TLS and Attacks

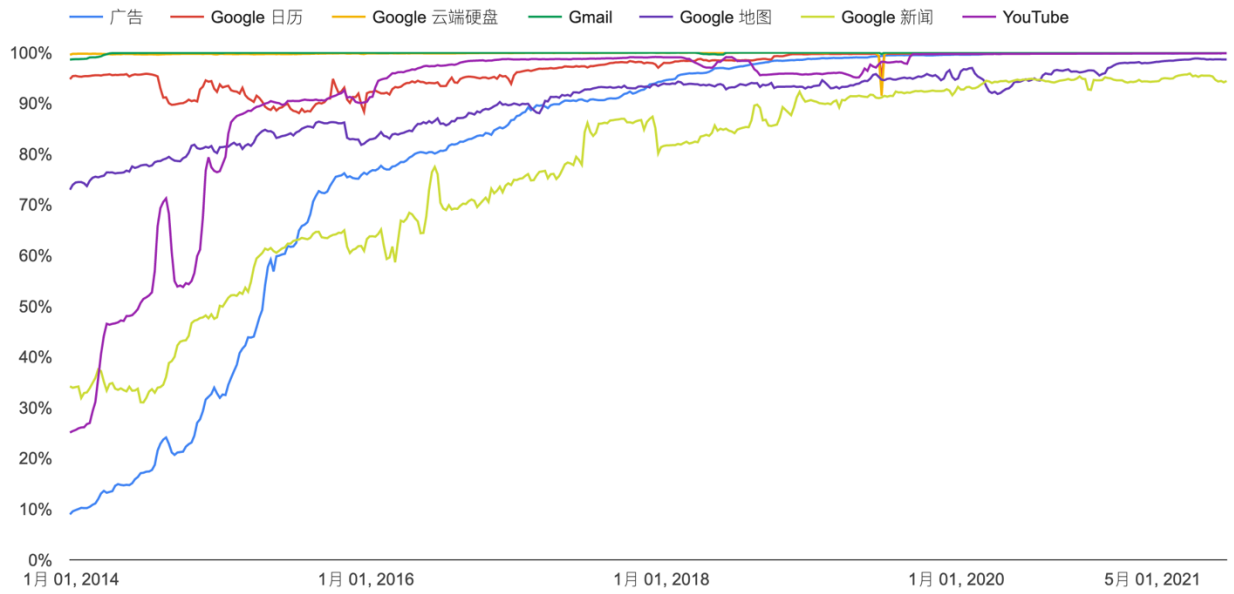
Haixin Duan

Google 统计HTTPS（HTTP/TLS）

Google 的网站

Google 产品和服务中的加密情况（按产品）

此图表提供了多款产品的加密流量的概况。其中的数据基于相应产品所带来的大多数 Google 流量。我们会继续努力解决那些导致部分 Google 产品很难支持加密的技术障碍。此图表会随着时间的推移而变化，以反映产品加密进展情况。



其他热门网站
(top 100)

我们使用了私有数据来源，以跟踪互联网上排名前 100 位的非 Google 网站的 HTTPS 状况。据我们估算，这些网站的流量约占全球所有网站流量的 25%。

默认使用 HTTPS 的网站

97 / 100

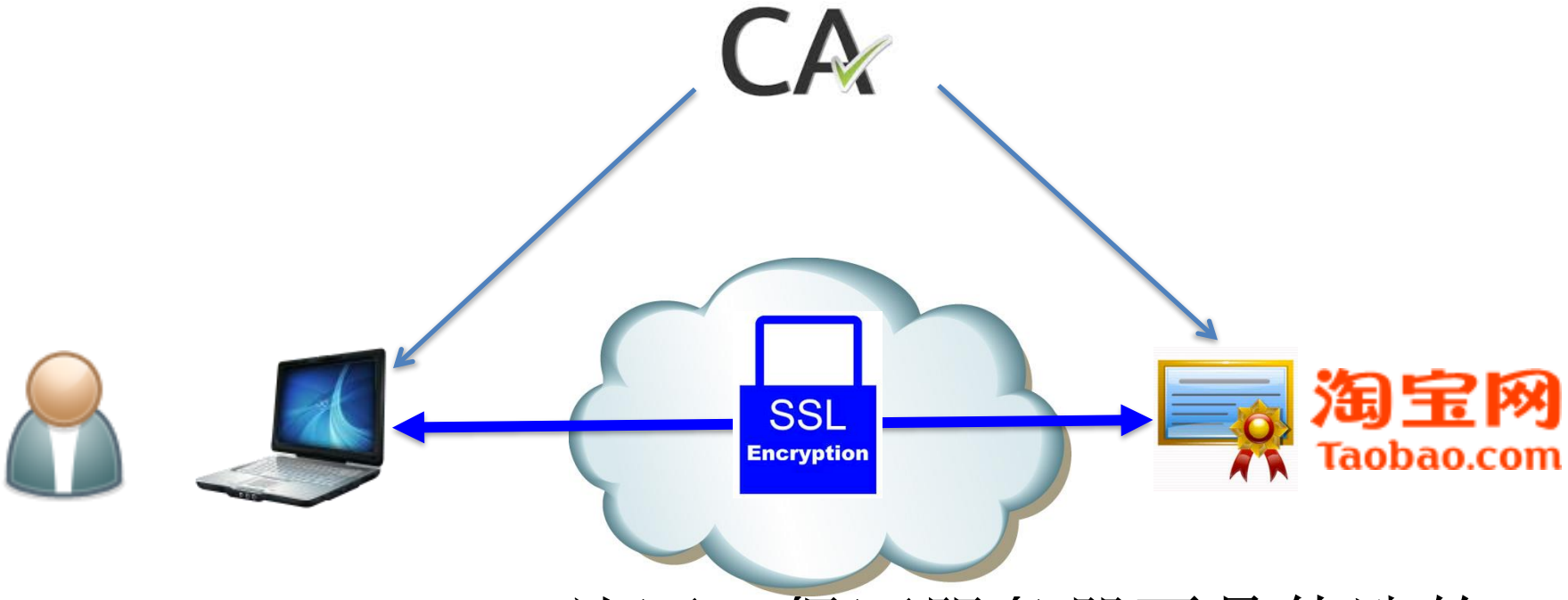
支持 HTTPS 的网站

100 / 100

在 2019年1月，我们归档了各大热门网站页面的 HTTPS 状况图表，并改为显示汇总图表。

开放式环境中的安全通信

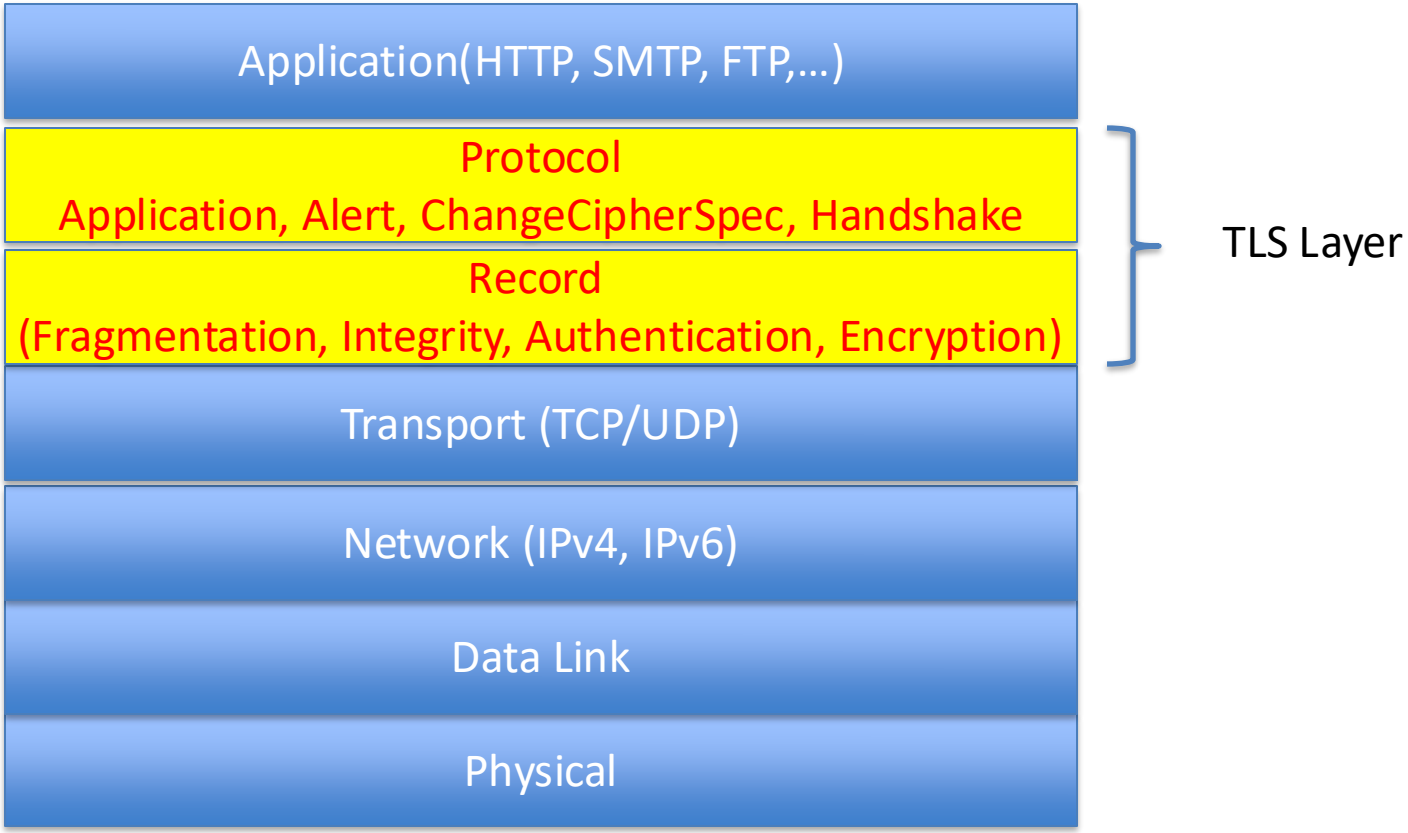
Public Key Infrastructure (Trusted Third Party)



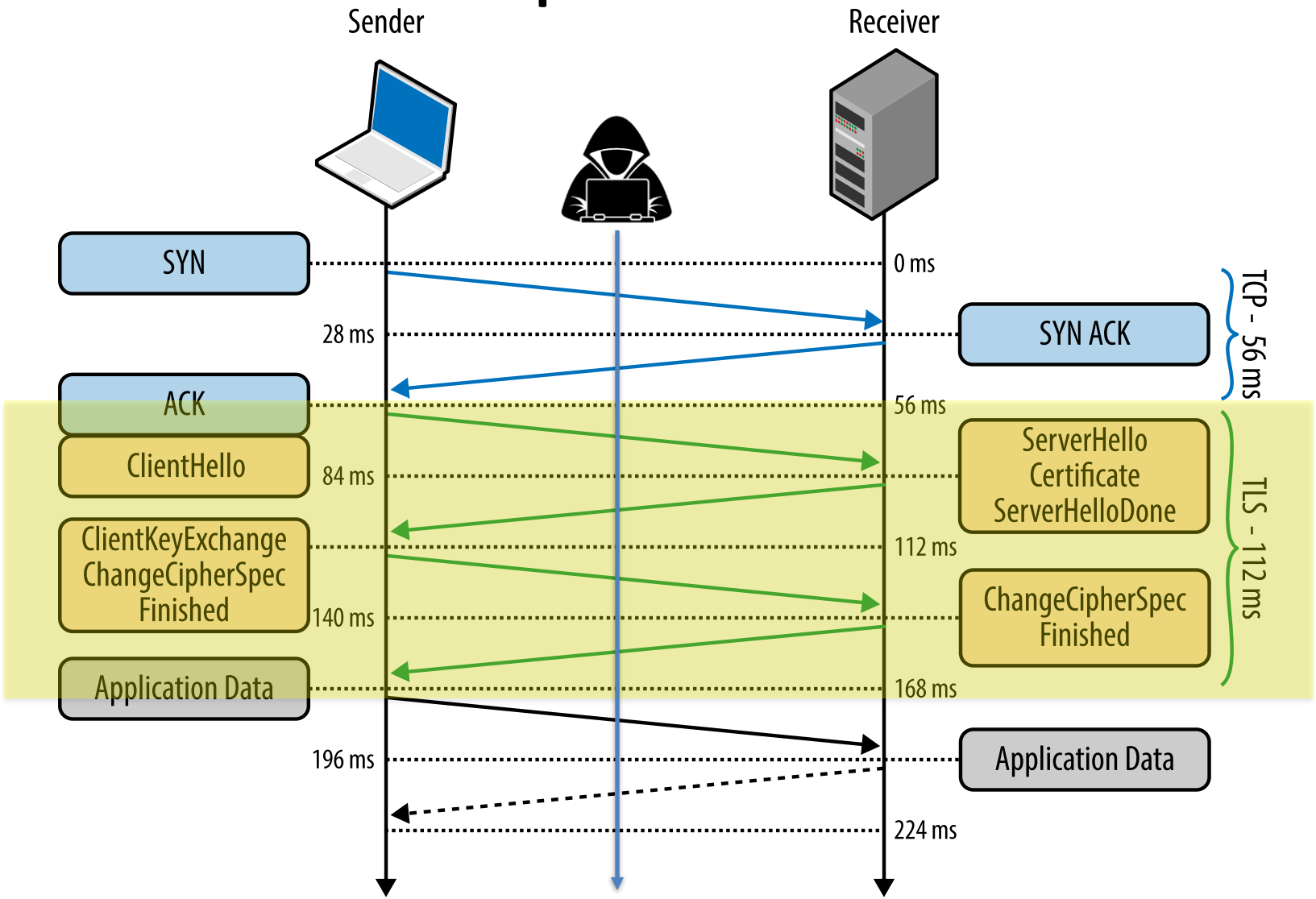
- Authentication: 认证，保证服务器不是伪造的
- Confidentiality： 数据加密传输防止窃听、泄密
- Integrity： 防止数据伪造、被篡改、重发等

Protocol stack

- The most widely used secure protocol
- For Web, it is a MUST, not optional



TLS protocol



MITM: Sniffering, Reply, modification

History of SSL/TLS

- **1994: SSL 2.0** released, by Netscape; SSL 2.0 deprecated(prohibited) in 2011(RFC6176)
  - Deficiencies:
    - Message Authentication: MD5
    - Handshake message not protected, MITM
    - integrity and message encryption use the same key
    - Sessions can be easily terminated
- 1995: Microsoft release PCT v1.0
- 1996: Netscape SSL v3.0

Taher Elgamal, father of SSL

Internet Engineering Task Force (IETF)
Request for Comments: 6101
Category: Historic
ISSN: 2070-1721

A. Freier
P. Karlton
Netscape Communications
P. Kocher
Independent Consultant

August 2011

The Secure Sockets Layer (SSL) Protocol Version 3.0

Abstract

This document is published as a historical record of the SSL 3.0 protocol. The original Abstract follows.

This document specifies version 3.0 of the Secure Sockets Layer (SSL 3.0) protocol, a security protocol that provides communications privacy over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.

Foreword

Although the SSL 3.0 protocol is a widely implemented protocol, a pioneer in secure communications protocols, and the basis for Transport Layer Security (TLS), it was never formally published by the IETF, except in several expired Internet-Drafts. This allowed no easy referencing to the protocol. We believe a stable reference to the original document should exist and for that reason, this document describes what is known as the last published version of the SSL 3.0 protocol, that is, the November 18, 1996, version of the protocol.

Analysis of the SSL 3.0 protocol

David Wagner
University of California, Berkeley
daw@cs.berkeley.edu

Bruce Schneier
Counterpane Systems
schneier@counterpane.com

Abstract

The SSL protocol is intended to provide a practical, application-layer, widely applicable connection-oriented mechanism for Internet client/server communications security. This note gives a detailed technical analysis of the cryptographic strength of the SSL 3.0 protocol. A number of minor flaws in the protocol and several new active attacks on SSL are presented; however, these can be easily corrected without overhauling the basic structure of the protocol. We conclude that, while there are still a few technical wrinkles to iron out, on the whole SSL 3.0 is a valuable contribution towards practical communications security.

1 Introduction

gives some background on SSL 3.0 and its predecessor SSL 2.0. Sections 3 and 4 explore several possible attacks on the SSL protocol and offer some technical discussion on the cryptographic protection afforded by SSL 3.0; this material is divided into two parts, with the SSL record layer analyzed in Section 3 and the SSL key-exchange protocol considered in Section 4. Finally, Section 5 concludes with a high-level view of the SSL protocol's strengths and weaknesses.

2 Background

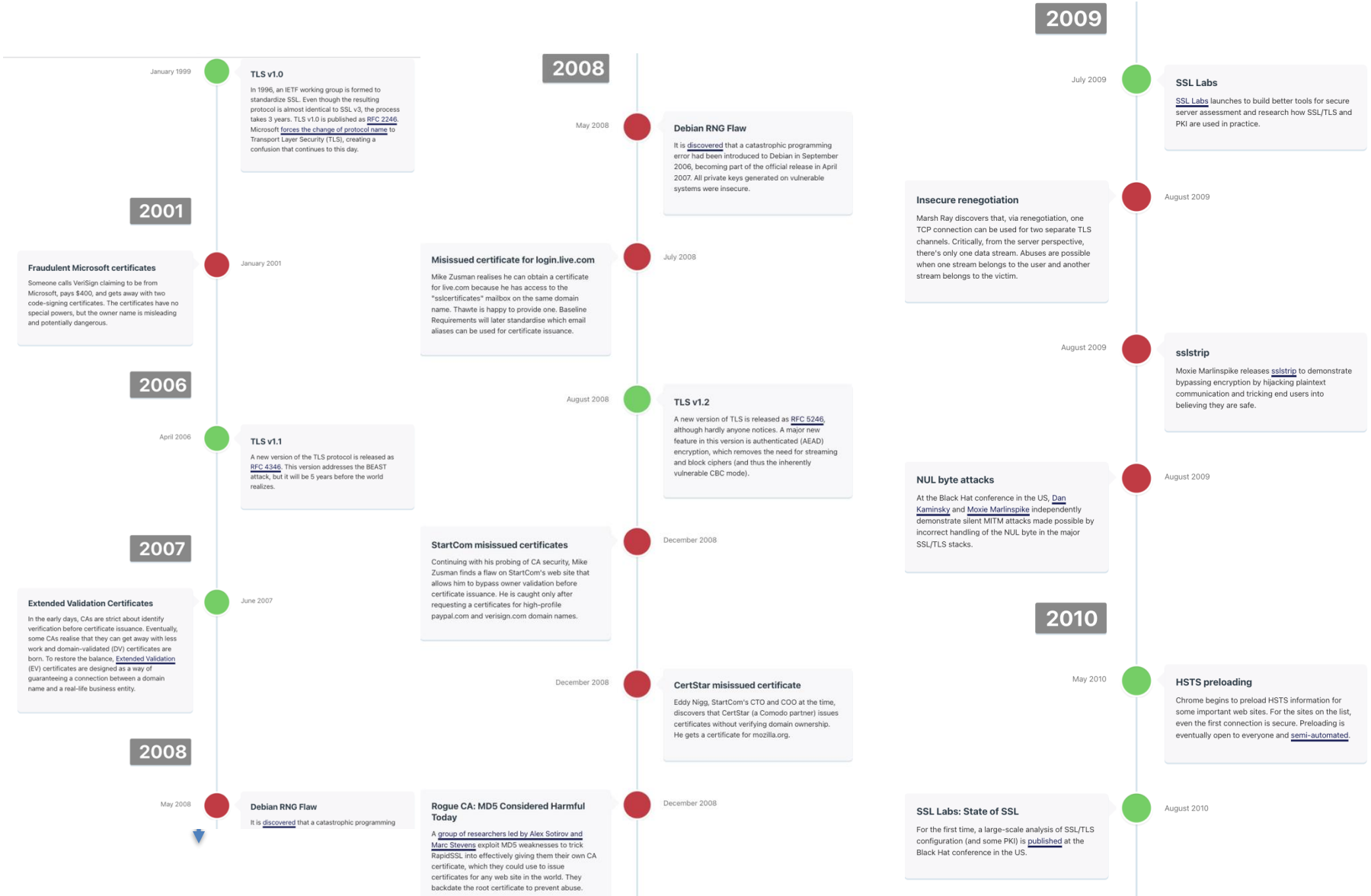
SSL is divided into two layers, with each layer using services provided by a lower layer and providing functionality to higher layers. The SSL record layer provides confidentiality, authenticity, and replay protection over a connection-oriented reliable

D. Wagner and B. Schneier, "Analysis of the SSL 3.0 protocol," *The Second USENIX Workshop on* Electronic Commerce, 1996.

History of SSL/TLS

- **1995: SSL 2.0** released, by Netscape; deprecated(prohibited) in 2011(RFC6167)
- **1996: SSL 3.0** draft 1996; RFC 6160, 2011, SSL deprecated in 2015, RFC 7568
- **1999: TLS 1.0** 1999 , RFC 2246 deprecated in 2021, RFC 8996
- **2006: TLS 1.1** 2006, RFC 4346 deprecated in 2021, RFC 8996
- **2008: TLS 1.2**, RFC 5246
- **2018: TLS 1.3**, RFC 8446, after 28 drafts

Timeline of TLS attacks and defenses



Timeline of TLS attacks and defenses

August 2010

EFF: SSL Observatory

At the same conference, the EFF launches SSL Observatory, another large-scale analysis, with more focus on PKI. These two efforts lead to an era of continuous scanning and analysis of public systems.

September 2010

SPDY

Google starts to use SPDY, a new multiplexing protocol designed to be more efficient than HTTP. Crucially, SPDY always encrypts traffic.

September 2010

False Start

Google starts to use False Start, a new technique that speeds up TLS handshake by allowing clients to send application data before the handshake is fully authenticated.

October 2010

Firesheep

Although it's always been known that plaintext communication is inherently insecure; a Firefox add-on called Firesheep captures the public's imagination after making sidejacking trivially easy. The tool is used for high-profile demonstrations, leading to many companies adopting encryption by default.

2011

March 2011

Prohibiting SSL Version 2.0

The IETF attempts to formally deprecate SSL v2 by publishing RFC 6176. According to SSL Labs, 54% HTTPS servers supported this obsolete protocol version in 2011.

March 2011

Comodo reseller breaches

Nine fraudulent certificates for seven high-profile domain names are obtained by someone calling themselves ComodoHacker. The attack was detected quickly and the certificates revoked within hours.

June 2011

BEAST

The BEAST attack is released to exploit predictable IVs in TLS 1.0. Even though this problem had been fixed in TLS 1.1 released in 2006 and TLS 1.2 in 2008, no one actually uses these newer protocol versions. Browsers will take about 2 more years to deploy TLS 1.2 and servers many years longer. We have no choice but to rely on browser counter-measures in the meantime. BEAST is the first attack to show how rich browser functionality can be abused to attack cryptography from multiple vantage points. Many similar attacks will follow.

August 2011

DigiNotar

A little-known Dutch CA spectacularly implodes after ComodoHacker gains access to all their systems and issues hundreds of fraudulent certificates. DigiNotar tries to cover up, but the incident is discovered when Chrome's public key pinning runs into a fraudulent Gmail certificate. This event leads to widespread unhappiness and attempts to design better approaches to public trust.

August 2011

Convergence

Moxie Marlinspike launches Convergence, an attempt to design a different system for public trust based on the concept of notaries. Although the project is promising, it's abandoned after browsers show little interest in it.

August 2011

Public key pinning

Google begins to use public key pinning in Chrome to verify identity of their own properties. In the following years, this mechanism successfully discovers a number of misissued certificates.

November 2011

Google deploys forward secrecy

In a major improvement, Google deploys forward secrecy, making sure private key compromises can't be used to retroactively uncover captured encrypted traffic.

2012

February 2012

Insufficient Entropy on Embedded Devices

Researchers study the quality of private keys on the public Internet and discover that about 0.5% RSA keys and 1% DSA keys can be compromised due to insufficient entropy at the time of creation. Mostly embedded devices affected.

February 2012

Chrome decides to stop checking for revocation

After many years of browsers not actually checking for certificate revocation properly, Google finally decides to get rid of the checks altogether. Other browser vendors eventually follow. The net effect is that it's now virtually impossible to revoke certificates. Although vendors develop their own proprietary revocation mechanisms, they can use them only for a small number of high-profile certificates.

March 2012

HTTP/2 call for proposals

After the success with SPDY there is a critical mass of developers interested in working on the next revision of the HTTP protocol. A period of intense debate and discussion begins.

April 2012

SSL Pulse

SSL Labs launches SSL Pulse in order to perform automated monthly scans and reporting on about 150,000 most visited web sites.

May 2012

Flame

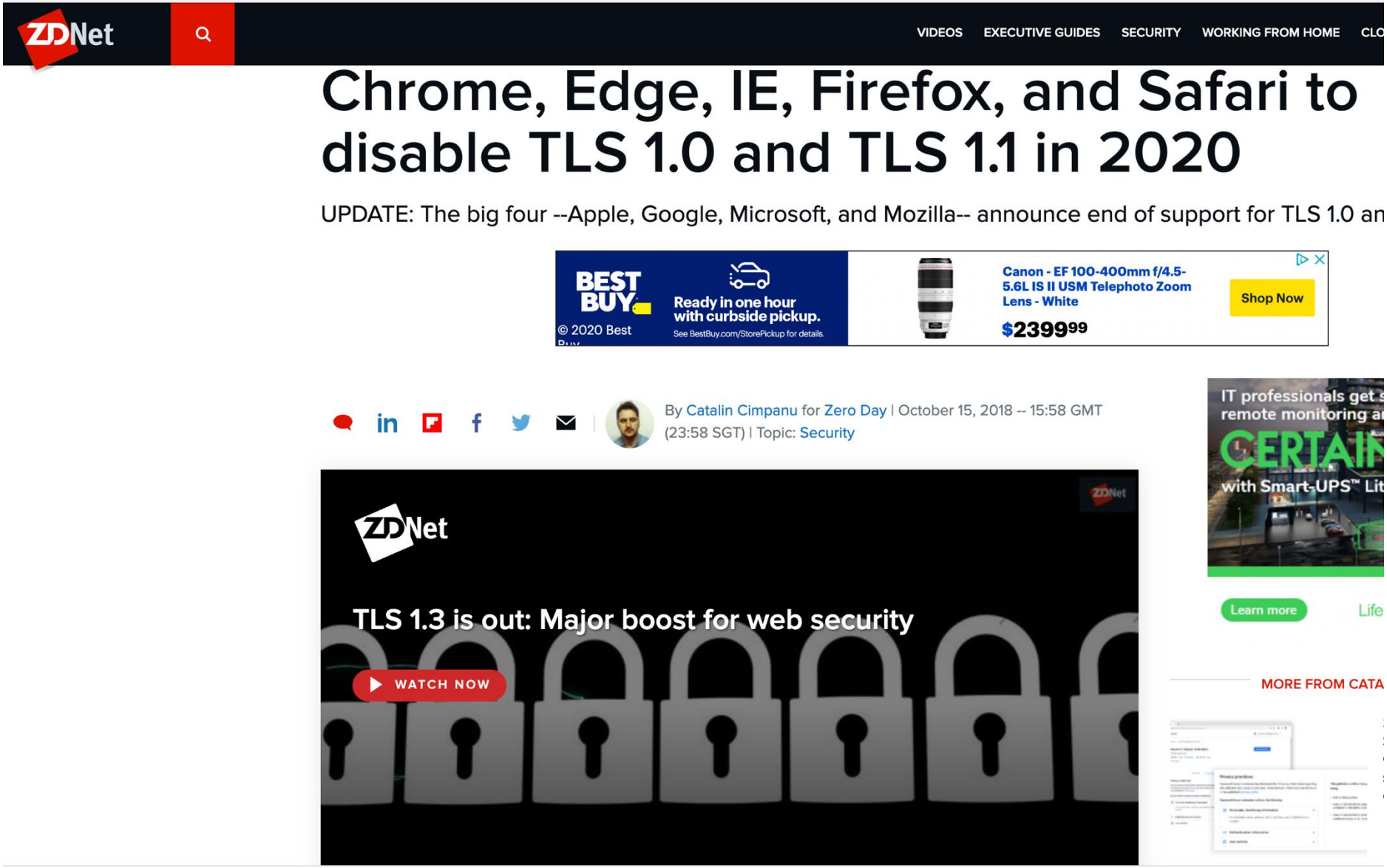
The Iranian CERT discloses the existence of Flame, a malware used in targeted cyber attacks against Iran. Flame is possibly the first cyberweapon that could have operated as early as 2007. In a stunning turn of events, it transpired that Flame exploited an MD5 collision of its own to obtain a fraudulent CA certificate.

