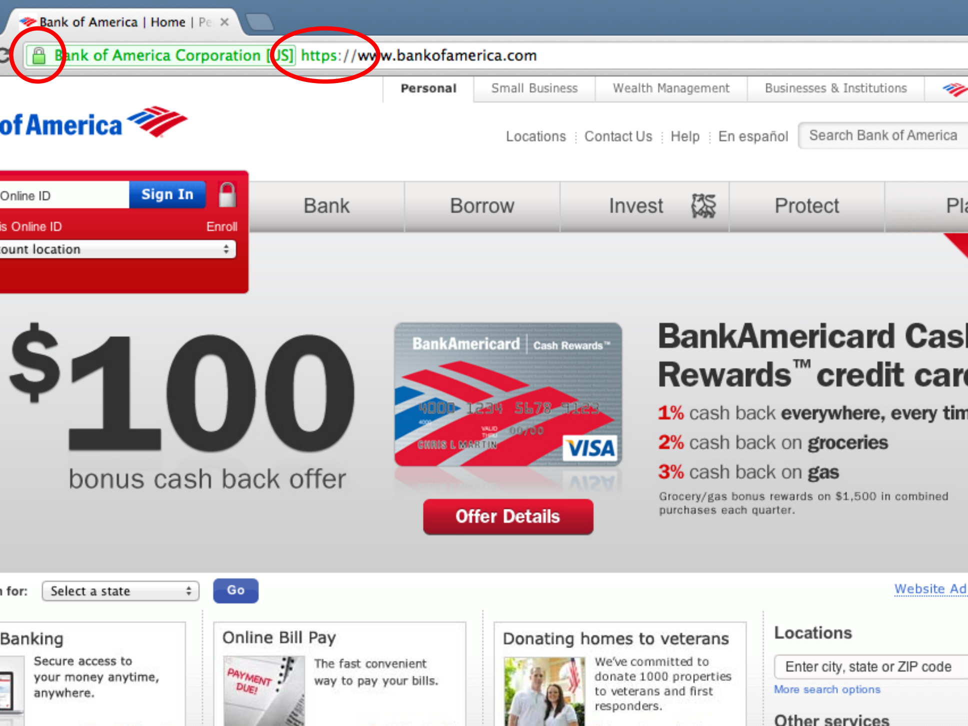
How not to generate random numbers

Nadia Heninger

University of Pennsylvania

May 13, 2015



Bank of America | Home | Pe

Bank of America Corporation [US] https://www.bankofamerica.com

Personal | Small Business | Wealth Management | Businesses & Institutions

of America

Locations | Contact Us | Help | En español | Search Bank of America

Online ID | Sign In

is Online ID | Enroll

ount location

Bank | Borrow | Invest | Protect | Pla

$100
bonus cash back offer

BankAmericard Cash Rewards™

4000 1234 5678 9123

CHRIS L MARTIN

VISA

Offer Details

BankAmericard Cas
Rewards™ credit car

1% cash back **everywhere, every tim**

2% cash back on **groceries**

3% cash back on **gas**

Grocery/gas bonus rewards on $1,500 in combined purchases each quarter.

n for: Select a state | Go

Website Ad

Banking

Secure access to your money anytime, anywhere.

Online Bill Pay

PAYMENT DUE!

The fast convenient way to pay your bills.

Donating homes to veterans

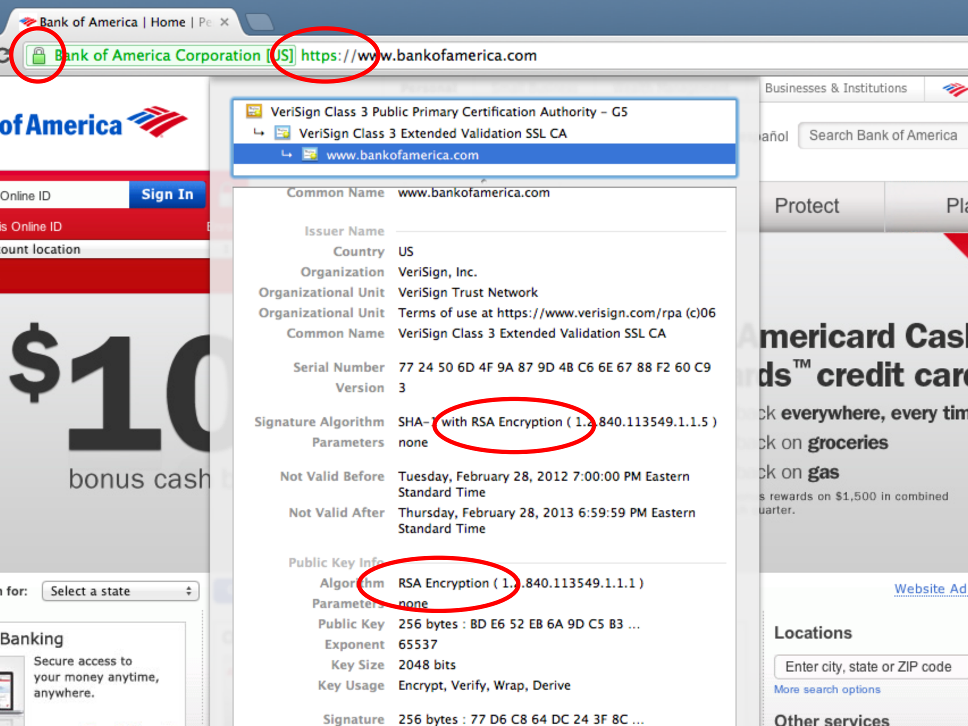
We've committed to donate 1000 properties to veterans and first responders.

