The legacy of export-grade cryptography in the 21st century

Nadia Heninger

University of Pennsylvania

October 6, 2016

International Traffic in Arms Regulations

April 1, 1992 version

Category XIII--Auxiliary Military Equipment ...

(b) Information Security Systems and equipment, cryptographic devices, software, and components specifically designed or modified therefore, including:

(1) Cryptographic (including key management) systems, equipment, assemblies, modules, integrated circuits, components or software with the capability of maintaining secrecy or confidentiality of information or information systems, except cryptographic equipment and software as follows:

(i) Restricted to decryption functions specifically designed to allow the execution of copy protected software, provided the decryption functions are not user-accessible.

(ii) Specially designed, developed or modified for use in machines for banking or money transactions, and restricted to use only in such transactions. Machines for banking or money transactions include automatic teller machines, self-service statement printers, point of sale terminals or equipment for the encryption of interbanking transactions.

...

Timeline of US cryptography export control

- Pre-1994: Encryption software requires individual export license as a munition.
- 1994: US State Department amends ITAR regulations to allow export of approved software to approved countries without individual licenses. 40-bit symmetric cryptography was understood to be approved under this scheme.
- 1995: Netscape develops initial SSL protocol.
- 1996: Bernstein v. United States; California judge rules ITAR regulations are unconstitutional because "code is speech"
- 1996: Cryptography regulation moved to Department of Commerce.
- 1999: TLS 1.0 standardized.
- 2000: Department of Commerce loosens regulations on mass-market and open source software.

Commerce Control List: Category 5 - Info. Security

(May 21, 2015 version)

a.1.a. A symmetric algorithm employing a key length in excess of 56-bits; or

a.1.b. An asymmetric algorithm where the security of the algorithm is based on any of the following:

a.1.b.1. Factorization of integers in excess of 512 bits (e.g., RSA);

a.1.b.2. Computation of discrete logarithms in a multiplicative group of a finite field of size greater than 512 bits (e.g., Diffie-Hellman over Z/pZ); or

a.1.b.3. Discrete logarithms in a group other than mentioned in 5A002.a.1.b.2 in excess of 112 bits (e.g., Diffie-Hellman over an elliptic curve);

a.2. Designed or modified to perform cryptanalytic functions;

"The government must be wary of suffocating [the encryption software] industry with regulation in the new digital age, but we must be able to strike a balance between the legitimate concerns of the law enforcement community and the needs of the marketplace." — U.S. Vice President Al Gore, September 1997

"The government must be wary of suffocating [the encryption software] industry with regulation in the new digital age, but we must be able to strike a balance between the legitimate concerns of the law enforcement community and the needs of the marketplace." — U.S. Vice President Al Gore, September 1997

"Because, if, in fact, you can't crack that [encryption] at all, government can't get in, then everybody is walking around with a Swiss bank account in their pocket – right? So there has to be some concession to the need to be able to get into that information somehow." — President Obama, March 2016

Historical experiment: How did this "compromise" work out for us?

Updated timeline of export control

- 1994: ITAR regulatory scheme.
- 1995: Netscape develops initial SSL protocol.
- 1996: Cryptography regulation moved to Department of Commerce.
- 1999: TLS 1.0 standardized.
- 2000: Department of Commerce loosens regulations on mass-market and open source software.
- ...

Updated timeline of export control

- 1994: ITAR regulatory scheme.
- 1995: Netscape develops initial SSL protocol.
- 1996: Cryptography regulation moved to Department of Commerce.
- 1999: TLS 1.0 standardized.
- 2000: Department of Commerce loosens regulations on mass-market and open source software.
- ...
- March 2015: **FREAK** attack; 10% of popular sites vulnerable.
- May 2015: **Logjam** attack; 8% of popular sites vulnerable.
- March 2016: **DROWN** attack; 25% of popular sites vulnerable.

The FREAK attack

A Messy State of the Union: Taming the Composite State Machines of TLS Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, Jean Karim Zinzindohoue *Oakland 2015*

Textbook RSA Encryption

[Rivest Shamir Adleman 1977]

Public Key

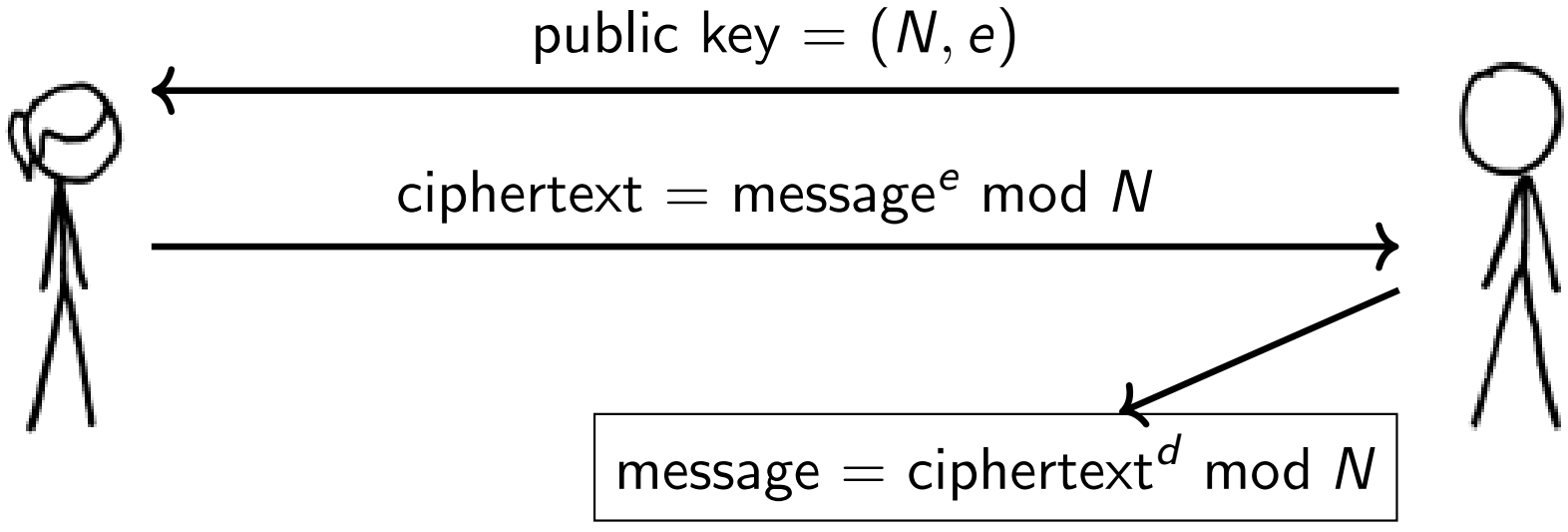
$N = pq$ modulus

e encryption exponent

Private Key

p, q primes

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



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 | Pl

$100

bonus cash back offer

Offer Details

BankAmericard Cash Rewards™ credit card

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

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

The fast convenient way to pay your bills.