July 2012

Baseline requirements

After their success with EV certificates, the CA/Browser Forum publishes Baseline Requirements to standardise issuance of all certificates.

TLS1.1及以下的版本2020年后禁用



Microsoft Office 365 & Apple SDK deorecate TLS 1.0 and 1.1

Microsoft | Docs Documentation Learn Q&A Code Samples

Microsoft 365 Solutions and architecture Apps and services Training Resources

Microsoft 365 / Manage information protection / Encryption / TLS 1.0 and 1.1 deprecation for Office 365

Version

Microsoft 365

Filter by title

- Email encryption
- SharePoint IRM
- Technical reference
- TLS 1.0 and 1.1 deprecation for Office 365
- TLS 1.0 and 1.1 deprecation in Office 365 GCC High and DoD
- TLS 1.2 in Office 365
- Data loss prevention (DLP)
- Manage information governance
- Manage eDiscovery
- Manage holds
- Manage auditing and alert policies
- Manage compliance risks
- Privacy management
- Hybrid compliance capabilities
- Microsoft 365 enterprise
- Microsoft 365 security
- Microsoft 365 Apps for business

Download PDF

Disabling TLS 1.0 and 1.1 for Microsoft 365

10/06/2021 • 4 minutes to read • +1 •

Applies to: Microsoft 365 Apps for enterprise, Office 365 Business, Office 365 Personal, Office Online Server, Office Web Apps

Important

We temporarily halted disablement of TLS 1.0 and 1.1 for commercial customers due to COVID-19. As supply chains have adjusted and certain countries open back up, we restarted the TLS 1.2 enforcement rollout on October 15, 2020. Rollout will continue over the following weeks and months.

As of October 31, 2018, the Transport Layer Security (TLS) 1.0 and 1.1 protocols are deprecated for the Microsoft 365 service. The effect for end-users is minimal. This change has been publicized for over two years, with the first public announcement made in December 2017. This article is only intended to cover the Office 365 local client in relation to the Office 365 service but can also apply to on-premises TLS issues with Office and Office Online Server/Office Web Apps.

For SharePoint and OneDrive, you'll need to update and configure .NET to support TLS 1.2. For information, see How to enable TLS 1.2 on clients.

Office 365 and TLS overview

查看简体中文页面 ›

Developer Discover Design Develop Distribute Support Acco

News and Updates New

SDK

TLS 1.0 and 1.1 deprecation update

September 21, 2021

Transport Layer Security (TLS) is a critical security protocol used to protect web traffic. It provides confidentiality and integrity of data in transit between clients and servers exchanging information. As part of ongoing efforts to modernize platforms, and to improve security and reliability, TLS 1.0 and 1.1 have been deprecated by the Internet Engineering Task Force (IETF) as of March 25, 2021. These versions have been deprecated on Apple platforms as of iOS 15, iPadOS 15, macOS 12, watchOS 8, and tvOS 15, and support will be removed in future releases.

If your app has enabled App Transport Security (ATS) on all connections, no changes are required. If your app continues to use legacy TLS 1.0 or 1.1, please make plans to transition to TLS 1.2 or later. We recommend supporting TLS 1.3, as it's faster and more secure. Make sure your web servers support the later versions and remove the following deprecated Security.framework symbols from your app:

- `tls_protocol_version_t.TLSv10`

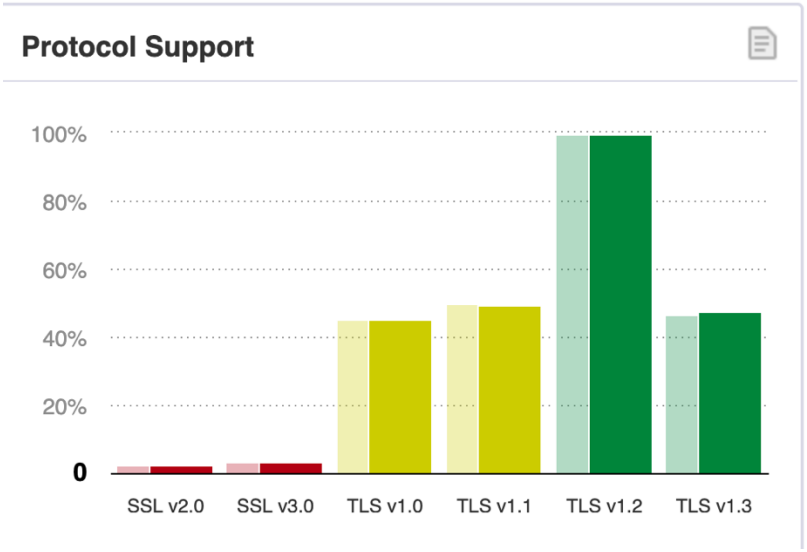
Only TLS 1.2 and 1.3 are ‘secure’

- ~~1995: SSL 2.0 released, by Netscape;~~
- ~~1996: SSL 3.0 draft 1996; RFC 6160, 2011~~
- ~~1999: TLS 1.0 1999 , RFC 2246~~
- ~~2006: TLS 1.1 2006, RFC 4346~~
- **2008: TLS 1.2, RFC 5246**
- 2011: SSL 2.0 deprecated(RFC6167)
- 2015: SSL 3.0 deprecated(RFC 7568)
- **2018: TLS 1.3, RFC 8446, after 28 drafts**
- 2021: TLS 1.0, 1.1 deprecated in RFC 8996

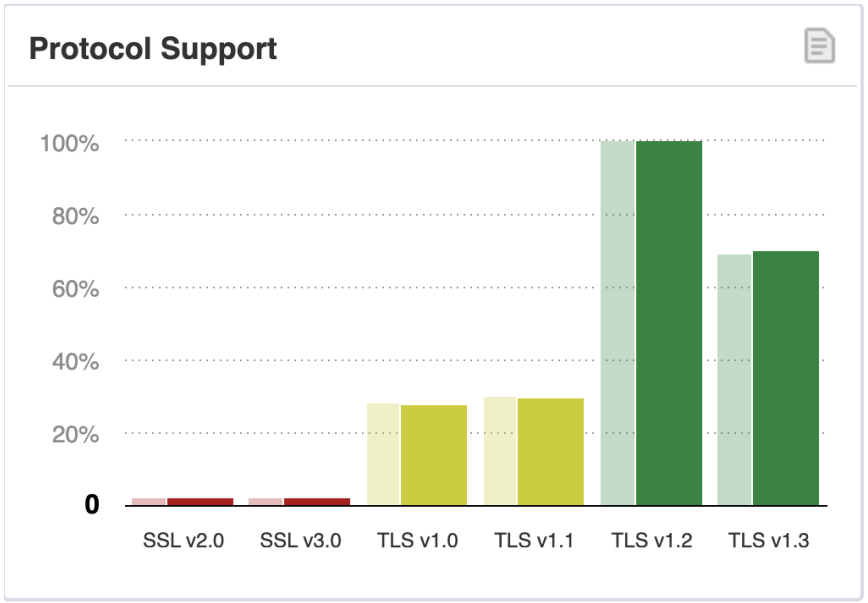
TLS 1.2 已经推出了13年了

Websites: protocols support

(by 2021/11/09)



(by 2024/11/21)



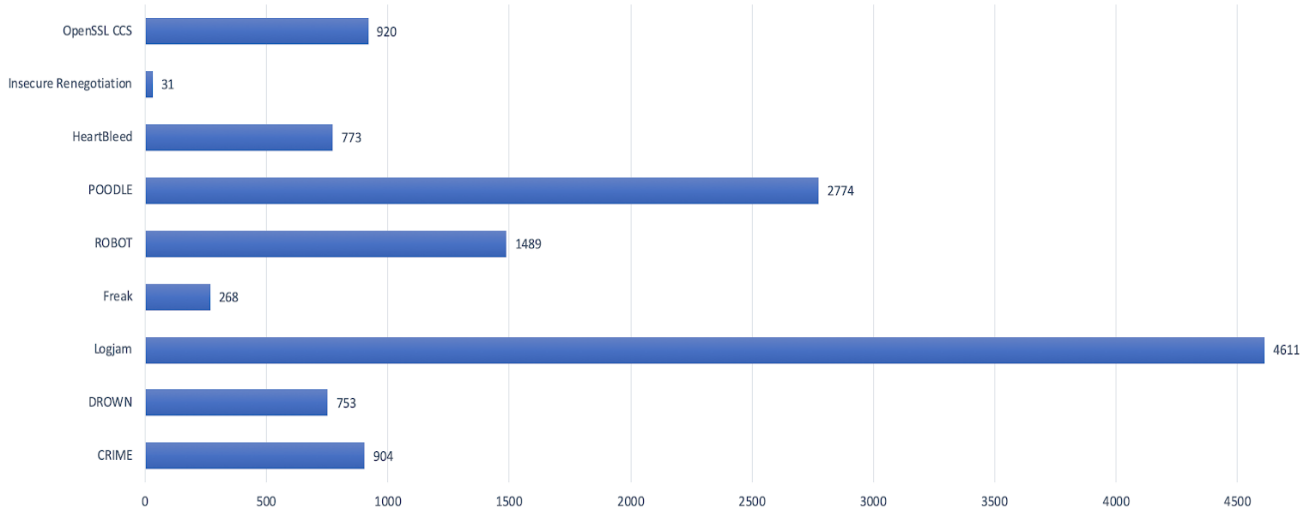
https://www.ssllabs.com/ssl-pulse/

2020年8月份中国TLS部署测量

- TLSv1.3(2018) 仍没有部署
- TLSv1.1及以下版本的84%

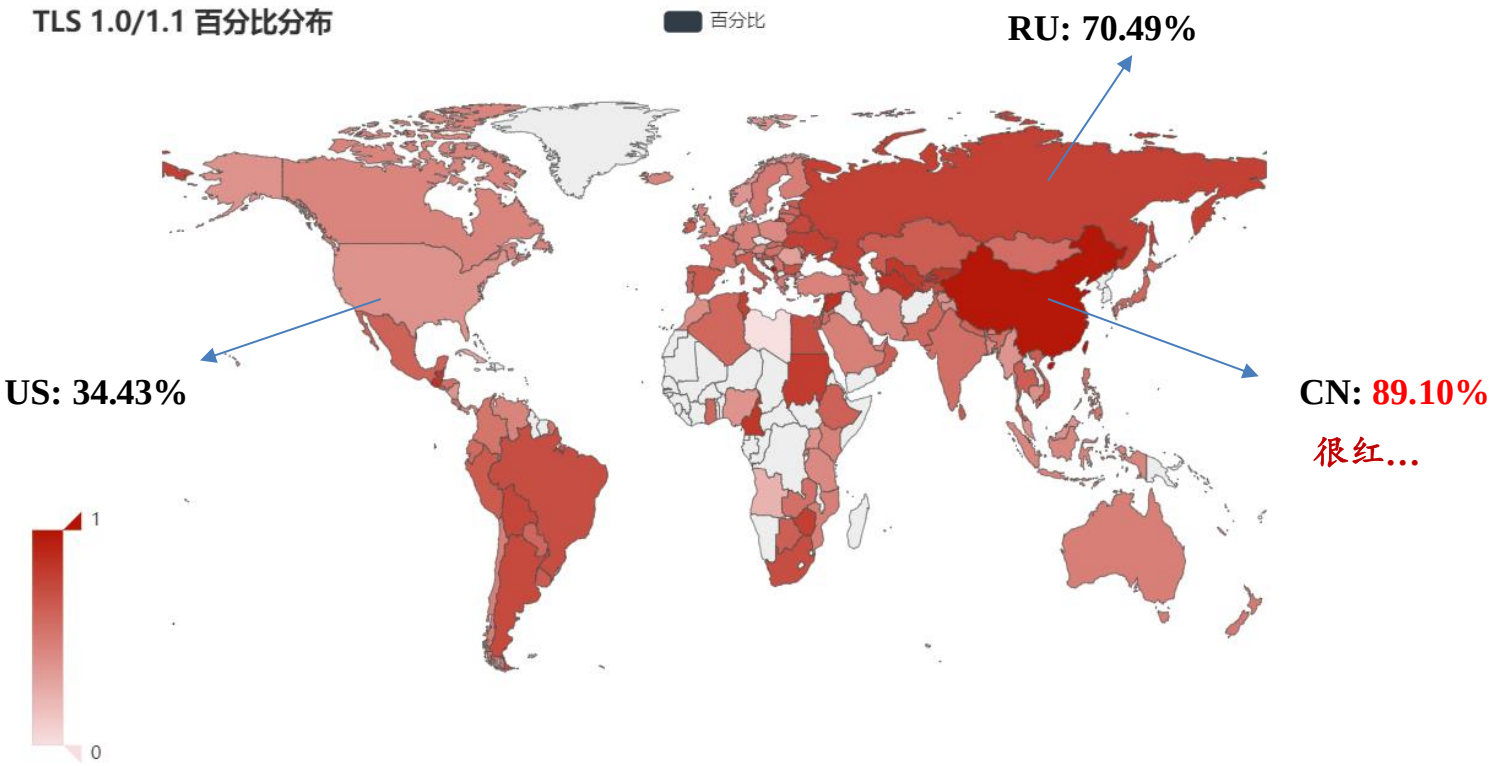
- 常见漏洞：Logjam, POODLE, CCS CRIME, Freak, HeartBleed

协议版本	比例
SSLv3	6%
TLSv1	77%
TLSv1_1	84%
TLSv1_2	94%
TLSv1_3	0%



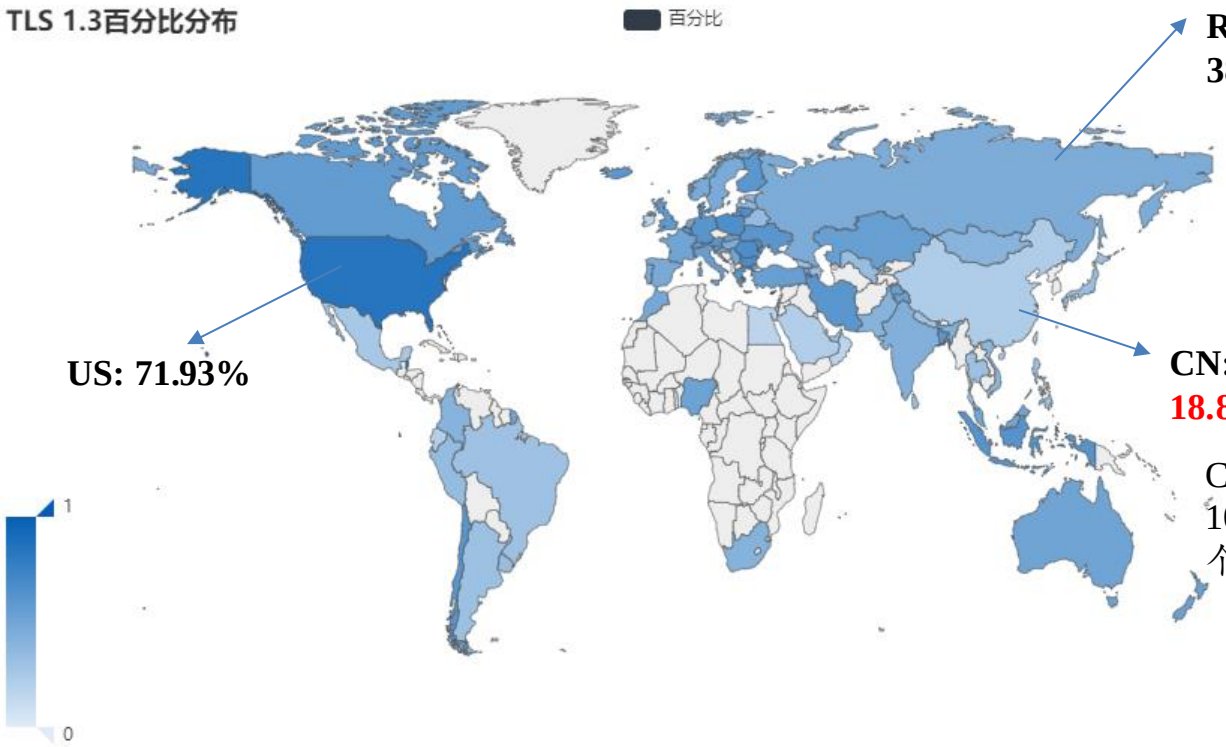
清华网研院张明明同学的测量数据

域名TOP列表中，TLS 1.0/1.1 存在情况的百分比分布



清华网研院李家琛同学的测量数据

域名TOP列表中，TLS 1.3 部署情况的百分比分布



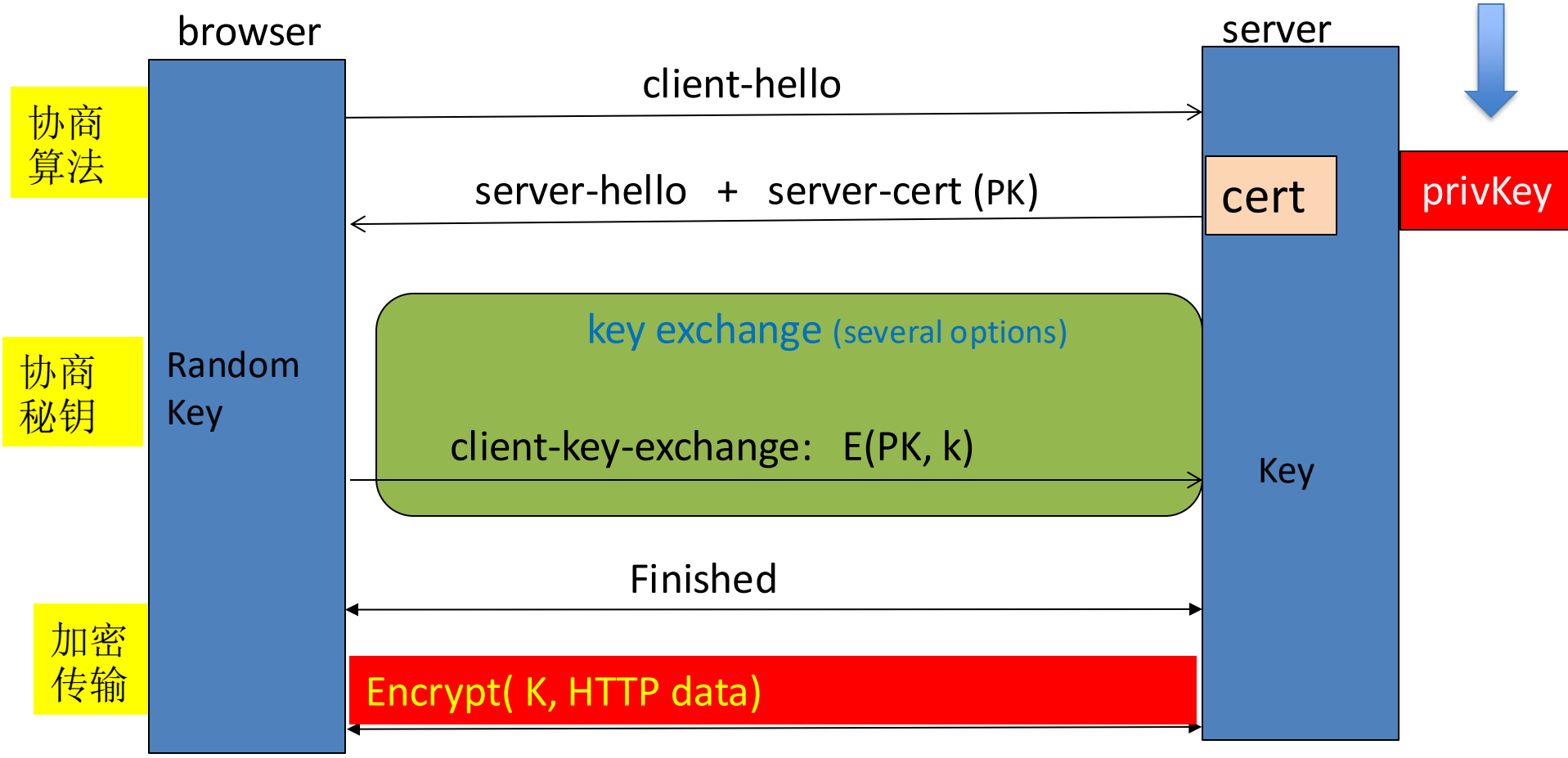
清华网研院李家琛同学的测量数据

几点思考

- 互联网的协议或标准，是经过检验的最佳实践，不是最理想的设计，也没有理论证明
- 所有标准和系统，都可能有安全漏洞，只能在攻击中逐步完善
- 很长一段时间，我们为了兼容，不得不容忍安全方面的缺陷
- 但是，向后兼容意味着降低安全。
- 协议的安全升级，需要漫长的过程

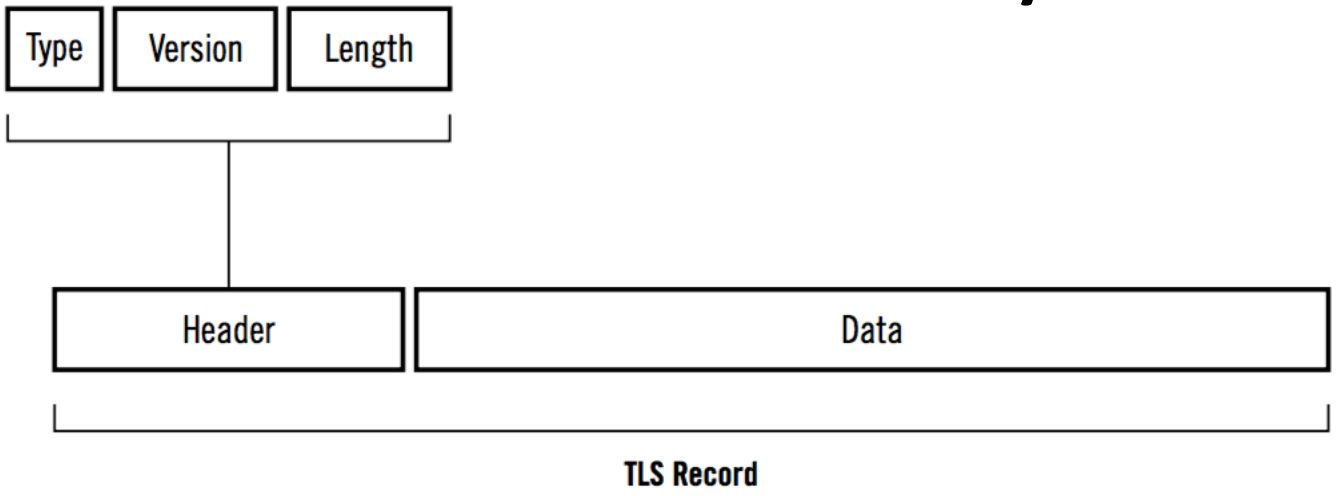
Overview of SSL/TLS

能证明服务器身份的，
是私钥而不是公钥证书



身份认证（Authentication）在哪里完成的？

Record Layer

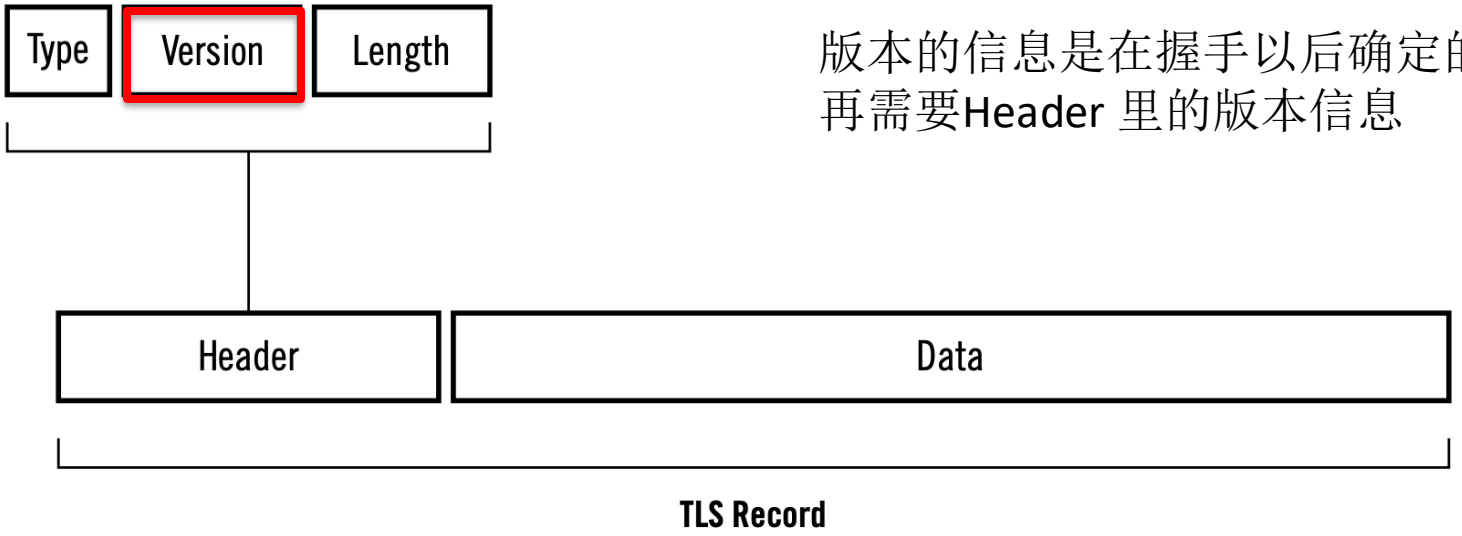



```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length; /* Maximum length is 2^14 (16,384) bytes. */
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

```
enum {
    change_cipher_spec (20),
    alert (21),
    handshake (22),
    application_data (23)
} ContentType;
```