Locations

Enter city, state or ZIP code

More search options

Other services



VeriSign Class 3 Public Primary Certification Authority – G5
↳ VeriSign Class 3 Extended Validation SSL CA
↳ www.bankofamerica.com

Common Name	www.bankofamerica.com
Issuer Name	
Country	US
Organization	VeriSign, Inc.
Organizational Unit	VeriSign Trust Network
Organizational Unit	Terms of use at https://www.verisign.com/rpa (c)06
Common Name	VeriSign Class 3 Extended Validation SSL CA
Serial Number	77 24 50 6D 4F 9A 87 9D 4B C6 6E 67 88 F2 60 C9
Version	3
Signature Algorithm	SHA-1 with RSA Encryption (1.2.840.113549.1.1.5)
Parameters	none
Not Valid Before	Tuesday, February 28, 2012 7:00:00 PM Eastern Standard Time
Not Valid After	Thursday, February 28, 2013 6:59:59 PM Eastern Standard Time
Public Key Info	
Algorithm	RSA Encryption (1.2.840.113549.1.1.1)
Parameters	none
Public Key	256 bytes : BD E6 52 EB 6A 9D C5 B3 ...
Exponent	65537
Key Size	2048 bits
Key Usage	Encrypt, Verify, Wrap, Derive
Signature	256 bytes : 77 D6 C8 64 DC 24 3F 8C ...

Textbook RSA

[Rivest Shamir Adleman 1977]

Public Key

$N = pq$ modulus

e encryption exponent

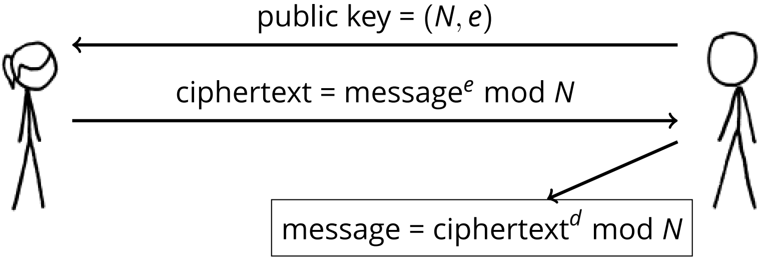
Private Key

p, q primes

d decryption exponent

$(d = e^{-1} \bmod (p-1)(q-1))$

Encryption



public key = (N, e)

ciphertext = $\text{message}^e \bmod N$

message = $\text{ciphertext}^d \bmod N$

Textbook RSA

[Rivest Shamir Adleman 1977]

Public Key

$N = pq$ modulus

e encryption exponent

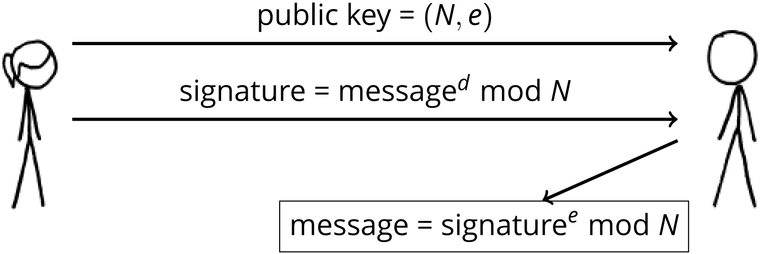
Private Key

p, q primes

d decryption exponent

$(d = e^{-1} \bmod (p-1)(q-1))$

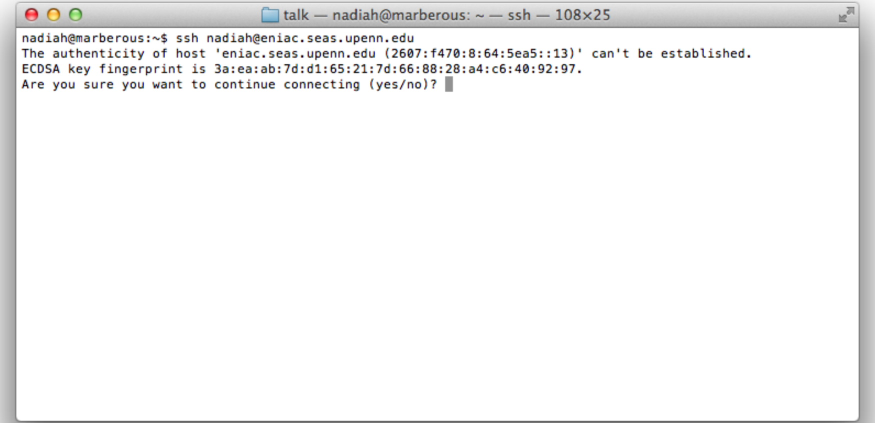
Signing



public key = (N, e)

signature = message^d mod N

$\boxed{\text{message} = \text{signature}^e \bmod N}$



```
nadiah@marberous:~$ ssh nadiah@eniac.seas.upenn.edu
The authenticity of host 'eniac.seas.upenn.edu (2607:f470:8:64:5ea5::13)' can't be established.
ECDSA key fingerprint is 3a:ea:ab:7d:d1:65:21:7d:66:88:28:a4:c6:40:92:97.
Are you sure you want to continue connecting (yes/no)? 
```

Textbook Diffie-Hellman

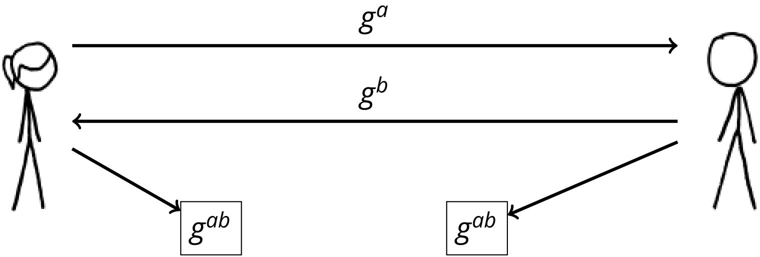
[Diffie Hellman 1976]