Donating homes to veterans

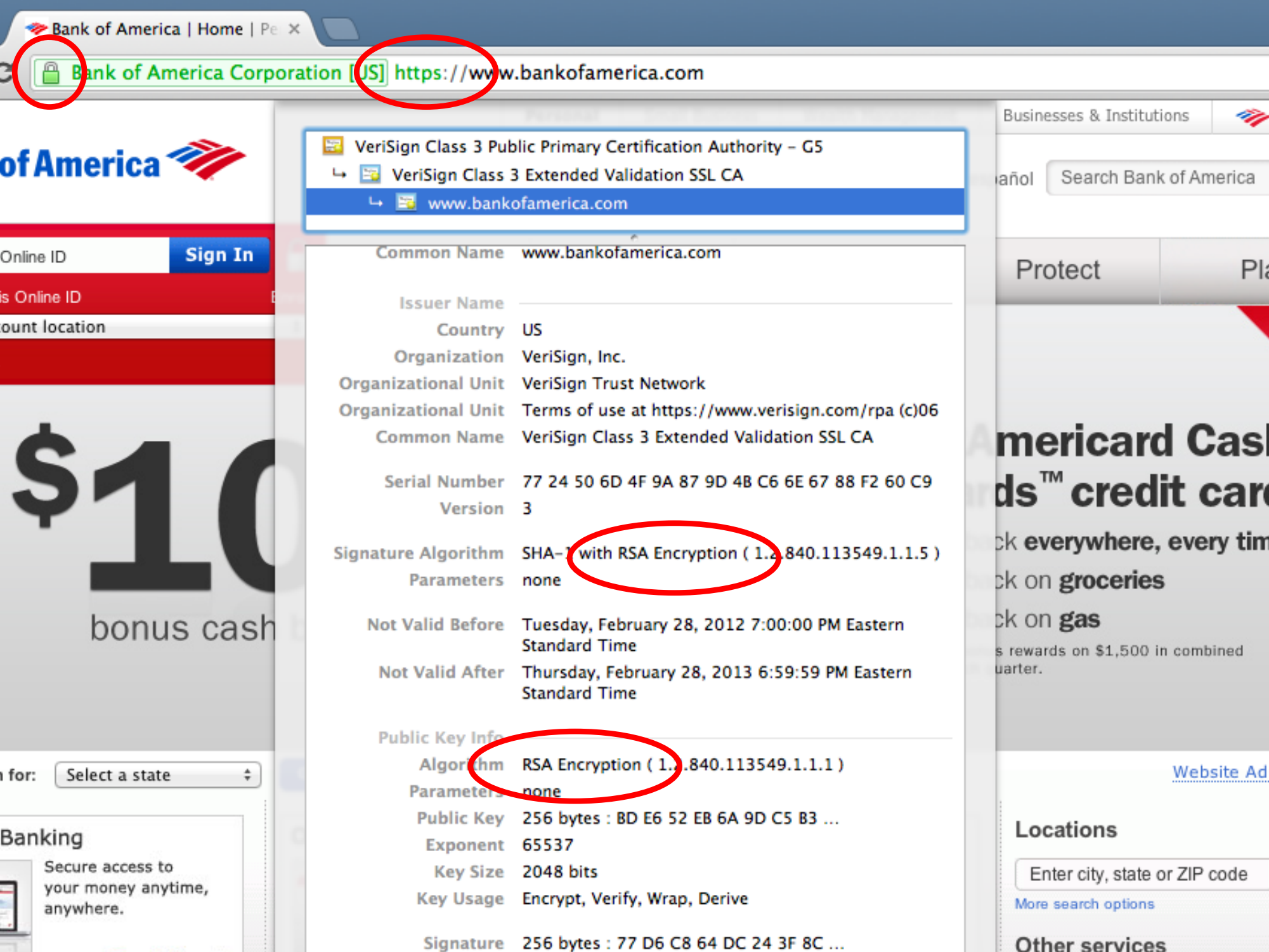
We've committed to donate 1000 properties to veterans and first responders.

Locations

Enter city, state or ZIP code

More search options

Other services


Bank of America | Home
Bank of America Corporation [US] https://www.bankofamerica.com
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 ...
Businesses & Institutions
Search Bank of America
Sign In
Protect
bonus cash
Banking
Secure access to your money anytime, anywhere.
Locations
Enter city, state or ZIP code
More search options
Other services

TLS RSA Key Exchange

client hello: client random

[. . . RSA . . .]

TLS RSA Key Exchange

client hello: client random

[. . . RSA . . .]

server hello: server random, [RSA]

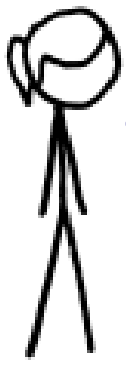
certificate = **RSA pubkey** k_{2048} + CA signatures

TLS RSA Key Exchange

client hello: client random

[. . . RSA . . .]

server hello: server random, [RSA]



certificate = **RSA pubkey** k_{2048} + CA signatures

client key exchange: $\mathbf{RSAenc_{k_{2048}}(pms)}$

client finished: $\text{Auth}_{k_{mc}}(\text{dialog})$

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS RSA Key Exchange

client hello: client random

[. . . RSA . . .]

server hello: server random, [RSA]

certificate = **RSA pubkey** k_{2048} + CA signatures

client key exchange: $\mathbf{RSAenc_{k_{2048}}(pms)}$

client finished: $\text{Auth}_{k_{m_c}}(\text{dialog})$

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

server finished: $\text{Auth}_{k_{m_s}}(\text{dialog})$

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS RSA Key Exchange

client hello: client random

[. . . RSA . . .]

server hello: server random, [RSA]

certificate = **RSA pubkey** k_{2048} + CA signatures

client key exchange: $\mathbf{RSAenc_{k_{2048}}(pms)}$

client finished: $\mathrm{Auth}_{k_{m_c}}(\mathrm{dialog})$

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

server finished: $\mathrm{Auth}_{k_{m_s}}(\mathrm{dialog})$

$\mathrm{Enc}_{k_e}(\mathrm{request})$

Question: How do you selectively weaken a protocol based on RSA?

Question: How do you selectively weaken a protocol based on RSA?

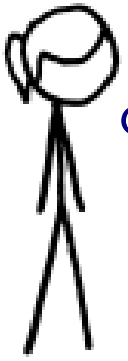
Export answer: Optionally use a small RSA key.

TLS RSA Export Key Exchange

client hello: client random

[. . . RSA_EXPORT . . .]

server hello: server random, [RSA_EXPORT]



certificate = **RSA pubkey k_{2048}** + CA signatures

server key exchange: **RSA pubkey k_{512}**

client key exchange: **RSAenc$_{k_{512}}$(pms)**

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

client finished: Auth$_{k_{m_c}}$(dialog)

KDF(*pms*, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

server finished: Auth$_{k_{m_s}}$(dialog)

Enc$_{k_e}$(request)

RSA export cipher suites in TLS

In March 2015, export cipher suites supported by 36.7% of the 14 million sites serving browser-trusted certificates!

`TLS_RSA_EXPORT_WITH_RC4_40_MD5`
`TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5`
`TLS_RSA_EXPORT_WITH_DES40_CBC_SHA`

Totally insecure, but no modern client would negotiate export ciphers. ... right?

Tracking the Freak Attack Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman `freakattack.com`

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

[. . . RSA . . .]