```
struct {
    uint8 major;
    uint8 minor;
} ProtocolVersion;
```

TLS 1.3

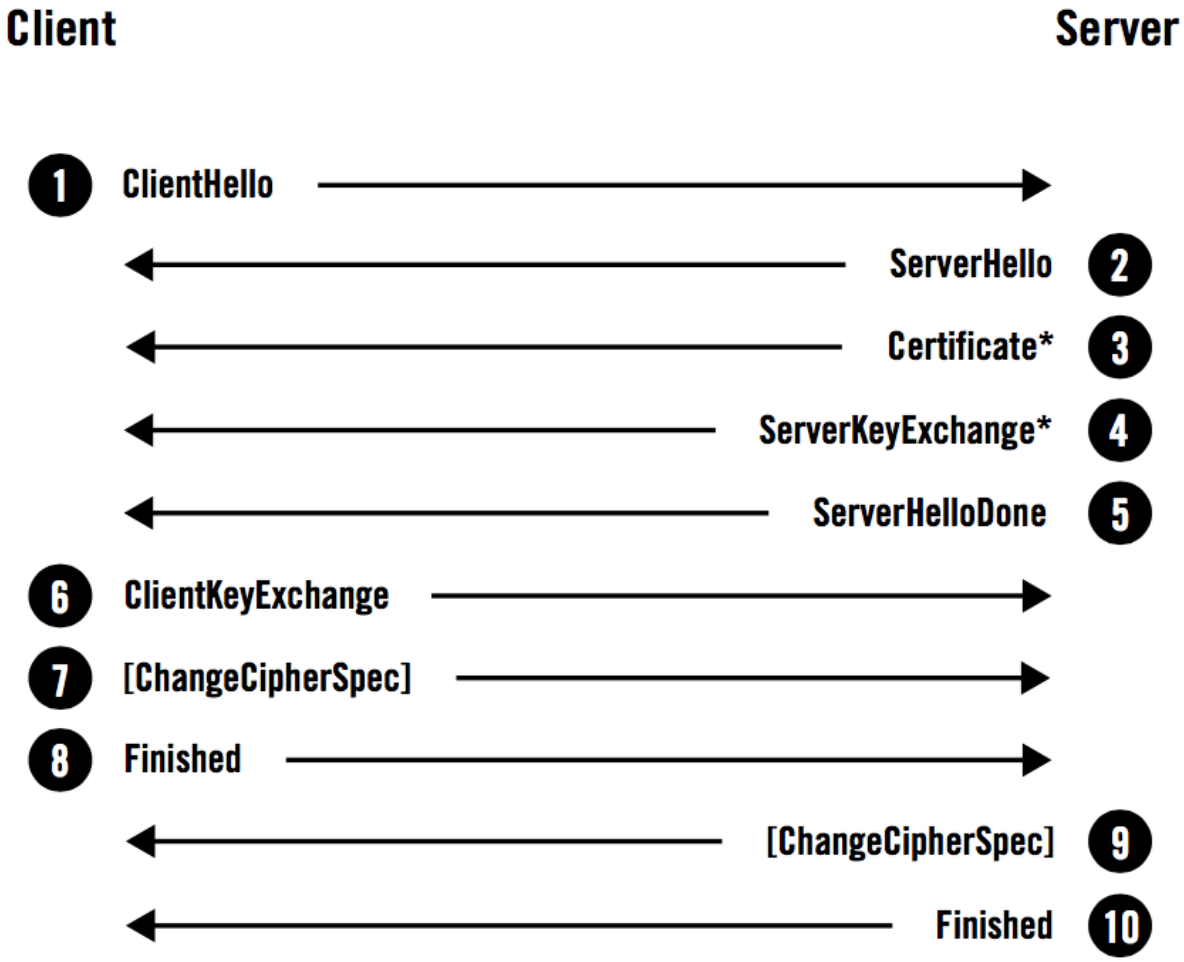


版本的信息是在握手以后确定的，TLS1.3 不再需要Header 里的版本信息

```
struct {
    ContentType type;
    ProtocolVersion legacy_record_version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

```
enum {
    invalid(0),
    change_cipher_spec(20),
    alert(21),
    handshake(22),
    application_data(23),
    (255)
} ContentType;
```

完整的SSL握手认证过程



* Optional message

[ ] ChangeCipherSpec protocol message

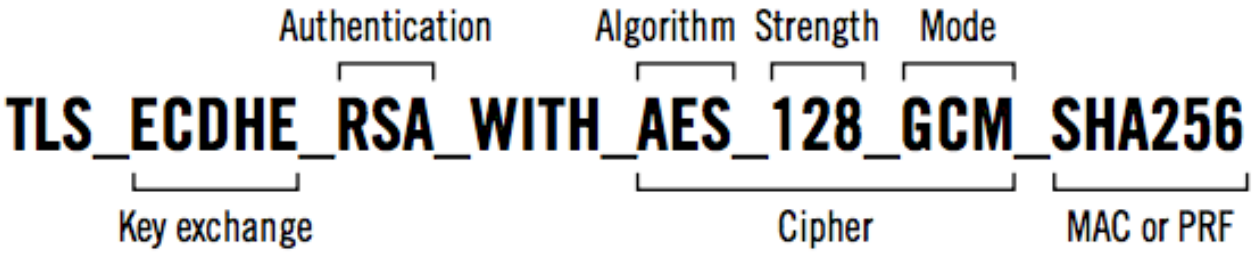
```
Handshake protocol: ClientHello
    Version: TLS 1.2
    Random
        Client time: May 22, 2030 02:43:46 GMT
        Random bytes: b76b0e61829557eb4c611adfd2d36eb232dc1332fe29802e321ee871
    Session ID: (empty)
    Cipher Suites
        Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
        Suite: TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
        Suite: TLS_RSA_WITH_AES_128_GCM_SHA256
        Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
        Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA
        Suite: TLS_RSA_WITH_AES_128_CBC_SHA
        Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA
        Suite: TLS_RSA_WITH_RC4_128_SHA
    Compression methods
        Method: null
    Extensions
        Extension: server_name
            Hostname: www.feistyduck.com
        Extension: renegotiation_info
        Extension: elliptic_curves
            Named curve: secp256r1
            Named curve: secp384r1
        Extension: signature_algorithms
            Algorithm: sha1/rsa
            Algorithm: sha256/rsa
            Algorithm: sha1/ecdsa
            Algorithm: sha256/ecdsa
```

随机数，用于防止重放(replay)攻击。

每个Session 协商一组密码套件，由ID 标识。

Server 保存一个session id cache， 记录每个session 的状态和参数

```
Handshake protocol: ClientHello
    Version: TLS 1.2
    Random
        Client time: May 22, 2030 02:43:46 GMT
        Random bytes: b76b0e61829557eb4c611adfd2d36eb232dc1332fe29802e321ee871
    Session ID: (empty)
    Cipher Suites
```




```
    Compression methods
        Method: null
    Extensions
        Extension: server_name
            Hostname: www.feistyduck.com
        Extension: renegotiation_info
        Extension: elliptic_curves
            Named curve: secp256r1
            Named curve: secp384r1
        Extension: signature_algorithms
            Algorithm: sha1/rsa
            Algorithm: sha256/rsa
            Algorithm: sha1/ecdsa
            Algorithm: sha256/ecdsa
```

客户端所支持的密码算法：

- 密钥交换算法：ECDHE
- 认证算法： RSA
- 加密算法： AES
- 强度（密钥长度）：128
- 加密模式：GCM
- 消息认证及伪随机数：SHA256

```
Handshake protocol: ClientHello
    Version: TLS 1.2
    Random
        Client time: May 22, 2030 02:43:46 GMT
        Random bytes: b76b0e61829557eb4c611adfd2d36eb232dc1332fe29802e321ee871
    Session ID: (empty)
    Cipher Suites
        Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
        Suite: TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
        Suite: TLS_RSA_WITH_AES_128_GCM_SHA256
        Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
        Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA
        Suite: TLS_RSA_WITH_AES_128_CBC_SHA
        Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA
        Suite: TLS_RSA_WITH_RC4_128_SHA
    Compression methods
        Method: null
    Extensions
        Extension: server_name
            Hostname: www.feistyduck.com
        Extension: renegotiation_info
        Extension: elliptic_curves
            Named curve: secp256r1
            Named curve: secp384r1
        Extension: signature_algorithms
            Algorithm: sha1/rsa
            Algorithm: sha256/rsa
            Algorithm: sha1/ecdsa
            Algorithm: sha256/ecdsa
```

应用层数据压缩方法，因导致侧信道攻击（如CRIME），弃用

SNI(Server Name Indication)

```
Handshake protocol: ClientHello
    Version: TLS 1.2
    Random
        Client time: May 22, 2030 02:43:46 GMT
        Random bytes: b76b0e61829557eb4c611adfd2d36eb232dc1332fe29802e321ee871
    Session ID: (empty)
    Cipher Suites
        Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
        Suite: TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
        Suite: TLS_RSA_WITH_AES_128_GCM_SHA256
        Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
        Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA
        Suite: TLS_RSA_WITH_AES_128_CBC_SHA
        Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA
        Suite: TLS_RSA_WITH_RC4_128_SHA
    Compression methods
        Method: null
    Extensions
        Extension: server_name
            Hostname: www.feistyduck.com
        Extension: renegotiation_info
        Extension: elliptic_curves
            Named curve: secp256r1
            Named curve: secp384r1
        Extension: signature_algorithms
            Algorithm: sha1/rsa
            Algorithm: sha256/rsa
            Algorithm: sha1/ecdsa
            Algorithm: sha256/ecdsa
```

TLS1.1 之后， Tag-length-value, 扩展协议， 可传送任意数据

SNI(Server Name Indication, 虚拟主机/CDN)

```
Handshake protocol: ClientHello
    Version: TLS 1.2
    Random
        Client time: May 22, 2030 02:43:46 GMT
        Random bytes: b76b0e61829557eb4c611adfd2d36eb232dc1332fe29802e321ee871
    Session ID: (empty)
    Cipher Suites
        Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
        Suite: TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
        Suite: TLS_RSA_WITH_AES_128_GCM_SHA256
        Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
        Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA
        Suite: TLS_RSA_WITH_AES_128_CBC_SHA
        Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA
        Suite: TLS_RSA_WITH_RC4_128_SHA
    Compression methods
        Method: null
    Extensions
        Extension: server_name
            Hostname: www.feistyduck.com
        Extension: renegotiation_info
        Extension: elliptic_curves
            Named curve: secp256r1
            Named curve: secp384r1
        Extension: signature_algorithms
            Algorithm: sha1/rsa
            Algorithm: sha256/rsa
            Algorithm: sha1/ecdsa
            Algorithm: sha256/ecdsa
```

虚拟主机/CDN环境中，指示访问哪一个主机（域名）

```
Handshake protocol: ClientHello
    Version: TLS 1.2
    Random
        Client time: May 22, 2030 02:43:46 GMT
        Random bytes: b76b0e61829557eb4c611adfd2d36eb232dc1332fe29802e321ee871
    Session ID: (empty)
    Cipher Suites
        Suite: TLS_ECDHE_RSA_WITH_AES_12
        Suite: TLS_DHE_RSA_WITH_AES_128_
        Suite: TLS_RSA_WITH_AES_128_GCM_
        Suite: TLS_ECDHE_RSA_WITH_AES_12
        Suite: TLS_DHE_RSA_WITH_AES_128_
        Suite: TLS_RSA_WITH_AES_128_CBC_
        Suite: TLS_RSA_WITH_3DES_EDE_CBC
        Suite: TLS_RSA_WITH_RC4_128_SHA
    Compression methods
        Method: null
    Extensions
        Extension: server_name
            Hostname: www.feistyduck.com
        Extension: renegotiation_info
        Extension: elliptic_curves
            Named curve: secp256r1
            Named curve: secp384r1
        Extension: signature_algorithms
            Algorithm: sha1/rsa
            Algorithm: sha256/rsa
            Algorithm: sha1/ecdsa
            Algorithm: sha256/ecdsa
```

2.7.1 Recommended Parameters secp256k1

The elliptic curve domain parameters over $\mathbb{F}_p$ associated with a Koblitz curve `secp256k1` are specified by the sextuple $T = (p, a, b, G, n, h)$ where the finite field $\mathbb{F}_p$ is defined by:

$$p = \texttt{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F}$$
$$= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$$

The curve E: $y^2 = x^3 + ax + b$ over $\mathbb{F}_p$ is defined by:

$$a = \texttt{00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000}$$
$$b = \texttt{00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007}$$

The base point G in compressed form is:

$$G = \texttt{02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798}$$

and in uncompressed form is:

$$G = \texttt{04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8}$$

Finally the order n of G and the cofactor are:

$$n = \texttt{FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141}$$
$$h = \texttt{01}$$

Server Hello

```
Handshake protocol: ServerHello
    Version: TLS 1.2
    Random
        Server time: Mar 10, 2059 02:35:57 GMT
        Random bytes: 8469b09b480c1978182ce1b59290487609f41132312ca22aacaf5012
    Session ID: 4cae75c91cf5adf55f93c9fb5dd36d19903b1182029af3d527b7a42ef1c32c80
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
    Compression method: null
    Extensions
        Extension: server_name
        Extension: renegotiation_info
```

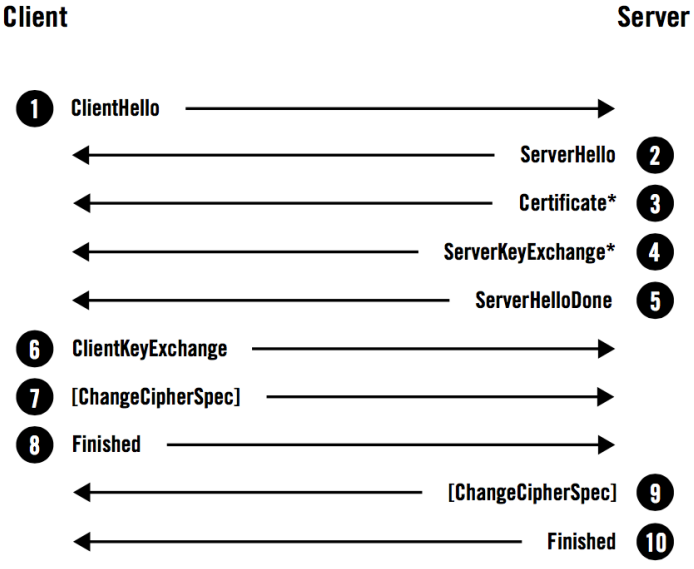
3-in-one

Certificate/Server Key Exchange/Server Hello Done

No.	Time	Source	Destination	Protocol	Length	Info
4	0.003331	10.0.1.6	202.112.57.31	TLSv1.2	249	Client Hello
5	0.005366	202.112.57.31	10.0.1.6	TCP	66	443→54525 [ACK] Seq=1 Ack=184 Win=15616 Len=0 TSval=3355045
6	0.014950	202.112.57.31	10.0.1.6	TLSv1.2	1514	Server Hello
7	0.015391	202.112.57.31	10.0.1.6	TLSv1.2	893	Certificate
8	0.015452	10.0.1.6	202.112.57.31	TCP	66	54525→443 [ACK] Seq=184 Ack=2276 Win=130240 Len=0 TSval=475
9	1.920475	10.0.1.6	202.112.57.31	TLSv1.2	141	Client Key Exchange

```
▷ Frame 7: 893 bytes on wire (7144 bits), 893 bytes captured (7144 bits)
▷ Ethernet II, Src: Apple_56:6a:53 (e8:8d:28:56:6a:53), Dst: Apple_d1:eb:d2 (34:36:3b:d1:eb:d2)
▷ Internet Protocol Version 4, Src: 202.112.57.31 (202.112.57.31), Dst: 10.0.1.6 (10.0.1.6)
▷ Transmission Control Protocol, Src Port: 443 (443), Dst Port: 54525 (54525), Seq: 1449, Ack: 184, Len: 827
▽ [2 Reassembled TCP Segments (1834 bytes): #6(1354), #7(480)]
    [Frame: 6, payload: 0-1353 (1354 bytes)]
    [Frame: 7, payload: 1354-1833 (480 bytes)]
    [Segment count: 2]
    [Reassembled TCP length: 1834]
    [Reassembled TCP Data: 16030307250b00072100071e00071b30820717308205ffa0...]
▽ Secure Sockets Layer
  ▷ TLSv1.2 Record Layer: Handshake Protocol: Certificate
▽ Secure Sockets Layer
  ▷ TLSv1.2 Record Layer: Handshake Protocol: Server Key Exchange
  ▽ TLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 4
    ▽ Handshake Protocol: Server Hello Done
        Handshake Type: Server Hello Done (14)
        Length: 0
```

```
00d0  30 37 32 35 32 37 5a 30  24 31 0b 30 09 06 03 55   072527Z0 $1.0...U
00e0  04 06 13 02 43 4e 31 15  30 13 06 03 55 04 03 0c   ....CN1. 0...U...
00f0  0c 69 6e 66 6f 72 73 65  63 2e 6f 72 67 30 82 01   .inforse c.org0..
0100  22 30 0d 06 09 2a 86 48  86 f7 0d 01 01 01 05 00   "0...*.H ........
0110  03 82 01 0f 00 30 82 01  0a 02 82 01 01 00 fa 78   .....0.. .......x
0120  c3 22 9e fc f1 07 e2 80  7f 67 33 b9 e4 fa f0 66   ."...... .g3....f
0130  37 09 99 0c 6f 2f 20 4a  69 d3 e6 57 89 51 da b3   7...o/ J i..W.Q..
0140  23 5e 5b 8e 9c 7e 96 ac  c9 2c 54 f6 2b e1 22 18   #^[..~.. .,T.+."
```



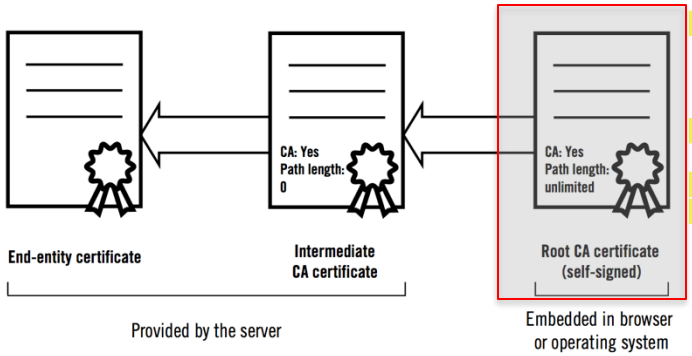
* Optional message

[ ] ChangeCipherSpec protocol message

CertificateS, chain without root

```
    4 0.003331    10.0.1.6          202.112.57.31     TLSv1.2    249 Client Hello
    5 0.005366    202.112.57.31     10.0.1.6          TCP         66 443→54525 [ACK] Seq=1 Ack=184 Win=15616 Len=0 TSval=3355045
    6 0.014950    202.112.57.31     10.0.1.6          TLSv1.2   1514 Server Hello
    7 0.015391    202.112.57.31     10.0.1.6          TLSv1.2    893 Certificate
    8 0.015452    10.0.1.6          202.112.57.31     TCP         66 54525→443 [ACK] Seq=184 Ack=2276 Win=130240 Len=0 TSval=475
    9 1.920475    10.0.1.6          202.112.57.31     TLSv1.2    141 Client Key Exchange
```

```
▽ TLSv1.2 Record Layer: Handshake Protocol: Certificate
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 1829
  ▽ Handshake Protocol: Certificate
      Handshake Type: Certificate (11)
      Length: 1825
      Certificates Length: 1822
    ▽ Certificates (1822 bytes)
        Certificate Length: 1819
      ▽ Certificate (id-at-commonName=inforsec.org,id-at-countryName=CN)
        ▽ signedCertificate
            version: v3 (2)
            serialNumber : 0x31ffaadf4c01bb6094fd9501c7d58065
          ▽ signature (sha256WithRSAEncryption)
              Algorithm Id: 1.2.840.113549.1.1.11 (sha256WithRSAEncryption)
          ▷ issuer: rdnSequence (0)
          ▷ validity
          ▽ subject: rdnSequence (0)
            ▽ rdnSequence: 2 items (id-at-commonName=inforsec.org,id-at-countryName=CN)
              ▽ RDNSequence item: 1 item (id-at-countryName=CN)
                ▽ RelativeDistinguishedName item (id-at-countryName=CN)
                    Id: 2.5.4.6 (id-at-countryName)
                    CountryName: CN
              ▽ RDNSequence item: 1 item (id-at-commonName=inforsec.org)
                ▽ RelativeDistinguishedName item (id-at-commonName=inforsec.org)
```




```
00d0  30 37 32 35 32 37 5a 30  24 31 0b 30 09 06 03 55   072527Z0 $1.0...U
00e0  04 06 13 02 43 4e 31 15  30 13 06 03 55 04 03 0c   ....CN1. 0...U...
00f0  0c 69 6e 66 6f 72 73 65  63 2e 6f 72 67 30 82 01   .inforse c.org0..
0100  22 30 0d 06 09 2a 86 48  86 f7 0d 01 01 01 05 00   "0...*.H ........
0110  03 82 01 0f 00 30 82 01  0a 02 82 01 01 00 fa 78   .....0.. .......x
0120  c3 22 9e fc f1 07 e2 80  7f 67 33 b9 e4 fa f0 66   ."...... .g3....f
0130  37 09 99 0c 6f 2f 20 4a  69 d3 e6 57 89 51 da b3   7...o/ J i..W.Q..
0140  23 5e 5b 8e 9c 7e 96 ac  c9 2c 54 f6 2b e1 22 18   #^[..~.. .,T.+.".
```

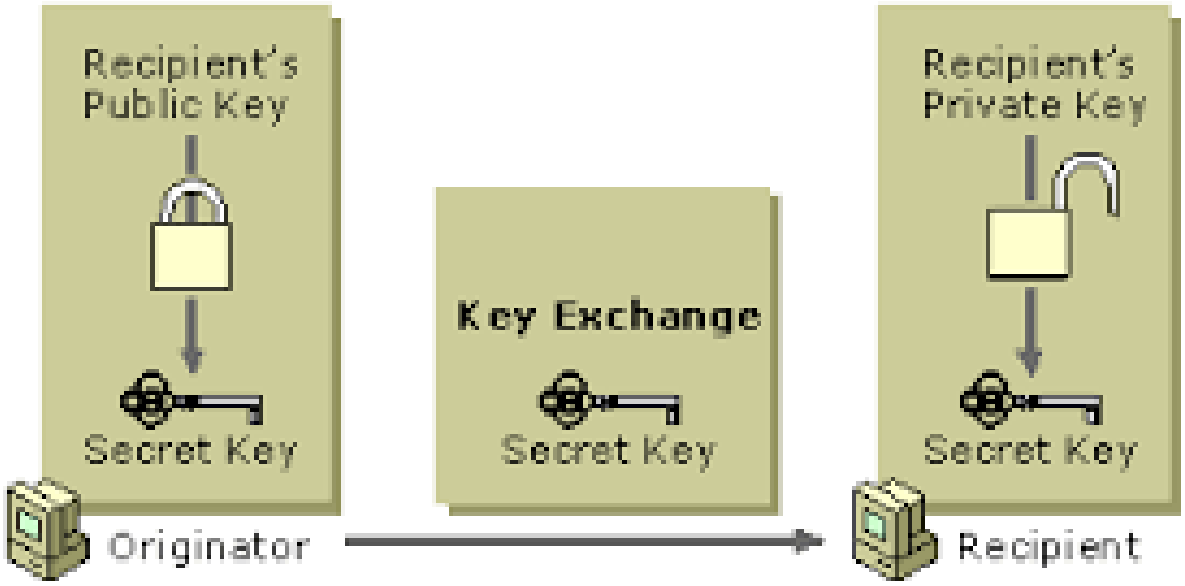
Server Key Exchange

Key Exchange	Description
dh_anon	Diffie-Hellman (DH) key exchange without authentication
dhe_rsa	Ephemeral DH key exchange with RSA authentication
ecdh_anon	Ephemeral Elliptic Curve DH (ECDH) key exchange without authentication (RFC 4492)
ecdhe_rsa	Ephemeral ECDH key exchange with RSA authentication (RFC 4492)
ecdhe_ecdsa	Ephemeral ECDH key exchange with ECDSA authentication (RFC 4492)
krb5	Kerberos key exchange (RFC 2712)
rsa	RSA key exchange and authentication
psk	Pre-Shared Key (PSK) key exchange and authentication (RFC 4279)
dhe_psk	Ephemeral DH key exchange with PSK authentication (RFC 4279)
rsa_psk	PSK key exchange and RSA authentication (RFC 4279)
srp	Secure Remote Password (SRP) key exchange and authentication (RFC 5054)

Key Exchange with RSA

Client encrypt premaster_secret (48 bytes RN) with public key of the server

Server decrypt the message and get the premaster_secret



Structs used by KeyExchange: RSA

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa:
            EncryptedPreMasterSecret;
        case dhe_dss:
        case dhe_rsa:
        case dh_dss:
        case dh_rsa:
        case dh_anon:
            ClientDiffieHellmanPublic;
        case ecdhe:
            ClientECDiffieHellmanPublic;
    } exchange_keys;
} ClientKeyExchange;
```

```
struct {
    select (KeyExchangeAlgorithm) {
        case dh_anon:
            ServerDHParams     params;
        case dhe_rsa:
            ServerDHParams     params;
            Signature          params_signature;
        case ecdh_anon:
            ServerECDHParams   params;
        case ecdhe_rsa:
        case ecdhe_ecdsa:
            ServerECDHParams   params;
            Signature          params_signature;
        case rsa:
        case dh_rsa:
            /* no message */
    };
} ServerKeyExchange;
```

Perfect Forward Secrecy(前向安全)

- A property of secure communication protocols in which compromise of long-term keys does not compromise **past session keys**
- Without Forward Secrecy, attacker can capture all encrypted traffic for **offline analysis later**
- If he can get private key later, he can decrypt all traffic, by decryption of secret key
- RSA is vulnerable to such attack, so it does NOT support forward secrecy

BIZ & IT —

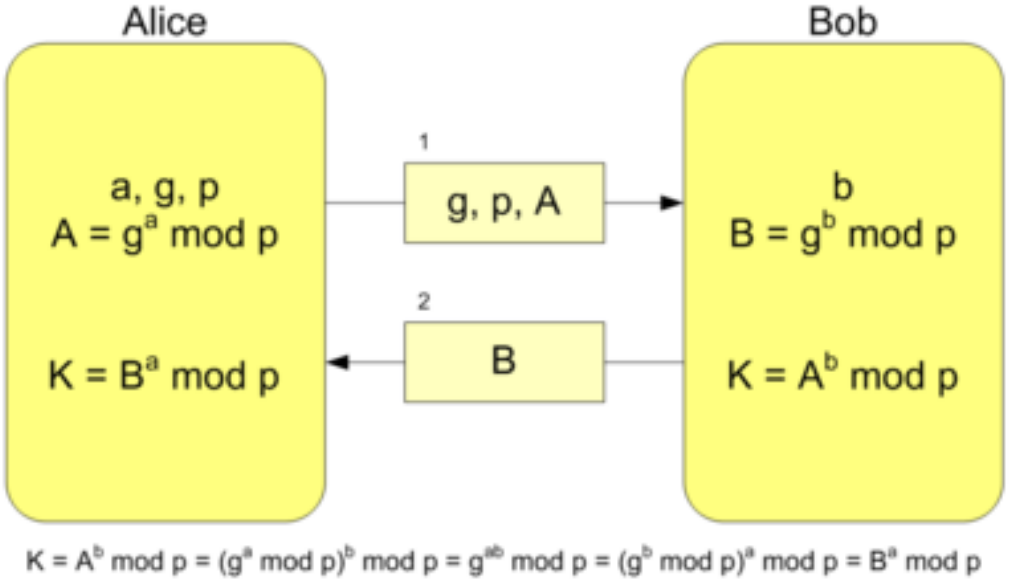
How the NSA can break trillions of encrypted Web and VPN connections

Researchers show how mass decryption is well within the NSA's $11 billion budget.

DAN GOODIN - 10/16/2015, 12:42 AM

Key Exchange with DH(Diffie-Hellman)

- Alice's private Key: a， Bob : b
- Attacker cannot calculate K with g, p, A and B



Structs used by KeyExchange: DH