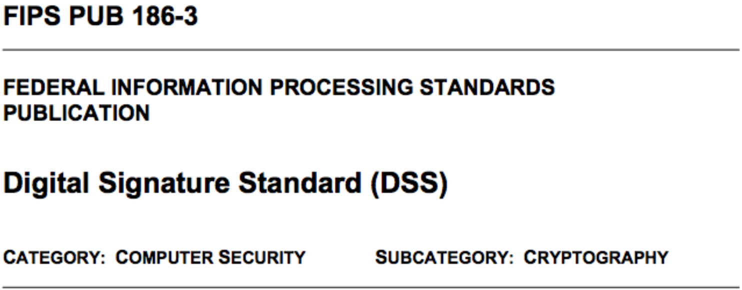
Public Parameters

G a group (e.g. $\mathbb{F}_p$, or an elliptic curve)

g group generator

Key Exchange





FIPS PUB 186-3

FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION

Digital Signature Standard (DSS)

CATEGORY: COMPUTER SECURITY **SUBCATEGORY: CRYPTOGRAPHY**

DSA/ECDSA Public Key

G group parameters

g group generator

$y = g^x$

Private Key

x private key

Motivating question:

What does cryptography look like on a broad scale?

Methodology:

1. Collect cryptographic data (keys, signatures...)
2. Look for interesting things.

Data Collection

Collecting HTTPS data

(Heninger, Durumeric, Wustrow, Halderman 2012)
(Durumeric, Wustrow, Halderman 2013)

zmap

Methodology:

- Scan entire IPv4 space on port 443.
- Download HTTPS certificates from live hosts.

Open port	Handshake	RSA	DSA	ECDSA	GOST
28,900,000	12,800,000	5,600,000	6,000	8	200

Scanning tools available at `zmap.io`, data at `scans.io`.

SSH

(Heninger, Durumeric, Wustrow, Halderman 2012)
(Bos, Halderman, Heninger, Moore, Naehrig, Wustrow 2013)

Methodology:

- Scan entire IPv4 space on port 22.
- Download host public keys, signatures, Diffie-Hellman key exchange.

Open port	Handshake	RSA	DSA	ECDSA	GOST
23,000,000	12,000,000	10,900,000	9,900,000	1,200,000	114

PGP

(Lenstra, Hughes, Augier, Bos, Kleinjung, Wachter 2012)

XKCD

PGP keys are used to

- sign and encrypt email messages.

Methodology:

- Download PGP key repository dump containing public keys, signatures.

RSA keys	DSA keys	ElGamal keys
700,000	2,100,000	2,100,000

Bitcoin

(Bos, Halderman, Heninger, Moore, Naehrig, Wustrow 2013)

Bitcoin uses ECDSA.

Addresses are public keys, transactions contain signatures.

Block chain is transferred to bitcoin clients.
Can also be downloaded in bulk.

August 2013:

keys	transactions
15,291,112	22,159,078

Taiwan Citizen Digital Certificate Smartcards

(Bernstein, Chang, Cheng, Chou, Heninger, Lange, van Someren 2013)

Taiwan's smart card IDs allow citizens to

- file income taxes,
- update car registrations,
- transact with government agencies,
- interact with companies (e.g. Chunghwa Telecom) online.

March 2012: Collected 3,002,000 certificates (all using RSA keys) from national LDAP directory.

2.3 million distinct 1024-bit RSA moduli, 700,000 2048-bit.

Cryptography relies on good randomness.

If you use bad randomness, an attacker might be able to guess your private key.

End of story?

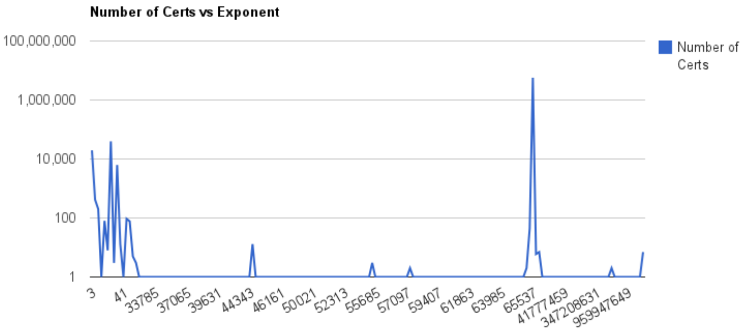
What could go wrong: Repeated keys

RSA Public Keys

$N = pq$ modulus

e encryption exponent

- Two hosts share e: not a problem.



What could go wrong: Repeated keys

RSA Public Keys

$N = pq$ modulus

e encryption exponent

- Two hosts share e: not a problem.
- Two hosts share N: $\rightarrow$ both know private key of the other.

Hosts share the same public and private keys, and can decrypt and sign for each other.

What could go wrong: Shared factors

If two RSA moduli share a common factor,

$$N_1 = pq_1 \qquad\qquad N_2 = pq_2$$

What could go wrong: Shared factors

If two RSA moduli share a common factor,

$$N_1 = pq_1 \qquad\qquad N_2 = pq_2$$

$$\mathrm{gcd}(N_1, N_2) = p$$

You can factor both keys with GCD algorithm.

Time to factor
768-bit RSA modulus:
2.5 calendar years
[Kleinjung et al. 2010]

Time to calculate GCD
for 1024-bit RSA moduli:
15μs

What could go wrong: Repeated DSA/ECDSA keys

DSA Public Key

G, g domain parameters

$y = g^x$

Private Key

x private key

- Two hosts have same public key → both know private key of the other.

