Cryptography worst practices

Daniel J. Bernstein
University of Illinois at Chicago

http://cr.yp.to/talks.html
#2012.03.08-1

realized i really
for some fake r

indeed.

You are currently offline. To chat with your friends, go online.

Alice and Bob are communicating.
Eve is eavesdropping.

What cryptography
promises to Alice and Bob:

Confidentiality despite espionage.
Maybe Eve wants to acquire data.

Integrity despite corruption.
Maybe Eve wants to change data.

Availability despite sabotage.
Maybe Eve wants to destroy data.

This talk: Some examples of what cryptography actually delivers in the real world.

- Failures of confidentiality.
- Failures of integrity.
- Failures of availability.

This talk: Some examples of what cryptography actually delivers in the real world.

- Failures of confidentiality.
- Failures of integrity.
- Failures of availability.

Often adding cryptography makes attacks *easier* than they were before.

This talk: Some examples
of what cryptography actually
delivers in the real world.

- Failures of confidentiality.
- Failures of integrity.
- Failures of availability.

Often adding cryptography
makes attacks *easier*
than they were before.

Sometimes cryptography
(deliberately?) gives users
a false sense of security.
Users then behave carelessly,
making attacks even easier.

Cryptography vs. blind attacks

"The attacker isn't sniffing our network packets so we're secure."

e.g. Browser has a cookie authorizing access to an account.
Cookie = account credentials, authenticated by server to itself.

Cryptography vs. blind attacks

"The attacker isn't sniffing our network packets so we're secure."

e.g. Browser has a cookie
authorizing access to an account.
Cookie = account credentials,
authenticated by server to itself.

Browser sends this cookie
to server through HTTP.
Might actually be secure
if Eve isn't sniffing the network;
but Eve *is* sniffing the network!
2010 example: Firesheep
stealing Facebook credentials.

Cryptography vs. passive attacks

"The attacker isn't forging network packets so we're secure."

Examples of this "security":

- TCP checking IP address.
- DNS checking IP address.
- New marketing stunt: Tcpcrypt.

Cryptography vs. passive attacks

"The attacker isn't forging network packets so we're secure."

Examples of this "security":

- TCP checking IP address.
- DNS checking IP address.
- New marketing stunt: Tcpcrypt.

"Compare this tcpdump output, which appears encrypted . . . with the cleartext packets you would see without tcpcryptd running. . . . Active attacks are much harder as they require listening and modifying network traffic."

Reality: Eve is modifying network traffic, often at massive scale.

2011.10 Wall Street Journal:
“A U.S. company that
makes Internet-blocking gear
acknowledges that Syria has been
using at least 13 of its devices
to censor Web activity there.”

Reality: Eve is modifying network traffic, often at massive scale.

2011.10 Wall Street Journal:
"A U.S. company that
makes Internet-blocking gear
acknowledges that Syria has been
using at least 13 of its devices
to censor Web activity there."

2012.02: Trustwave (one of the SSL CAs trusted by your browser) admits selling a transparent HTTPS interception box to a private company.

Integrity über alles

"We detect corrupt data
so we're secure."

What about availability?
What about confidentiality?

Many "security solutions"
ignore these issues.
Sometimes adding crypto
allows *easier* attacks against
availability and confidentiality.

Interesting example: DNSSEC.

A brief history of DNSSEC server deployment:

1993.11: DNSSEC design begins.

A brief history of
DNSSEC server deployment:

1993.11: DNSSEC design begins.

2008.07: Kaminsky announces apocalypse, saves the world.

A brief history of
DNSSEC server deployment:

1993.11: DNSSEC design begins.

2008.07: Kaminsky announces apocalypse, saves the world.
⇒ New focus on DNSSEC.

A brief history of
DNSSEC server deployment:

1993.11: DNSSEC design begins.

2008.07: Kaminsky announces apocalypse, saves the world.
⇒ New focus on DNSSEC.

2009.08.09 DNSSEC servers:
941 IP addresses worldwide.

A brief history of
DNSSEC server deployment:

1993.11: DNSSEC design begins.