```
struct {
    select (KeyExchangeAlgorithm) {
        case dh_anon:
            ServerDHParams      params;
        case dhe_rsa:
            ServerDHParams      params;
            Signature           params_signature;
        case ecdh_anon:
            ServerECDHParams    params;
        case ecdhe_rsa:
        case ecdhe_ecdsa:
            ServerECDHParams    params;
            Signature           params_signature;
        case rsa:
        case dh_rsa:
            /* no message */
    };
} ServerKeyExchange;
```

```
struct {
    opaque dh_p;
    opaque dh_g;
    opaque dh_Ys;
} ServerDHParams;
```

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa:
            EncryptedPreMasterSecret;
        case dhe_dss:
        case dhe_rsa:
        case dh_dss:
        case dh_rsa:
        case dh_anon:
            ClientDiffieHellmanPublic;
        case ecdhe:
            ClientECDiffieHellmanPublic;
    } exchange_keys;
} ClientKeyExchange;
```

```
struct {
    select (PublicValueEncoding) {
        case implicit:
            /* empty; used when the client's public
               parameter is embedded in its certificate */
        case explicit:
            opaque dh_Yc;
    } dh_public;
} ClientDiffieHellmanPublic;
```

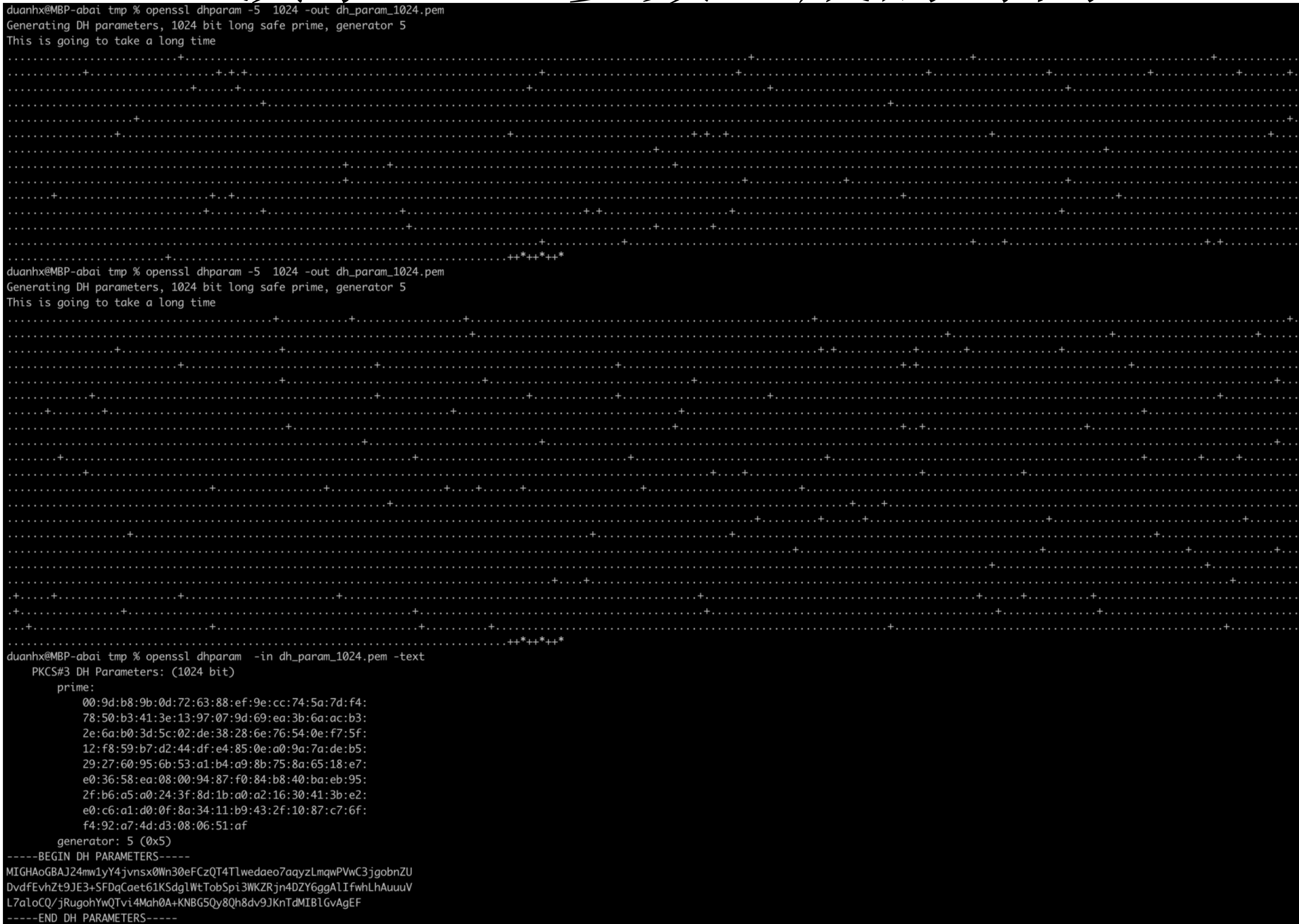
DHE(Ephemeral DH) and Forward Secrecy

- DH is static, in which p and g will not change
  - p, g is static; server will store priv_key (g^X mod p)
- Ephemeral Diffie-Hellman(DHE) supports forward secrecy
- DHE
  - None of the parameters will be re-used
  - Test:

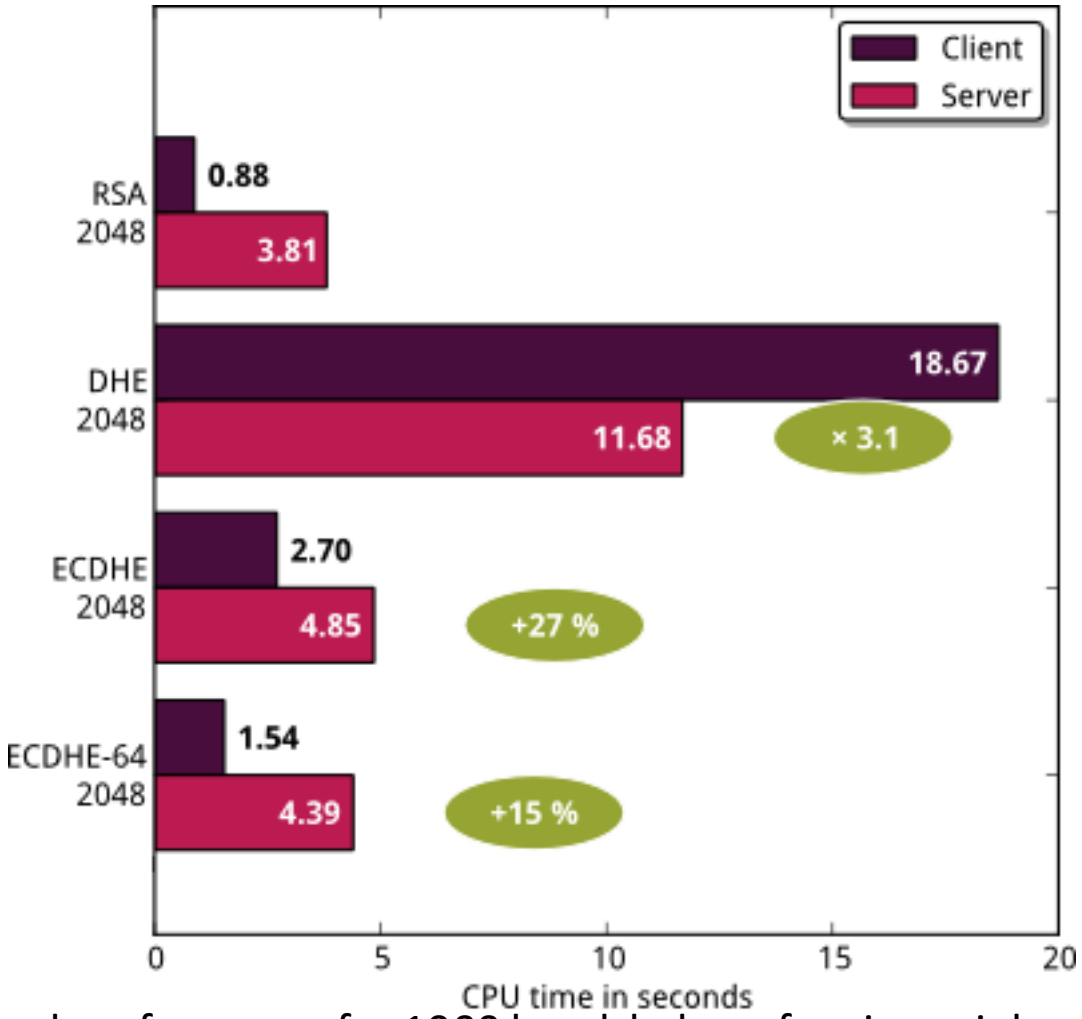
    $ openssl dhparam -5 1024 -out dh_param_1024.pem

    $ openssl dhparam -in dh_param_1024.pem -text

演示：DH参数生成的时间



Performance



Compared performance for 1000 handshakes of various cipher suites (RSA 2048, DHE, ECDHE, optimized ECDHE)

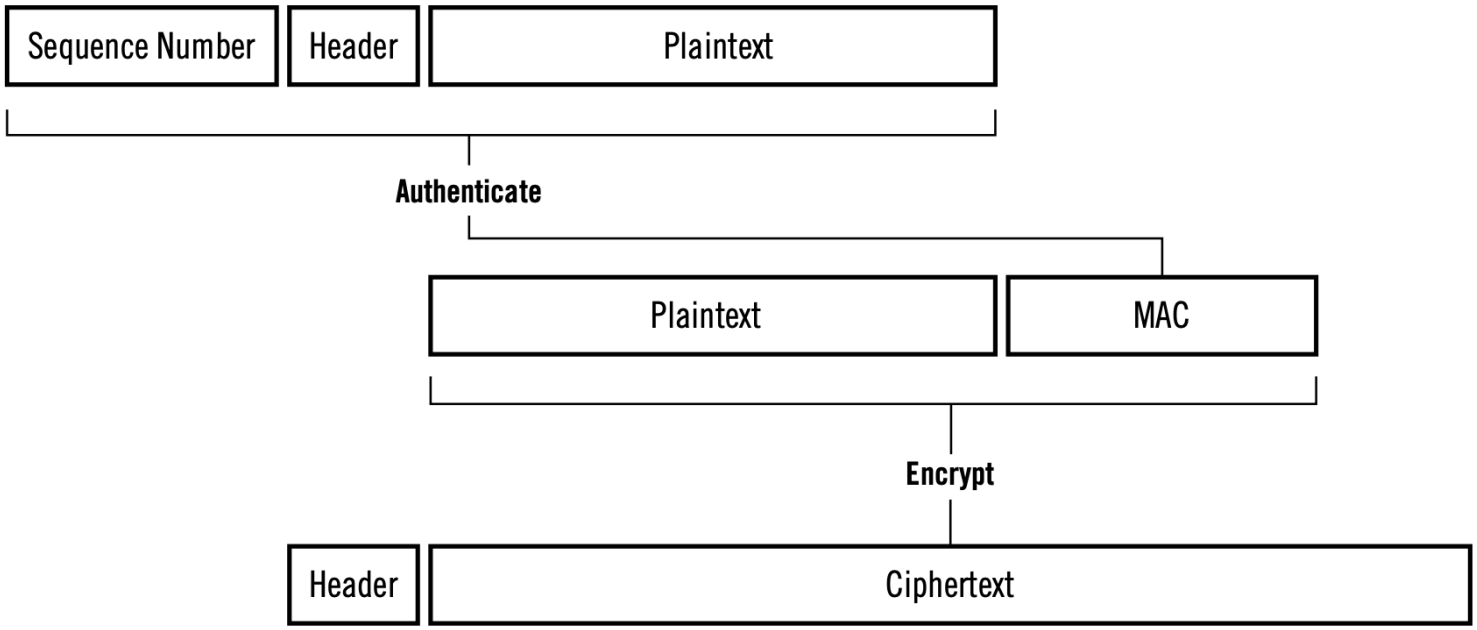
http://vincent.bernat.im/en/blog/2011-ssl-perfect-forward-secrecy.html

Authentication is implicit

- RSA key exchange:
  - the client generates a random value as the premaster secret and sends it encrypted with the server's public key.
- DHE and ECDHE exchanges
  - The parameters are signed with its private key.

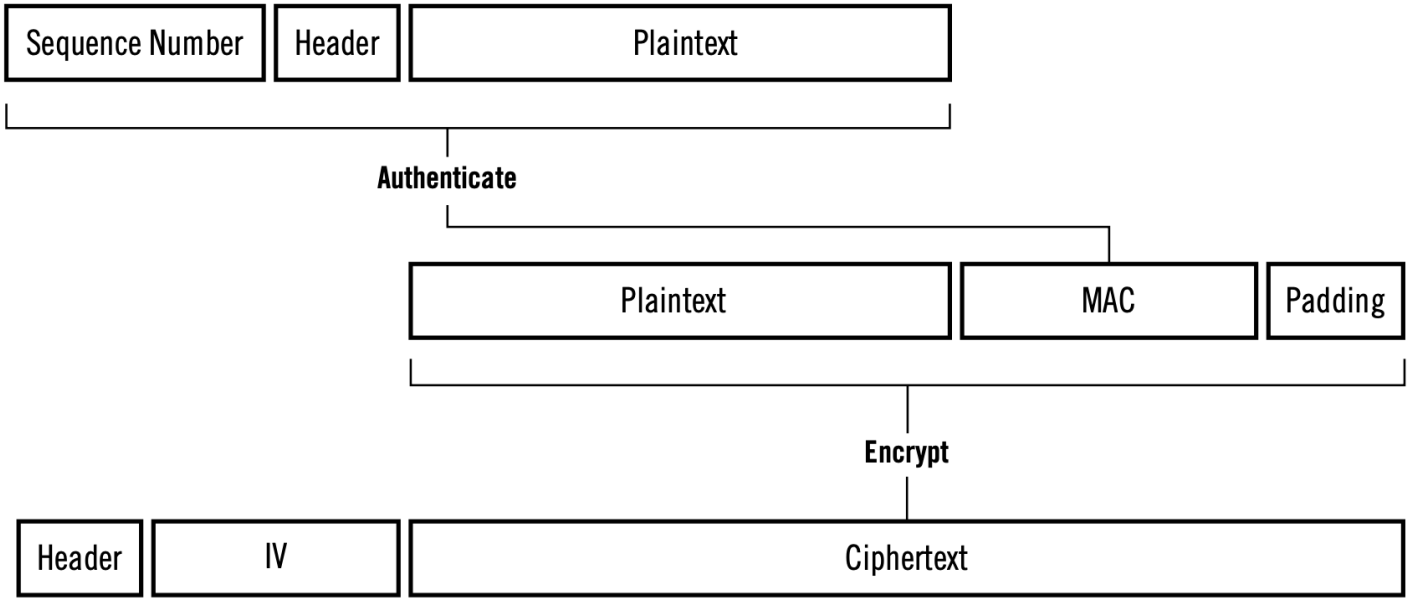
Encryption: Stream cipher

- E.g. RC4
- MAC then Encrypt



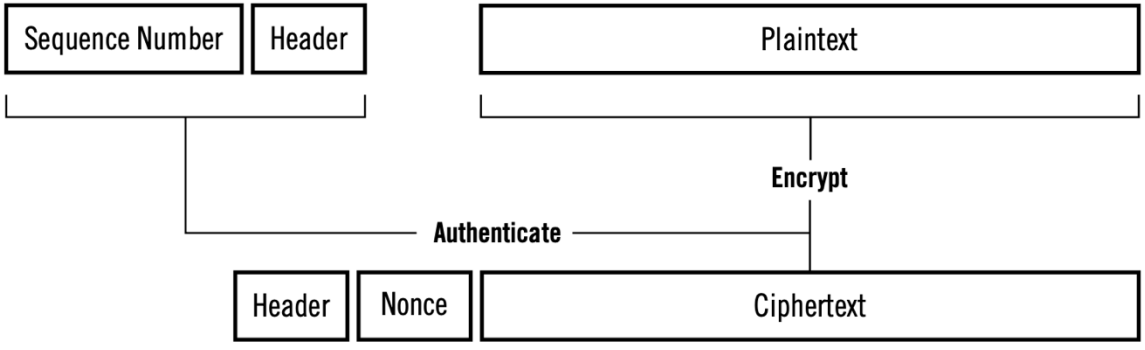
Encryption: Block cipher

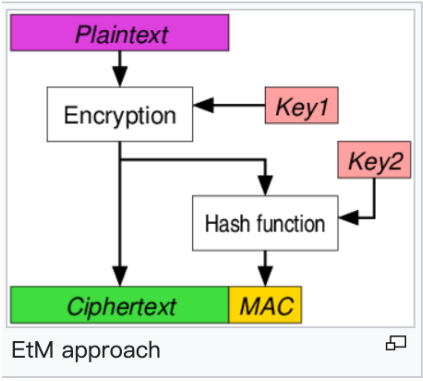
- AES, Chacha2,…
- MAC then Encrypt: BEAST Attack
- Padding : padding oracle attack

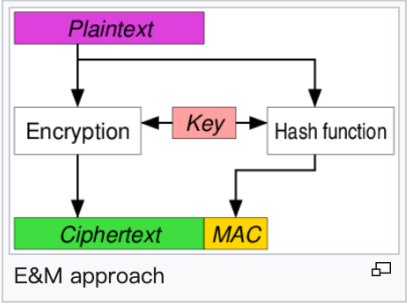


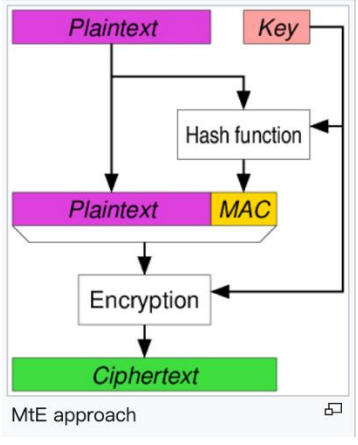
Authentication Encryption

- Authenticated Encryption with associated data(AEAD)
  - No padding, no IV, Noce

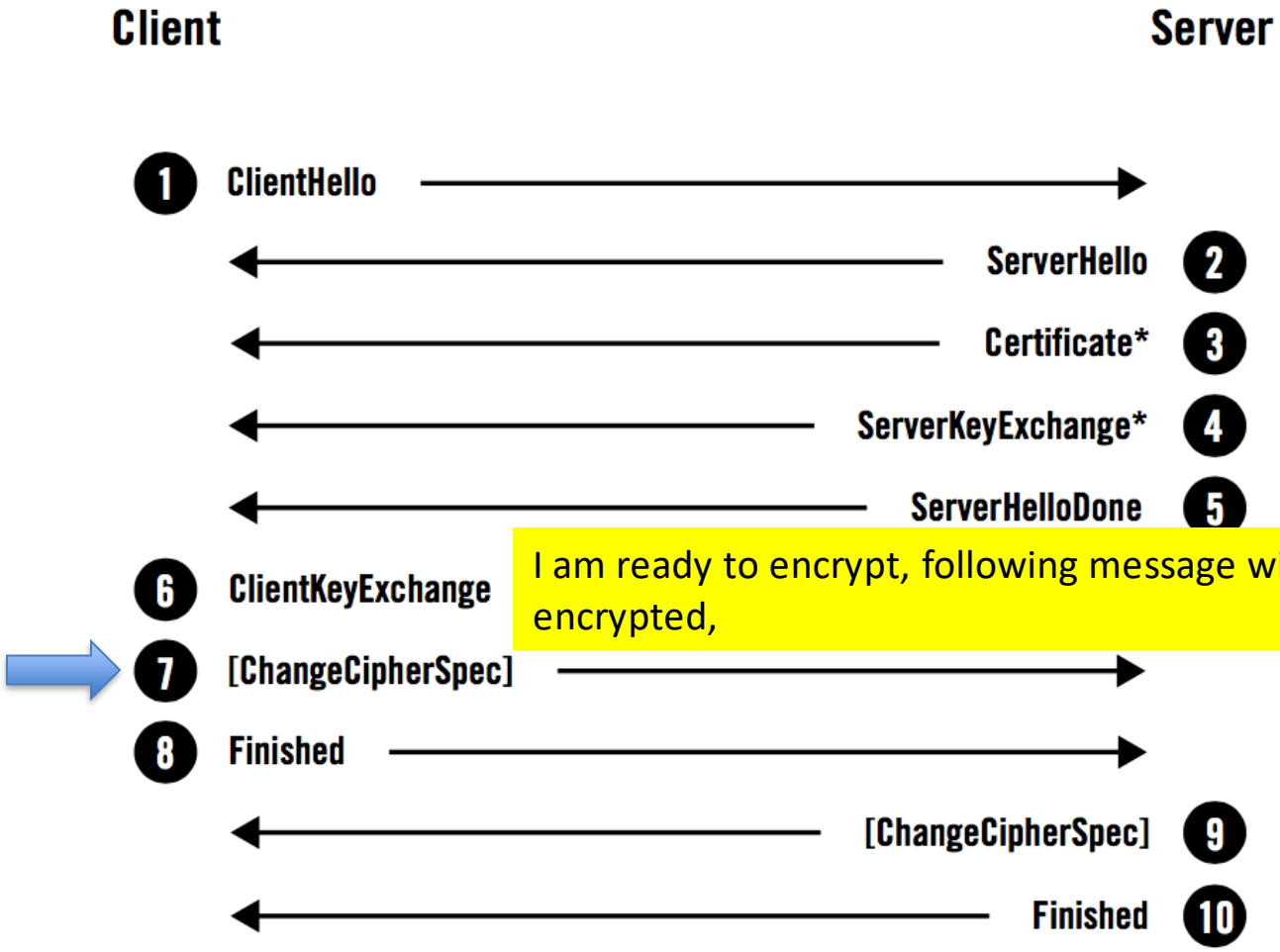








ChangeCipherSpec



* Optional message

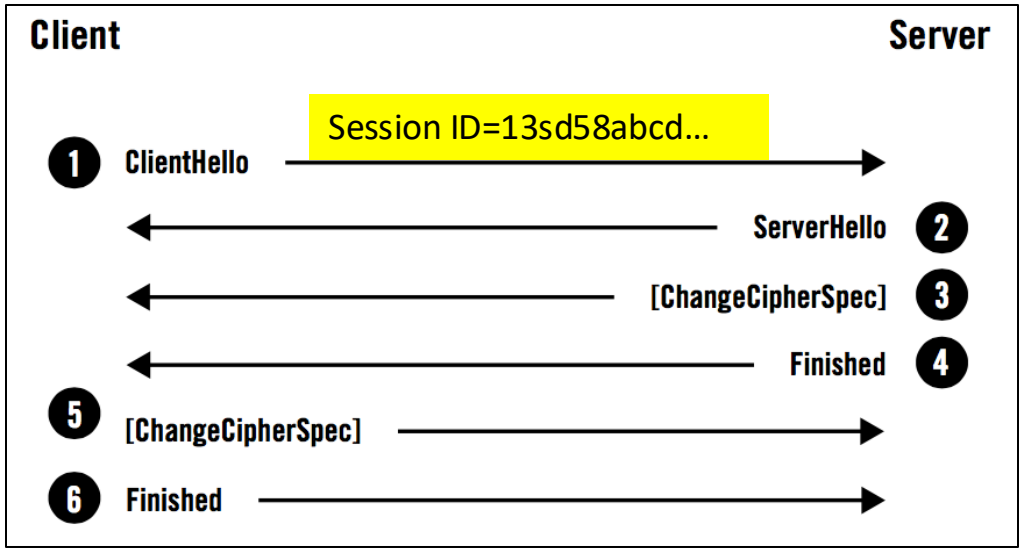
[ ] ChangeCipherSpec protocol message

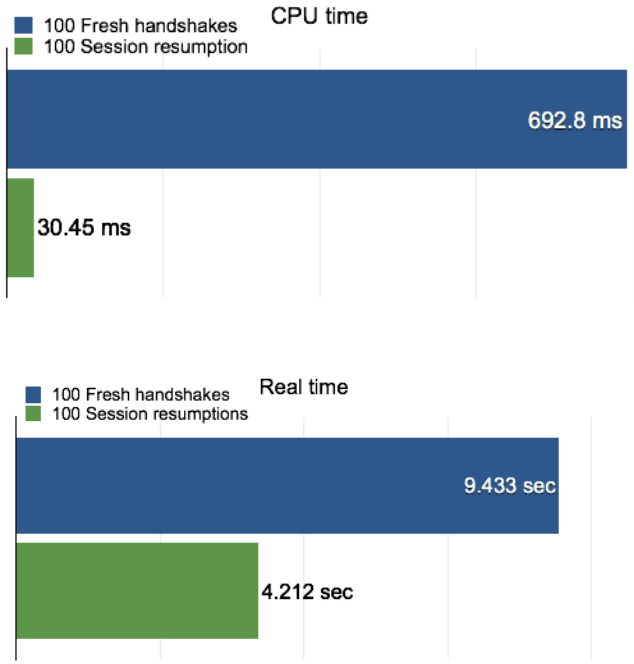
Finish

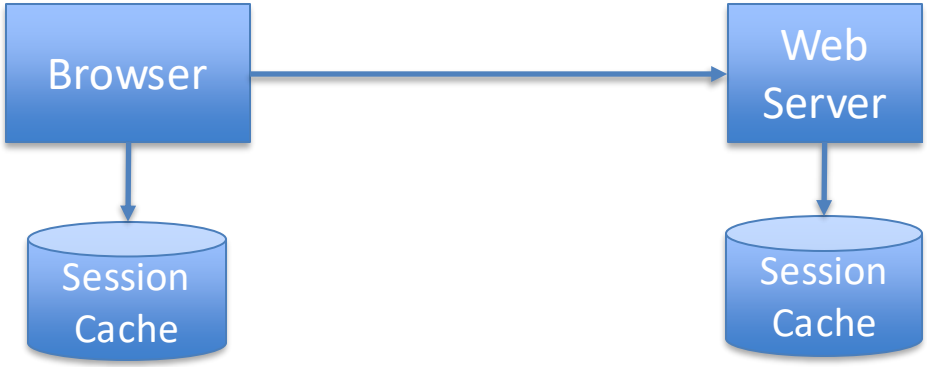
- To verify the handshake integrity, the client and server both send cryptographic signatures of the exchanged data.

```
finished_key =
       HKDF-Expand-Label(BaseKey, "finished", "", Hash.length)

   Structure of this message:

      struct {
          opaque verify_data[Hash.length];
      } Finished;

   The verify_data value is computed as follows:

      verify_data =
          HMAC(finished_key,
               Transcript-Hash(Handshake Context,
                               Certificate*, CertificateVerify*))

      * Only included if present.
```

