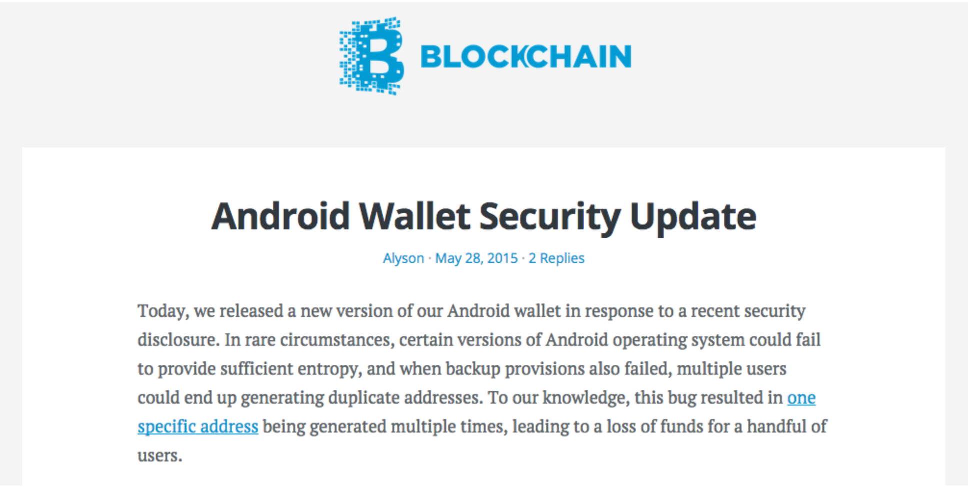
How not to generate random numbers

Nadia Heninger

University of Pennsylvania

June 5, 2015

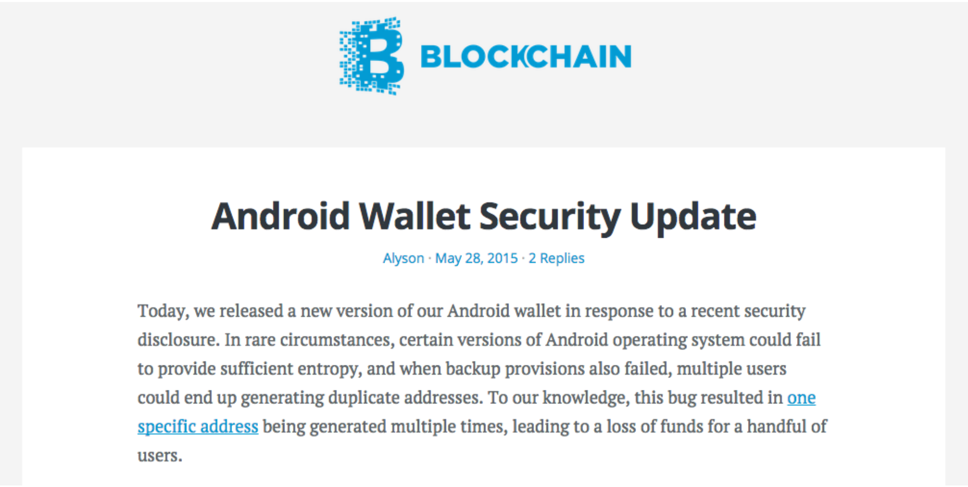


BLOCKCHAIN

Android Wallet Security Update

Alyson · May 28, 2015 · 2 Replies

Today, we released a new version of our Android wallet in response to a recent security disclosure. In rare circumstances, certain versions of Android operating system could fail to provide sufficient entropy, and when backup provisions also failed, multiple users could end up generating duplicate addresses. To our knowledge, this bug resulted in one specific address being generated multiple times, leading to a loss of funds for a handful of users.



BLOCKCHAIN

Android Wallet Security Update

Alyson · May 28, 2015 · 2 Replies

Today, we released a new version of our Android wallet in response to a recent security disclosure. In rare circumstances, certain versions of Android operating system could fail to provide sufficient entropy, and when backup provisions also failed, multiple users could end up generating duplicate addresses. To our knowledge, this bug resulted in one specific address being generated multiple times, leading to a loss of funds for a handful of users.

/u/murbul:

And the final mistake: They were using HTTP instead of HTTPS to make the webservice call to random.org. On Jan 4, random.org started enforcing HTTPS and returning a 301 Permanently Moved error for HTTP - see https://www.random.org/news/. So since that date, the entropy has actually been the error message (turned into bytes) instead of the expected 256-bit number. Using that seed, SecureRandom will generate the private key for address 1Bn9ReEocMG1WEW1qYjuDrdFzEFFDCq43F 100% of the time. Ouch. This is around the time that address first appears, so the timeline matches.

Motivating question:

What does cryptography look like on a broad scale?

Methodology:

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

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. *Financial Cryptography 2014*

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.*

Data Collection

Collecting HTTPS data

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

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)

PGP keys are used to

- sign and encrypt email messages.

HOW TO USE PGP TO VERIFY THAT AN EMAIL IS AUTHENTIC:
LOOK FOR THIS TEXT AT THE TOP:
REPLY
-----BEGIN PGP SIGNED MESSAGE-----
HASH: SHA256
HEY,
IF IT'S THERE, THE EMAIL IS PROBABLY FINE.