FREAK: MITM downgrade attack to export RSA

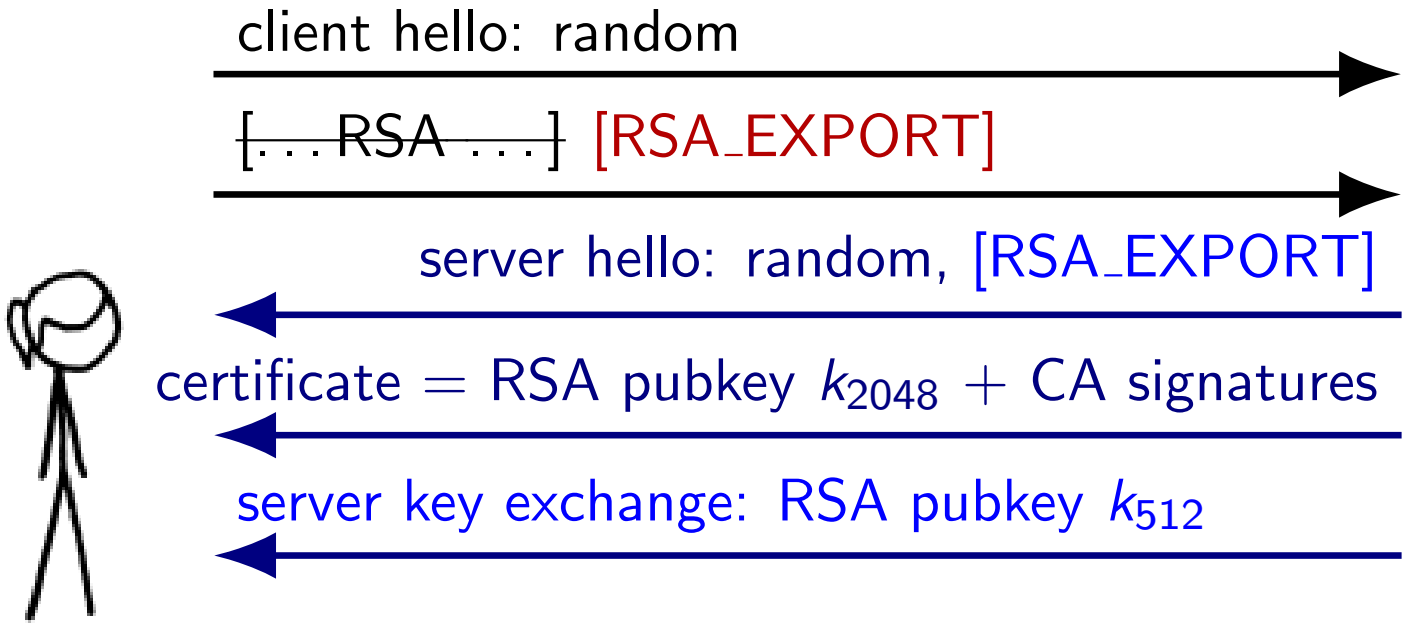
Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[. . . RSA . . .]~~ [RSA_EXPORT]

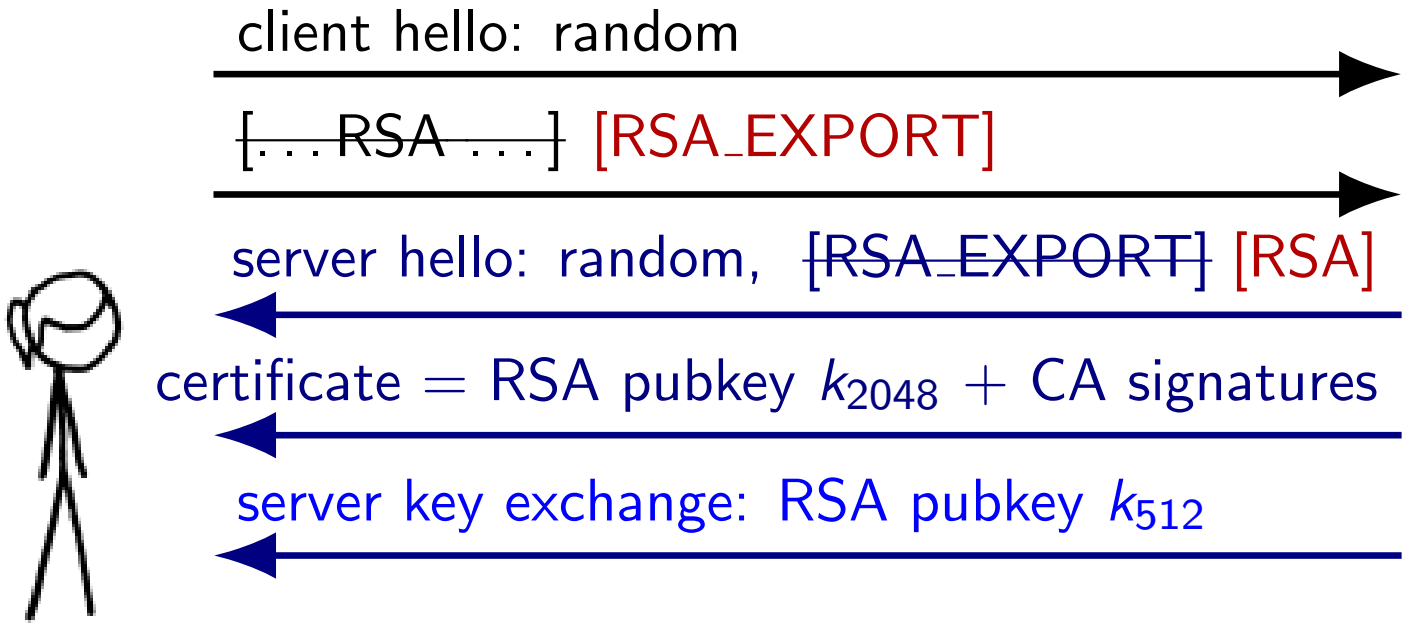
FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]



FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]



FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[...RSA...]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[. . . RSA . . .]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

client finished: $\mathrm{Auth}_{k_{m_c}}(\mathrm{dialog})$

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[...RSA...]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

client finished: $\mathrm{Auth}_{k_{mc}}$(modified dialog)

KDF(*pms*, randoms) $\rightarrow$ k_{mc}, k_{ms}, k_e

KDF(*pms*, randoms) $\rightarrow$ k_{mc}, k_{ms}, k_e

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[...RSA...]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

client finished: $\mathrm{Auth}_{k_{mc}}$(modified dialog)

server finished: $\mathrm{Auth}_{k_{mc}}$(dialog)

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[. . . RSA . . .]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

client finished: $\mathrm{Auth}_{k_{mc}}$(modified dialog)

server finished: $\mathrm{Auth}_{k_{ms}}$(modified dialog)

KDF(pms, randoms) $\rightarrow$ k_{mc}, k_{ms}, k_e

KDF(pms, randoms) $\rightarrow$ k_{mc}, k_{ms}, k_e

FREAK: MITM downgrade attack to export RSA

Implementation flaw: Most major browsers accept unexpected server key exchange messages. [BDFKPSZZ 2015]

client hello: random

~~[. . . RSA . . .]~~ [RSA_EXPORT]

server hello: random, ~~[RSA_EXPORT]~~ [RSA]

certificate = RSA pubkey k_{2048} + CA signatures

server key exchange: RSA pubkey k_{512}

client key exchange: $\mathrm{RSAenc}_{k_{512}}(pms)$

client finished: $\mathrm{Auth}_{k_{mc}}$(modified dialog)

server finished: $\mathrm{Auth}_{k_{ms}}$(modified dialog)