2008.07: Kaminsky announces apocalypse, saves the world.
⇒ New focus on DNSSEC.

2009.08.09 DNSSEC servers:
941 IP addresses worldwide.

2010.12.24 DNSSEC servers:
2536 IP addresses worldwide.

A brief history of
DNSSEC server deployment:

1993.11: DNSSEC design begins.

2008.07: Kaminsky announces apocalypse, saves the world.
⇒ New focus on DNSSEC.

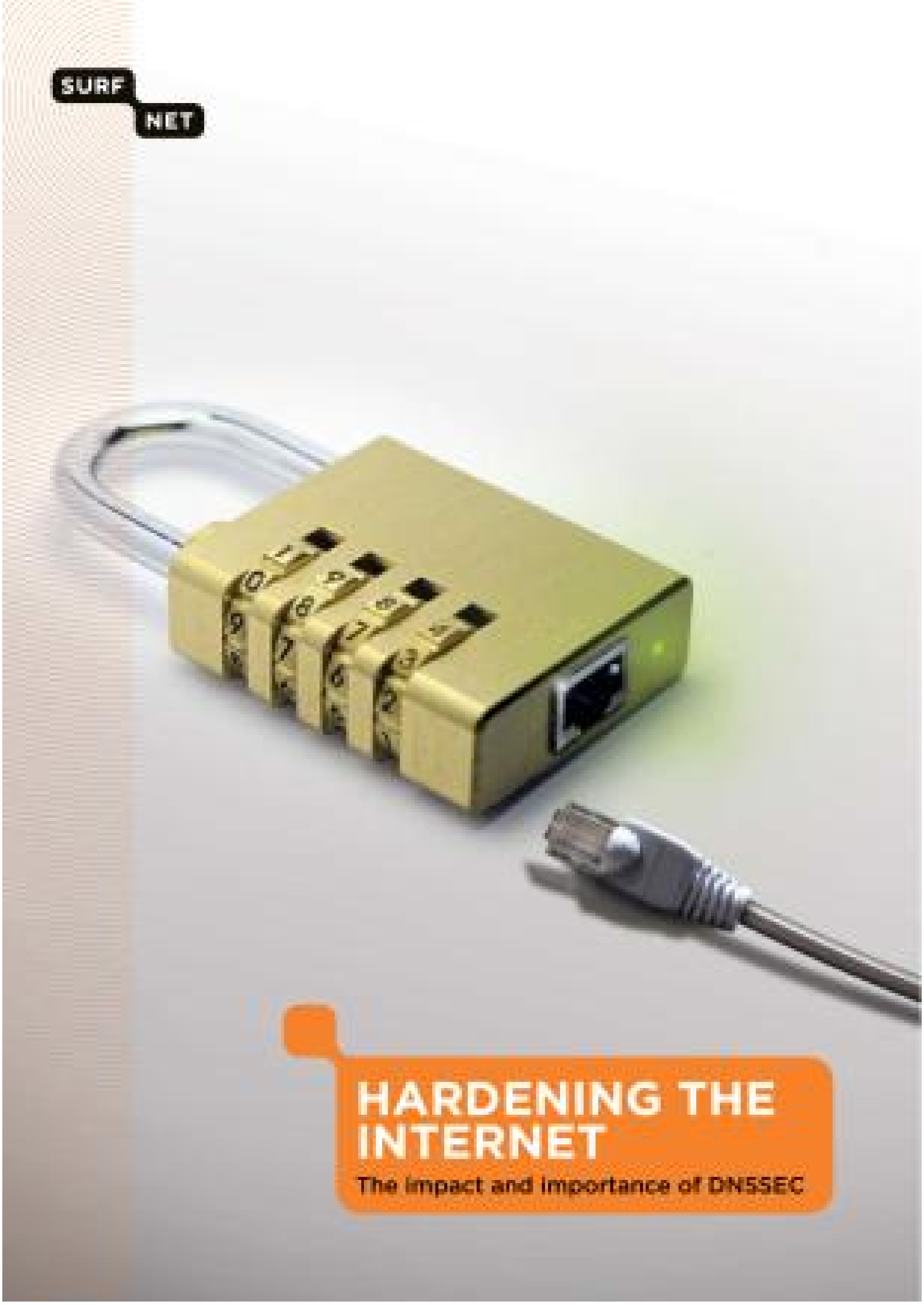
2009.08.09 DNSSEC servers:
941 IP addresses worldwide.