XKCD

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:

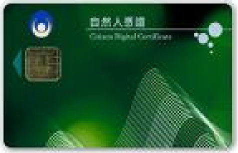
keys	transactions
15,291,112	22,159,078

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?

Are repeated public exponents a problem?

RSA Public Keys

$N = pq$ modulus

e encryption exponent

TLS e values	
65537	5,689,766
17	39,637
3	19,629
35	6,272
5	418
7	201
47	94
11	80
59	77
65535	44
37	13
44611	13
13	8
65543	7
2147483647	7
65539	6
257	5
19	3

Are repeated public moduli a problem?

Public Key		Private Key	
$N = pq$	modulus	p, q	primes
e	encryption exponent	d	decryption exponent ($d = e^{-1} \bmod (p-1)(q-1)$)

- Two hosts share N: $\rightarrow$ both know private key of the other. Factorization is unique.

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

What happens if we look for repeated moduli?

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

What happens if we look for repeated moduli?

$>$ 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 moduli?

$>$ 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.

What happens if we look for repeated moduli?

$>$ 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.

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%)

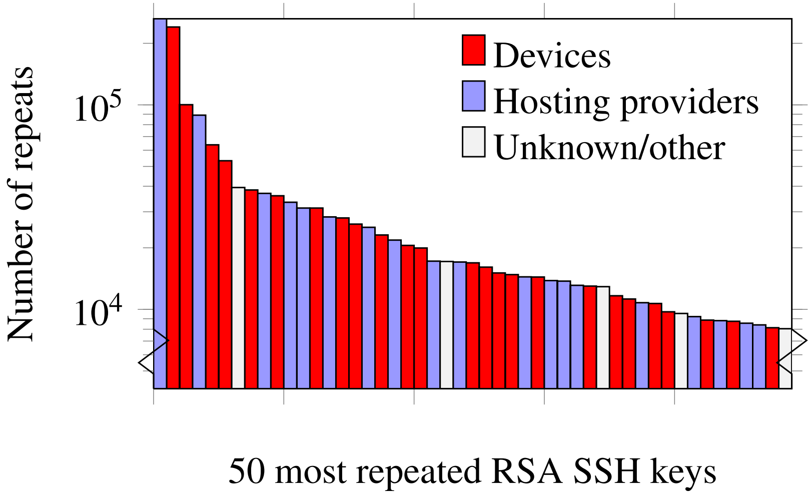
Subjects of most repeated TLS Certificates

```
C=TW, ST=HsinChu, L=HuKou, O=DrayTek Corp., OU=DrayTek Support, CN=Vigor Rou
C=UA, ST=Califonia, L=Irvine, O=Broadcom, OU=Broadband, CN=Daniel/emailAddre
C=US, ST=AL, L=Huntsville, O=ADTRAN, Inc., CN=NetVanta/emailAddress=tech.sup
C=CA, ST=Quebec, L=Gatineau, O=Axentraserver Default Certificate 863B4AB, CN
C=US, ST=California, L=Santa Clara, O=NETGEAR Inc., OU=Netgear Prosafe, CN=N
C=--, ST=SomeState, L=SomeCity, O=SomeOrganization, OU=SomeOrganizationalUni
C=US, ST=Texas, L=Round Rock, O=Dell Inc., OU=Remote Access Group, CN=iDRAC6
C=--, ST=SomeState, L=SomeCity, O=SomeOrganization, OU=SomeOrganizationalUni
C=IN, ST=WA, L=WA, O=lxlabs, OU=web, CN=*.lxlabs.com/emailAddress=sslsign@lx
C=TW, ST=none, L=Taipei, O=NetKlass Techonoloy Inc, OU=NetKlass, CN=localhos
C=--, ST=SomeState, L=SomeCity, O=SomeOrganization, OU=SomeOrganizationalUni
C=US, CN=ORname_Jungo: OpenRG Products Group
C=--, ST=SomeState, L=SomeCity, O=SomeOrganization, OU=SomeOrganizationalUni
C=LT, L=Kaunas, O=Ubiquiti Networks Inc., OU=devint, CN=ubnt/emailAddress=su
C=PL, ST=Some-State, O=Mini Webservice Ltd
C=US, ST=Texas, L=Round Rock, O=Dell Inc., OU=Remote Access Group, CN=DRAC5
C=AU, ST=Some-State, O=Internet Widgits Pty Ltd, CN=TS Series NAS
C=DE, ST=NRW, L=Wuerselen, O=LANCOM Systems, OU=Engineering, CN=www.lancom s
```