What could go wrong: Weak DSA/ECDSA signatures

Public Key

G, g domain parameters

$y = g^x$

Private Key

x private key

DSA and ECDSA signatures contain a random nonce.

- DSA nonce known $\rightarrow$ easily compute private key.

What could go wrong: Weak DSA/ECDSA signatures

Public Key

G, g domain parameters

$y = g^x$

Private Key

x private key

DSA and ECDSA signatures contain a random nonce.

- DSA nonce known $\rightarrow$ easily compute private key.
- DSA nonce reused to sign distinct messages $\rightarrow$ easily compute nonce.

Should we expect to find key collisions in the wild?

Experiment: Compute GCD of each pair of *M* RSA moduli randomly chosen from *P* primes.

What *should* happen? **Nothing.**

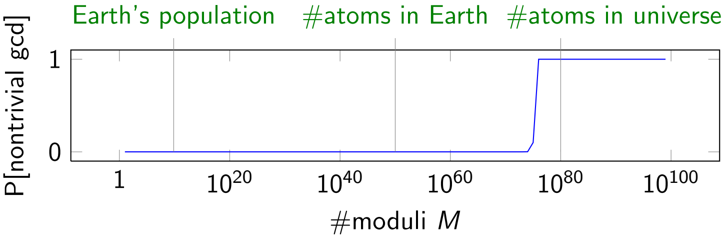
Should we expect to find key collisions in the wild?

Experiment: Compute GCD of each pair of M RSA moduli randomly chosen from P primes.

What *should* happen? **Nothing.**

Prime Number Theorem:
$\sim 10^{150}$ 512-bit primes

Birthday bound:
$\Pr[\text{nontrivial gcd}] \approx 1 - e^{-2M^2/P}$



How to efficiently compute pairwise GCDs

Computing pairwise gcd(N_i, N_j) the naive way on all of the RSA keys in the above datasets would take

$$15\mu\text{s} \times \binom{14 \times 10^6}{2} \text{pairs} \approx 1100 \text{ years}$$

of computation time.

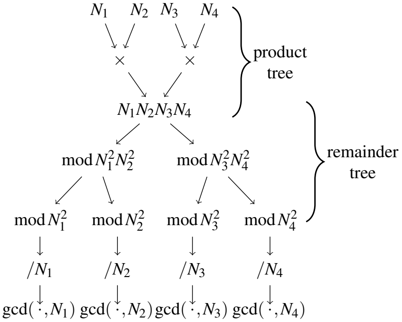
How to efficiently compute pairwise GCDs

Computing pairwise $\gcd(N_i, N_j)$ the naive way on all of the RSA keys in the above datasets would take

$$15\mu s \times \xcancel{\binom{14 \times \text{[illegible]}^{6}}{2}} \text{ pairs} \approx 1100 \text{ years}$$

of computation time.

Algorithm from (Bernstein 2004)
A few hours for datasets.
Implementation available at
https://factorable.net.



What *does* happen when we GCD all the keys?

What *does* happen when we GCD all the keys?

Compute private keys for

- 64,081 HTTPS servers (0.50%).
- 2,459 SSH servers (0.03%).
- 2 PGP users (and a few hundred invalid keys).
- 103 Taiwanese citizens.

What happens if we look for repeated DSA nonces?

Compute private keys for

- 105,728 (1.03%) of SSH DSA servers.
- 158 Bitcoin addresses.

What happens if we look for repeated keys?

$>$ 60% of HTTPS and SSH hosts served non-unique public keys.

What happens if we look for repeated keys?

$>$ 60% of HTTPS and SSH hosts served non-unique public keys.

Many valid (and common) reasons to share keys:

- Shared hosting situations. Virtual hosting.
- A single organization registers many domain names with the same key.
- Expired certificates that are renewed with the same key.

What happens if we look for repeated keys?

$>$ 60% of HTTPS and SSH hosts served non-unique public keys.

Common (and unwise) reasons to share keys:

- Device default certificates/keys.
- Apparent entropy problems in key generation.
- Virtual machine snapshots post key-generation.

What happens if we look for repeated keys?

$>$ 60% of HTTPS and SSH hosts served non-unique public keys.

Common (and unwise) reasons to share keys:

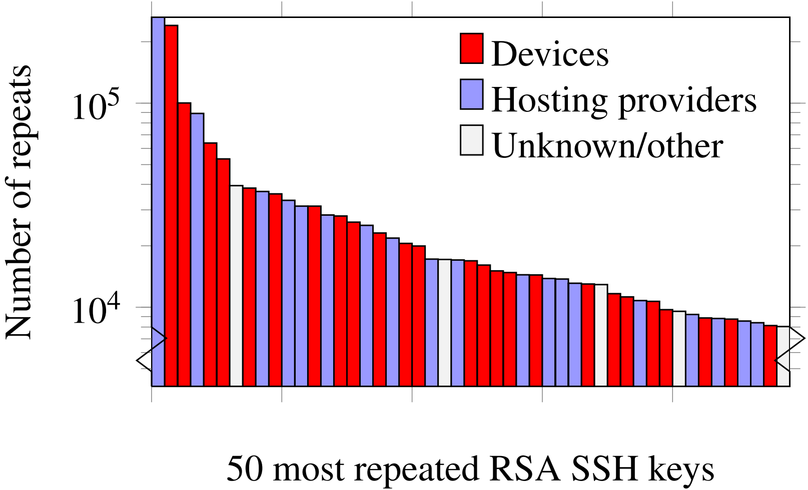
- Device default certificates/keys.
- Apparent entropy problems in key generation.
- Virtual machine snapshots post key-generation.