Session Resumption with session ID Cache





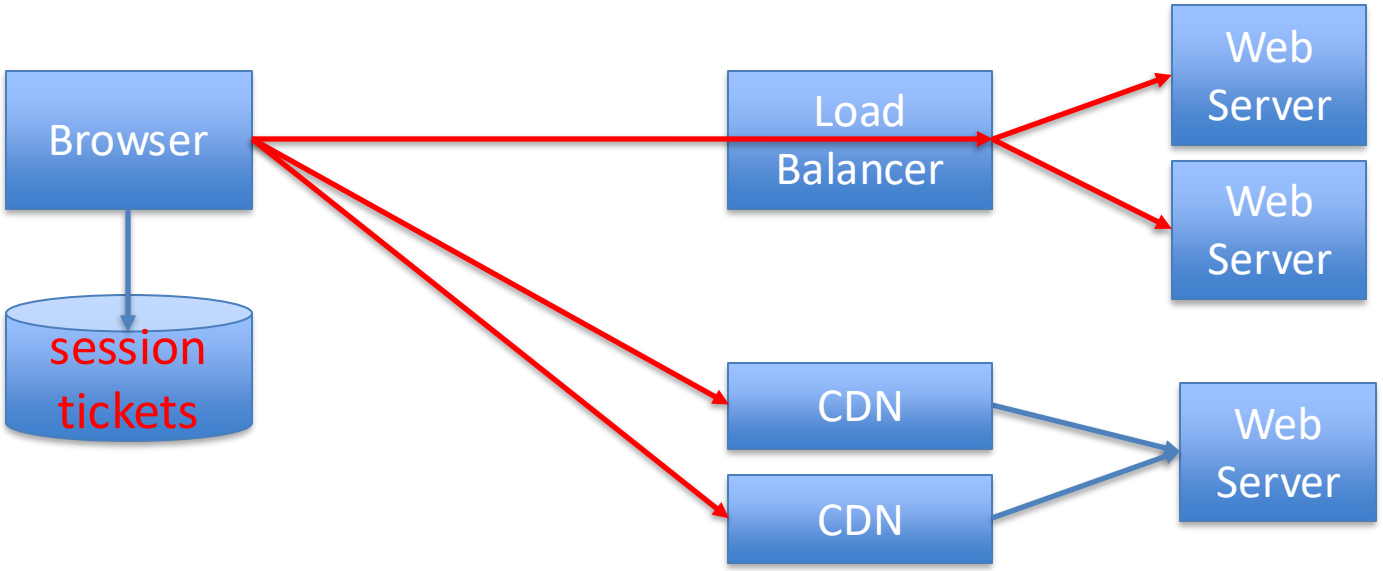


缺点：

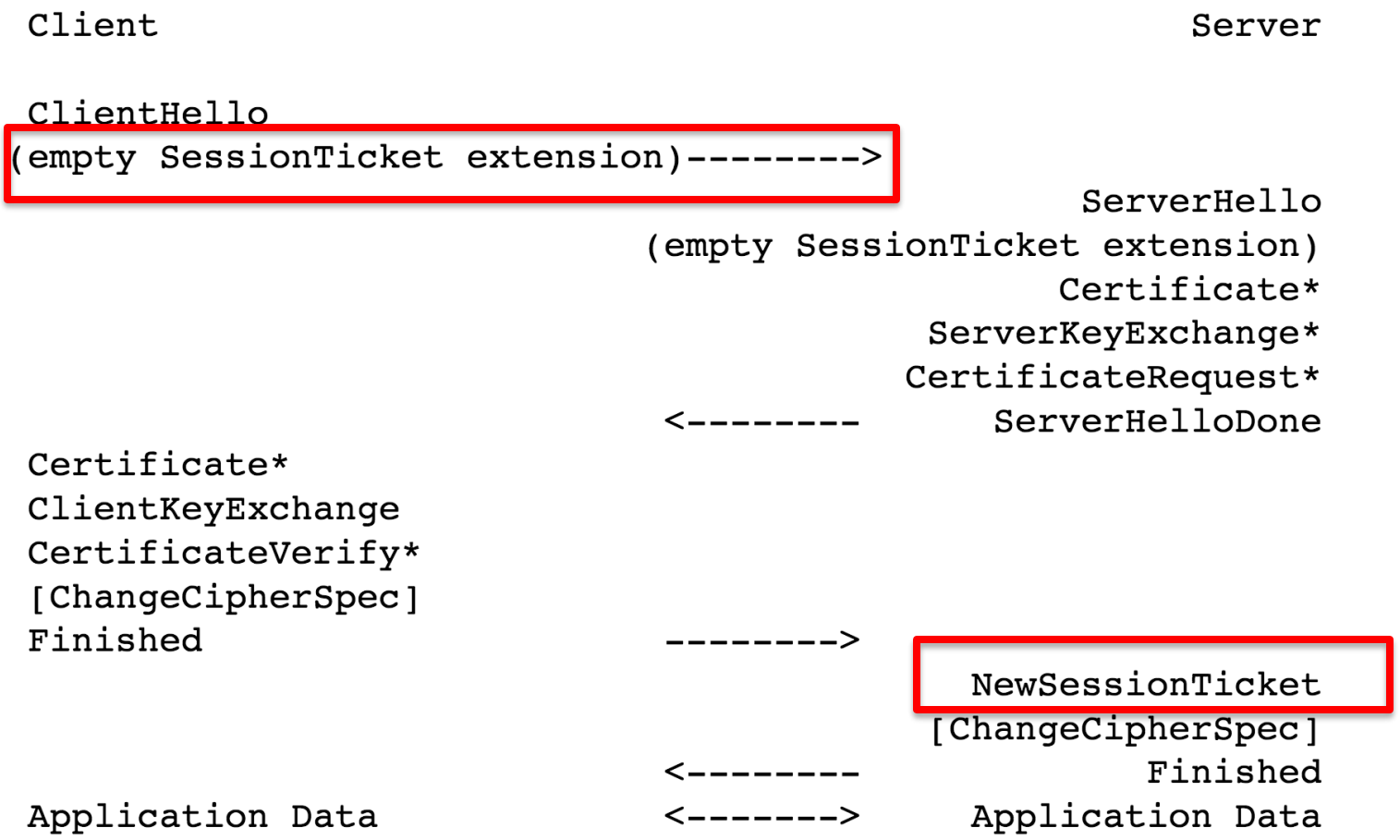
1. 长期存储状态，消耗服务器资源，可扩展性差
2. 不适用于服务器负载均衡、CDN场景

Session Tickets, RFC 5077, 2008 added to TLS 1.2

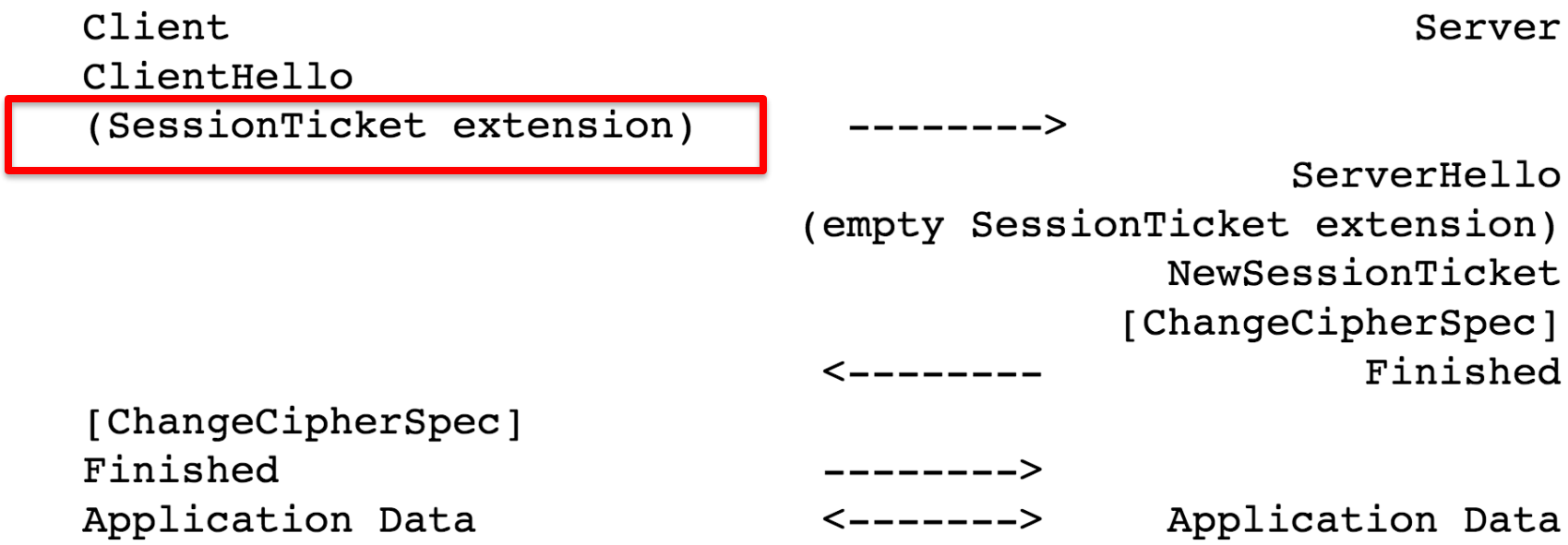
- New session resumption mechanism that doesn't require any server-side storage.
- Like HTTP Cookie, server store state information into Client(e.g., browser)



Transport Layer Security (TLS) Session Resumption without Server-Side State



Transport Layer Security (TLS) Session Resumption without Server-Side State



TLS Extensions, 2003, RFC3546

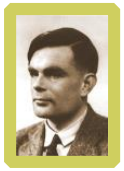
- To add functionality without changing TLS

Table 2.3. A selection of commonly seen TLS extensions

Type	Name	Description
0	server_name	Contains the intended secure virtual host for the connection
5	status_request	Indicates support for OCSP stapling
13 (0x0d)	signature_algorithms	Contains supported signature algorithm/hash function pairs
15 (0x0f)	heartbeat	Indicates support for the Heartbeat protocol
16 (0x10)	application_layer_protocol_negotiation	Contains supported application-layer protocols that the client is willing to negotiate
18 (0x12)	signed_certificate_timestamp	Used by servers to submit the proof that the certificate had been shared with the public; part of Certificate Transparency
21 (0x15)	padding	Used as a workaround for certain bugs in the F5 load balancers[a]
35 (0x23)	session_ticket	Indicates support for stateless session resumption
13172 (0x3374)	next_protocol_negotiation	Indicates support for Next Protocol Negotiation
65281 (0xff01)	renegotiation_info	Indicates support for secure renegotiation

[a] A TLS padding extension (Internet-Draft, A. Langley, January 2014)

The three laws of security:



- Absolutely secure systems do not exist
- To halve your vulnerability, you have to double your expenditure
- Cryptography is typically bypassed, not penetrated

A. Shamir. Cryptography: State of the science.

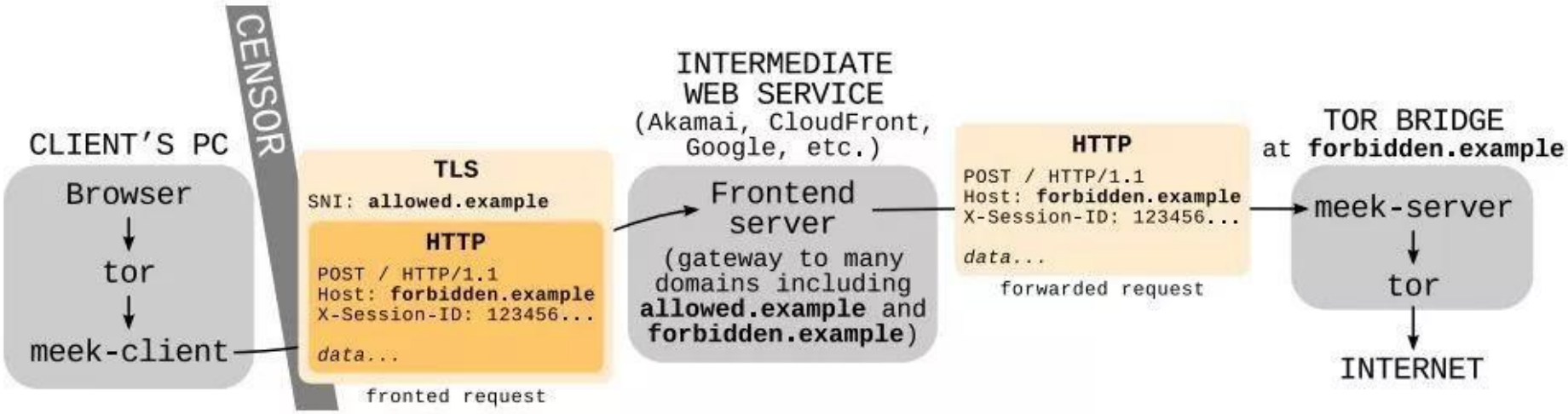
ACM A. M. Turing Award Lecture, June 8 2003.

Limitation of TLS

- Metadata of TCP and all other lower layers remains in plaintext, e.g. IP, server name, TCP connection, ...
- TLS handshake is exposed as plaintext, e.g. client fingerprint, SNI, user identity,..
- Some protocol information remains in the clear: the observer can see the subprotocol and length of each message.

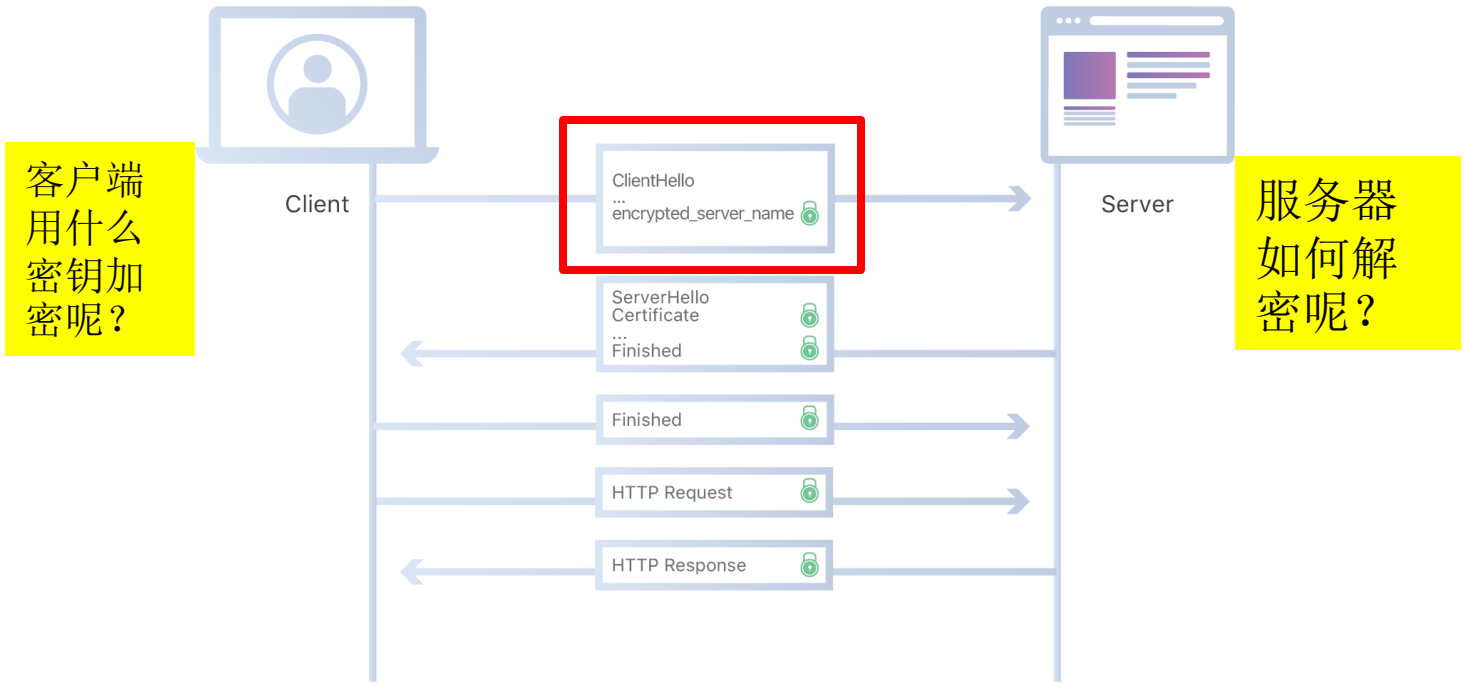
Server Name Indication(SNI)

- To support virtual host
- Enable Domain Fronting attack

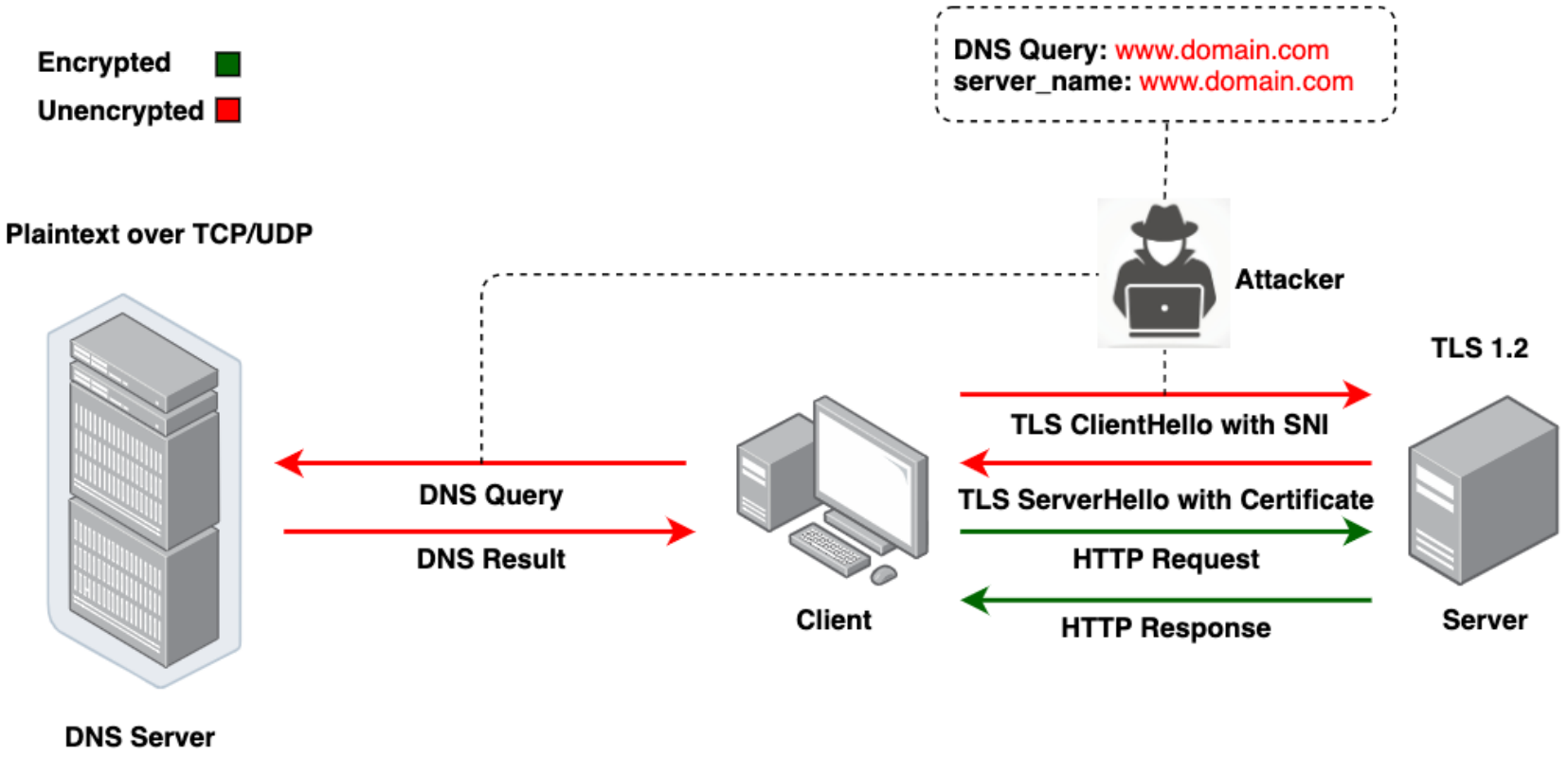


Fifield, David; Lan, Chang; Hynes, Rod; Wegmann, Percy; Paxson, Vern (2015). "Blocking-resistant communication through domain fronting" (PDF). *Proceedings on Privacy Enhancing Technologies*. **2015**
Disabled by Google, Amazon, 2018

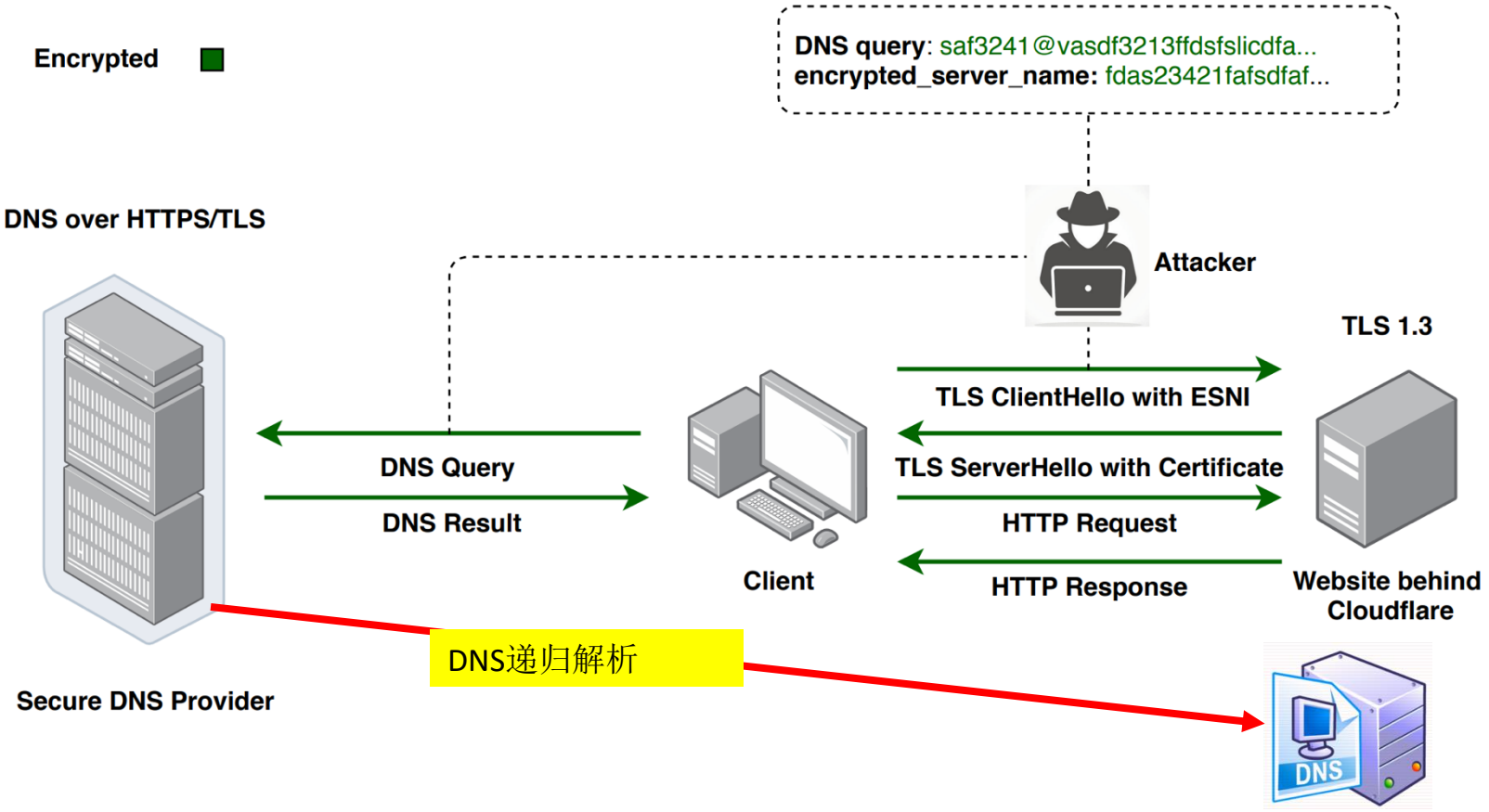
TLS 1.3 新功能： TLS加密的服务器名字(ESNI)



DNS, TLS1.2都可以被过滤



DoH 和 TLS 1.3 之后....

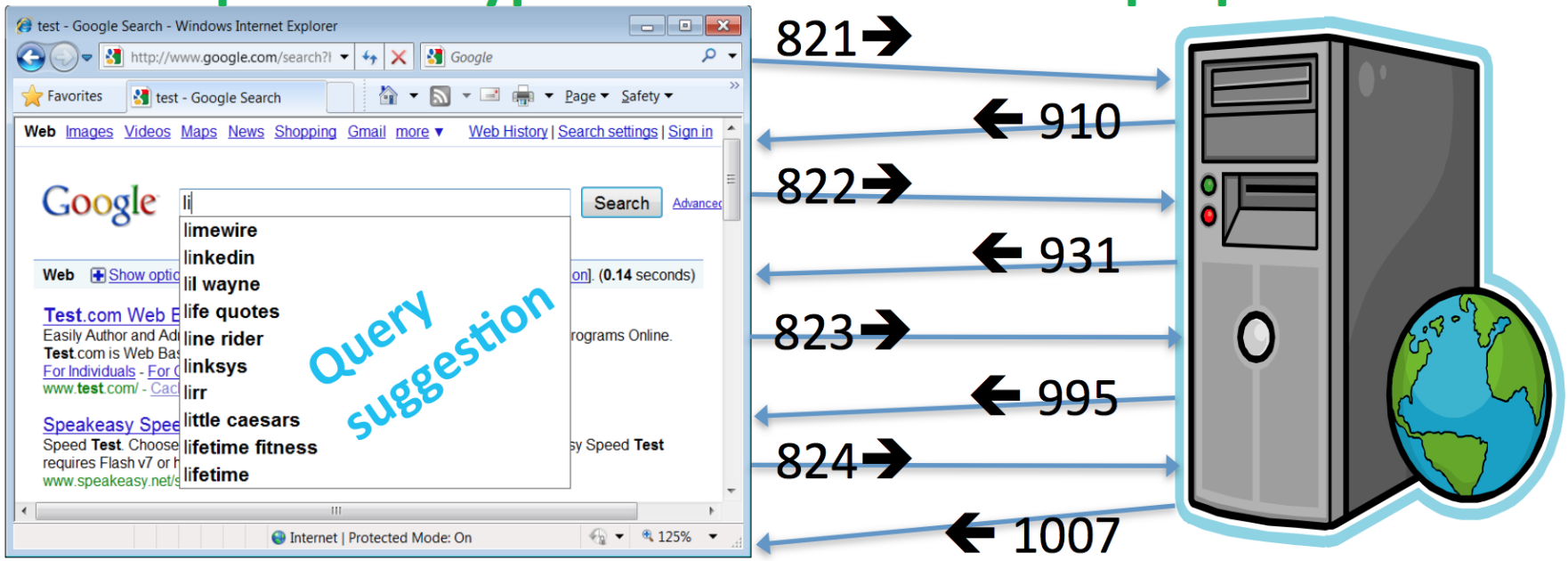


Side channel: packet size

Google bing YAHOO!