x509 Subject Alt Name of Repeated Trusted TLS Certificates

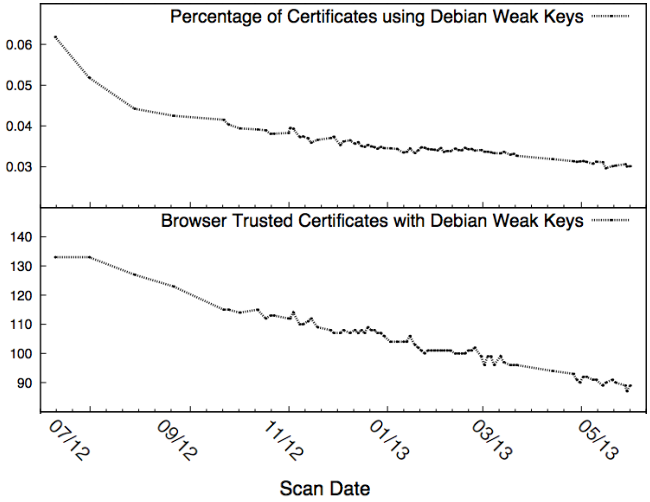
```
DNS:*.opentransfer.com, DNS:opentransfer.com
DNS:*.home.pl, DNS:home.pl
DNS:a248.e.akamai.net, DNS:*.akamaihd.net, DNS:*.akamaihd-staging.net
DNS:*.c11.hesecure.com, DNS:c11.hesecure.com
DNS:*.pair.com, DNS:pair.com
DNS:*.c12.hesecure.com, DNS:c12.hesecure.com
DNS:*.c10.hostexcellence.com, DNS:c10.hostexcellence.com
DNS:*.securesitehosting.net, DNS:securesitehosting.net
DNS:*.sslcert19.com, DNS:sslcert19.com
DNS:*.c11.ixsecure.com, DNS:c11.ixsecure.com
DNS:*.c9.hostexcellence.com, DNS:c9.hostexcellence.com
DNS:*.naviservers.net, DNS:naviservers.net
DNS:*.c10.ixwebhosting.com, DNS:c10.ixwebhosting.com
DNS:*.google.com, DNS:google.com, DNS:*.atggl.com, DNS:*.youtube.com, DNS:yo
DNS:*.hospedagem.terra.com.br
DNS:*.c8.ixwebhosting.com, DNS:c8.ixwebhosting.com
DNS:www.control.tierra.net, DNS:control.tierra.net
```

Classifying repeated SSH host keys

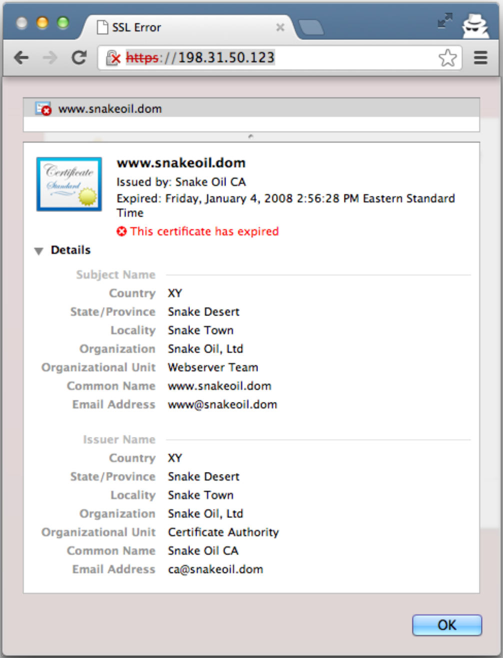


Debian OpenSSL Weak Keys

31,111 (0.34%) of RSA SSH hosts



Durumeric Wustrow Halderman 2013

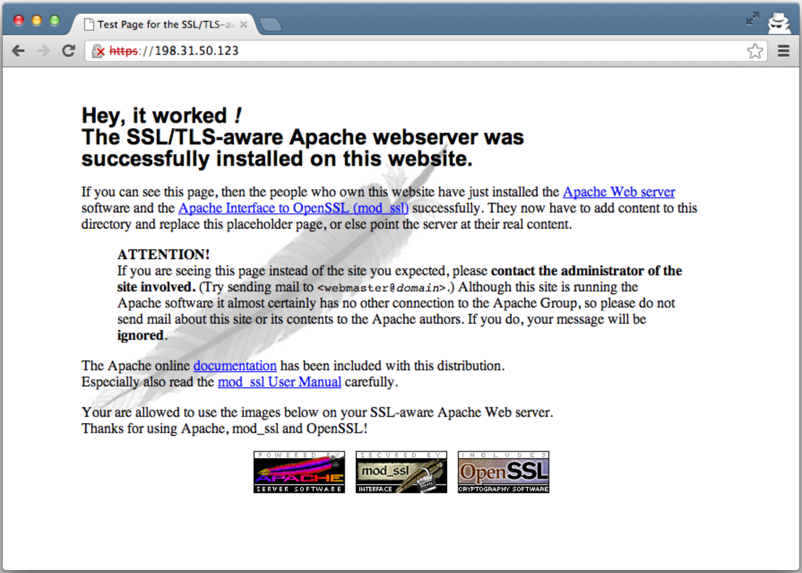


www.snakeoil.dom

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!

POWERED BY APACHE SERVER SOFTWARE
SECURED BY mod_ssl INTERFACE
INCLUDES OpenSSL CRYPTOGRAPHY SOFTWARE

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,

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

$$\gcd(N_1, N_2) = p$$

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

Do we actually expect to find key collisions in the wild?