2010.12.24 DNSSEC servers:
2536 IP addresses worldwide.

2011.12.14 DNSSEC servers:
3393 IP addresses worldwide.

What is DNSSEC?

What is DNSSEC?
Is it a lock for the Internet?



What is DNSSEC?

Is it a lock for the Internet?

Or is it more like this?

What is DNSSEC?
Is it a lock for the Internet?
Or is it more like this?

Let's see what DNSSEC can do
as an amplification tool for
denial-of-service attacks.

Download DNSSEC zone list:

```
wget -m -k -I / \
  secspider.cs.ucla.edu
cd secspider.cs.ucla.edu
awk '
  /GREEN.*GREEN.*GREEN.*Yes/ {
    split($0,x,/<TD>/)
    sub(/<\/TD>/,"",x[5])
    print x[5]
  }
' ./*--zone.html \
| sort -u | wc -l
```

Make list of DNSSEC names:

```
( cd secspider.cs.ucla.edu
  echo ./*--zone.html \
  | xargs awk '
    /^Zone <STRONG>/ { z = $2
      sub(/<STRONG>/,"",z)
      sub(/<\/STRONG>/,"",z)
    }
    /GREEN.*GREEN.*GREEN.*Yes/ {
      split($0,x,/<TD>/)
      sub(/<\/TD>/,"",x[5])
      print x[5],z,rand()
    }'
) | sort -k3n \
| awk '{print $1,$2}' > SERVERS
```

For each domain: Try query, estimate DNSSEC amplification.

```
while read ip z
do
  dig +dnssec +ignore +tries=1 \
  +time=1 any "$z" "@$ip" | \
  awk -v "z=$z" -v "ip=$ip" '{
    if ($1 != ";;") next
    if ($2 != "MSG") next
    if ($3 != "SIZE") next
    if ($4 != "rcvd:") next
    est = (22+$5)/(40+length(z))
    print est,ip,z
  }'
done < SERVERS > AMP
```

For each DNSSEC server,
find domain estimated to have
maximum DNSSEC amplification:

```
sort -nr AMP | awk '{
  if (seen[$2]) next
  if ($1 < 30) next
  print $1,$2,$3
  seen[$2] = 1
}' > MAXAMP
head -1 MAXAMP
wc -l MAXAMP
```

Output (last time I tried it):

```
95.6279 156.154.102.26 fi.
2326 MAXAMP
```

Can that really be true?
$>$ 3000 DNSSEC servers
around the Internet, each
providing $> 30\times$ amplification
of incoming UDP packets?

Can that really be true?
> 3000 DNSSEC servers
around the Internet, each
providing $> 30\times$ amplification
of incoming UDP packets?

Let's verify this.

Choose quiet test machines
on two different networks
(without egress filters).

e.g. Sender: 1.2.3.4.
Receiver: 5.6.7.8.

Run network-traffic monitors on 1.2.3.4 and 5.6.7.8.

On 1.2.3.4, set response address to 5.6.7.8, and send 1 query/second:

```
ifconfig eth0:1 \
  5.6.7.8 \
  netmask 255.255.255.255
while read est ip z
do
  dig -b 5.6.7.8 \
  +dnssec +ignore +tries=1 \
  +time=1 any "$z" "@$ip"
done < MAXAMP >/dev/null 2>&1
```

I sustained 51× amplification
of actual network traffic
in a US-to-Europe experiment
on typical university computers
at the end of 2010.

I sustained 51× amplification
of actual network traffic
in a US-to-Europe experiment
on typical university computers
at the end of 2010.

Attacker sending 10Mbps
can trigger 500Mbps flood
from the DNSSEC drone pool,
taking down typical site.

I sustained 51× amplification
of actual network traffic
in a US-to-Europe experiment
on typical university computers
at the end of 2010.