Scenario: search using encrypted Wi-Fi WPA/WPA2.

Example: user types "*list*" on a WPA2 laptop.



Attacker's effort: linear, not exponential.

Consequence: Anybody on the street knows our search queries.


产品设计


系统的部署和运维

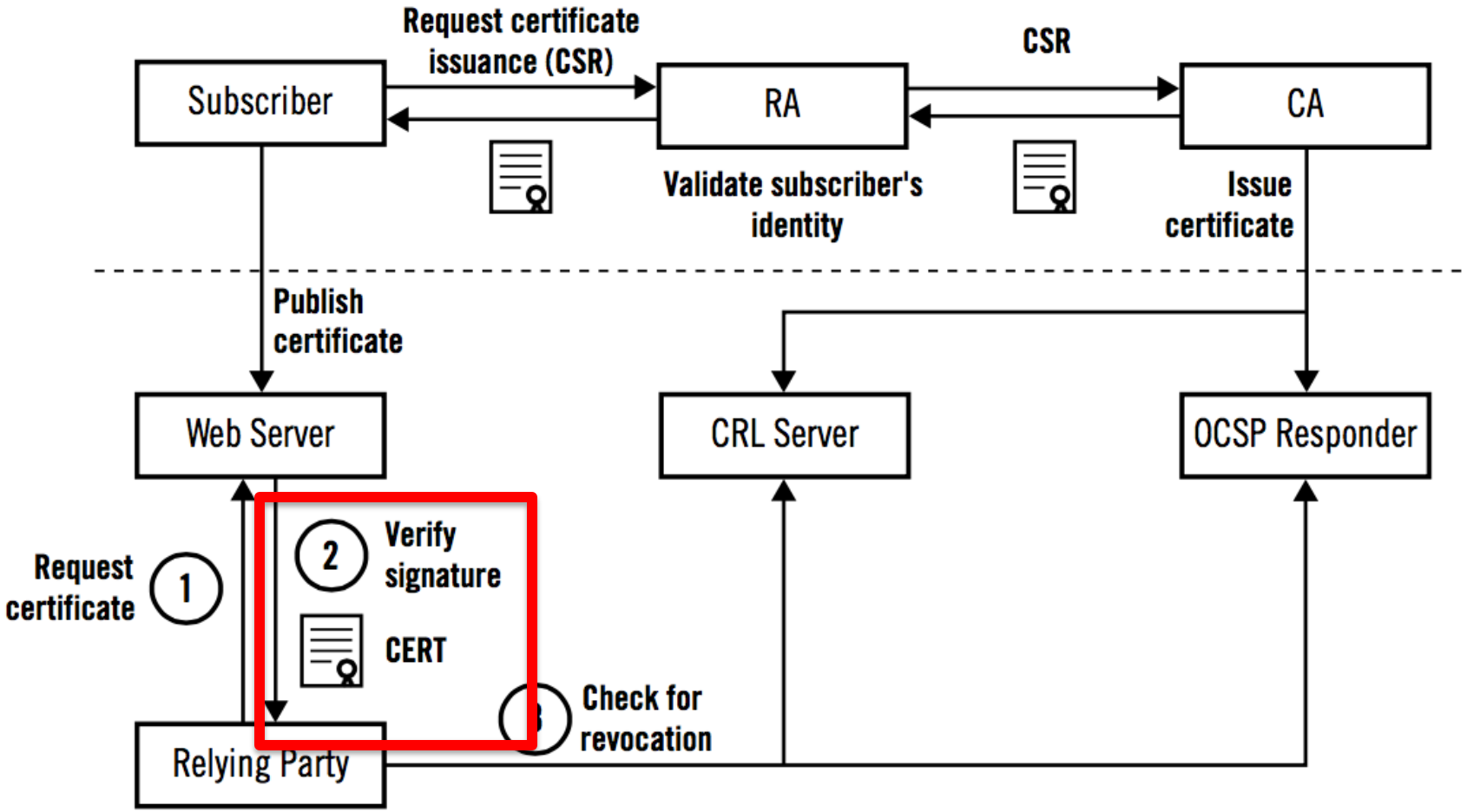
TLS Vulnerabilities and Attacks

Haixin Duan

Outline

- Attacks against PKI
- Implementation Issues
- Protocol attacks

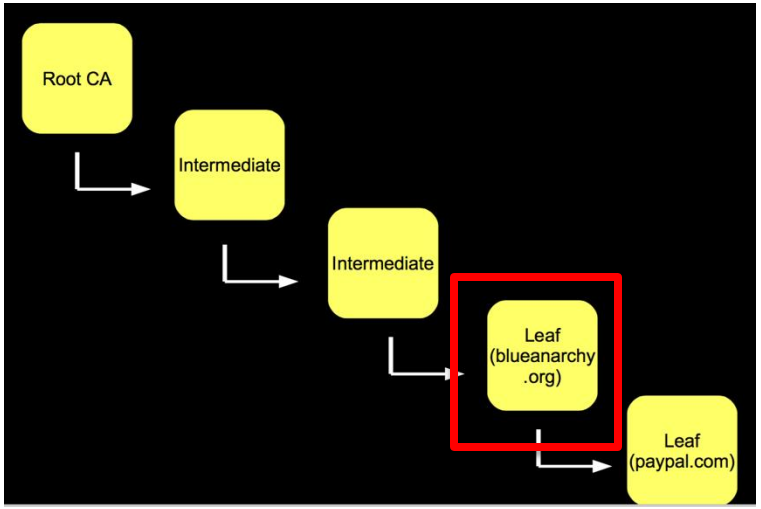
Certificate validation problems



Implementation Issues

- **Certificate Validation Flaws**
  - **Library and Platform Validation Failures**
    - Basic Constraints check failure in MS CryptoAPI (2002)

What if a certificate signed by a leaf node?



Internet Explorer SSL Vulnerability ,
Moxie Marlinspike, 2002

```
Authority Information Access:
    OCSP - URI:http://ocsp.sca1b.amazontr
    CA Issuers - URI:http://crt.sca1b.amc

X509v3 Basic Constraints: critical
    CA:FALSE
```

https://moxie.org/ie-ssl-chain.txt

Using Frankencerts for Automated Adversarial Testing of Certificate Validation

test certificate

TLS/SSL Implementations

Brubaker, C., Jana, S., Ray, B., Khurshid, S., & Shmatikov, V.. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. *IEEE Symposium on Security and Privacy*. 2014

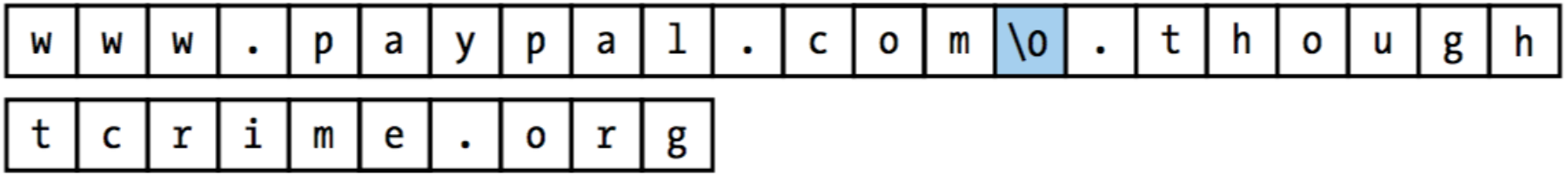
Test results summary

- Tested 14 different SSL/TLS implementations
- 208 discrepancies due to 15 root causes
- Multiple bugs
  - Accepting fake and unauthorized intermediate Certificate Authorities (CAs)

attacker can impersonate any website!

e.g., any server with a valid X.509 version 1 certificate can act as a rogue CA and issue fake certificates for any domain, enabling man-in-the-middle attacks against MatrixSSL and GnuTLS.

Hostname Validation Issues

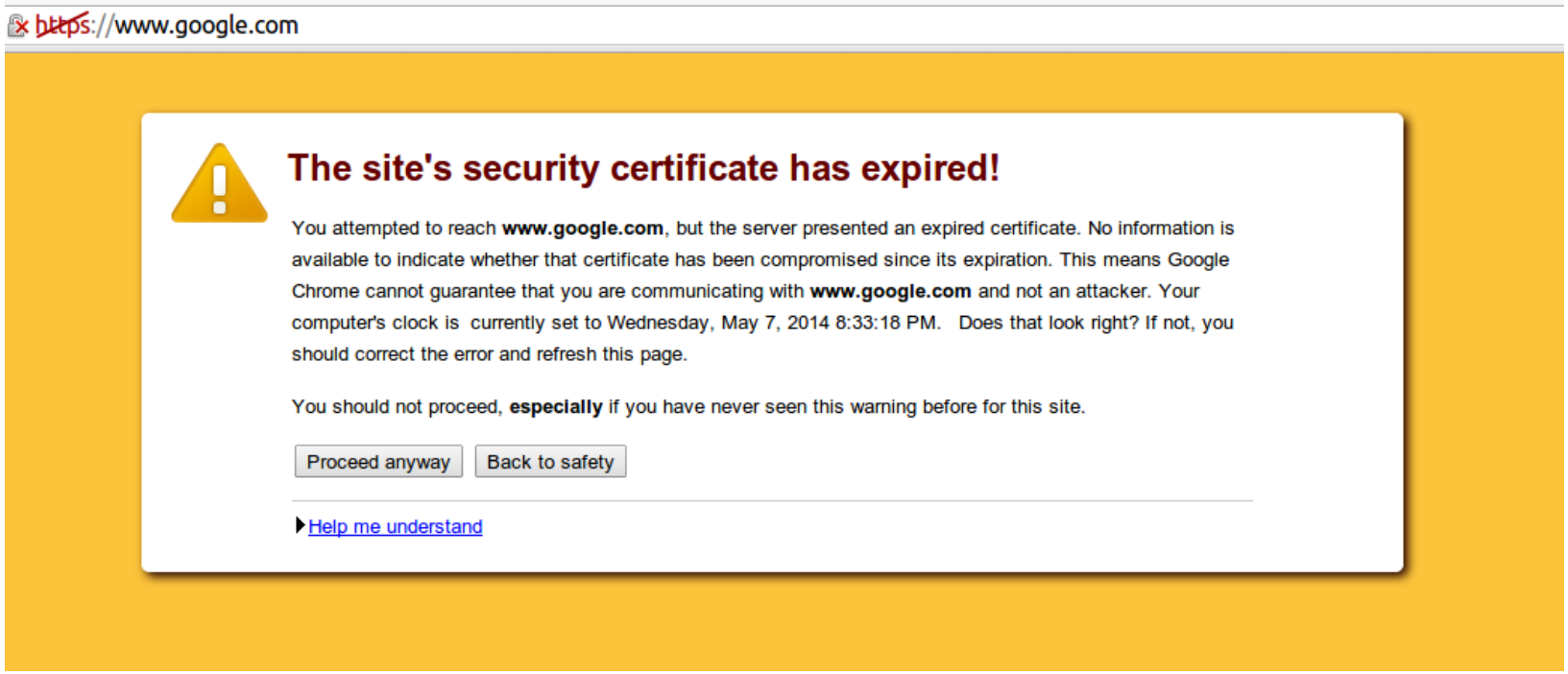


- CA think this domain belongs to *thoughtcrime.org*
- Browser think this certificate belongs to *www.papal.com*

Implementation Issues

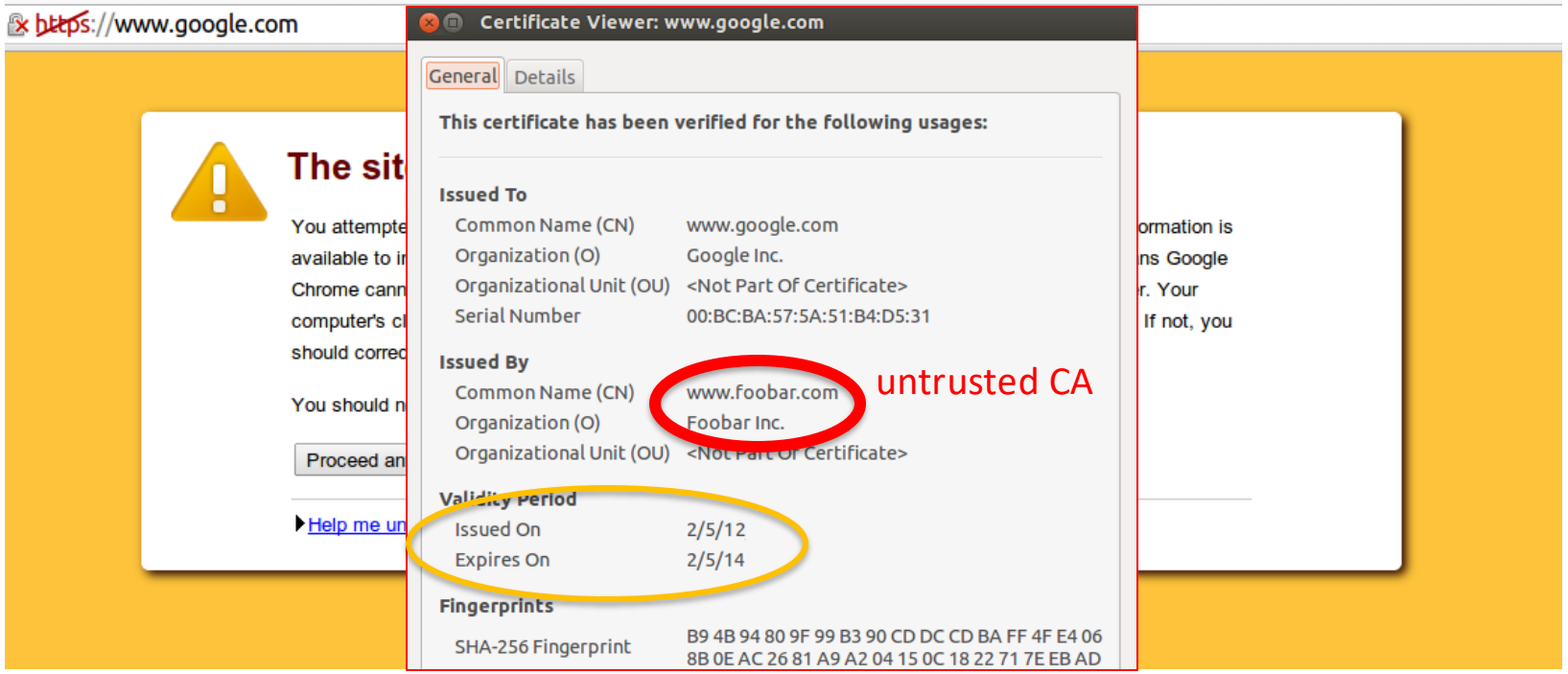
- **Certificate Validation Flaws**
  - **Library and Platform Validation Failures**
    - Basic Constraints check failure in MS CryptoAPI (2002)
    - Basic Constraints check failure in iOS (2011)
    - OpenSSL ChangeCipherSpec Injection (2014)
  - Application Validation Failures
    - Georgiev et al., The most dangerous code in the world: validating SSL certicates in non-browser software, CCS'12

Exhibit 2: Google Chrome



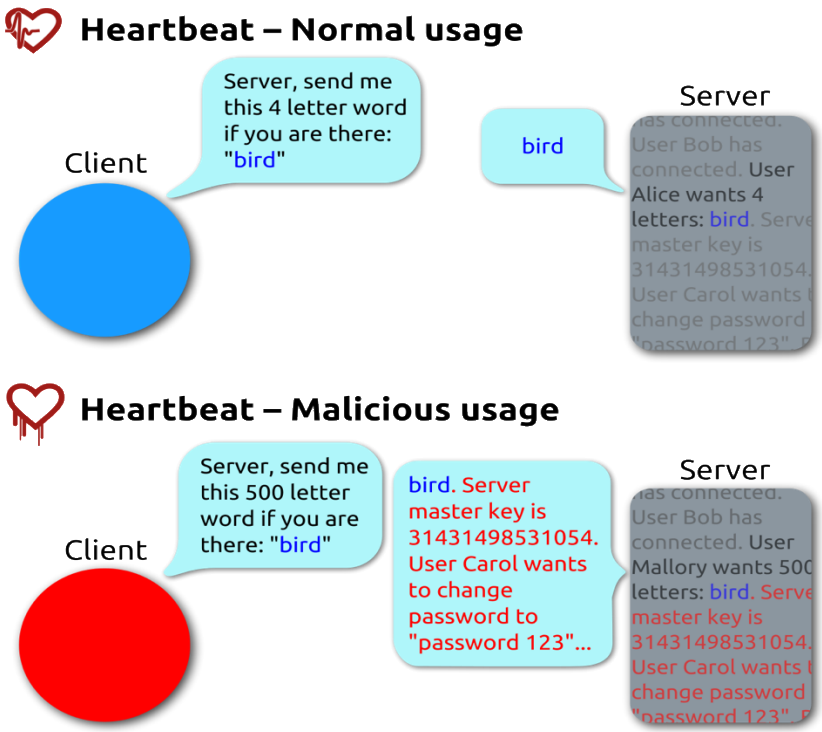
Not a big problem… just to click through?

Exhibit 2: Google Chrome



Implementation Issues: HeartBleed

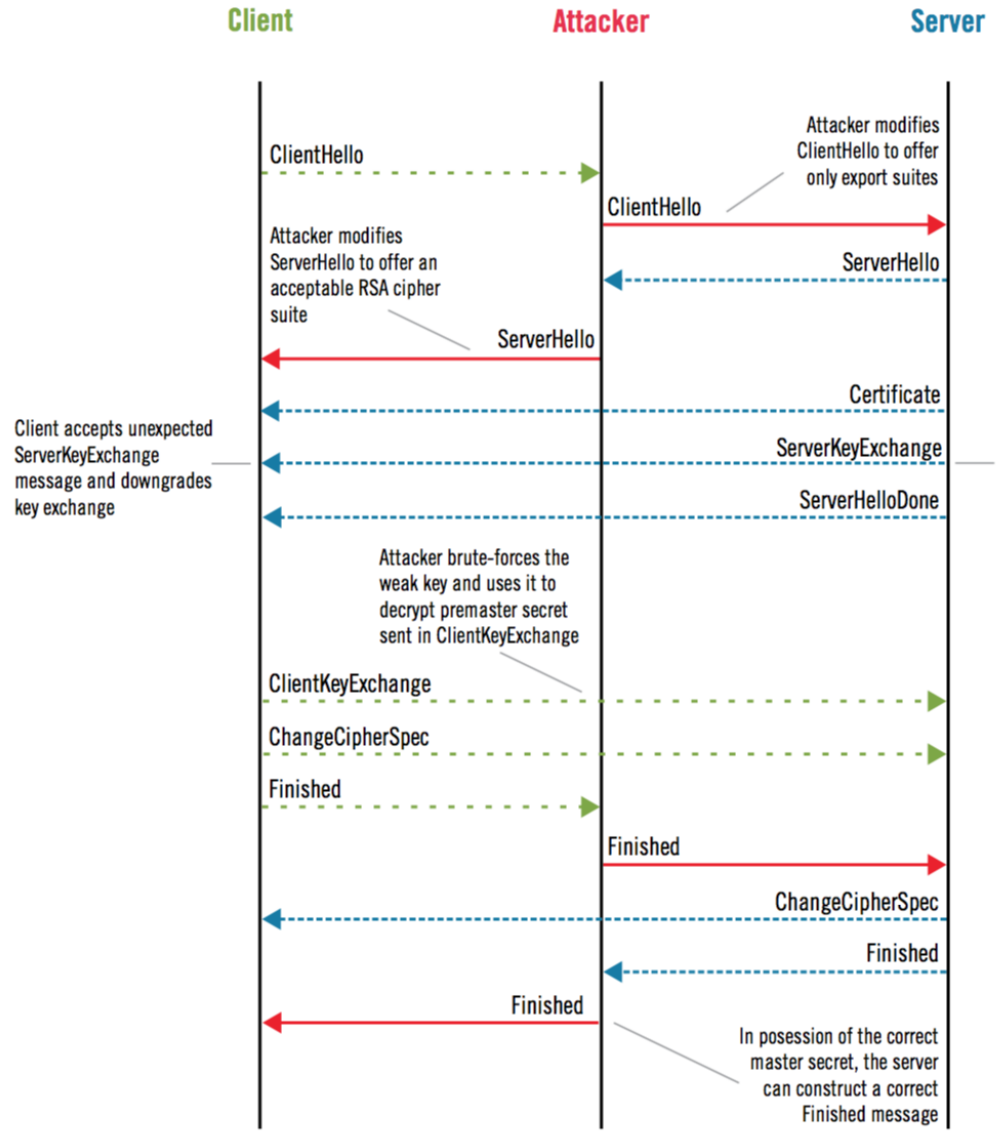
- **Heartbleed (**CVE-2014-0160, 2014**)**
  - Heartbeat extension for DTLS , RFC6520, 2012
  - Software bug of OpenSSL: over read
  - not a cryptographic failure.
  - Poor code quality of OpenSSL



http://heartbleed.com/

Implementation Issues:

FREAK(*Factoring RSA Export Keys*)



- *Downgrade to a lower secure cipher*
- *export-strength* suites: 40bit shared-key, 512 RSA

e.g., TLS_RSA_EXPORT_WITH_DES40_CBC_SHA, TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA, etc

spent about $100 on cloud computing resources (EC2), and broke the key in about seven hours.

Reuse the 512 bit Key

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[†]

[*] INRIA Paris-Rocquencourt [†] INRIA Nancy-Grand Est, CNRS, and Université de Lorraine
[‖] Microsoft Research [‡] University of Pennsylvania [§] Johns Hopkins [¶] University of Michigan

For additional materials and contact information, visit WeakDH.org.

ABSTRACT

We investigate the security of Diffie-Hellman key exchange as used in popular Internet protocols and find it to be less secure than widely believed. First, we present Logjam, a novel flaw in TLS that lets a man-in-the-middle downgrade connections to "export-grade" Diffie-Hellman. To carry out this attack, we implement the number field sieve discrete log algorithm. After a week-long precomputation for a specified 512-bit group, we can compute arbitrary discrete logs in that group in about a minute. We find that 82% of vulnerable servers use a single 512-bit group, allowing us to compromise connections to 7% of Alexa Top Million HTTPS sites. In response, major browsers are being changed to reject short groups.

We go on to consider Diffie-Hellman with 768- and 1024-bit groups. We estimate that even in the 1024-bit case, the computations are plausible given nation-state resources. A small number of fixed or standardized groups are used by millions of servers; performing precomputation for a single 1024-bit group would allow passive eavesdropping on 18% of popular HTTPS sites, and a second group would allow decryption of traffic to 66% of IPsec VPNs and 26% of SSH servers. A close reading of published NSA leaks shows that the agency's attacks on VPNs are consistent with having achieved such a break. We conclude that moving to stronger key exchange methods should be a priority for the Internet community.

coded, or widely shared Diffie-Hellman parameters has the effect of dramatically reducing the cost of large-scale attacks, bringing some within range of feasibility today.

The current best technique for attacking Diffie-Hellman relies on compromising one of the private exponents (a, b) by computing the discrete log of the corresponding public value $(g^a \bmod p,\ g^b \bmod p)$. With state-of-the-art number field sieve algorithms, computing a single discrete log is more difficult than factoring an RSA modulus of the same size. However, an adversary who performs a large precomputation for a prime p can then quickly calculate arbitrary discrete logs in that group, amortizing the cost over all targets that share this parameter. Although this fact is well known among mathematical cryptographers, it seems to have been lost among practitioners deploying cryptosystems. We exploit it to obtain the following results:

Active attacks on export ciphers in TLS. We introduce Logjam, a new attack on TLS by which a man-in-the-middle attacker can downgrade a connection to export-grade cryptography. This attack is reminiscent of the FREAK attack [7] but applies to the ephemeral Diffie-Hellman ciphersuites and is a TLS protocol flaw rather than an implementation vulnerability. We present measurements that show that this attack applies to 8.4% of Alexa Top Million HTTPS sites and 3.4% of all HTTPS servers that have browser-trusted certificates.

To exploit this attack, we implemented the number field

Logjam: Active downgrade attack to export Diffie-Hellman

Protocol flaw: Server does not sign chosen cipher suite.

hello, client random

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

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

certificate = public RSA key + CA signatures

p_{512}, g, g^a, $\text{Sign}_{\text{RSAkey}}(p_{512}, g, g^a)$

g^b

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

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

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

Enc_{k_e}(request)

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

Most hosts use the same parameters

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

Top ten primes accounted for 99% of hosts.

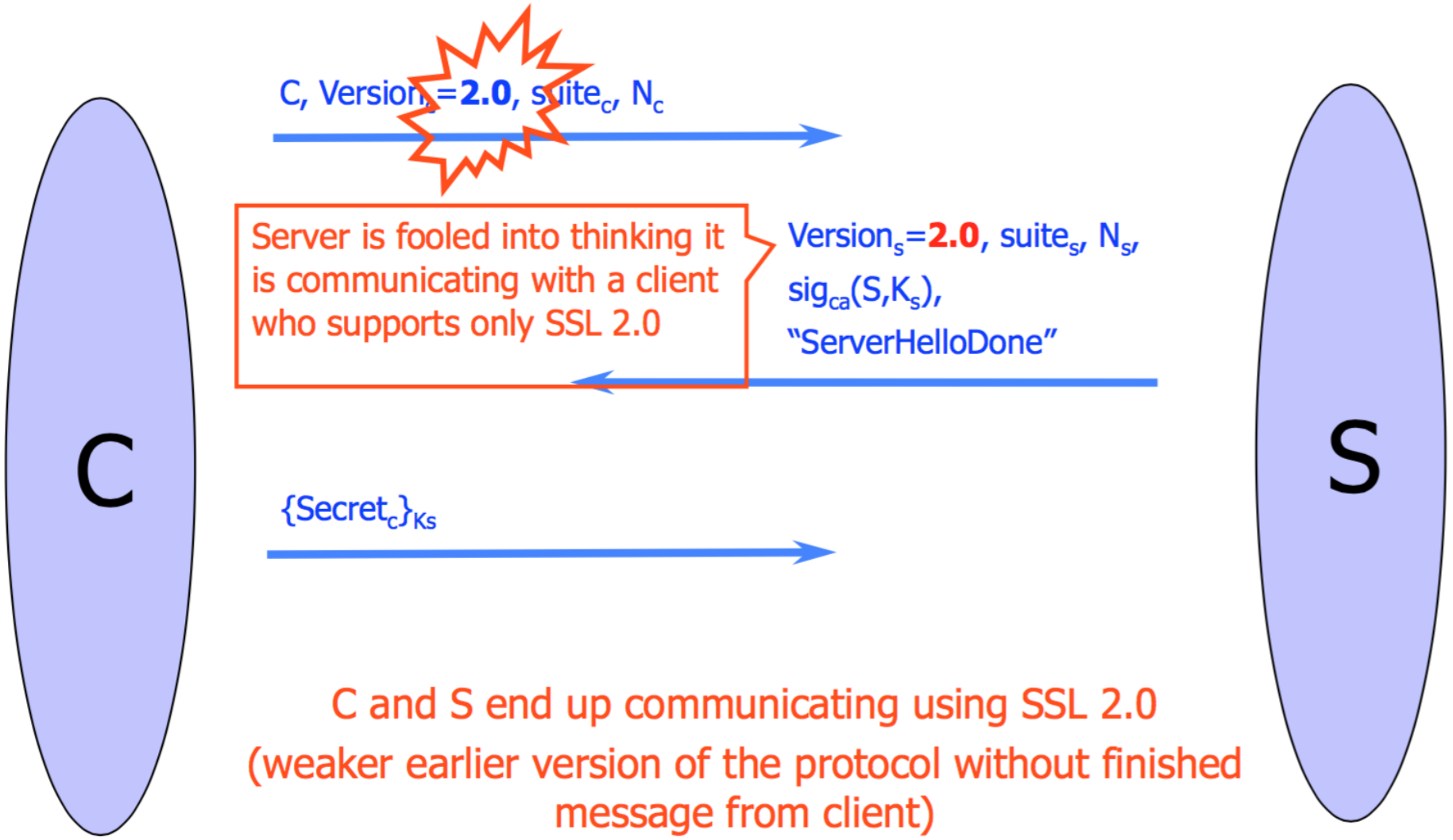
Computing 512-bit discrete logs

- Carried out precomputation for Apache, mod_ssl primes.