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

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

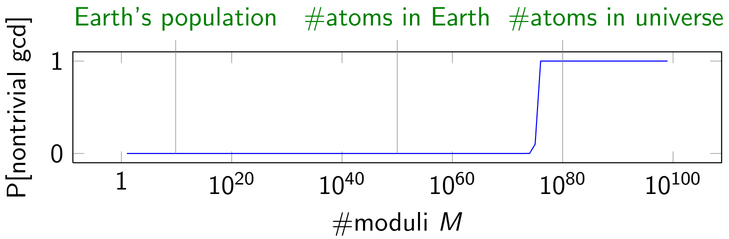
Do we actually expect to find key collisions in the wild?

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

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

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

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



Naively computing pairwise GCDs

Euclid's algorithm $\gcd(a, b)$

```
if $b = 0$:
    return $a$
else:
    return $\gcd(b, a \bmod b)$
```

a, b have n bits $\rightarrow O(n^2)$ time.

Naively computing pairwise GCDs

Euclid's algorithm $\gcd(a, b)$

```
if $b = 0$:
    return $a$
else:
    return $\gcd(b, a \bmod b)$
```

a, b have n bits $\rightarrow O(n^2)$ time.

Naively computing pairwise GCDs

Euclid's algorithm gcd(a, b)

```
if $b = 0$:
    return $a$
else:
    return gcd($b$, $a$ mod $b$)
```

a, b have n bits $\rightarrow O(n^2)$ time.

Use fast integer arithmetic for $O(n(\lg n)^2 \lg\lg n)$ time.

"Fast multiplication and its applications" Bernstein 2008

Naively computing pairwise GCDs

Euclid's algorithm $\gcd(a, b)$

```
~~if $b = 0$:
    return $a$
else:
    return $\gcd(b, a \bmod b)$~~
```

a, b have n bits $\rightarrow O(n^2)$ time.

Use fast integer arithmetic for $O(n(\lg n)^2 \lg\lg n)$ time.

"Fast multiplication and its applications" Bernstein 2008

Naive pairwise GCDs:

```
for all pairs $(N_i, N_j)$:
    if $\gcd(N_i, N_j) \neq 1$:
        add $(N_i, N_j)$ to list
```

Naively computing pairwise GCDs

Euclid's algorithm $\gcd(a, b)$

```
if $b = 0$:
    return $a$
else:
    return $\gcd(b, a \bmod b)$
```

a, b have n bits $\rightarrow O(n^2)$ time.

Use fast integer arithmetic for $O(n(\lg n)^2 \lg \lg n)$ time.

"Fast multiplication and its applications" Bernstein 2008

Naive pairwise GCDs:

```
for all pairs $(N_i, N_j)$:
    if $\gcd(N_i, N_j) \neq 1$:
        add $(N_i, N_j)$ to list
```

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

$$\approx 1100 \text{ years}$$

Naively computing pairwise GCDs

Euclid's algorithm $\gcd(a, b)$

```
if $b = 0$:
    return $a$
else:
    return $\gcd(b, a \bmod b)$
```

a, b have n bits $\rightarrow O(n^2)$ time.

Use fast integer arithmetic for $O(n(\lg n)^2 \lg\lg n)$ time.

"Fast multiplication and its applications" Bernstein 2008

Naive pairwise GCDs:

```
for all pairs $(N_i, N_j)$:
    if $\gcd(N_i, N_j) \neq 1$:
        add $(N_i, N_j)$ to list
```

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

$$\approx 1100 \text{ years}$$

Efficiently computing pairwise GCDs

An efficient algorithm due to [Bernstein 2004].

$$\begin{tikzcd}
N_1 \arrow[dr] & & N_2 \arrow[dl] & & N_3 \arrow[dr] & & N_4 \arrow[dl] & \\
& \times \arrow[drr] & & & & \times \arrow[dll] & & \text{product tree} \\
& & & N_1N_2N_3N_4 \arrow[dll] \arrow[drr] & & & & \\
& \bmod N_1^2N_2^2 \arrow[dl] \arrow[dr] & & & & \bmod N_3^2N_4^2 \arrow[dl] \arrow[dr] & & \text{remainder tree} \\
\bmod N_1^2 \arrow[d] & & \bmod N_2^2 \arrow[d] & & \bmod N_3^2 \arrow[d] & & \bmod N_4^2 \arrow[d] & \\
/N_1 \arrow[d] & & /N_2 \arrow[d] & & /N_3 \arrow[d] & & /N_4 \arrow[d] & \\
\gcd(\cdot, N_1) & & \gcd(\cdot, N_2) & & \gcd(\cdot, N_3) & & \gcd(\cdot, N_4) &
\end{tikzcd}$$

$O(mn \operatorname{polylog}(mn))$ time for m n-bit integers, a few hours for datasets. Implementation available at https://factorable.net.

What happens if we compute GCDs of some RSA moduli?

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

What happens if we compute GCDs of some RSA moduli?

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).

... 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.

