 appendix 3.1

DSS, FIPS 186-2 appendix 3.2

ANSI X9.31 appendix A

ANSI X9.62 annex A.4.

http://csrc.nist.gov/rng/index.html

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Case Studies

Case I: TCP Sequence #s

Michal Zalewski's article:

http://lcamtuf.coredump.cx

(presented separately)

(C) 2004 - JL@Hisown.com - All Rights Reserved

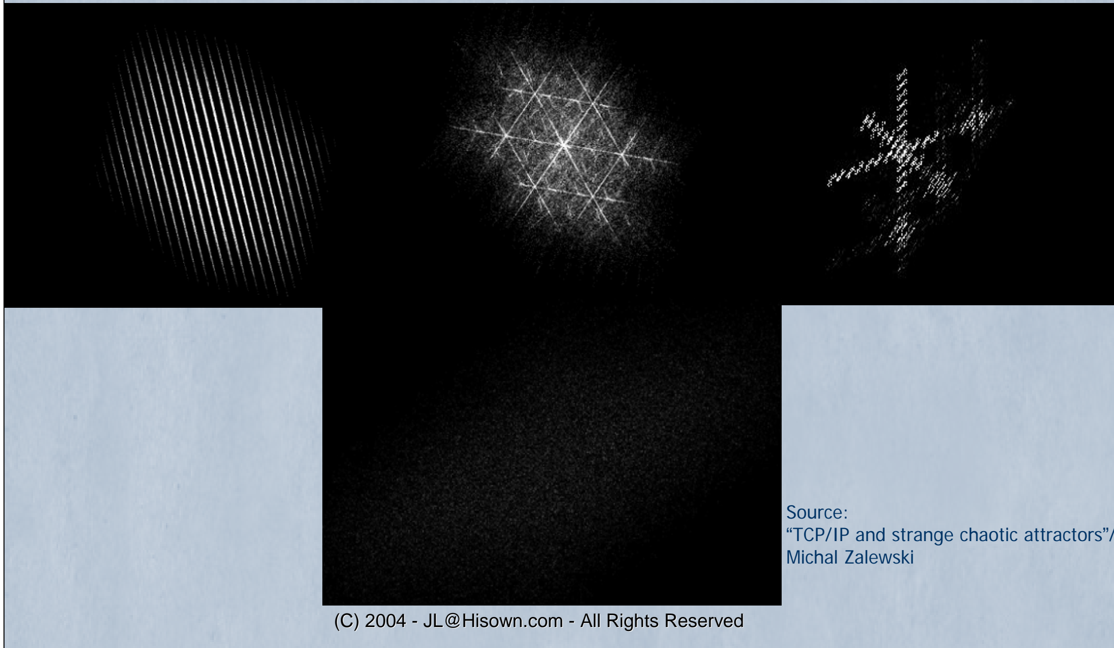
Michal Zalewski's article is revolutionary, in many ways – owing the fact that he proves just how predictable OS random number generators are. Zalewski chooses to use this predictable for TCP/IP Spoofing – that is, guessing the TCP Sequence #s, which are pretty much the only thing that keeps IPv4 Internet reasonably usable. If not for these #s, which are required as part of the 3-Way handshake, TCP would be just as easily spoofed as UDP.

Zalewski's results are quite alarming. With percentages usually ranging between about 12% (Windows 2000) and 100% (HP-UX), he can allegedly guess the next sequence #s by simply attempting a few SYN/ACK exchanges with the host, then spoofing a SYN from some other IP, following with a spoofed ACK (in reply to the SYN/ACK). Zalewski's study was further expanded in http://www.ee.oulu.fi/research/ouspg/frontier/sota/whitepaper-prng/ - work by Finnish researchers. Both researches show that nearly ALL implementations of Random #s in operating systems are predictable.

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Case Studies

Phase Space Analysis



Source:
“TCP/IP and strange chaotic attractors”/
Michal Zalewski

(C) 2004 - JL@Hisown.com - All Rights Reserved

The above figures (taken from Michal Zalewski’s excellent article, “TCP/IP and strange chaotic attractors” aren’t modern art – they are the result of mathematic studies of Random Number Generators.

Turns out that RNGs tend to have “attractors” – much like the Chaotic attractors discovered by Lorenz and others. These are numbers near which the random distribution is more dense.

In addition to fancy, often symmetrical figures, these have an actual use – guessing random numbers and predicting sequence numbers.