HTTPS:
default certificates/keys:
670,000 hosts (5%)

low-entropy repeated keys:
40,000 hosts (0.3%)

SSH:
default or low-entropy keys:
1,000,000 hosts (10%)

Classifying repeated keys



... only two of the factored https certificates were signed by a CA, and both are expired. The web pages aren't active.

… only two of the factored https certificates were signed by a CA, and both are expired. The web pages aren't active.

Look at subject information for certificates:

```
CN=self-signed, CN=system generated, CN=0168122008000024
CN=self-signed, CN=system generated, CN=0162092009003221
CN=self-signed, CN=system generated, CN=0162122008001051
C=CN, ST=Guangdong, O=TP-LINK Technologies CO., LTD., OU=TP-LINK SOFT, CN=TL-R478+1145D5C30089/emailAddres
C=CN, ST=Guangdong, O=TP-LINK Technologies CO., LTD., OU=TP-LINK SOFT, CN=TL-R478+139819C30089/emailAddres
CN=self-signed, CN=system generated, CN=0162072011000074
CN=self-signed, CN=system generated, CN=0162122009008149
CN=self-signed, CN=system generated, CN=0162122009000432
CN=self-signed, CN=system generated, CN=0162052010005821
CN=self-signed, CN=system generated, CN=0162072008005267
C=US, O=2Wire, OU=Gateway Device/serialNumber=360617088769, CN=Gateway Authentication
CN=self-signed, CN=system generated, CN=0162082009008123
CN=self-signed, CN=system generated, CN=0162072008005385
CN=self-signed, CN=system generated, CN=0162082008000317
C=CN, ST=Guangdong, O=TP-LINK Technologies CO., LTD., OU=TP-LINK SOFT, CN=TL-R478+3F5878C30089/emailAddres
CN=self-signed, CN=system generated, CN=0162072008005597
CN=self-signed, CN=system generated, CN=0162072010002630
CN=self-signed, CN=system generated, CN=0162032010008958
CN=109.235.129.114
CN=self-signed, CN=system generated, CN=0162072011004982
CN=217.92.30.85
CN=self-signed, CN=system generated, CN=0162112011000190
CN=self-signed, CN=system generated, CN=0162062008001934
CN=self-signed, CN=system generated, CN=0162112011004312
CN=self-signed, CN=system generated, CN=0162072011000946
C=US, ST=Oregon, L=Wilsonville, CN=141.213.19.107, O=Xerox Corporation, OU=Xerox Office Business Group,
CN=XRX0000AAD53FB7.eecs.umich.edu, CN=(141.213.19.107|XRX0000AAD53FB7.eecs.umich.edu)
CN=self-signed, CN=system generated, CN=0162102011001174
CN=self-signed, CN=system generated, CN=0168112011001015
CN=self-signed, CN=system generated, CN=0162012011000446
```

Attributing SSL and SSH vulnerabilities to implementations

Evidence strongly suggested *widespread implementation problems*.

Clue #1: Vast majority of weak keys generated by network devices:

- Juniper network security devices
- Cisco routers
- IBM server management cards
- Intel server management cards
- Innominate industrial-grade firewalls
- . . .

Identified devices from $>$ 50 manufacturers

Random number generation in software

crypto keys

Random number generation in software

crypto keys

To generate random keys, we need a source of randomness.

Random number generation in software

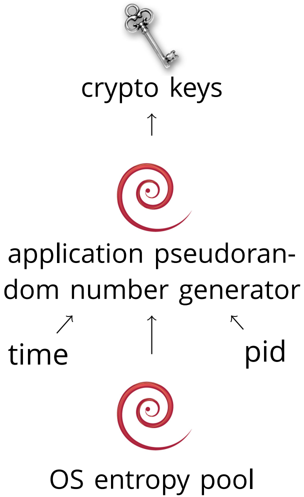
crypto keys

↑

application pseudorandom number generator

To generate random keys, we need a source of randomness.

Random number generation in software



crypto keys

↑

application pseudorandom number generator

↗ ↑ ↖

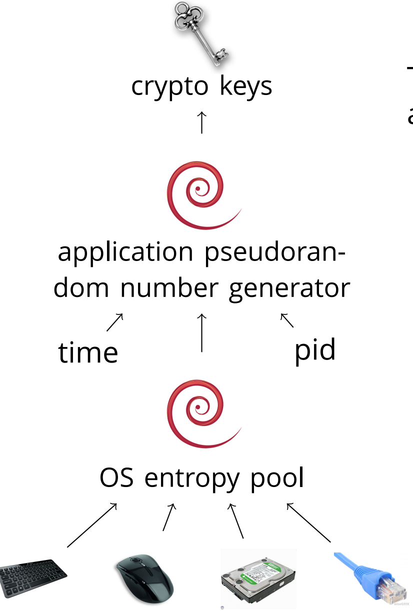
time pid

OS entropy pool

To generate random keys, we need a source of randomness.

> *"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin."*
>
> –John von Neumann

Random number generation in software



crypto keys

↑

application pseudorandom number generator

↗ ↑ ↖

time pid

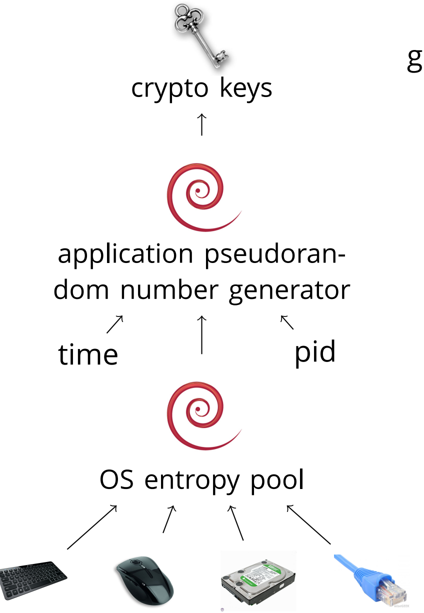
OS entropy pool

↗ ↗ ↖ ↖

To generate random keys, we need a source of randomness.