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

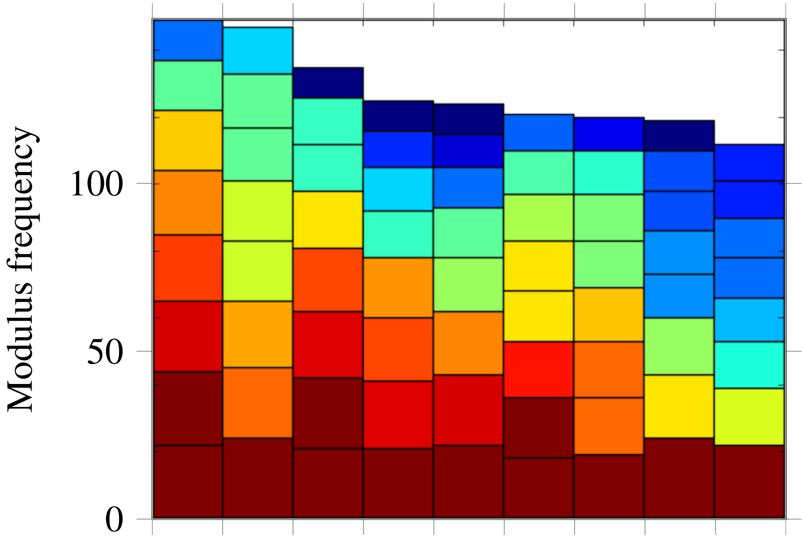
Attributing SSL and SSH vulnerabilities to implementations

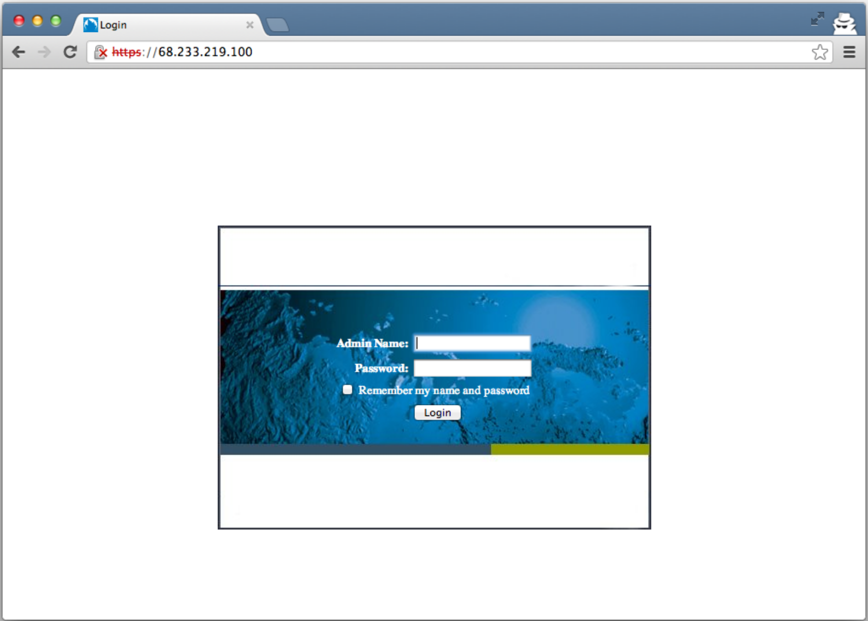
Evidence strongly suggested *widespread implementation problems*.

Clue #2: Very different behavior for different devices. Different companies, implementations, underlying software, distributions of prime factors.

Distribution of prime factors

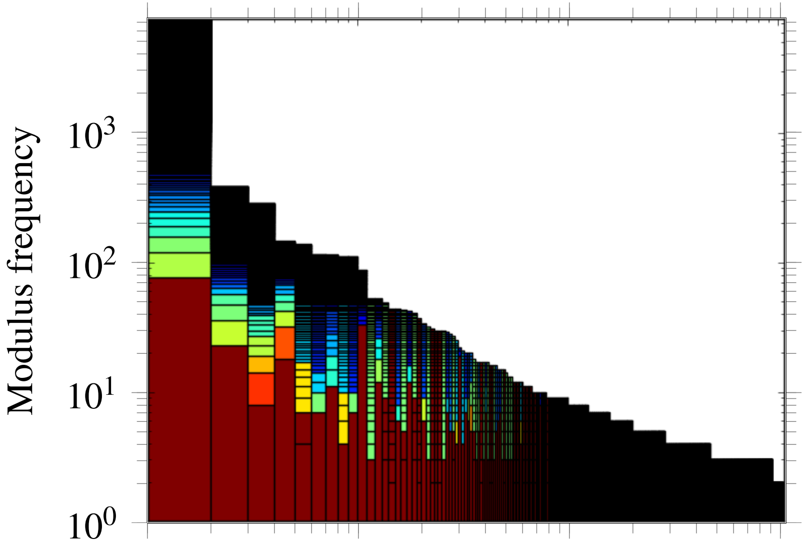
IBM Remote Supervisor Adapter II and Bladecenter Management Module