Case II: DNS Cache Poisoning

http://www.securityfocus.com/guest/17905/

(presented separately)

And.. Introducing the "Birthday Paradox"

(C) 2004 - JL@Hisown.com - All Rights Reserved

A second interesting case study shows all too well what happens when your sample space (= range of possible values) is too small. In the case of DNS Query IDs: 16-bit, or 65,536 values. Since DNS is stateless (implemented over UDP), the Query ID is the only thing tying a query to a response. The URL above demonstrates a vicious attack that can poison most DNS implementations (due to an implementation bug, that has since been corrected in newer DNSs), with about 800 packets!

The means to do so is by a "Birthday attack", which we will describe further.

The Birthday Paradox

An anecdote: if you know 23 people, there’s a >50% chance that two of them share the same birthday.

A major surprise: If you know 45 people, it’s almost 100% certain two of them share the same birthday

Assuming babies are born uniformly, that’s quite amazing!
(Given a 1/365.25 chance of being born on some day)

(C) 2004 - JL@Hisown.com - All Rights Reserved

The “Birthday Paradox” is a fun little party-game to try out, with surprising results. (It actually works!).

Even in random spaces, collisions can occur – and the paradox show that they can occur far more often than one would initially dare to presume.

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Birthdays? Random #s? What?

The “Birthday Paradox” extends to far more than birthdays!

ANY function over ANY sample space may be attacked

If the sample space size is **n**, *= 365*

even if the chance of a collision is **1/n**, *= Two people sharing the same birthday*

chances of a sample pair colliding are over 50%

if the amount of samples is O(**1.2sqrt(n)**).

*= 1.2 * 19.11 = 22.93 ~ 23*

(C) 2004 - JL@Hisown.com - All Rights Reserved

The above shows the actual application of the Paradox for any sample space, not just days in a year. The application known to most people is the intuitive birthday one, but this is just a special case thereof.

Remember the term “Collision”. For us, it means two people with the same birthdays. But, as we’ll see, this term has a very broad meaning, (and it’s almost always an insecure one).

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

The Birthday Paradox

The birthday paradox "breaks" DNS implementations, and would break them even if they had been truly random.

It's the most efficient attack vs. generic hash functions (described in the appendix)

(C) 2004 - JL@Hisown.com - All Rights Reserved

The paradox can thus be applied for DNS Query IDs (recall the case study URL, http://securityfocus.com/guest/17905/) for DNS cache poisoning, hash functions (as shown in the appendix) and more. In fact, the birthday paradox is the reason why session IDs as well as hash values are chosen to be 128-bits at the bare minimum – as their effective strength is at the square root of the message space size.

Case III: Netscape 4.x's SSL

In earlier versions of Netscape 4.x, the SSL session keys, as generated by the browser, were based on weak entropy – essentially, gettimeofday() and the pid.

As a result, SSL connections could be broken, once achieving the symmetric phase.

(C) 2004 - JL@Hisown.com - All Rights Reserved

When creating secret keys – if your source of entropy is weak – your keys will be easily guessable. Caveat emptor!

Case IV: Session IDs

Due to HTTPs statelessness, application programmers need to provide an alternate means to preserve state in between client calls.

Session IDs (passed in GET, POST, or via cookies) are the most common method used for this purpose.

However..

Guessing the Session ID would hijack the session!

(C) 2004 - JL@Hisown.com - All Rights Reserved

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Generating Session IDs

Most session IDs are generated by the underlying web server. AND BETTER THAT WAY.

Although sometimes faults do exist

e.g. WebSphere 4.0 Session IDs:

(see securiTeam post 2001-10/0008)

Still, sometimes you need your own session ID, for one purpose or the other. In that case, take note:

(C) 2004 - JL@Hisown.com - All Rights Reserved

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Generating Session IDs

Using Windows Globally Unique IDs is BAD practice.

GUIDs are 128-bit, BUT are derived from the MAC, current time and clock skew

(q.v. Internet Draft: draft-leach-uuids-guids)

Do not rely on GUIDs in any way, for security (this includes COM identifiers, etc).

(C) 2004 - JL@Hisown.com - All Rights Reserved

Attacks on Session IDs

Using Java Servlet IDs (128-bit) isn't that hot, either.

A recent (2/2005) paper by Zvi Gutterman demonstrates session IDs may be broken by an $O(2^{64})$ brute force attack.