> *"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin."*
>
> –John von Neumann

Random number generation in software



crypto keys

application pseudorandom number generator

time

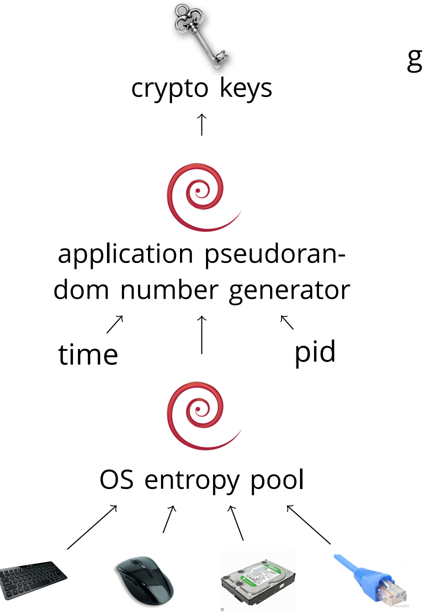
pid

OS entropy pool

Hypothesis: Devices automatically generate crypto keys on first boot.

Random number generation in software

Hypothesis: Devices automatically generate crypto keys on first boot.



crypto keys

↑

application pseudorandom number generator

↗ ↑ ↖

time

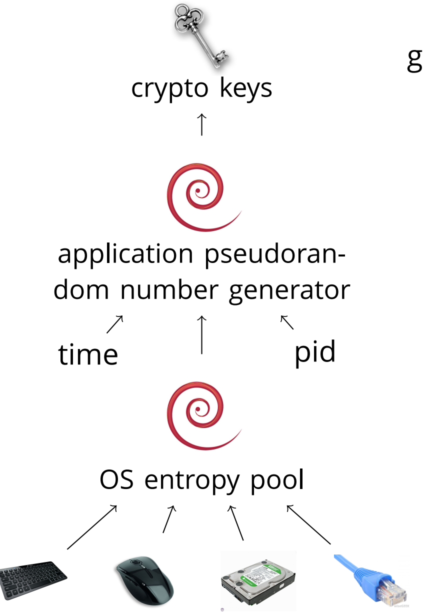
pid

OS entropy pool

↗ ↗ ↖ ↖

- Headless or embedded devices may lack these entropy sources.

Random number generation in software



crypto keys

application pseudorandom number generator

time

pid

OS entropy pool

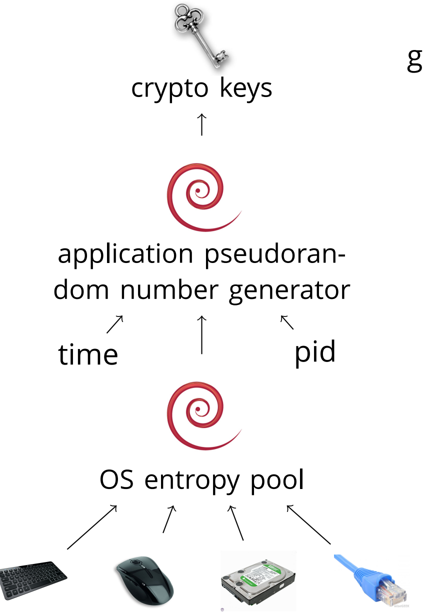
Hypothesis: Devices automatically generate crypto keys on first boot.

- OS random number generator may not have incorporated any entropy when queried by software.

- Headless or embedded devices may lack these entropy sources.

Random number generation in software

Hypothesis: Devices automatically generate crypto keys on first boot.



crypto keys

application pseudorandom number generator

time

pid

OS entropy pool

- OS random number generator may not have incorporated any entropy when queried by software.

  Experimentally verified Linux “boot-time entropy hole”

- Headless or embedded devices may lack these entropy sources.

Linux random number generators

`/dev/random`	`/dev/urandom`
“high-quality” randomness	pseudorandomness
blocks if insufficient entropy available	never blocks

“As a general rule, `/dev/urandom` should be used for everything except long-lived GPG/SSL/SSH keys.”—`man random`

`random`’s conservative blocking behavior is a usability problem.

This results in many developers using `urandom` for cryptography.

```
 /* We’ll use /dev/urandom by default, since
/dev/random is too much hassle.  If system developers
aren’t keeping seeds between boots nor getting any
entropy from somewhere it’s their own fault.  */
#define DROPBEAR_RANDOM_DEV "/dev/urandom"
```

Generating vulnerable RSA keys in software

- Insufficiently random seeds for pseudorandom number generator $\implies$ we should see repeated keys.

```
prng.seed()
p = prng.random_prime()
q = prng.random_prime()
N = p*q
```

- We do:
  - $>$ 60% of hosts share keys
  - At least 0.3% due to bad randomness.
- Repeated keys may be a sign that implementation is vulnerable to a targeted attack.

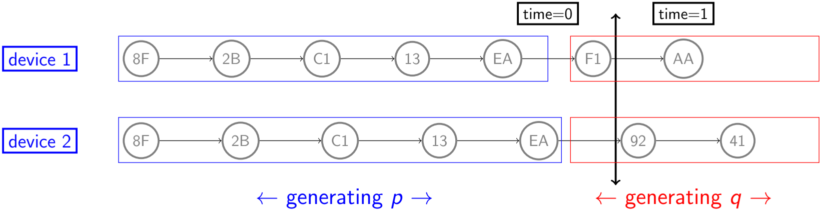
But why do we see factorable keys?