Attacker sending 10Mbps
can trigger 500Mbps flood
from the DNSSEC drone pool,
taking down typical site.

Attacker sending 200Mbps
can trigger 10Gbps flood,
taking down very large site.

I sustained 51× amplification
of actual network traffic
in a US-to-Europe experiment
on typical university computers
at the end of 2010.

Attacker sending 10Mbps
can trigger 500Mbps flood
from the DNSSEC drone pool,
taking down typical site.

Attacker sending 200Mbps
can trigger 10Gbps flood,
taking down very large site.

Want even more: 100Gbps?
Tell people to install DNSSEC!

RFC 4033 says
“DNSSEC provides no protection against denial of service attacks.”

RFC 4033 says
“DNSSEC provides no protection against denial of service attacks.”

RFC 4033 doesn’t say
“DNSSEC is a pool of remote-controlled attack drones, the worst DDoS amplifier on the Internet.”

RFC 4033 says
“DNSSEC provides no protection against denial of service attacks.”

RFC 4033 doesn’t say
“DNSSEC is a pool of remote-controlled attack drones, the worst DDoS amplifier on the Internet.”

Not covered in this talk:
other types of DoS attacks.
e.g. DNSSEC advertising says zero server-CPU-time cost.
How much server CPU time can attackers actually consume?

But wait, there’s more!

RFC 4033 says “DNSSEC does not provide confidentiality.” DNSSEC doesn’t encrypt queries or responses.

But wait, there's more!

RFC 4033 says "DNSSEC does not provide confidentiality." DNSSEC doesn't encrypt queries or responses.

RFC 4033 doesn't say "DNSSEC damages confidentiality of data in DNS databases."

But wait, there's more!

RFC 4033 says "DNSSEC does not provide confidentiality." DNSSEC doesn't encrypt queries or responses.

RFC 4033 doesn't say "DNSSEC damages confidentiality of data in DNS databases."

DNSSEC has leaked a huge number of private DNS names such as `acadmedpa.org.br`.

Why does DNSSEC leak data? An interesting story!

Core DNSSEC data flow:

`kuleuven.be` DNS database
includes precomputed signatures
from Leuven administrator.

(Hypothetical example.
In the real world,
Leuven isn't using DNSSEC.)

What about *dynamic* DNS data?

Core DNSSEC data flow:

`kuleuven.be` DNS database
includes precomputed signatures
from Leuven administrator.

(Hypothetical example.
In the real world,
Leuven isn't using DNSSEC.)

What about *dynamic* DNS data?
DNSSEC purists say "Answers
should always be static."

What about *old* DNS data?
Are the signatures still valid?

Can an attacker replay
obsolete signed data?

e.g. You move IP addresses.
Attacker grabs old address,
replays old signature.

What about *old* DNS data?
Are the signatures still valid?

Can an attacker replay
obsolete signed data?

e.g. You move IP addresses.
Attacker grabs old address,
replays old signature.

If clocks are synchronized
then signatures can
include expiration times.
But frequent re-signing
is an administrative disaster.

Some DNSSEC suicide examples:

2010.09.02: `.us` killed itself.

2010.10.07: `.be` killed itself.

Some DNSSEC suicide examples:

2010.09.02: `.us` killed itself.

2010.10.07: `.be` killed itself.

2012.02.23: ISC administrators killed some `isc.org` names.

Some DNSSEC suicide examples:

2010.09.02: `.us` killed itself.

2010.10.07: `.be` killed itself.

2012.02.23: ISC administrators killed some `isc.org` names.

2012.02.28: “`Last night I was unable to check the weather forecast, because the fine folks at NOAA.gov / weather.gov broke their DNSSEC.`”

Some DNSSEC suicide examples:

2010.09.02: `.us` killed itself.

2010.10.07: `.be` killed itself.

2012.02.23: ISC administrators killed some `isc.org` names.

2012.02.28: "`Last night I was unable to check the weather forecast, because the fine folks at NOAA.gov / weather.gov broke their DNSSEC.`"

2012.02.28, ISC's Evan Hunt: "`dnssec-accept-expired yes`"

What about *nonexistent* data?

What about *nonexistent* data?