While this is not currently feasible, it definitely borders on feasibility. Taking Moore's law into account, 128-bit SIDs may end up vulnerable soon.

(C) 2004 - JL@Hisown.com - All Rights Reserved

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Morals for Session IDs

In general, consider using additional secrets for sessions:

- The IP Address (if possible, e.g. Intranet apps)
- POST fields
- User-Agent fields, etc.
- Cookies (And - Consider HTTPOnly/Secure)

Additionally, beware of URL mangling if using external links (since browsers will send Referer:... string).

Last but not least – implicitly EXPIRE sessions

(C) 2004 - JL@Hisown.com - All Rights Reserved

The methods described in this slides mostly revolve around one idea: Add more entropy to your sessions, and distribute the secret over multiple channels. They are by NO means fool proof, and may be bypassed by a determined hacker. BUT, they do make life a lot harder.

The IP Address and User-Agent data is a good approach to make life a bit harder for the hacker, whilst the client remains totally agnostic. An attacker would now have to guess the exact IP, perform spoofing successfully, and know which browser the spoofed user is using. A user, however, will not switch an IP address (unless load balancers are involved) – and will not suddenly switch User-Agent from say, Firefox to Opera, while trying to maintain the same session. Supplying extra post fields, or more cookie data, is also a nice way of strengthening your session.

All the above imply ***defense in depth*** – Rather than relying solely on the session ID (which may be spoofed, or stolen via XSS), we employ several mechanisms. If one fails, it's enough for us to know something is amiss.

Finally, implicitly expire sessions after a given timeout. Most users are likely to just close their browser and not explicitly logout anyway. By expiring sessions, a hacker now has to attempt a hijack only while a user is actively conducting one – which is a LOT harder.

As an added bonus, implement auditing. This will clue you in when session hijack attempts are being conducted.

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

More Morals

If the need arises for a random number..
Be it a session ID, a key, or the roll of a virtual die:

- Do NOT use the default PRNGs
- If (available)
  use the secure variants
  Else
  read random data from OS sources

- ALWAYS ensure the PRNG is seeded properly

(C) 2004 - JL@Hisown.com - All Rights Reserved

Hash Functions

A HASH is a function that takes an arbitrary input (of any length) and produces a fixed length output, satisfying (among other things) three requirements:

- **Ease of Computation:** for any M, H(M) is easily deduced
- **Irreversibility**: H(M) yields no information on M.
- **Collision Resistance**: Given M, it is hard to find another M', that hashes to the very same value.

(C) 2004 - JL@Hisown.com - All Rights Reserved

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Hash Functions - Examples

This is better demonstrated in practice:

H(M) = 0 if M is even, 1 if M is odd (1-bit “hash” ☺)

H(M) = M mod 2^{32} (32-bit)

H(M) = CRC(M) (32-bit)

So, it is far from trivial to generate a hash function!

(the above are NOT real hash functions!)

(C) 2004 - JL@Hisown.com - All Rights Reserved

Better demonstrated by examples, this slide lists all sorts of “would-be” hashes, but – as we can see, they do not satisfy the requirements stated in the previous slide.

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Real Hash functions

The last slide didn't demonstrate real hashes, like these:

MD4: By Ron Rivest (used in Windows, obsolete) - 128 bit

MD5: By Ron Rivest (ubiquitous) – 128 bit

SHA-1: NSA(?) (likewise prevalent) – 160 bit

And numerous variants, as well as SHA-256, SHA-384, SHA-512…

(C) 2004 - JL@Hisown.com - All Rights Reserved

These are the "real" hashes, used in applications today. Hashes are prevalent in code signing, digital certificates, system fingerprints, and much, much more.

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Colliding Hashes

ANY Hash function will ALWAYS contain collisions!

(This is due to the Pigeon Hole Principle: take n pigeons, in n-1 pigeonholes – and two will have to squeeze in, or one stays out..)

(further, it will contain an INFINITE number of them!)

Remember: Colliding a hash = breaking it!

(C) 2004 - JL@Hisown.com - All Rights Reserved

Turns out, requirement #3 is the hardest to fulfill. Due to (yet another) mathematical principle, known as the “Pigeon Hole Principle”, collisions are inevitable. Not only that, but you can expect an infinite number of them! This is quite clear, if you think about compressing an infinite number of possibilities into such a tiny (yes, even 2^{160} is ‘tiny’ compared to infinity!) finite space (Think, the Genie in Aladdin “all the power in the world… in such a teeeeeny weeny space!” ☺).