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

- After 1 week precomputation, median individual log time 70s.
- Many ways attacker can work around delay.
- Logjam and our precomputations can be used to break connections to 8% of the HTTPS top 1M sites!

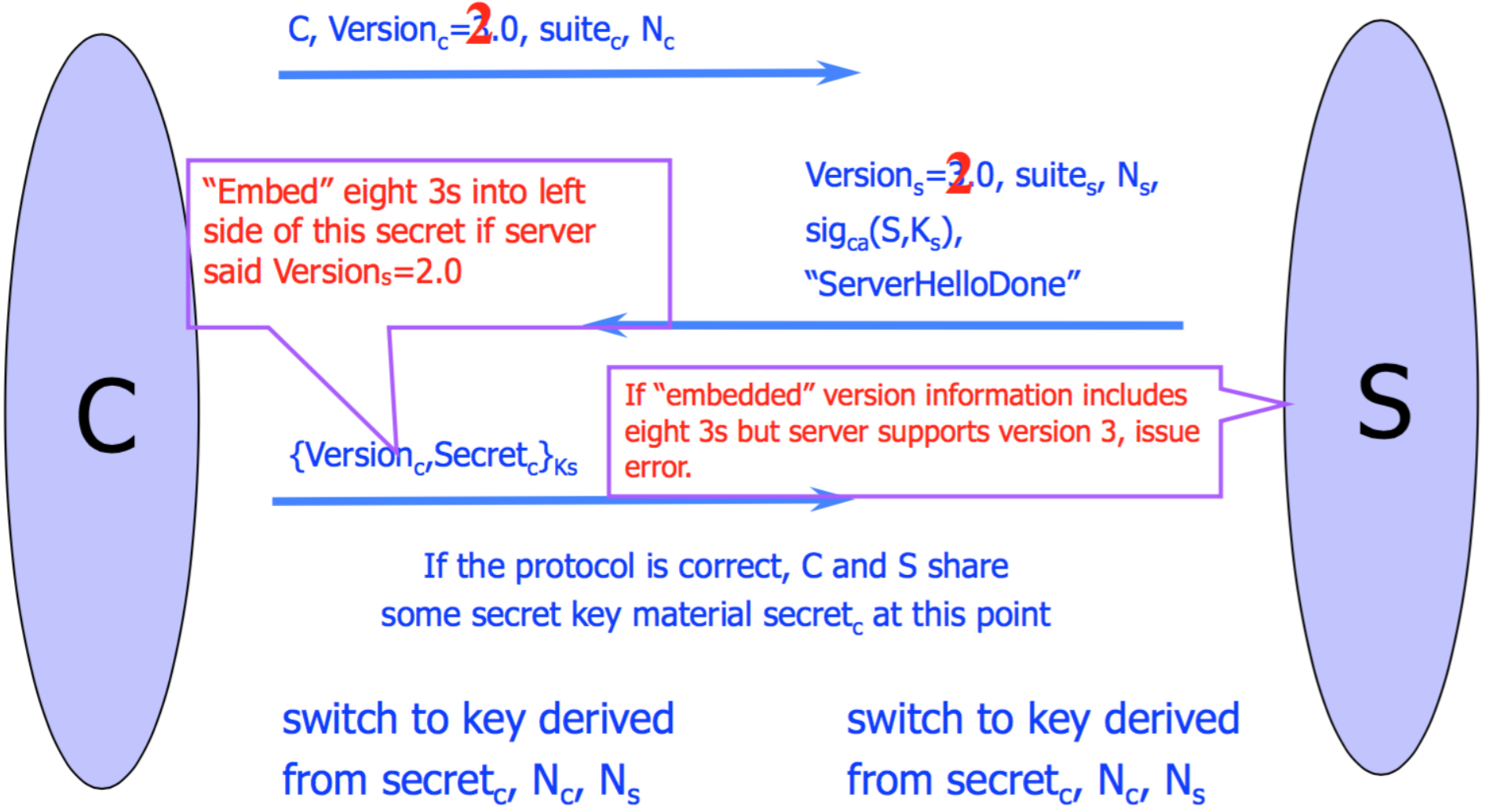
Implementation Issues:
Protocol Rollback Attacks



C and S end up communicating using SSL 2.0
(weaker earlier version of the protocol without finished message from client)

Protocol Rollback Attacks

RFC 6101: The SSL Protocol Version 3.0, Section E.2. ,2011



Downgrade protection in TLS 1.3

- ServerHello.Random:
  - If negotiating TLS 1.2/1.1, TLS 1.3 servers MUST set the last 8 bytes of their Random value to the bytes: 44 4F 57 4E 47 52 44 01/00
- Finished message

```
verify_data =
    HMAC(finished_key,
         Transcript-Hash(Handshake Context,
                         Certificate*, CertificateVerify*))
```

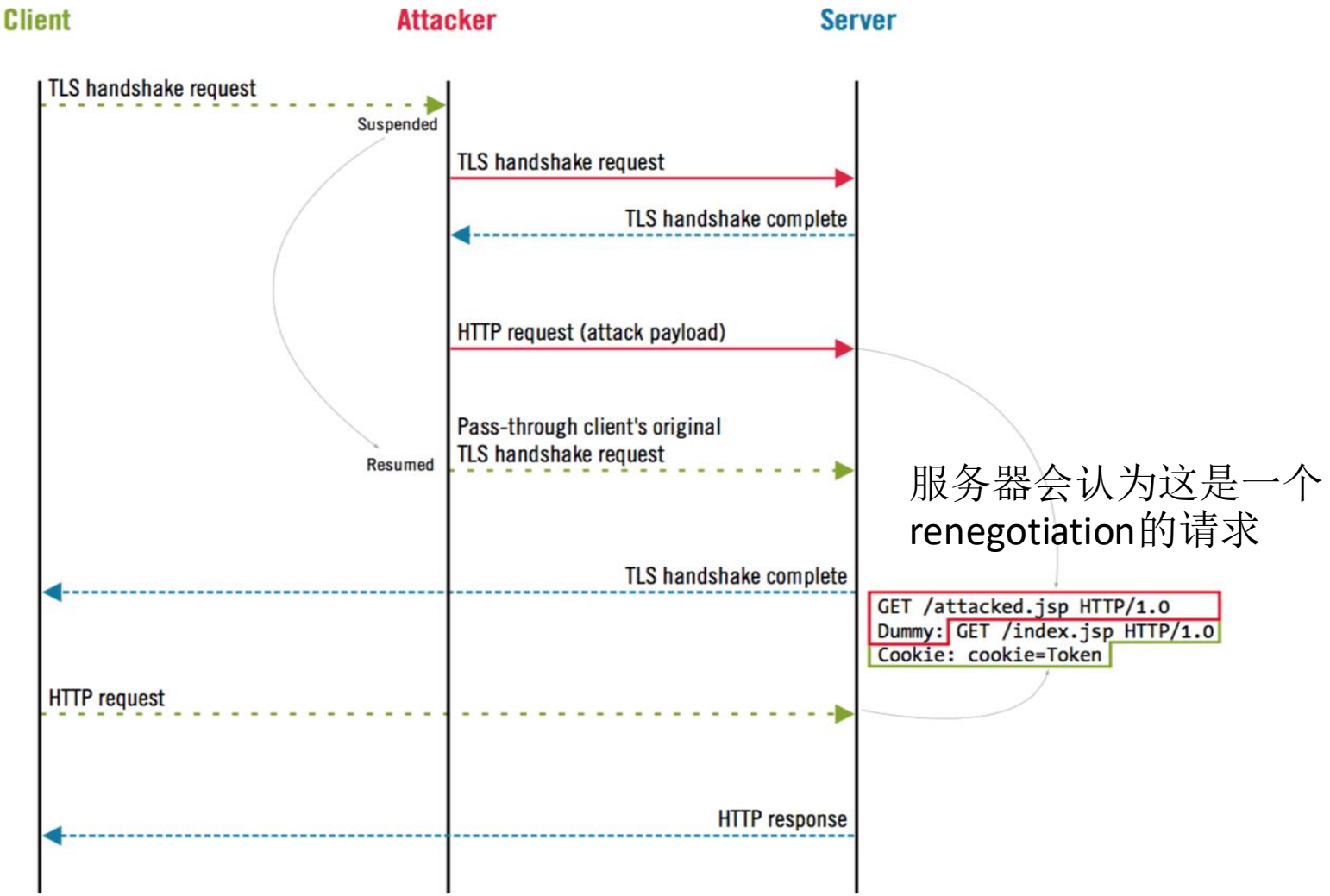
Outline

- Attacks against PKI
- Implementation Issues
- Protocol attacks

Renegotiation

- Request Client Certificate after TLS connection
  - User friendly
- Information Hiding
  - Plain:Client/Server Hello, Certificate, KeyExchange
  - Keep client identity private
- Change of encryption strength
- TLS Record counter overflow

Insecure renegotiation



1. Server/Client is not able to know the parties behind the same TCP
2. Application(e,g. HTTP) is not aware under-layer TLS renegotiation

Renegotiation attacks on HTTP

- Execution of arbitrary GET request
  - Bypass authentication
  - Attacker can not get the response (get not get the credential)

```
GET /path/to/resource.jsp HTTP/1.0
X-Ignore: GET /index.jsp HTTP/1.0
Cookie: JSESSIONID=B3DF4B07AE33CA7DF207651CDB42136A
```

Renegotiation attacks on HTTP

- Execution of arbitrary GET request
  - Bypass authentication
  - Attacker can not get the response (get not get the credential)

```
GET /transfer/do.jsp?from=victim&to=attacker HTTP/1.0
X-Ignore: GET /index.jsp HTTP/1.0
Cookie: JSESSIONID=B3DF4B07AE33CA7DF207651CDB42136A
```

Renegotiation attacks on HTTP

- Credential theft

```
POST /statuses/update.xml HTTP/1.0
Authorization: Basic [attacker's credentials]
Content-Type: application/x-www-form-urlencoded
Content-Length: [estimated body length]

status=POST /statuses/update.xml HTTP/1.1
Authorization: Basic [victim's credentials]
```

Mitigation

Upgrade to support secure renegotiation

In early 2010, the *Renegotiation Indication* extension was released to address the problem with renegotiation at the protocol level.[10] Today, several years later, you should expect that all products can be upgraded to support secure renegotiation. If you're dealing with products that cannot be upgraded, it's probably an opportunity to consider if they're still worth using.

The `renegotiation_info` extension improves TLS with verification that renegotiation is being carried out between the same two parties that negotiated the previous handshake.

Initially (during the first handshake on a connection), this extension is used by both parties to inform each other that they support secure renegotiation; for this, they simply send the extension without any data. To secure SSL 3, which doesn't support extensions, clients can instead use a special signaling suite, `TLS_EMPTY_RENEGOTIATION_INFO_SCSV` (`0xff`).

On subsequent handshakes, the extension is used to submit proof of knowledge of the previous handshake. Clients send the `verify_data` value from their previous `Finished` message. Servers send two values: first the client's `verify_data` and then their own. The attacker couldn't have obtained these values, because the `Finished` message is always encrypted.

Mitigation

Upgrade to support secure renegotiation

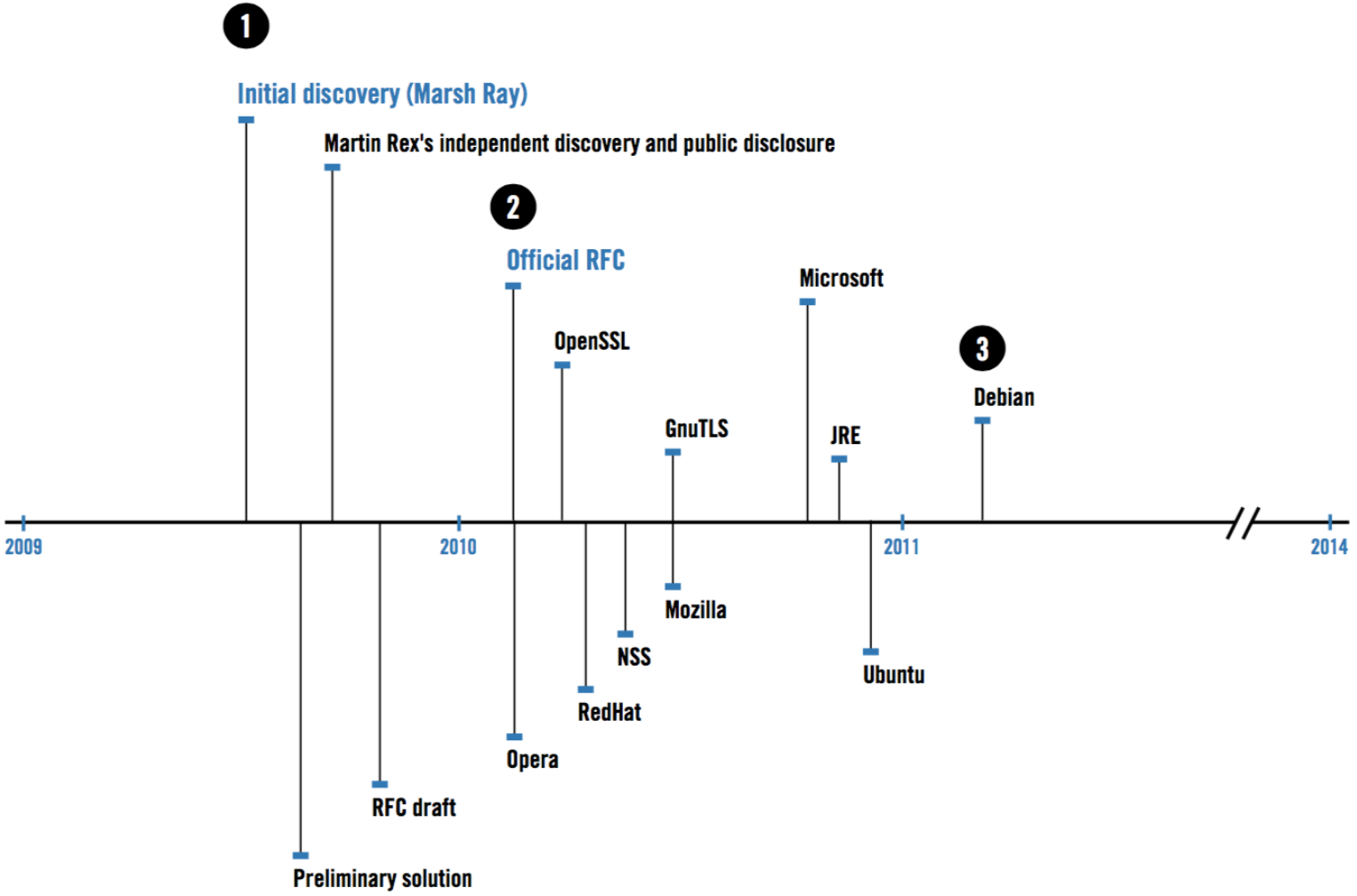
In early 2010, the *Renegotiation Indication* extension was released to address the problem with renegotiation at the protocol level.[10] Today, several years later, you should expect that all products can be upgraded to support secure renegotiation. If you're dealing with products that cannot be upgraded, it's probably an opportunity to consider if they're still worth using.

Disable renegotiation

In the first several months after the discovery, disabling renegotiation was the only mitigation option.

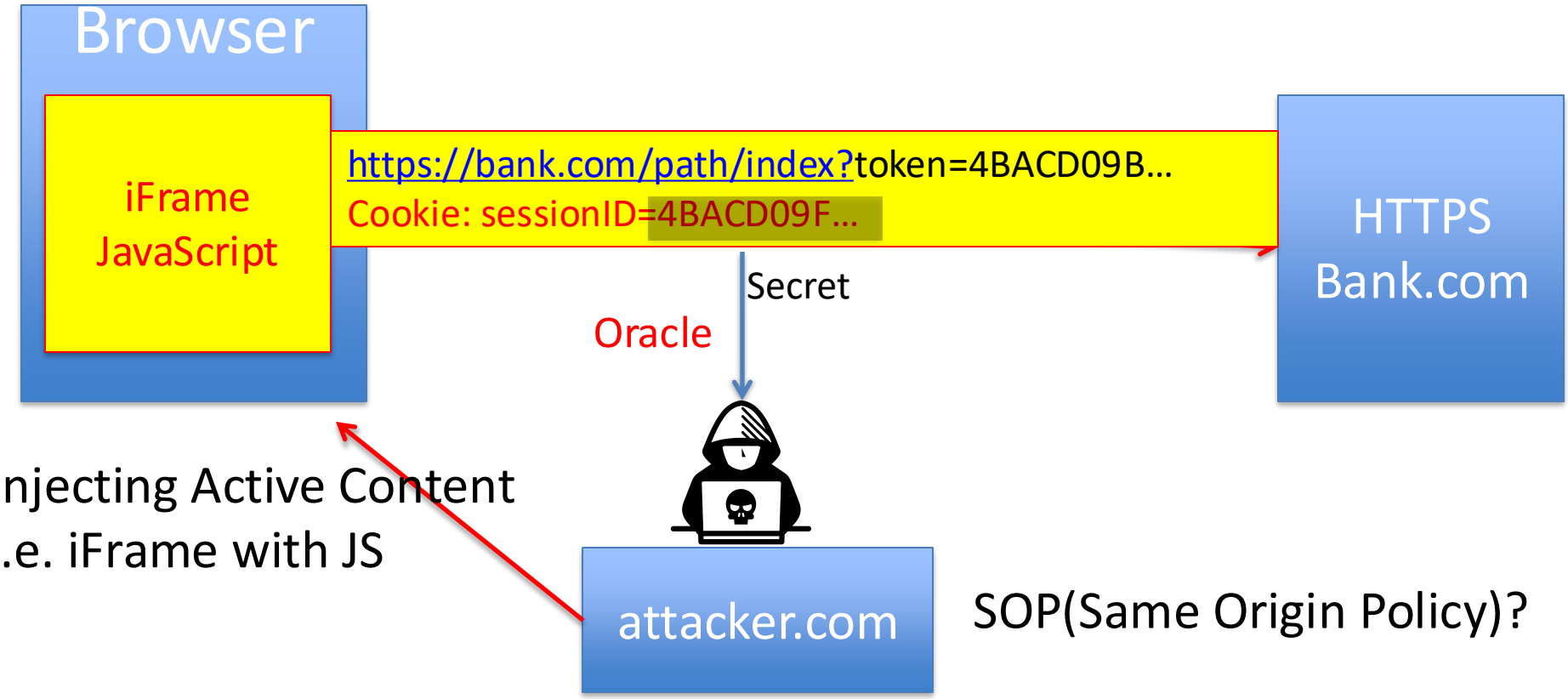
This approach is inferior to supporting secure renegotiation. First, some deployments actually need renegotiation (typically when deploying client certificate authentication). Second, not supporting secure renegotiation promotes renegotiation uncertainty on the Web, effectively preventing users from protecting themselves.

Insecure renegotiation remediation timeline



Web Attack: Chosen-plaintext attack

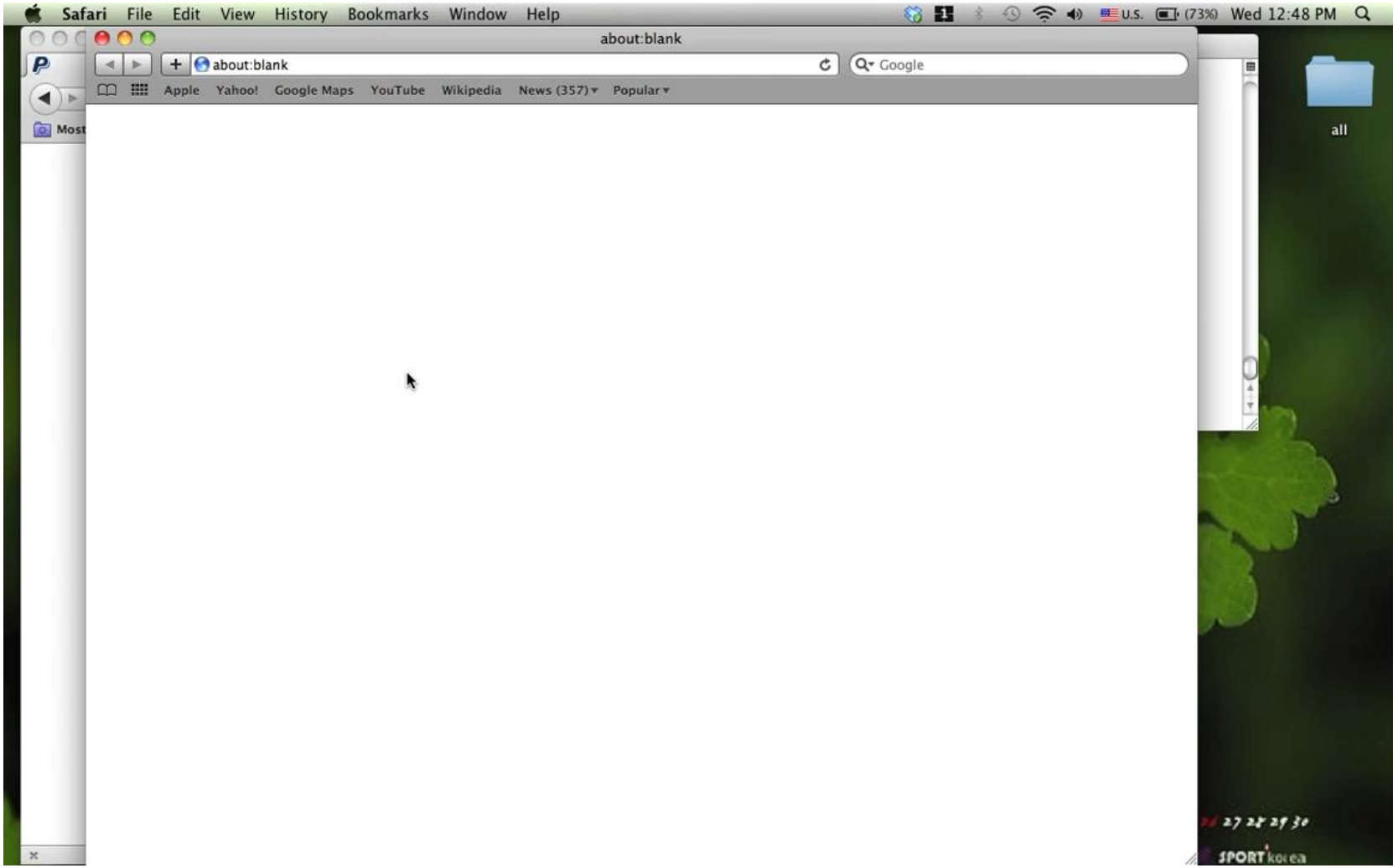
- Man in the middle, or On-Path



Threat model for attacks such as BEAST, CRIME, BEACH, TIME ….