Enc_{k_e}(request)

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(pms, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

FREAK vulnerability in practice

- Implementation flaw affected OpenSSL, Microsoft SChannel, IBM JSSE, Safari, Android, Chrome, BlackBerry, Opera, IE

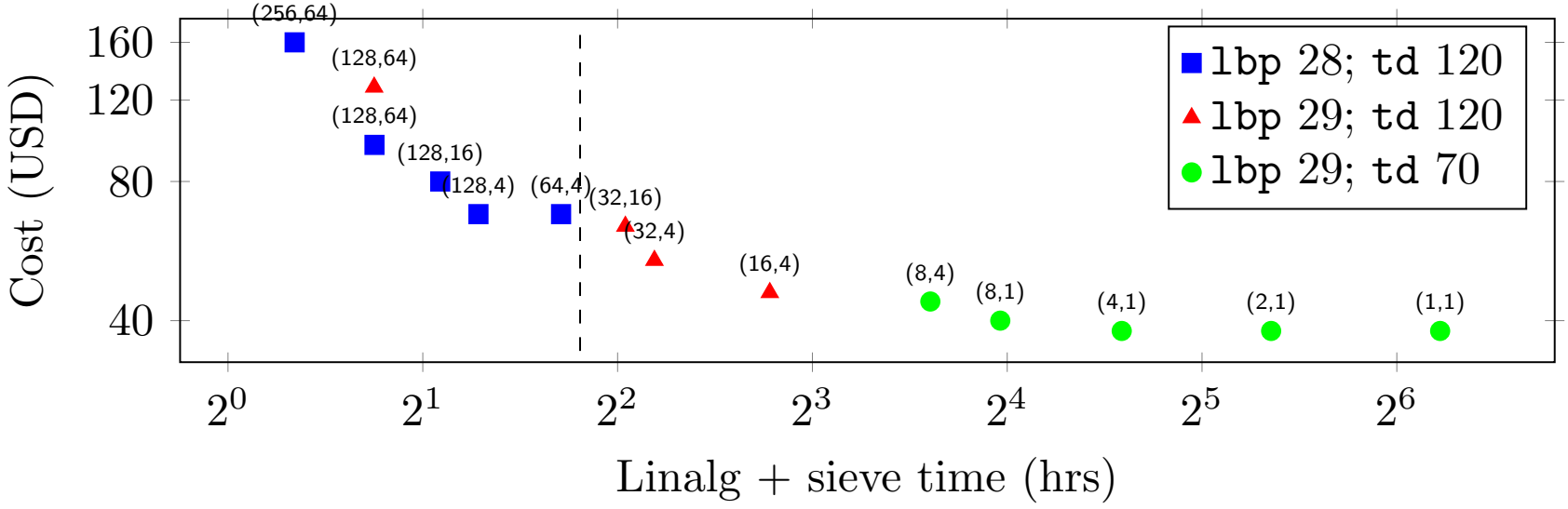
FREAK vulnerability in practice

- Implementation flaw affected OpenSSL, Microsoft SChannel, IBM JSSE, Safari, Android, Chrome, BlackBerry, Opera, IE
- Attack outline:
  1. MITM attacker downgrades connection to export, learns server's ephemeral 512-bit RSA export key.
  2. Attacker factors 512-bit modulus to obtain server private key.
  3. Attacker uses private key to forge client/server authentication for successful downgrade.

FREAK vulnerability in practice

- Implementation flaw affected OpenSSL, Microsoft SChannel, IBM JSSE, Safari, Android, Chrome, BlackBerry, Opera, IE
- Attack outline:
  1. MITM attacker downgrades connection to export, learns server's ephemeral 512-bit RSA export key.
  2. Attacker factors 512-bit modulus to obtain server private key.
  3. Attacker uses private key to forge client/server authentication for successful downgrade.
- Attacker challenge: Need to know 512-bit private key before connection times out
- Implementation shortcut: "Ephemeral" 512-bit RSA server keys generated only on application start; last for hours, days, weeks, months.

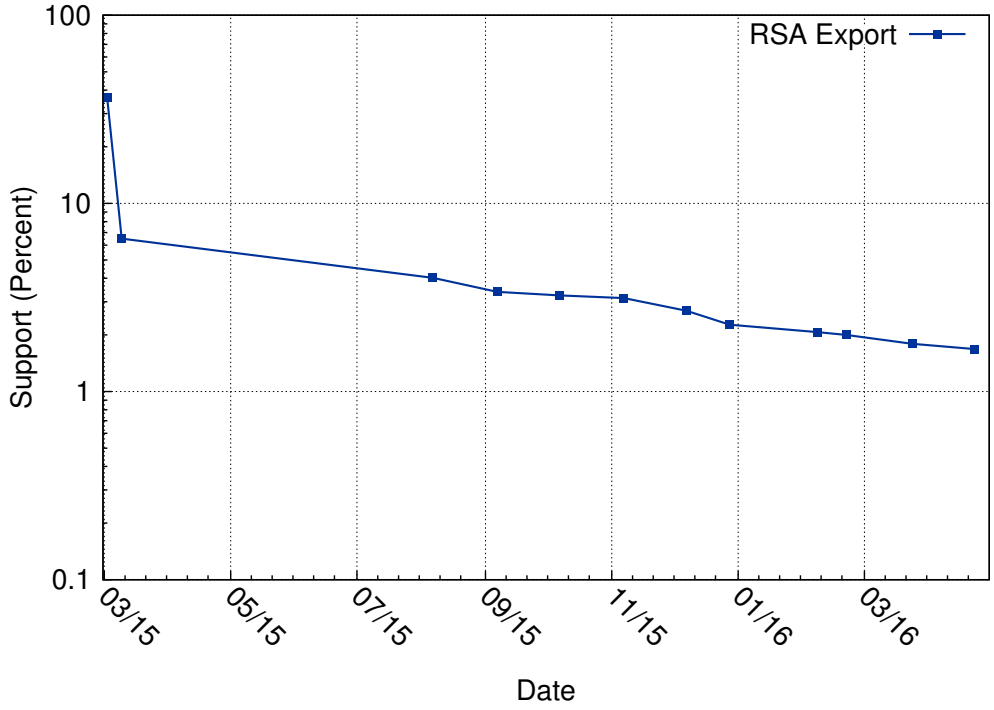
Factoring 512-bit RSA in the cloud



Factoring as a Service Luke Valenta, Shaanan Cohney, Alex Liao, Joshua Fried, Satya Bodduluri, and Nadia Heninger. *FC 2016.* `seclab.upenn.edu/projects/faas/`

FREAK mitigation

- All major browsers pushed bug fixes.
- Server operators encouraged to disable export cipher suites.



But still enabled for about 2% of trusted sites today.