Does Leuven administrator precompute signatures on "`aaaaa.kuleuven.be does not exist`", "`aaaab.kuleuven.be does not exist`", etc.?

What about *nonexistent* data?

Does Leuven administrator precompute signatures on "`aaaaa.kuleuven.be does not exist`", "`aaaab.kuleuven.be does not exist`", etc.?

Crazy! Obvious approach:
"We sign each record that exists, and don't sign anything else."

What about *nonexistent* data?

Does Leuven administrator precompute signatures on "`aaaaa.kuleuven.be does not exist`", "`aaaab.kuleuven.be does not exist`", etc.?

Crazy! Obvious approach:
"We sign each record that exists, and don't sign anything else."

User asks for nonexistent name.
Receives *unsigned* answer saying the name doesn't exist.
Has no choice but to trust it.

User asks for `www.google.com`.
Receives unsigned answer,
a packet forged by Eve,
saying the name doesn't exist.
Has no choice but to trust it.

Clearly a violation of availability.
Sometimes a violation of integrity.
This is not a good approach.

User asks for `www.google.com`.
Receives unsigned answer,
a packet forged by Eve,
saying the name doesn't exist.
Has no choice but to trust it.

Clearly a violation of availability.
Sometimes a violation of integrity.
This is not a good approach.

Alternative: DNSSEC's "NSEC".
e.g. `nonex.clegg.com` query
returns "`There are no names between nick.clegg.com and start.clegg.com`" + signature.
(This is a real example.)

Attacker learns
all n names in `clegg.com`
(with signatures guaranteeing
that there are no more)
using n DNS queries.

This is not a good approach.

DNSSEC purists disagree:
"It is part of the design
philosophy of the DNS
that the data in it is public."
But this notion is so extreme
that it became a PR problem.

New DNSSEC approach:

1. “NSEC3” technology:
Use a “one-way hash function”
such as (iterated salted) SHA-1.
Reveal *hashes* of names
instead of revealing names.
“`There are no names with hashes between ... and ...`”

New DNSSEC approach:

1. "NSEC3" technology:
Use a "one-way hash function"
such as (iterated salted) SHA-1.
Reveal *hashes* of names
instead of revealing names.
"`There are no names with hashes between ... and ...`"

2. Marketing:
Pretend that NSEC3 is
less damaging than NSEC.

ISC: "NSEC3 does not allow enumeration of the zone."

Reality: Attacker grabs the hashes by abusing DNSSEC’s NSEC3; computes the same hash function for many different name guesses; quickly discovers almost all names (and knows # missing names).

Reality: Attacker grabs the hashes by abusing DNSSEC's NSEC3; computes the same hash function for many different name guesses; quickly discovers almost all names (and knows # missing names).

DNSSEC purists: "You could have sent all the same guesses as queries to the server."

Reality: Attacker grabs the hashes by abusing DNSSEC's NSEC3; computes the same hash function for many different name guesses; quickly discovers almost all names (and knows # missing names).

DNSSEC purists: "You could have sent all the same guesses as queries to the server."

4Mbps flood of queries is under 500 million noisy guesses/day. NSEC3 allows typical attackers 1000000 million to 1000000000 million silent guesses/day.

Misdirected cryptography

"We're cryptographically protecting X so we're secure."

Misdirected cryptography

"We're cryptographically protecting X so we're secure."

Is X the complete communication from Alice to Bob, all the way from Alice to Bob?

Misdirected cryptography

"We're cryptographically protecting X so we're secure."

Is X the complete communication from Alice to Bob, all the way from Alice to Bob?

Often X doesn't reach Bob.

Misdirected cryptography

"We're cryptographically protecting X so we're secure."

Is X the complete communication from Alice to Bob, all the way from Alice to Bob?

Often X doesn't reach Bob.
Example: Bob views Alice's web page on his Android phone.
Phone asked hotel DNS cache for web server's address.
Eve forged the DNS response!
DNS cache checked DNSSEC but the phone didn't.

Often X isn't Alice's data.

Often X isn't Alice's data.