So – that’s the bad news. Collisions are there, and an infinite number of them. Yet – we do not want any way to predict collisions.

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Colliding Hashes

The Birthday paradox enables a birthday attack that will break any hash, at a sqrt of the message space size.

That's why all hashes start out at higher (128 and greater) bit lengths.

Our aim is to make the birthday attack the most effective attack, while maintaining its unfeasibility.

(C) 2004 - JL@Hisown.com - All Rights Reserved

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

Colliding Hashes

The Birthday paradox enables a birthday attack that will break any hash, at a sqrt of the message space size.

That's why all hashes start out at higher (128 and greater) bit lengths.

Our aim is to make the birthday attack the most effective attack, while maintaining its unfeasibility.

(C) 2004 - JL@Hisown.com - All Rights Reserved

The birthday attack will have the attacker generating $2^{m/2}$ variants of one, innocuous message, and $2^{m/2}$ variants of another, bogus message. With >50% probability, there will be a matching pair. Say M_{51497} and $M'_{10241975}$ match. (Doesn't matter WHICH two match, so long as we find a match).

Thus, the attacker gives the innocuous end of the pair for someone to sign. The entity signing will not sign the message, but its hash(!). So while it signs the original message hash – it has in fact also signed its "evil twin" (this attack is also known as an evil twin attack in some places).

Note, this can be applied vs. ANY hash, no matter how perfect. The Birthday Paradox is an inherent mathematical fact. If your hashes were a mere 64-bit, it would suffice to generate $O(2^{32})$ message – about 4-8 Billion or so, to achieve great percentages of success. With 128-bit and above hashes, though, the $O(2^{64})$ needed is less feasible.

In order to break this "Evil Twin" connection, all one needs to do is simply insert a random change of any type in to the message given to sign. This would break the "quantum coupling", and radically alter the hash. So even though the attack is a valid threat, it is fairly easily mitigated.

(O(n) = "Order of" n. This is used to approximate the number of attempts required. Actual results may vary).

(C) 2004/2005 - JL@Hisown.com - All Rights Reserved

The Status (as of 4/6/2005)

MD4: Collisions found by using pen and paper!
(Wang, Feng et al, http://eprint.iacr.org/2004/199.pdf
and http://www.rsasecurity.com/rsalabs/node.asp?id=2738)

MD5: Collisions found at less than 2^{64}
(Wang, Feng et al, http://eprint.iacr.org/2004/199.pdf)

SHA-0*: Effectively collided by numerous people
(Biham, Chen, Joux, and others)

SHA-1: Collisions found at 2^{69}
(Wang, Yin, et al, 2/15/2005)

* - See note, below

(C) 2004 - JL@Hisown.com - All Rights Reserved

The “BIG Deal” about all the recent developments is, that it turns out that vs. particular functions, there exist ways far more efficient than the birthday paradox to find arbitrary collisions. The most recent development (at the time of writing) brings SHA-1 from 2^{80} to 2^{69}. Not exactly feasible, still, but this result is likely to be improved as time goes by.

SHA-0 is the (flawed) precursor of SHA-1. It is worth noting, that it has been recalled by the NSA (fairly quickly, back in the mid ’90s), and SHA-1 was set to the standard. The real difference between the SHAs is a mere left shift of just one bit(!) in one of the phase. But that, to paraphrase Frost, has made all the difference.

The good(?) news

We're not doomed!! (yet)

2/15/2005 – Moscone Center, SF, CA

These attacks are STILL not a 2nd pre-image attack.

(when THAT happens, kiss your hash goodbye).

(C) 2004 - JL@Hisown.com - All Rights Reserved

This is mentioned in Bruce Schneier's great Blog:

http://www.schneier.com/blog/archives/2005/02/sha1_broken.html

And further elaborated in:

http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

(If you haven't picked up a copy of "Applied Cryptography" 2nd Edition – now's a great time to do so!)

Should a 2nd-Preimage attack be concocted, by some wily hacker, Chinese Wizard, or NSA expert – the foundations of Internet security will truly crumble. Digital Certificates could be forged. Viruses could be embedded in signed code. Signatures would no longer be acceptable. BUT.. That has yet to happen (at the time of writing!).

The End (for now)

(C) 2004 - JL@Hisown.com - All Rights Reserved