The Logjam attack

Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice
David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, Paul Zimmermann *CCS 2015* `weakdh.org`

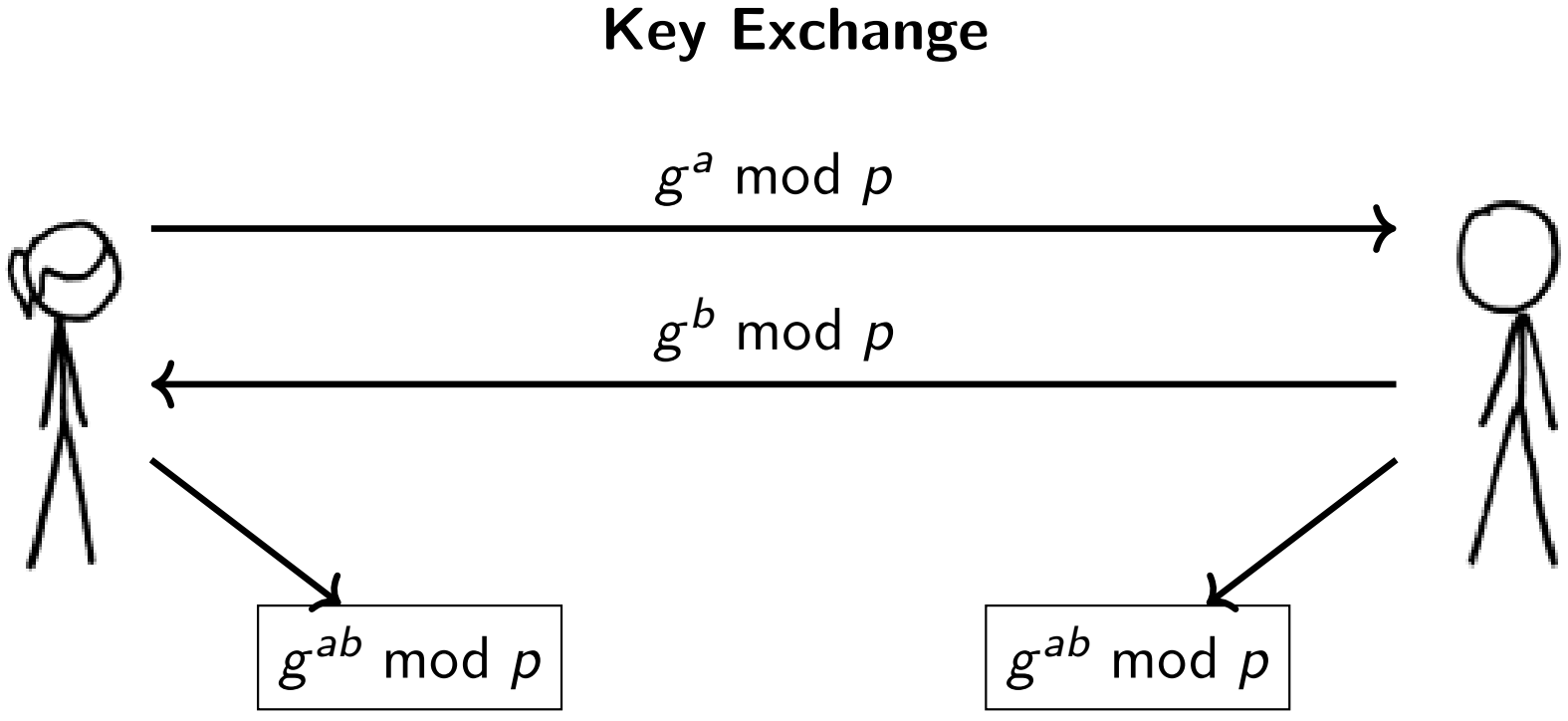
Textbook (Finite-Field) Diffie-Hellman

[Diffie Hellman 1976]

Public Parameters

p a prime (so $\mathbb{F}_p^*$ is a cyclic group)

$g < p$ group generator (often 2 or 5)



TLS Diffie-Hellman Key Exchange

client hello: client random

[. . . DHE . . .]

TLS Diffie-Hellman Key Exchange

client hello: client random

[. . . DHE . . .]

server hello: server random, [DHE]

certificate = public RSA key + CA signatures

server kex: $p, g, \mathbf{g^a}$, $\text{Sign}_{\text{RSAkey}}(p, g, g^a)$

TLS Diffie-Hellman Key Exchange

client hello: client random

[. . . DHE . . .]

server hello: server random, [DHE]

certificate = public RSA key + CA signatures

server kex: $p, g, \mathbf{g^a}$, $\text{Sign}_{\text{RSAkey}}(p, g, g^a)$

client kex: $\mathbf{g^b}$

client finished: $\text{Auth}_{k_{m_c}}(\text{dialog})$

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS Diffie-Hellman Key Exchange

client hello: client random

[. . . DHE . . .]

server hello: server random, [DHE]

certificate = public RSA key + CA signatures

server kex: $p, g, \mathbf{g^a}$, $\text{Sign}_{\text{RSAkey}}(p, g, g^a)$

client kex: $\mathbf{g^b}$

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

client finished: $\text{Auth}_{k_{m_c}}$(dialog)

server finished: $\text{Auth}_{k_{m_s}}$(dialog)

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

TLS Diffie-Hellman Key Exchange

client hello: client random

[. . . DHE . . .]

server hello: server random, [DHE]

certificate = public RSA key + CA signatures

server kex: $p, g, \mathbf{g^a}$, $\text{Sign}_{\text{RSAkey}}(p, g, g^a)$

client kex: $\mathbf{g^b}$

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

client finished: $\text{Auth}_{k_{m_c}}$(dialog)

server finished: $\text{Auth}_{k_{m_s}}$(dialog)

Enc_{k_e}(request)

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

Question: How do you selectively weaken a protocol based on Diffie-Hellman?

Question: How do you selectively weaken a protocol based on Diffie-Hellman?

Export answer: Optionally use a smaller prime.

TLS Diffie-Hellman Export Key Exchange

client hello: client random

[. . . DHE_EXPORT . . .]

server hello: server random, [DHE_EXPORT]

certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

client kex: $\mathbf{g^b}$

client finished: $\mathrm{Auth}_{k_{mc}}(\mathrm{dialog})$

server finished: $\mathrm{Auth}_{k_{ms}}(\mathrm{dialog})$

$\mathrm{Enc}_{k_e}(\mathrm{request})$

KDF(g^{ab}, randoms) $\rightarrow$ k_{mc}, k_{ms}, k_e

KDF(g^{ab}, randoms) $\rightarrow$ k_{mc}, k_{ms}, k_e

Diffie-Hellman export cipher suites in TLS

```
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_DH_Anon_EXPORT_WITH_RC4_40_MD5
TLS_DH_Anon_EXPORT_WITH_DES40_CBC_SHA
```

April 2015: 8.4% of Alexa top 1M HTTPS support `DHE_EXPORT`.

Totally insecure, but no modern client would negotiate export ciphers. ... right?

Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.

client hello: random

[. . . DHE . . .]