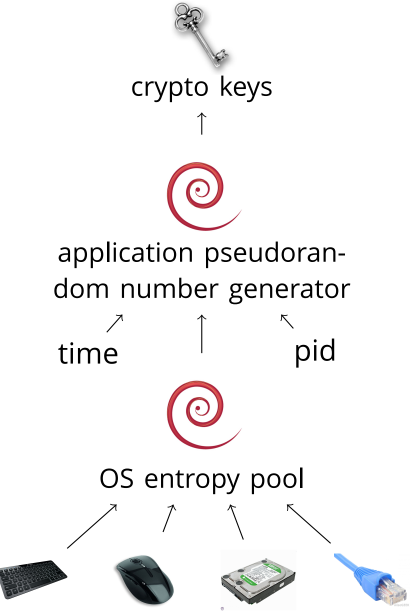

Login
https://68.233.219.100
Admin Name:
Password:
Remember my name and password
Login

Distribution of prime factors

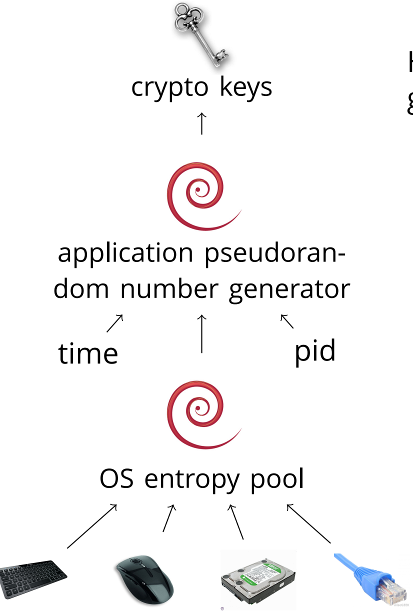
Juniper SRX branch devices



Random number generation in software

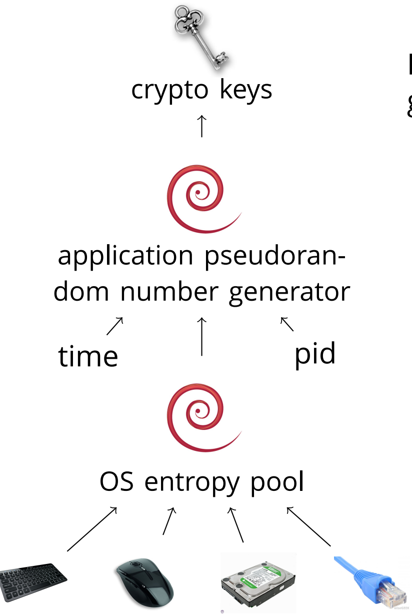


Random number generation in software



Hypothesis: Devices automatically generate crypto keys on first boot.

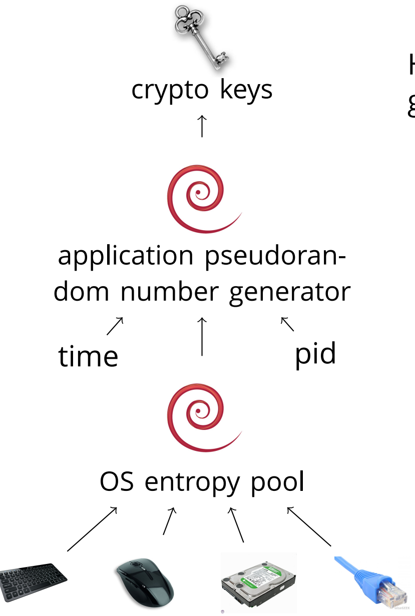
Random number generation in software



Hypothesis: Devices automatically generate crypto keys on first boot.

- Headless or embedded devices may lack these entropy sources.

Random number generation in software



Hypothesis: Devices automatically generate crypto keys on first boot.

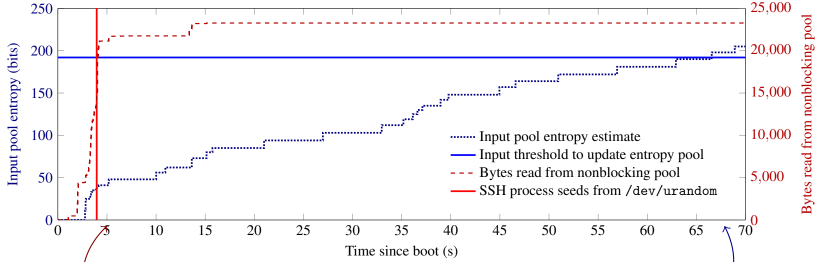
- OS random number generator may not have incorporated any entropy when queried by software.

- Headless or embedded devices may lack these entropy sources.

Linux boot-time entropy hole

Experiment: Instrument Linux kernel to track entropy estimates.

Ubuntu Server 10.04



SSH process starts

entropy pool updated

Patched since July 2012.

Generating vulnerable RSA keys in software

- Insufficiently random seeds for pseudorandom number generator $\Longrightarrow$ 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.

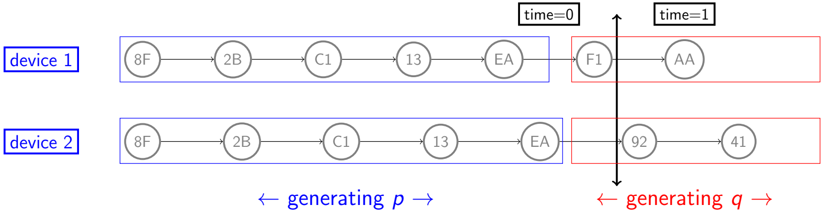
But why do we see factorable keys?