Generating factorable RSA keys in software

```
prng.seed()
p = prng.random_prime()
prng.add_randomness()
q = prng.random_prime()
N = p*q
```

OpenSSL adds time in seconds

Insufficient randomness can lead to factorable keys.



Experimentally verified OpenSSL generates factorable keys in this situation.

Devices generating weak DSA signatures

Step 1: Low-entropy DSA key generation

Step 2: Low-entropy seed for PRNG generating signature nonce.

Host 1	Host 2
50	84
58	24
9	13
36	89
84	85
24	68
13	52
89	69
85	47

Step 3: Two sequences in same state → colliding nonces.

Investigating Taiwanese smartcard weak keys

Most common factor appears 46 times

```
c0000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
000000000000000000000000000002f9
```

Investigating Taiwanese smartcard weak keys

Most common factor appears 46 times

```
c0000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
000000000000000000000000000002f9
```

which is the next prime after $2^{511} + 2^{510}$.

Investigating Taiwanese smartcard weak keys

Most common factor appears 46 times

```
c0000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
000000000000000000000000000002f9
```

which is the next prime after $2^{511} + 2^{510}$.
The next most common factor, repeated 7 times, is

```
c9242492249292499249492449242492
24929249924949244924249224929249
92494924492424922492924992494924
492424922492924992494924492424e5
```

Factored 80 more keys by extrapolating patterns.

Why are government-certified smartcards generating weak keys?

Best practices and standards in hardware random number generation require

- designers to characterize entropy from source
- testing of the signal from the entropy source at run time
- post-processing by running output through cryptographic hash function

These cards are clearly doing none of these things, even though they claimed FIPS compliance.

Why are government-certified smartcards generating weak keys?

Best practices and standards in hardware random number generation require

- designers to characterize entropy from source
- testing of the signal from the entropy source at run time
- post-processing by running output through cryptographic hash function

These cards are clearly doing none of these things, even though they claimed FIPS compliance.

Hypothesized failure:

- Hadware RNG has underlying weakness that causes failure in some situations.
- Card software not operated in FIPS mode
  $\Longrightarrow$ no testing or post-processing RNG output

PGP: Implementation errors?

Why did GCD factor two PGP keys?

They were both $>$ 10 years old.

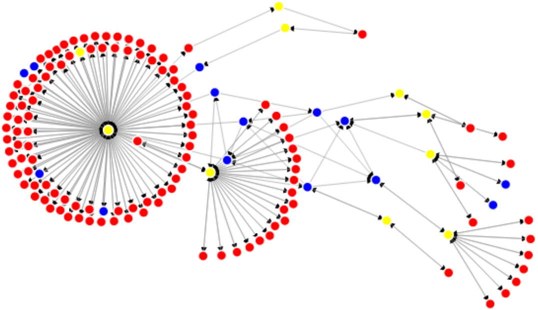
Seems to have been a rare implementation error.

Bitcoin

Several explanations so far:

- Android Java RNG vulnerability publicized August 2013.
- Test implementations.
- Developer error in uncommon bitcoin implementations.

Bitcoin address `1HKywxiL4JziqXrzLKhmB6a74ma6kxbSDj` has stolen 59 bitcoins from weak addresses so far.



red = vulnerable keys

Disclosure for HTTPS and SSH vulnerabilities

- Wrote disclosures to 61 companies.
- 13 had Security Incident Response Team contact information available.
- Received responses from 28.
- 13 told us they fixed the problem
- 5 informed us of security advisories
- Coordinated through US-CERT, ICS CERT, JP-CERT
- Linux kernel has been patched.
- Since publication in August 2012, 20% decrease in number of hosts serving factorable RSA keys.

Disclosure for Taiwan ID card vulnerabilities

Disclosed vulnerability to Taiwan MOICA (Ministry of Interior).

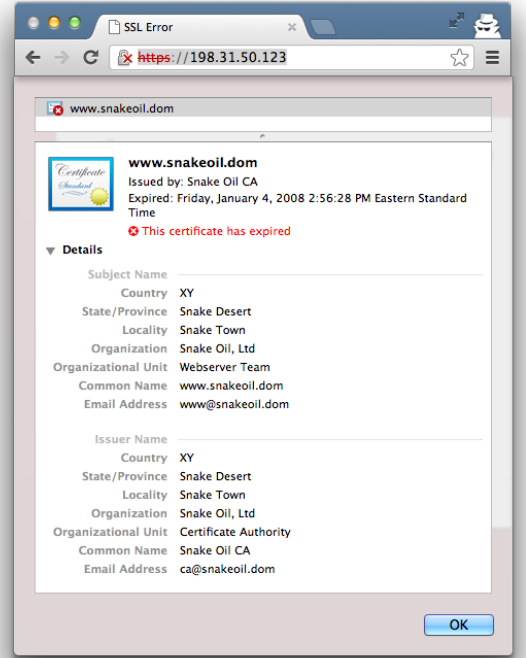
- Have replaced cards for users directly impacted by GCD vulnerabilities.
- Promised to replace cards from particular vulnerable batch.