Logjam: Active downgrade attack to export Diffie-Hellman

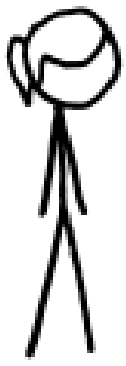
Protocol flaw: Server does not sign chosen cipher suite.

client hello: random

~~[. . . DHE . . .]~~ [DHE_EXPORT]

Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.



client hello: random

~~[. . . DHE . . .]~~ [DHE_EXPORT]

server hello: random, [DHE_EXPORT]

certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.

client hello: random

~~[...DHE...]~~ [DHE_EXPORT]

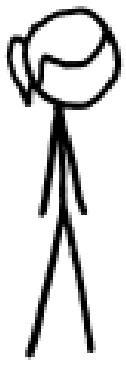
server hello: random, ~~[DHE_EXPORT]~~[DHE]

certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.



client hello: random

~~[. . . DHE . . .]~~ [DHE_EXPORT]

server hello: random, ~~[DHE_EXPORT]~~[DHE]

certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

client kex: $\mathbf{g^b}$

client finished: $\mathrm{Auth}_{k_{m_c}}(\mathrm{dialog})$

$\mathrm{KDF}(g^{ab}, \mathrm{randoms}) \rightarrow k_{m_c}, k_{m_s}, k_e$

$\mathrm{KDF}(g^{ab}, \mathrm{randoms}) \rightarrow k_{m_c}, k_{m_s}, k_e$

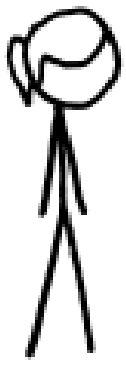
Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.

client hello: random

~~[...DHE...]~~ [DHE_EXPORT]

server hello: random, ~~[DHE_EXPORT]~~[DHE]



certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

client kex: $\mathbf{g^b}$

client finished: $\mathrm{Auth}_{k_{m_c}}$(modified dialog)

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

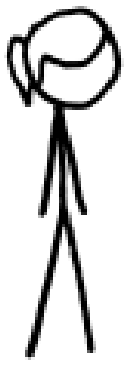
Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.

client hello: random

~~[. . . DHE . . .]~~ [DHE_EXPORT]

server hello: random, ~~[DHE_EXPORT]~~[DHE]



certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

client kex: $\mathbf{g^b}$

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

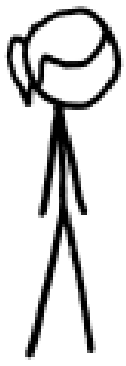
client finished: $\mathrm{Auth}_{k_{m_c}}$(modified dialog)

server finished: $\mathrm{Auth}_{k_{m_c}}$(dialog)

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.



client hello: random

~~[...DHE...]~~ [DHE_EXPORT]

server hello: random, ~~[DHE_EXPORT]~~[DHE]

certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

client kex: $\mathbf{g^b}$

client finished: $\mathrm{Auth}_{k_{m_c}}$(modified dialog)

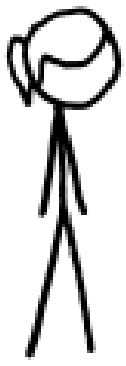
server finished: $\mathrm{Auth}_{k_{m_s}}$(modified dialog)

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.



client hello: random

~~[...DHE...]~~ [DHE_EXPORT]

server hello: random, ~~[DHE_EXPORT]~~[DHE]

certificate = public RSA key + CA signatures

server kex: $\mathbf{p_{512}}, g, \mathbf{g^a}$, $\mathrm{Sign}_{\mathrm{RSAkey}}(p_{512}, g, g^a)$

client kex: $\mathbf{g^b}$

client finished: $\mathrm{Auth}_{k_{mc}}$(modified dialog)

server finished: $\mathrm{Auth}_{k_{ms}}$(modified dialog)

Enc_{k_e}(request)

KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

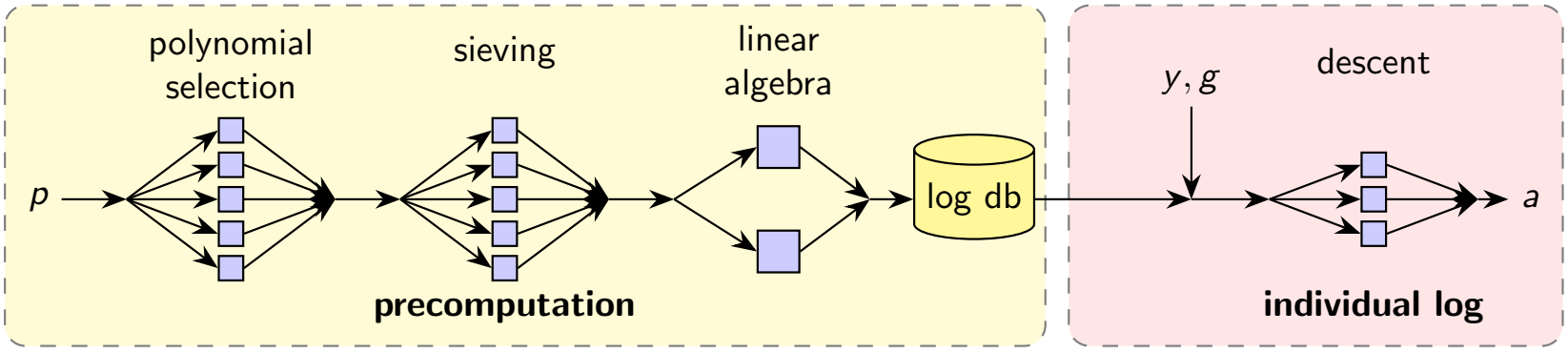
KDF(g^{ab}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

Carrying out the Diffie-Hellman export downgrade attack

1. Attacker man-in-the-middles connection, changing messages as necessary.
2. Attacker computes discrete log of g^a or g^b to learn session keys.
3. Attacker uses session keys to forge client, server finished messages.

- Attacker challenge: compute client or server ephemeral Diffie-Hellman secrets before connection times out
- For export Diffie-Hellman, most servers actually generate per-connection secrets.

How long does it take to compute discrete logs?



	polysel	sieving	linalg	descent
	2000-3000 cores		288 cores	36 cores
DH-512	3 hours	15 hours	120 hours	70 seconds

Precomputation can be done once and reused for many individual logs!

Implementing Logjam

Parameters hard-coded in implementations or built into standards.

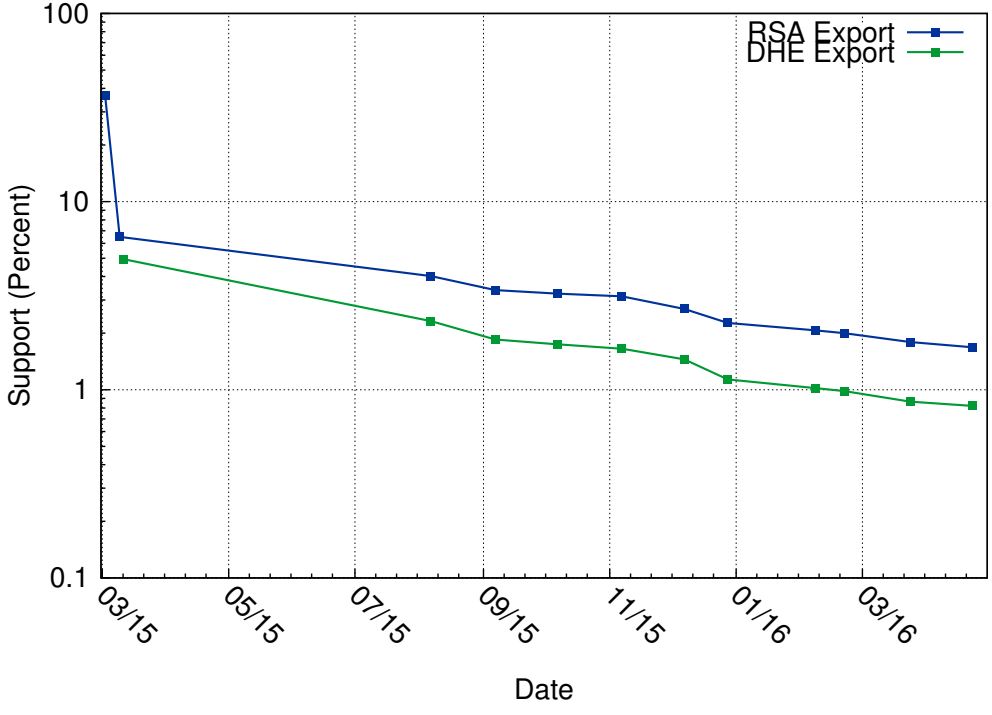
97% of `DHE_EXPORT` hosts choose one of three 512-bit primes.