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

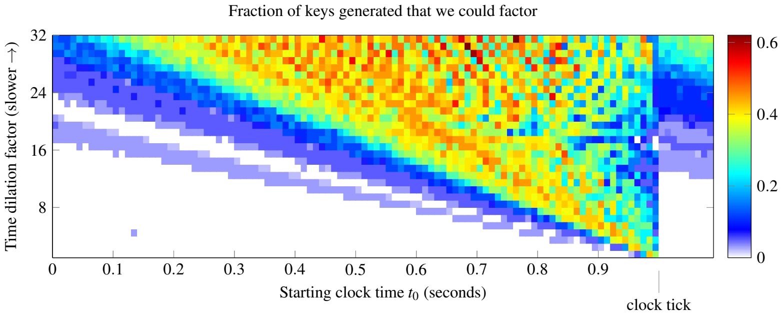
Insufficient randomness can lead to factorable keys.



Experimentally verified OpenSSL generates factorable keys in this situation.

Experimentally generating factorable keys in OpenSSL

Experiment: Generate keys in OpenSSL with time as only entropy source.



time as entropy source + asynchronous clocks → factorable keys

Unexplained oddities

Here are some prime factors of SSH keys (changed to protect the guilty):

```
d80000000000000...00000000000000000000000000000000001b3
bc0000000000000...00000000000000000000000000000000000c9
c60000000000000...00000000000000000000000000000000001af
```

Unexplained oddities

Here are some other prime factors of HTTPS keys we found:

```
c3a64ae7fc4d4d9f75cd2a49ec2d9f7...
c3a64ae7fc4d4d9f75cd2e5f2fc56c9...
c3a64ae7fc4d4d9f75cdee869c62229...

ee93536e58a60b0f56bf95faedc7ca42a9c9809a0aae2...
ee93536e58a60b0f56bf95faedc7ca42a9c9809a2cf5b...
ee93536e58a60b0f56bf95faedc7ca42a9c9809aad4a8...
ee93536e58a60b0f56bf95faedc7ca42a9c9809abb02d...
ee93536e58a60b0f56bf95faedc7ca42a9c9809acef6f...
```

PGP

Most likely a bad third-party plugin.

Textbook Diffie-Hellman

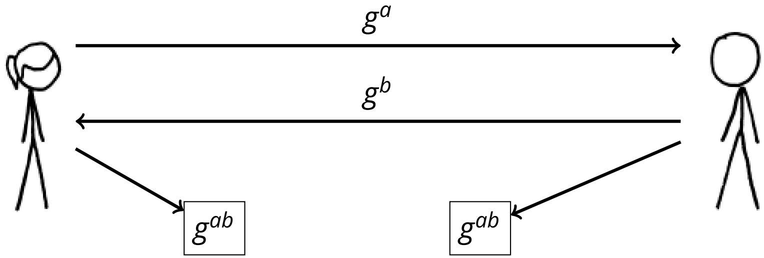
[Diffie Hellman 1976]

Public Parameters

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

g group generator

Key Exchange



Is a repeated g^a a vulnerability?

- Yes, if unrelated parties know discrete log/private key a.
- Yes, if repeated values signal entropy issues.

Is a repeated g^a a vulnerability?

- Yes, if unrelated parties know discrete log/private key a.
- Yes, if repeated values signal entropy issues.

ECDH TLS scans:

5.4*M* key exchanges 5.2*M* unique values.

Is a repeated g^a a vulnerability?

- Yes, if unrelated parties know discrete log/private key a.
- Yes, if repeated values signal entropy issues.

ECDH TLS scans:
5.4*M* key exchanges 5.2*M* unique values.

Possible explanation: OpenSSL using ephemeral-static ECDH. (Keys ephemeral per application instance and not per handshake.)

121,000 values presented by $>$ 1 IP address, most common on 2,000 hosts.

Mostly shared hosting. One Netasq device always presents same key for ECDHE.

The DSA Algorithm

DSA Public Key

p	prime
q	prime, divides $(p-1)$
g	generator of subgroup of order q mod p
y	$= g^x \bmod p$

Private Key

x	private key

Verify

$$u_1 = H(m)s^{-1} \bmod q$$
$$u_2 = rs^{-1} \bmod q$$
$$r \overset{?}{=} g^{u_1}y^{u_2} \bmod p \bmod q$$

Sign

Generate random k.

$$r = g^k \bmod p \bmod q$$
$$s = k^{-1}(H(m) + xr) \bmod q$$

ECDSA

ECDSA Public Key

G generator $\in E(\mathbb{F}_p)$

$Q = dG$

Private Key

d private key

Sign

Generate random k.

$(x, y) = kG$ $r = x \bmod n$

$s = k^{-1}(H(m) + dr) \bmod n$

What could go wrong: Repeated keys

DSA public keys

Public key

p	prime
q	prime, divides $(p-1)$
g	generator of subgroup of order q mod p
y	$= g^x \bmod p$

- Two hosts have same public key $\rightarrow$ both know private key of the other.