".ORG becomes the first open TLD to sign their zone with DNSSEC … Today we reached a significant milestone in our effort to bolster online security for the .ORG community. We are the first open generic Top-Level Domain to successfully sign our zone with Domain Name Security Extensions (DNSSEC). To date, the .ORG zone is the largest domain registry to implement this needed security measure."

What did .org actually sign?

2012.03.07 test: Ask .org about wikipedia.org.

The response has a *signed* statement "There might be names with hashes between h9rsfb7fpf2l8hg35cmpc765tdk23rp6, hheprfsv14o44rv9pgcndkt4thnraomv. We haven't signed any of them. Sincerely, .org"

Plus an *unsigned* statement "The wikipedia.org name server is 208.80.152.130."

Often X is horribly incomplete.

Often X is horribly incomplete.

Example: X is a server address, with a DNSSEC signature.

What Alice is sending to Bob are web pages, email, etc. Those aren't the same as X!

Often X is horribly incomplete.

Example: X is a server address, with a DNSSEC signature.

What Alice is sending to Bob are web pages, email, etc. Those aren't the same as X!

Alice can use HTTPS to protect her web pages . . . but then what attack is stopped by DNSSEC?

DNSSEC purists criticize HTTPS:
“Alice can’t trust her servers.”

DNSSEC signers are offline
(preferably in guarded rooms).
DNSSEC precomputes signatures.
DNSSEC doesn’t trust servers.

DNSSEC purists criticize HTTPS:
“Alice can’t trust her servers.”

DNSSEC signers are offline
(preferably in guarded rooms).
DNSSEC precomputes signatures.
DNSSEC doesn’t trust servers.

. . . but X is still wrong!
Alice’s servers still control
all of Alice’s web pages,
unless Alice uses PGP.

With or without PGP, what
attack is stopped by DNSSEC?

Variable-time cryptography

"Our cryptographic computations expose nothing but incomprehensible ciphertext to the attacker, so we're secure."

Reality: The attacker often sees ciphertexts *and* how long Alice took to compute the ciphertexts *and* how long Bob took to compute the plaintexts.

Timing variability often makes the cryptography easier to attack, sometimes trivial.

Ancient example, shift cipher:
Shift each letter by k,
where k is Alice's secret key.

e.g. Caesar's key: 3.
Plaintext HELLO.
Ciphertext KHOOR.

Ancient example, shift cipher:
Shift each letter by k,
where k is Alice's secret key.

e.g. Caesar's key: 3.
Plaintext HELLO.
Ciphertext KHOOR.

e.g. My key: 1.
Plaintext HELLO.
Ciphertext IFMMP.
See how fast that was?

Ancient example, shift cipher:
Shift each letter by k,
where k is Alice's secret key.

e.g. Caesar's key: 3.
Plaintext HELLO.
Ciphertext KHOOR.

e.g. My key: 1.
Plaintext HELLO.
Ciphertext IFMMP.
See how fast that was?

e.g. Your key: 13.
Plaintext HELLO.
Exercise: Find ciphertext.

This is a very bad cipher:
easily figure out key
from some ciphertext.

But it's even worse
against timing attacks:
instantly figure out key,
even for 1-character ciphertext.

This is a very bad cipher:
easily figure out key
from some ciphertext.

But it’s even worse
against timing attacks:
instantly figure out key,
even for 1-character ciphertext.

Our computers are using
much stronger cryptography,
but most implementations
leak secret keys via timing.

1970s: TENEX operating system
compares user-supplied string
against secret password
one character at a time,
stopping at first difference.

`AAAAAA` vs. `SECRET`: stop at 1.
`SAAAAA` vs. `SECRET`: stop at 2.
`SEAAAA` vs. `SECRET`: stop at 3.

Attackers watch comparison time,
deduce position of difference.
A few hundred tries
reveal secret password.

Objection: “Timings are noisy!”

Objection: “Timings are noisy!”

Answer #1: Even if noise
stops simplest attack,
does it stop *all* attacks?

Objection: “Timings are noisy!”

Answer #1: Even if noise stops simplest attack, does it stop *all* attacks?

Answer #2: Eliminate noise using statistics of many timings.