Hosts	Source	Year	Bits
80%	Apache 2.2	2005	512
13%	mod_ssl 2.3.0	1999	512
4%	JDK	2003	512

- Carried out precomputation for common primes.
- After 1 week precomputation, median individual log time 70s.
- Logjam and our precomputations can be used to break connections to 8% of the HTTPS top 1M sites!

Logjam mitigation

- Server operators encouraged to disable export cipher suites.



- Major browsers have raised minimum DH lengths:
  IE, Chrome, Firefox to 1024 bits; Safari to 768.

- TLS 1.3 draft includes anti-downgrade flag in client random.

Ramifications for 1024-bit Diffie-Hellman

- 1024-bit discrete log precomputation within range of large governments
- Widespread reuse of groups may explain some decryption abilities.

New Result:

- Can trapdoor 1024-bit primes in computationally undetectable way
- We computed 1024-bit discrete log in 2 months on 2000–3000 cores

A kilobit hidden SNFS discrete logarithm computation

Joshua Fried, Pierrick Gaudry, Nadia Heninger, Emmanuel Thomé
`eprint.iacr.org/2016/961`

The DROWN attack

DROWN: Breaking TLS using SSLv2

Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt

USENIX Security 2016. `https://drownattack.com`

A brief history of SSL/TLS

SSL 1.0	Terribly insecure; never released.
SSL 2.0	Released 1995; terribly insecure.
SSL 3.0	Released 1996; considered insecure since 2014/POODLE.
TLS 1.0	Released 1999.
TLS 1.1	Released 2006.
TLS 1.2	Released 2008.
TLS 1.3	Under development.

Clients will negotiate highest supported version, so it's ok for servers to support old versions for compatibility ...right?

Question: How do you selectively weaken a protocol that uses symmetric ciphers?

Question: How do you selectively weaken a protocol that uses symmetric ciphers?

SSLv2 export answer: Optionally send all but 40 bits of secret key in public.

$mk = mk_{clear} \parallel mk_{secret}$

SSLv2 Handshake

client hello: [cipher suites], challenge

server hello: [cipher suites], connection_ID

certificate = RSA pubkey + CA signatures

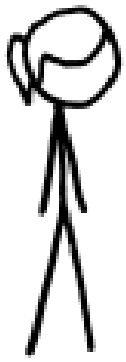
$\mathbf{mk_{clear}}$ + RSAenc($\mathbf{mk_{secret}}$)

server verify: Enc_{k_e}(challenge)

client finished

server finished

Enc_{k_e}(request)



KDF(mk_{clear}, mk_{secret}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

KDF(mk_{clear}, mk_{secret}, randoms) $\rightarrow$ k_{m_c}, k_{m_s}, k_e

Notable properties of SSLv2

- Devastating MITM attacks.
- RSA key exchange only.
- `master_key` varies in size according to symmetric cipher. For TLS, premaster secret always has 48 bytes.
- Both encryption and authentication use 40-bit symmetric secret for export cipher suites. (TLS export cipher suites extract 40-bit secret for encryption from 48-byte PMS.)
- Server authenticates first. (Not well specified in spec; implementations agree.)

RSA PKCS #1 v1.5 padding

`m = 00 02 [random padding string] 00 [data]`

- Encrypter pads message, then encrypts padded message using RSA public key.
- Decrypter decrypts using RSA private key, strips off padding to recover original data.

Q: What happens if a decrypter decrypts a message and the padding isn't in correct format?

A: Throw an error?

[Bleichenbacher 1998] padding oracle attacks

- **Attack:** If no padding error, attacker learns that first two bytes of plaintext are 00 02.
- Error messages are *oracle* for first two bytes of plaintext.

Adaptive chosen ciphertext attack:

1. Attacker wishes to decrypt RSA ciphertext c.
2. Attacker queries oracle on $s^e c \bmod N$ for random values of s.
3. Attacker successively narrows range of possible m until unique answer remains . . . eventually. *"Million message attack."*

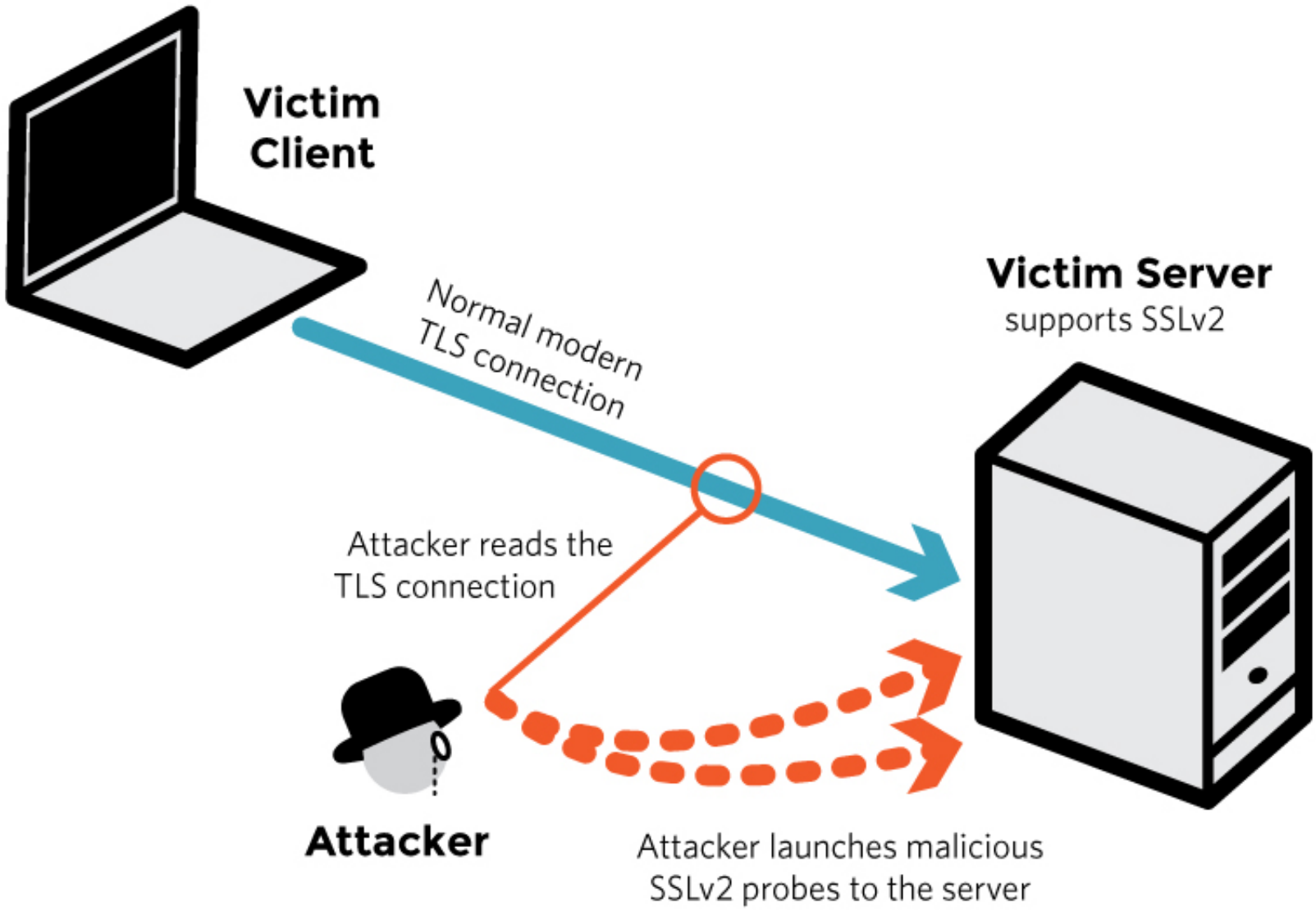
Mitigation: ~~Use OAEP.~~ Implementations don't reveal errors to attacker; proceed with protocol using fake random plaintext.