What could go wrong: Weak DSA signature nonce

Public Key

p, q, g domain parameters

$y = g^x \bmod p$

Private Key

x private key

Signature: (r, s_1)

$r = g^k \bmod p \bmod q$

$s_1 = k^{-1}(H(m_1) + xr) \bmod q$

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

What could go wrong: Weak DSA signature nonce

Public Key

p, q, g domain parameters

$y = g^x \bmod p$

Private Key

x private key

Signature: (r, s_1)

$r = g^k \bmod p \bmod q$

$s_1 = k^{-1}(H(m_1) + xr) \bmod q$

Signature: (r, s_2)

$r = g^k \bmod p \bmod q$

$s_2 = k^{-1}(H(m_2) + xr) \bmod q$

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

$$s_1 - s_2 = k^{-1}(H(m_1) - H(m_2)) \bmod q$$

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

What happens if we look for repeated DSA nonces?

Compute private keys for

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

Generating weak DSA signatures

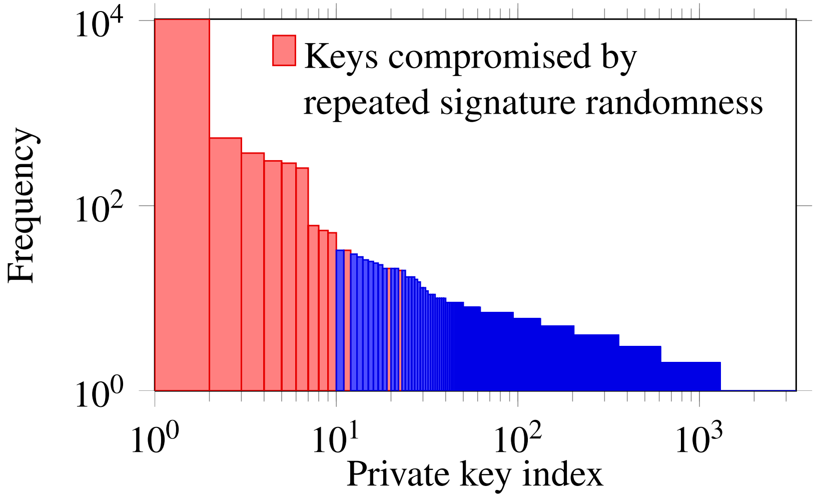
Step 1: Low-entropy DSA key generation

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 $\rightarrow$ colliding nonces.

Compromised DSA keys from Gigaset DSL routers

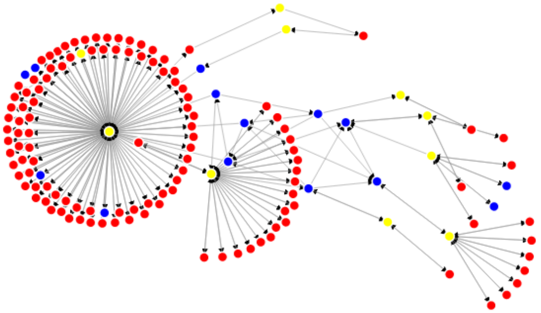


Bitcoin

Several explanations so far:

- Multiple Android RNG vulnerabilities
- 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.

Vendor responses

"When running the testing, would you be able to provide the software on the .. and the firmware on the … along with model numbers on the …."

"Attached is a document on the security the … uses." (It was empty.)

"Would you be able to provide the login credentials for the 3 test IP Addresses you provided. I would like to login to the device to gather the software and firmware installed."

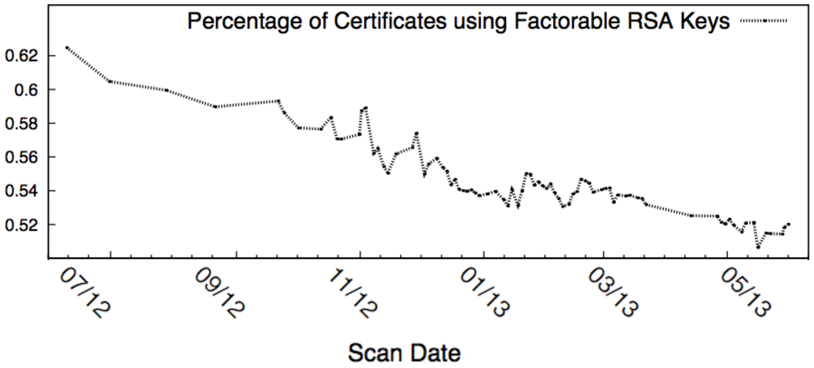
"Hi. What is your billing address, so that I can fwd your email to the appropriate Account Executive."

"some IT auditor somewhere is handed your paper, alarm bells sound in his or her head, and things start to get unnecessarily emergent, network admins start calling us, CSIRTs start engaging us to figure out what's going one, etc., etc."

Disclosure to end-users

- Attempted to contact end-users with signed certificates sharing keys with default certificates.
- Certificates belonged to Fortune 500 companies, insurance providers, law firms, a major public transit authority, and the US Navy.

Factorable TLS keys over time



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 *Financial Cryptography 2014*

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.*