Objection: “Timings are noisy!”

Answer #1: Even if noise
stops simplest attack,
does it stop *all* attacks?

Answer #2: Eliminate noise
using statistics of many timings.

Answer #3, what the
1970s attackers actually did:
Increase timing signal
by crossing page boundary,
inducing page faults.

1996 Kocher
extracted RSA keys
from local RSAREF timings:
small numbers were
processed more quickly.

2003 Boneh–Brumley
extracted RSA keys
from an OpenSSL web server.

2011 Brumley–Tuveri:
minutes to steal another
machine's OpenSSL ECDSA key.

Most IPsec software uses
`memcmp` to check authenticators.
Exercise: Forge IPsec packets.

Obvious source of problem:

`if(...)` leaks . . . into timing.

Obvious source of problem:
`if(...)` leaks ... into timing.

Almost as obvious:
`x[...]` leaks ... into timing.

Obvious source of problem:
`if(...)` leaks ... into timing.

Almost as obvious:
`x[...]` leaks ... into timing.

Usually these timings
are correlated with
total encryption time.

Obvious source of problem:
`if(...)` leaks ... into timing.

Almost as obvious:
`x[...]` leaks ... into timing.

Usually these timings are correlated with total encryption time.

Also have fast effect (via cache state, branch predictor, etc.) on timing of other threads and processes on same machine— even in other virtual machines!

Fast AES implementations
for most types of CPUs
rely critically on `[...]`.

2005 Bernstein recovered
AES key from a network server
using OpenSSL's AES software.

2005 Osvik–Shamir–Tromer
in 65ms stole Linux AES key
used for hard-disk encryption.
Attack process on same CPU,
using hyperthreading.

Many clumsy "countermeasures";
many followup attacks.

Hardware side channels (audio, video, radio, etc.) allow many more attacks for attackers close by, sometimes farther away.

Compare 2007 Biham–Dunkelman–Indesteege–Keller–Preneel (a feasible computation recovers one user's Keeloq key from an hour of ciphertext) to 2008 Eisenbarth–Kasper–Moradi–Paar–Salmasizadeh–Shalmani (power consumption revealed master Keeloq secret; recover any user's Keeloq key in seconds).

Decrypting unauthenticated data

“We authenticate our messages before we encrypt them, and of course we check for forgeries after decryption, so we’re secure.”

Decrypting unauthenticated data

"We authenticate our messages before we encrypt them, and of course we check for forgeries after decryption, so we're secure."

Theoretically it's possible to get this right, but it's terribly fragile.

1998 Bleichenbacher: Attacker
steals SSL RSA ciphertext
by observing server responses
to $\approx 10^6$ variants of ciphertext.

SSL inverts RSA, then checks
for correct “PKCS padding”
(which many forgeries have).
Subsequent processing applies
more serious integrity checks.

Server responses reveal
pattern of PKCS forgeries;
pattern reveals plaintext.

Typical defense strategy:
try to hide differences
between padding checks and
subsequent integrity checks.
But nobody gets this right.

More recent attacks
exploiting server responses:

2009 Albrecht–Paterson–Watson recovered some SSH plaintext.

2011 Paterson–Ristenpart–Shrimpton distinguished 48-byte SSL encryptions of `YES` and `NO`.

2012 Alfardan–Paterson recovered DTLS plaintext from OpenSSL and GnuTLS.

Let's peek at the 2011 attack.

Alice authenticates `NO` as `NO` $+$ 10-byte authenticator. (10: depends on SSL options.)

Alice authenticates `NO`
as `NO` + 10-byte authenticator.
(10: depends on SSL options.)

Then hides length by padding
to 16 or 32 or 48 or . . . bytes
(choice made by sender).
Padding 12 bytes to 32:
append bytes `19 19 19 . . . .`

Alice authenticates `NO`
as `NO` + 10-byte authenticator.
(10: depends on SSL options.)