Protocol flaw in SSLv2 is a Bleichenbacher oracle

- Server sends `ServerVerify` message before client authenticates!
- Attacker can learn 2 most significant + 6 least significant bytes of plaintext by brute forcing 40 bits:
  `m = 00 02 [padding] 00 [`mk_{secret}`]`

Observation: Servers use the same certificate/RSA public key for all SSL/TLS protocol versions.

DROWN: Using SSLv2 to decrypt TLS



DROWN attack complexity for 2048-bit RSA

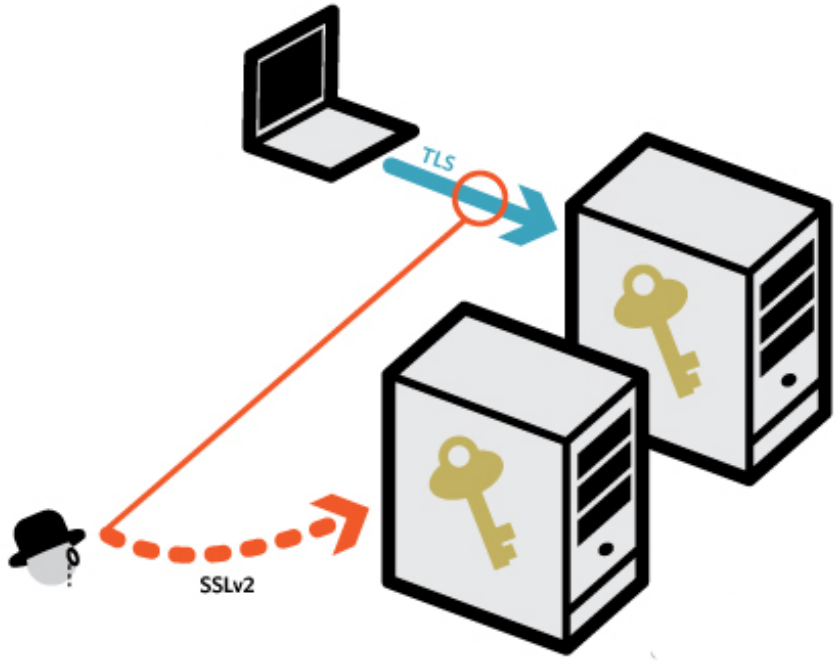
Optimizing	**Ciphertexts**	**Fractions**	**SSLv2 connections**	**Offline work**
offline work	12,743	1	50,421	$2^{49.64}$
offline work	1,055	10	46,042	$2^{50.63}$
compromise	4,036	2	41,081	$2^{49.98}$
online work	2,321	3	38,866	$2^{51.99}$
online work	906	8	39,437	$2^{52.25}$

How many servers were vulnerable to DROWN?

- At disclosure, 1.7M (10%) of HTTPS servers with browser-trusted certificates supported SSLv2.

How many servers were vulnerable to DROWN?

- At disclosure, 1.7M (10%) of HTTPS servers with browser-trusted certificates supported SSLv2.
- However, many more were vulnerable, due to key reuse across servers and *across protocols*.



- Overall, 22% of HTTPS servers with trusted certs (25% of the Top Million) were vulnerable to DROWN.

DROWN mitigations

- Update OpenSSL.
  OpenSSL team patched several bugs, disabled SSLv2 by default.
  One month after disclosure, only 15% of HTTPS hosts had patched!
- Fully disable SSLv2.
  Don't only disable export ciphers. If only ciphers are disabled, make sure they're actually disabled (CVE-2015-3197).
- Have single-use keys.
  Prudent to use different keys across different protocols and protocol versions.

Technical Lessons

- Obsolete cryptography considered harmful.

  Maintaining support for old services for backward compatibility isn't harmless.

- Limit complexity.

  Cryptographic APIs and state machines often overly complex.
  Design protocols to limit implementation mistakes.
  Design APIs to limit usage mistakes.

- Weakened cryptography considered harmful.

  Twenty years later, all three forms of SSL/TLS export crypto led to devastating attacks:

  - Export RSA (FREAK attack)
  - Export DHE (Logjam)
  - Export symmetric (DROWN).

Lessons for Policy

- *Technical backdoors in our infrastructure don't go away even when the political environment changes.*

  Twenty years of computing progress has brought attacks within range of modest attackers.

- Cannot assign cryptography based on nationality.

- Technological evidence opposes backdooring cryptography.

  Complexity of export cipher suites seems particularly prone to implementation vulnerabilities.

The good news: TLS can be secure

- TLS 1.2 with good choice of ciphers can be secure.[1]
- TLS 1.3 aggressively banning bad options.
  - Eliminating RSA key exchange.
  - Mminimum 2048 bits for FF-DHE.

[1] . . . as far as is publicly known, assuming implementations are correct, and assuming domain and key aren't exposed elsewhere in a weaker configuration.

A Messy State of the Union: Taming the Composite State Machines of TLS Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, Jean Karim Zinzindohoue. *Oakland 2015*.

Factoring as a Service Luke Valenta, Shaanan Cohney, Alex Liao, Joshua Fried, Satya Bodduluri, and Nadia Heninger. *FC 2016*. `seclab.upenn.edu/projects/faas/`

Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, Paul Zimmermann. *CCS 2015*. `weakdh.org`

DROWN: Breaking TLS using SSLv2 Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. To appear in *Usenix Security 2016*. `drownattack.com`

A Kilobit Hidden SNFS Discrete Logarithm Computation

Joshua Fried, Pierrick Gaudry, Nadia Heninger, Emmanuel Thomé `eprint.iacr.org/2016/961`