Gallery of horrors

Debian RNG

Debian weak keys served on:

- 4,147 (0.03%) of HTTPS hosts
- 31,111 (0.34%) of RSA SSH hosts
- 22,030 (0.34%) of DSA SSH hosts



www.snakeoil.dom

www.snakeoil.dom

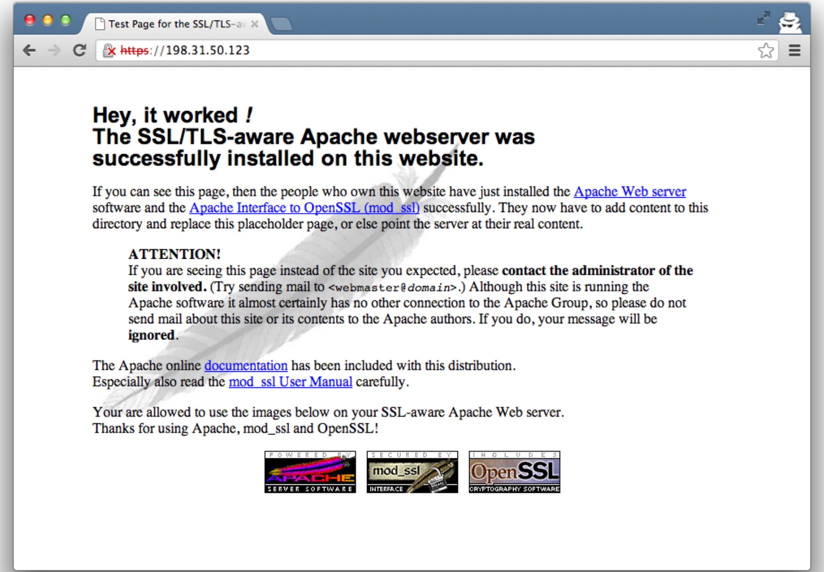
Issued by: Snake Oil CA

Expired: Friday, January 4, 2008 2:56:28 PM Eastern Standard Time

This certificate has expired

Details

Subject Name	
Country	XY
State/Province	Snake Desert
Locality	Snake Town
Organization	Snake Oil, Ltd
Organizational Unit	Webserver Team
Common Name	www.snakeoil.dom
Email Address	www@snakeoil.dom
Issuer Name	
Country	XY
State/Province	Snake Desert
Locality	Snake Town
Organization	Snake Oil, Ltd
Organizational Unit	Certificate Authority
Common Name	Snake Oil CA
Email Address	ca@snakeoil.dom



Hey, it worked !
The SSL/TLS-aware Apache webserver was successfully installed on this website.

If you can see this page, then the people who own this website have just installed the Apache Web server software and the Apache Interface to OpenSSL (mod_ssl) successfully. They now have to add content to this directory and replace this placeholder page, or else point the server at their real content.

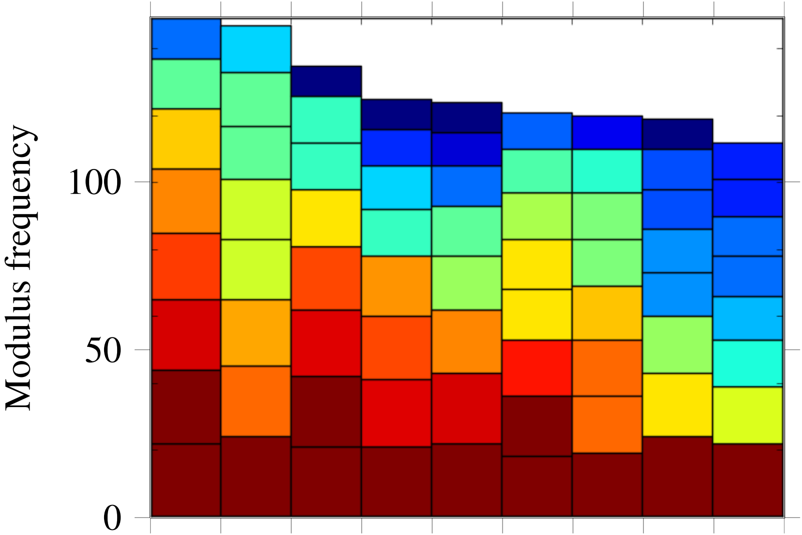
> **ATTENTION!**
> If you are seeing this page instead of the site you expected, please **contact the administrator of the site involved.** (Try sending mail to `<webmaster@`*`domain`*`>`.) Although this site is running the Apache software it almost certainly has no other connection to the Apache Group, so please do not send mail about this site or its contents to the Apache authors. If you do, your message will be **ignored**.

The Apache online documentation has been included with this distribution.
Especially also read the mod_ssl User Manual carefully.

Your are allowed to use the images below on your SSL-aware Apache Web server.
Thanks for using Apache, mod_ssl and OpenSSL!

Distribution of prime factors

IBM Remote Supervisor Adapter II and Bladecenter Management Module



Practical mitigations

Developers and manufacturers:

- Defense in depth: test, post-process, use multiple sources of randomness.
- Gather entropy more aggressively, add hardware sources.
- Seed devices with entropy at the factory.
- Generate keys on use rather than on boot.

CAs:

- Test for repeated, factorable, and other weak keys.

Users:

- Check against known weak keys. (See factorable.net)
- Replace default certificates.

Weak keys: Lessons

Systems:

- New insights from taking a macroscopic view of crypto practice.
- Cryptographic entropy is hard to get right.

Cryptography:

- Need to design cryptosystems resilient to random number generation problems. (“Hedged” crypto)

Theory:

- Many interesting algorithmic problems related to efficiently and obliviously mining data sets for cryptographic vulnerabilities.

Mining your Ps and Qs: Widespread Weak Keys in Network Devices
Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman *Usenix Security 2012* https://factorable.net

“Ron was wrong, Whit is right” published as
Public Keys Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter *Crypto 2012*

Elliptic Curve Cryptography in Practice Joppe W. Bos, J. Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow.

Factoring RSA keys from certified smart cards: Coppersmith in the wild
Daniel J. Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko van Someren, Asiacrypt 2013.