Then hides length by padding
to 16 or 32 or 48 or . . . bytes
(choice made by sender).
Padding 12 bytes to 32:
append bytes `19 19 19 ....`

Then puts 16 random bytes in
front, encrypts in "CBC mode".
Encryption of 48 bytes
R, P_1, P_2 is R, C_1, C_2 where
$C_1 = \text{AES}(R \oplus P_1)$,
$C_2 = \text{AES}(C_1 \oplus P_2)$.

Bob receives R, C_1, C_2;
computes $P_1 = R \oplus \text{AES}^{-1}(C_1)$;
computes $P_2 = C_1 \oplus \text{AES}^{-1}(C_2)$;
checks padding and authenticator.

Bob receives R, C_1, C_2;
computes $P_1 = R \oplus \text{AES}^{-1}(C_1)$;
computes $P_2 = C_1 \oplus \text{AES}^{-1}(C_2)$;
checks padding and authenticator.

What if Eve sends R', C_1 where
$R' = R \oplus 0 \ldots 0\ 16\ 16\ 16\ 16$?

Bob receives R, C_1, C_2;
computes $P_1 = R \oplus \text{AES}^{-1}(C_1)$;
computes $P_2 = C_1 \oplus \text{AES}^{-1}(C_2)$;
checks padding and authenticator.

What if Eve sends R', C_1 where
$R' = R \oplus 0 \ldots 0\,16\,16\,16\,16$?

Bob computes
$P_1' = P_1 \oplus 0 \ldots 0\,16\,16\,16\,16$.
Padding is still valid,
as is the authenticator.

Bob receives R, C_1, C_2;
computes $P_1 = R \oplus \text{AES}^{-1}(C_1)$;
computes $P_2 = C_1 \oplus \text{AES}^{-1}(C_2)$;
checks padding and authenticator.

What if Eve sends R', C_1 where
$R' = R \oplus$ `0 ... 0 16 16 16 16`?

Bob computes
$P_1' = P_1 \oplus$ `0 ... 0 16 16 16 16`.
Padding is still valid,
as is the authenticator.

If plaintext had been `YES` then
Bob would have rejected R', C_1
for having a bad authenticator.

Bad crypto primitives

"We're using a cryptographic standard so we're secure."

Examples of this "security":

- DES.
- 512-bit RSA.
- 768-bit RSA.
- MD5-based certificates.

Bad crypto primitives

"We're using a cryptographic standard so we're secure."

Examples of this "security":
- DES.
- 512-bit RSA.
- 768-bit RSA.
- MD5-based certificates.

1996 Dobbertin–Bosselaers–Preneel: "It is anticipated that these techniques can be used to produce collisions for MD5". Standardization committees didn't pay attention; why would they?

Speed over security

Crypto performance problems often lead users to reduce cryptographic security levels or give up on cryptography.

Example 1:
Google SSL uses RSA-1024.

Security note:
Analyses in 2003 concluded that RSA-1024 was breakable; e.g., 2003 Shamir–Tromer estimated 1 year, $\approx 10^7$ USD.
RSA Labs and NIST response:
Move to RSA-2048 by 2010.

“1024-bit factorization is expensive. Our data isn’t worth that much, so we’re secure.”

"1024-bit factorization is expensive. Our data isn't worth that much, so we're secure."

Is it really so expensive?

Hardware keeps getting better.

“1024-bit factorization is expensive. Our data isn’t worth that much, so we’re secure.”

Is it really so expensive?
Hardware keeps getting better.

Are *you* the only target?
Can attack many keys at once, spreading costs over those keys: batch NFS, batch ECDL, etc.

"1024-bit factorization is expensive. Our data isn't worth that much, so we're secure."

Is it really so expensive?
Hardware keeps getting better.

Are *you* the only target?
Can attack many keys at once, spreading costs over those keys: batch NFS, batch ECDL, etc.

Is the attacker paying?
Conficker broke into 10 million PCs around the Internet.

Example 2: Tor uses RSA-1024.

Example 3: DNSSEC uses RSA-1024: “tradeoff between the risk of key compromise and performance. . . ”

Example 4: OpenSSL continues to use secret AES array indices.

Example 5: `https://sourceforge.net/account` is protected by SSL but `https://sourceforge.net/develop` redirects browser to `http://sourceforge.net/develop`, turning off the cryptography.