Demo: BEAST attack

Thai Duong Juliano Rizzo, 2011



https://vnhacker.blogspot.com/2011/09/beast.html

BEAST(**B**rowser **E**xploit **A**gainst **S**SL/**T**LS)

ECB(Electronic Code Book)

- $C_1=E(M_1),\ C_2=E(M_2)$
- $C_1=C_2 \Rightarrow M_1=M_2$

Downgrade

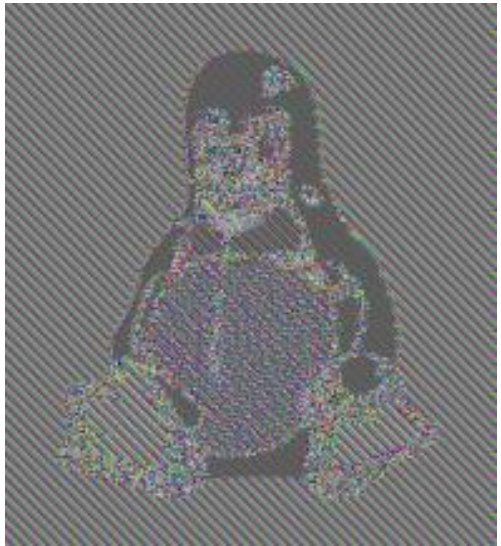
Predictable IV

CBC(Cipher Block Chaining)

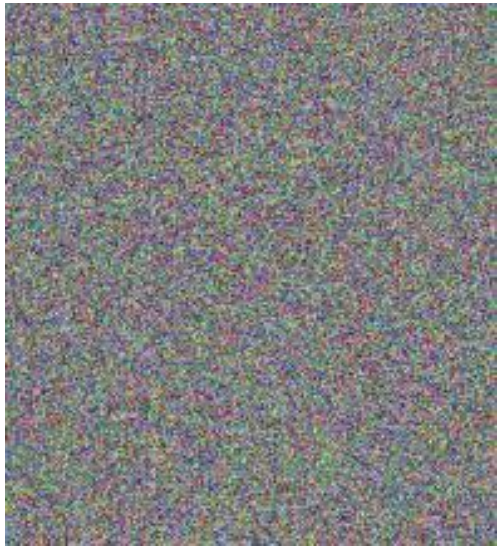
- $C_1= E(IV_1 \oplus M_1),\ IV_2=C_1$
- $C_2= E(IV_2 \oplus M_2)$,

$(IV_1 \neq IV_2) \wedge (C_1=C_2) \Rightarrow M_1=M_2$

ECB

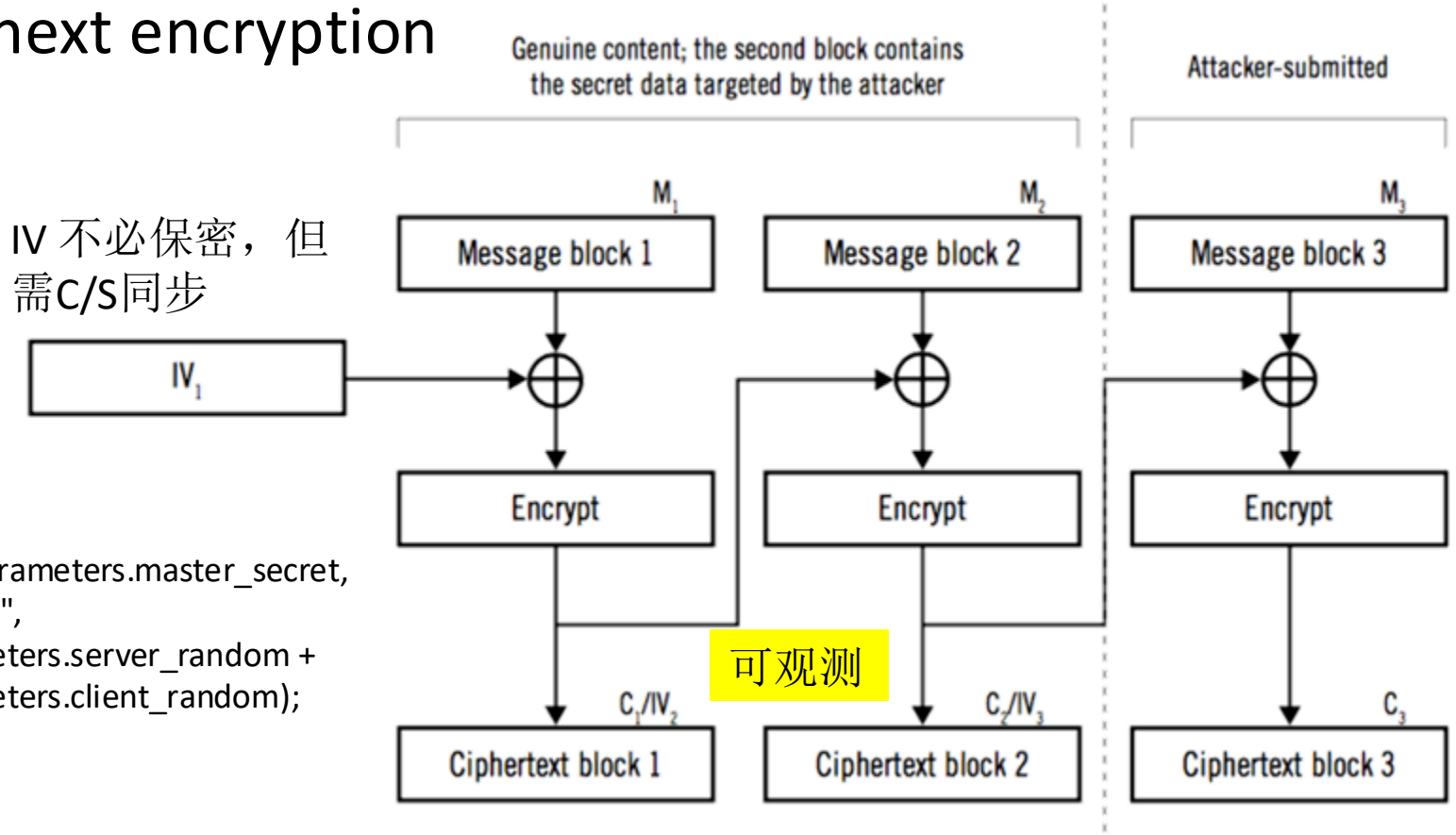


CBC



BEAST(CVE-2011-3389, 2011)

- Phillip Rogaway, 2002
- TLS 1.0 - , predictable IV(implicit IV): the encrypted block from the previous TLS record is used as the IV for the next encryption



key_block = PRF(SecurityParameters.master_secret, "key expansion", SecurityParameters.server_random + SecurityParameters.client_random);

BEAST

$$C_2 = E(M_2 \oplus IV_2) = E(M_2 \oplus C_1)$$
$$C_3 = E(M_3 \oplus IV_3) = E(M_3 \oplus C_2)$$

用猜测的M_g构造：$M_3 = M_g \oplus C_1 \oplus C_2$

可以捕获流量得到

$$C_3 = E(M_3 \oplus C_2) :$$
$$= E(M_g \oplus C_1 \oplus C_2 \oplus C_2) = E(M_g \oplus C_1)$$

BEAST

$$C_2 = E(M_2 \oplus IV_2) = E(M_2 \oplus C_1)$$
$$C_3 = E(M_3 \oplus IV_3) = E(M_3 \oplus C_2)$$

用猜测的M_g构造：$M_3 = M_g \oplus C_1 \oplus C_2$

可以观察到

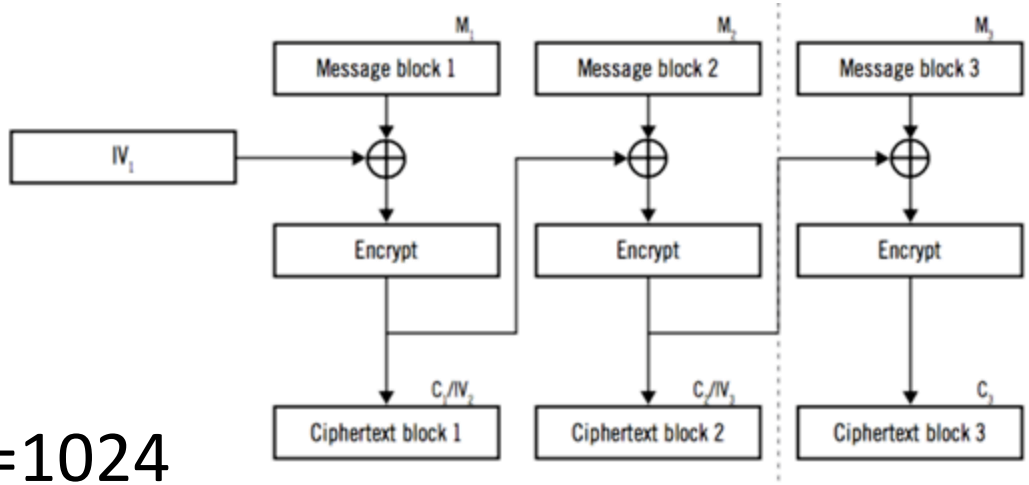
$$C_3 = E(M_3 \oplus C_2) :$$
$$= E(M_g \oplus C_1 \oplus C_2 \oplus C_2) = E(M_g \oplus C_1)$$

如果猜测正确 (即$M_g = M_2$)，则$C_3 = C_2$

$$C_3 = E(M_g \oplus C_1) = E(M_2 \oplus C_1) = C_2$$

攻击复杂性

- AES: 128 bits
- 2^{128} ?
- Cookie : Base64, 2^6
- Byte by byte: $16*2^6=1024$



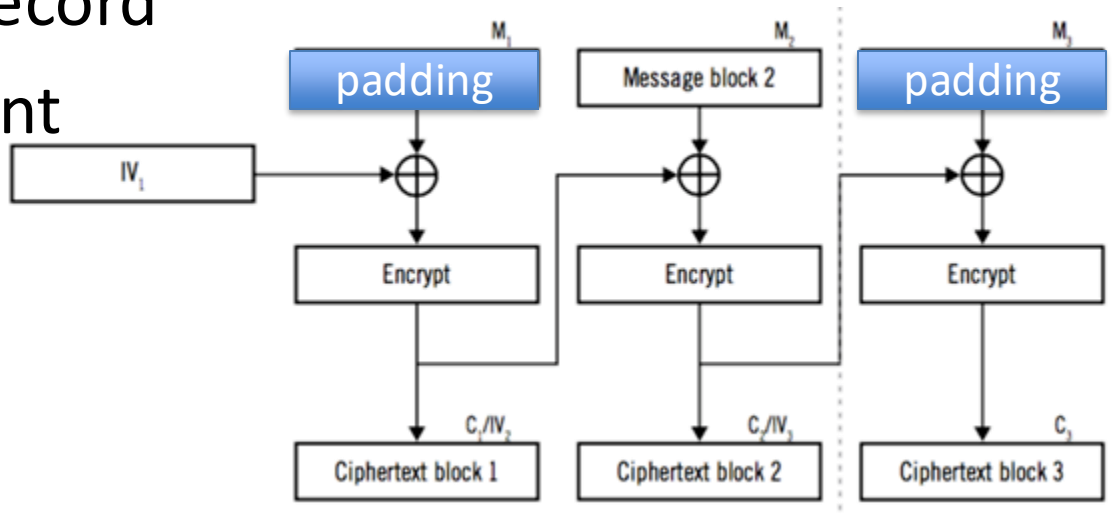
GET /PADDING_TO_FILL_RECORD<...>PADDING HTTP/1.1\r\n
Host: example.com\r\nAccept-Encoding:.... Cookie: sessionid=7

--- record 2 ---
xc89f94wa96fd7cb4cb0031ba249ca2
Accept-Language: en-US,en;q=0.8

通过填充URL，使得最后一块（16bytes）只有一个字节是未知的

Mitigation

- Disable CBC?
  - Stream cipher, RC4 ? recover plaintext with 2^26 sessions. Disabled in RFC 7645, 2015
- Empty TLS record:
  - injecting an empty (no data) TLS record before each real TLS record
  - Fast Deployment
  - IE problem



Explicit IVs, TLS1.1+

In order to make the PRF as secure as possible, it uses two hash algorithms in a way that should guarantee its security if either algorithm remains secure.

First, we define a data expansion function, P_hash(secret, data) that uses a single hash function to expand a secret and seed into an arbitrary quantity of output:

```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...
```

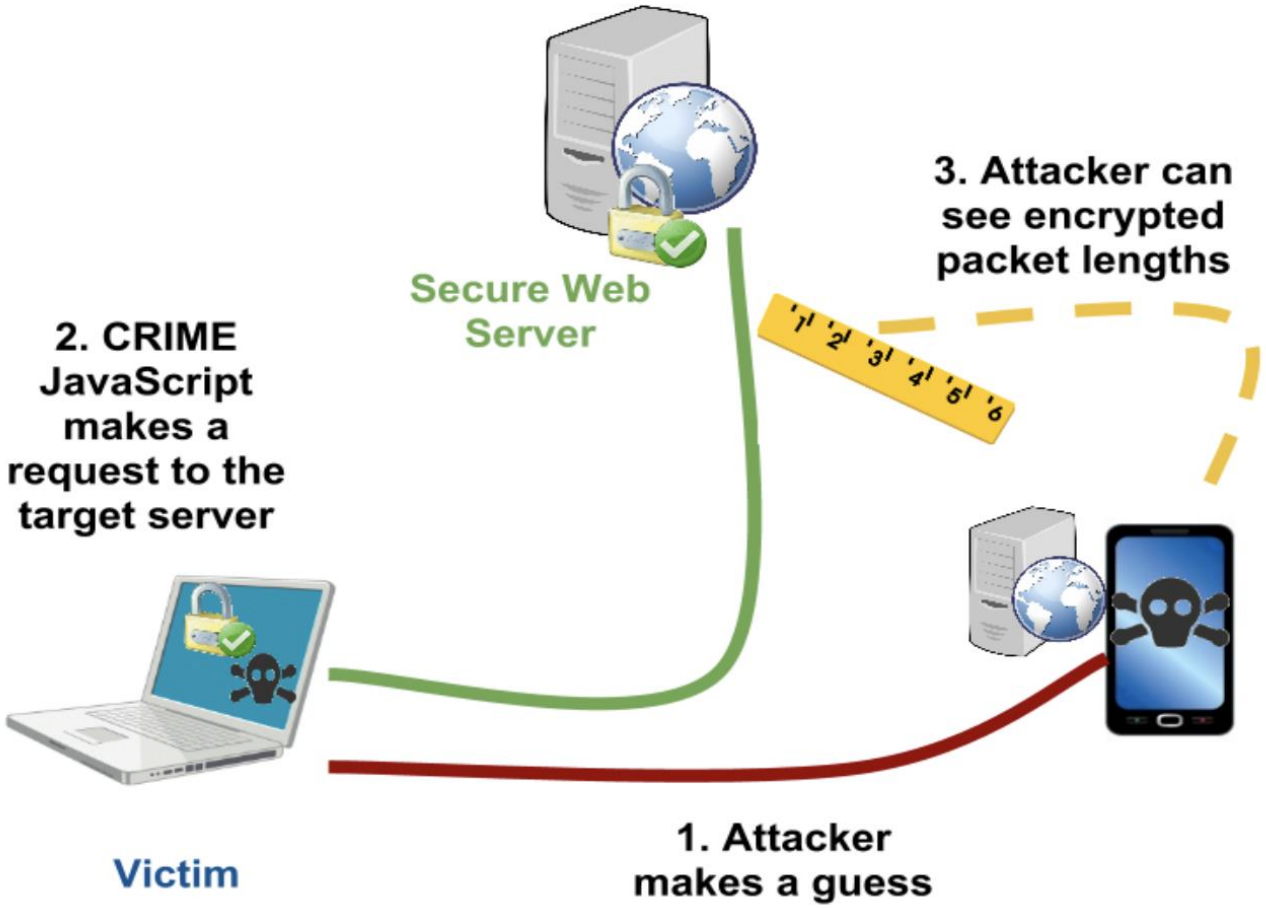
Where + indicates concatenation.

A() is defined as:

```
A(0) = seed
A(i) = HMAC_hash(secret, A(i-1))
```

P_hash can be iterated as many times as is necessary to produce the required quantity of data. For example, if P_SHA-1 is being used to create 64 bytes of data, it will have to be iterated 4 times (through A(4)), creating 80 bytes of output data; the last 16 bytes of the final iteration will then be discarded, leaving 64 bytes of output data.

Compression Side Channel Attacks (CRIME) , 2012



Compression Side Channel Attacks (CRIME) , 2012

http://www.ekoparty.org/archive/2012/CRIME_ekoparty2012.pdf

- Inject chosen plaintext into a victim's requests
- Measure the size of encrypted traffic

```
GET /JSESSIONID=X HTTP/1.1
Host: www.example.com
Cookie: JSESSIONID=B3DF4B07AE33CA
```

Incorrect guess:
73 bytes compressed

```
GET /JSESSIONID=B HTTP/1.1
Host: www.example.com
Cookie: JSESSIONID=B3DF4B07AE33CA
```

Correct guess:
72 bytes compressed

ME in CRIME is mass exploitation

- Worked for 45% of browsers: Chrome and Firefox.
- Worked for all SPDY servers: Gmail, Twitter, etc.
- Worked for 40% of SSL/TLS servers: Dropbox, GitHub, etc.

CRIME is the new BEAST

- BEAST opened the path to CRIME
  - Easy to perform chosen-plaintext attack against HTTPS.
  - Use URL path to decrypt cookie.
  - Move data across layer boundary.

- What's new?
  - SSL compressed record length info-leak, instead of CBC mode with chained IVs vulnerability.
  - New boundaries: compressor window size and TLS record size, instead of block cipher's block size.

Compression Side Channel Attacks (CRIME) , 2012

http://www.ekoparty.org/archive/2012/CRIME_ekoparty2012.pdf

- Inject chosen plaintext into a victim's requests
- Measure the size of encrypted traffic

```
GET /JSESSIONID=X HTTP/1.1
Host: www.example.com
Cookie: JSESSIONID=B3DF4B07AE33CA
```

Incorrect guess:
73 bytes compressed

```
GET /JSESSIONID=B HTTP/1.1
Host: www.example.com
Cookie: JSESSIONID=B3DF4B07AE33CA
```

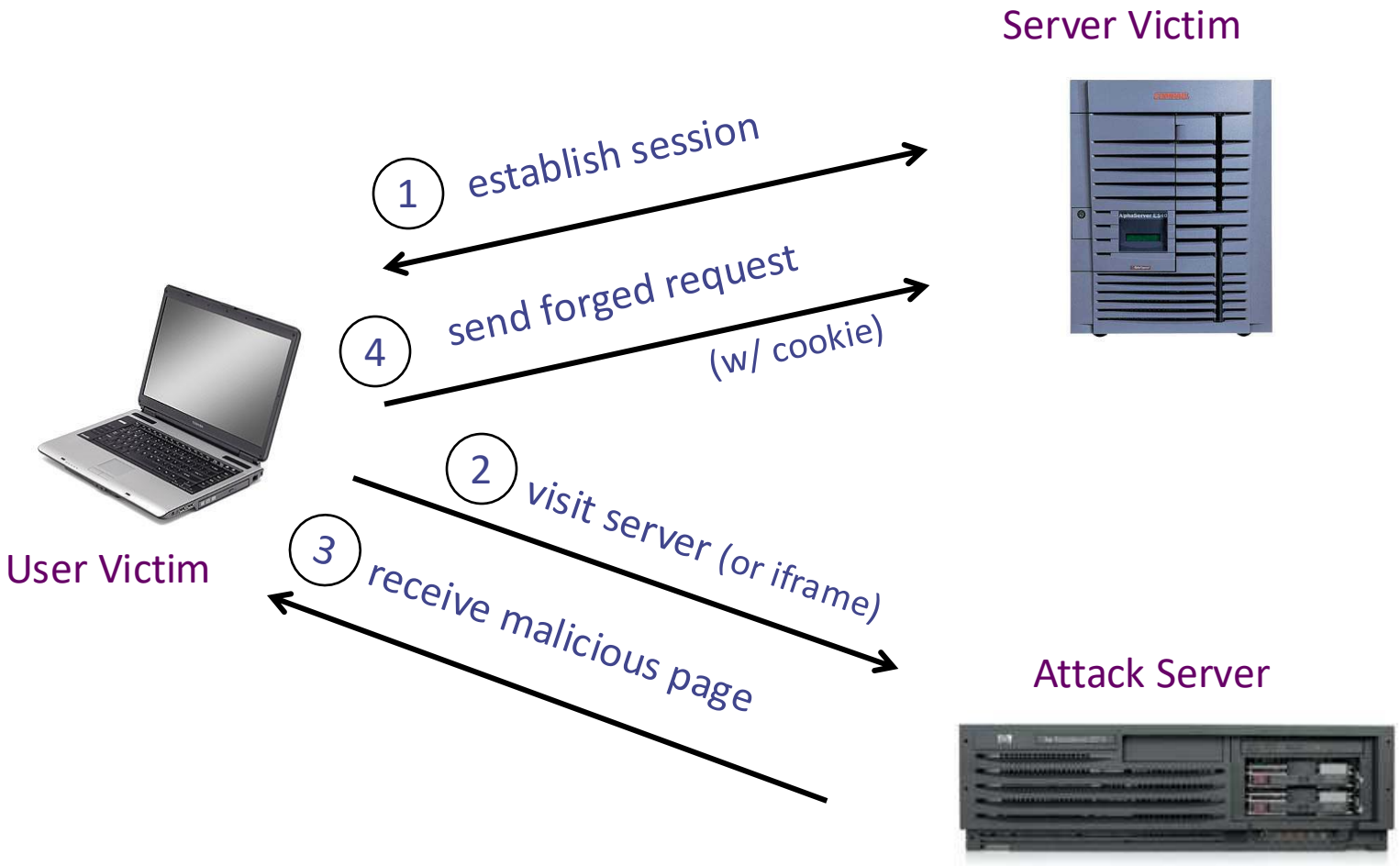
Correct guess:
72 bytes compressed

Mitigation: disable TLS compression

BREACH :Reviving the CRIME attack, 2013

- **Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext(BREACH)**
- A web application is vulnerable if :
  - Be served from a server that uses HTTP-level compression
  - Reflect user-input in HTTP response bodies
  - Reflect a secret (such as a CSRF token) in HTTP response bodies

Cross Site Request Forgery



Q: how long do you stay logged in to Gmail? Facebook? ….

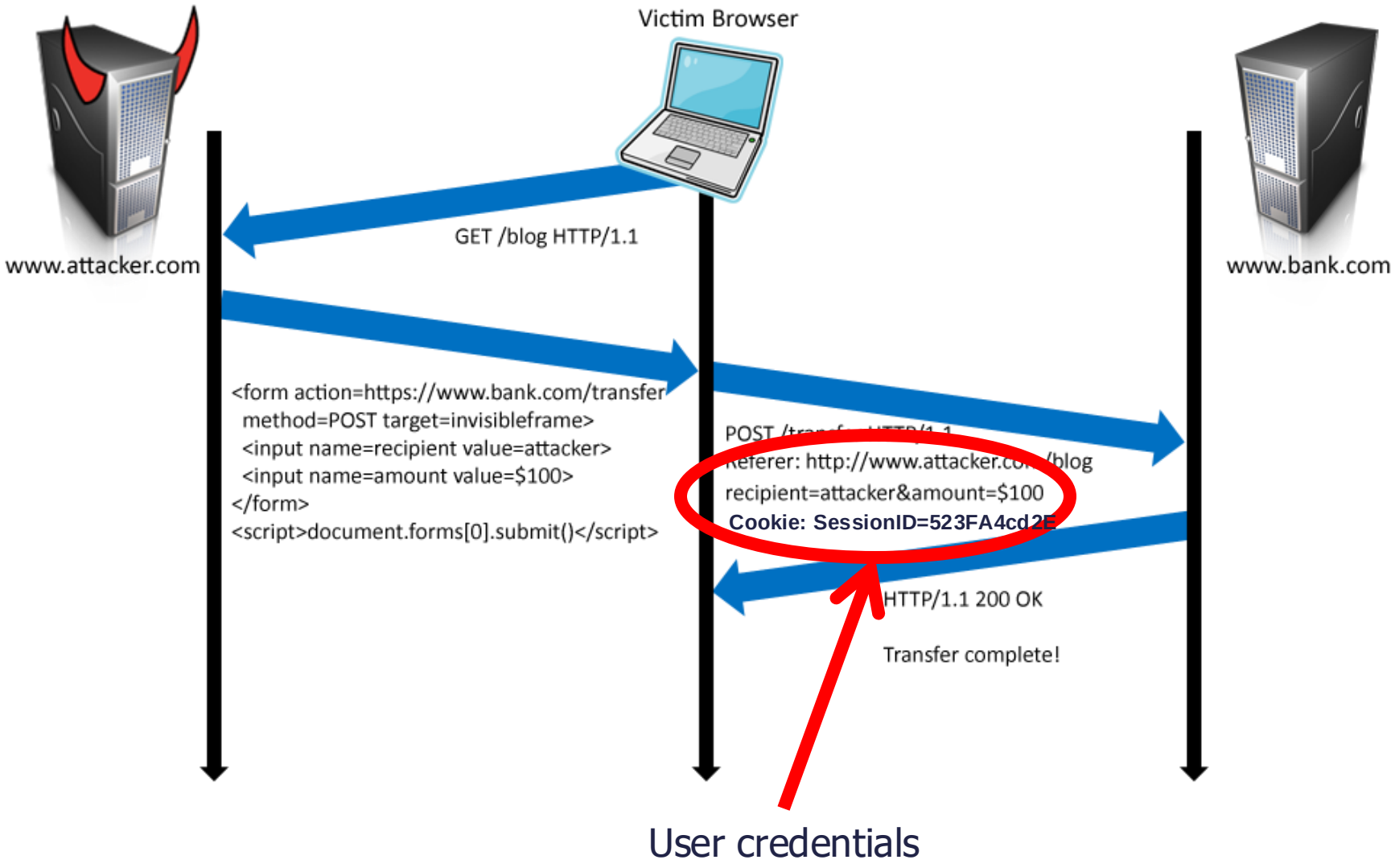
Cross Site Request Forgery (CSRF)

- Example:
  - User logs in to bank.com
    - Session cookie remains in browser state
  - User visits another site containing:

```
<form name=F action=http://bank.com/BillPay.php>
  <input name=recipient value=badguy> …
<script> document.F.submit(); </script>
```

  - Browser sends user auth cookie with request
    - Transaction will be fulfilled
- Problem:
  - cookie auth is insufficient when side effects occur

Form post with cookie



Defense : CSRF_Token


```
POST /transfer/do.jsp?from=victim&to=attacker  HTTP/1.0
.....
<BODY>
        <Form>   <input  type=hidden name=CSRF_Token
                                        value=some_secret />
         </Form>
</BODY>
```

Response, Reflect, CSRF token

Compromised HTTP request:

POST /owa/?ae=Item&t=IPM.Note&a=New&id=canary=<guess>

***HTTP response*:**

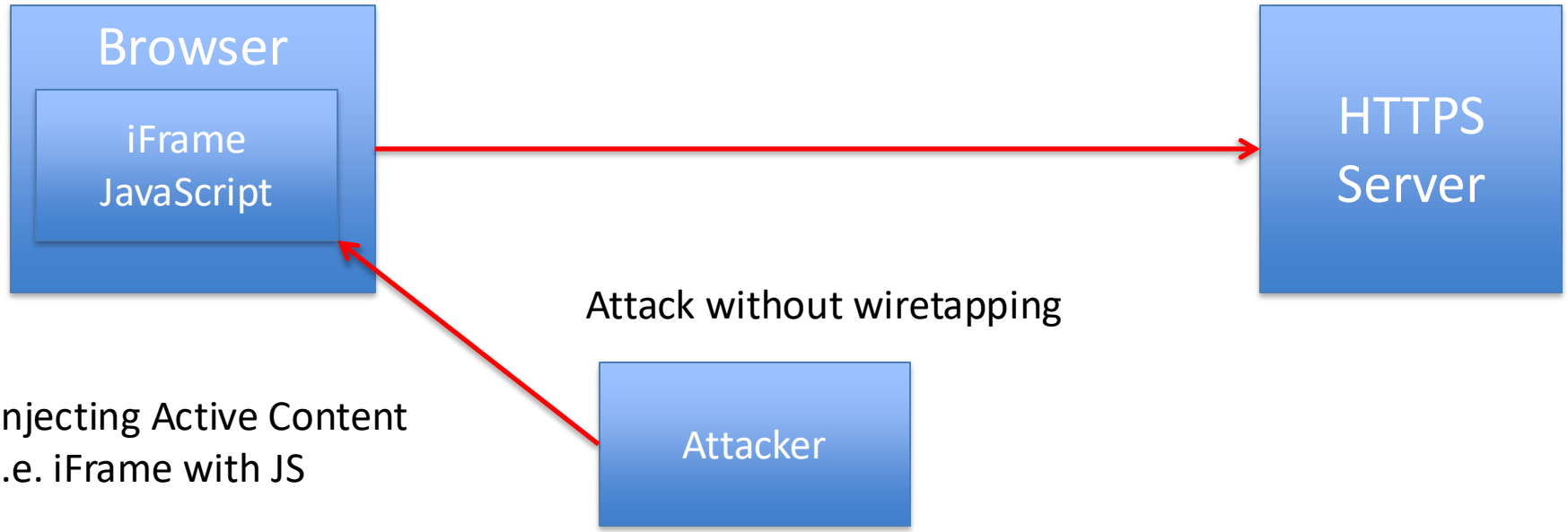
https://malbot.net:443/owa/forms/
basic/BasicEditMessage.aspx?ae=Item&t=IPM.Note&
amp;a=New&id=canary=<guess>

...

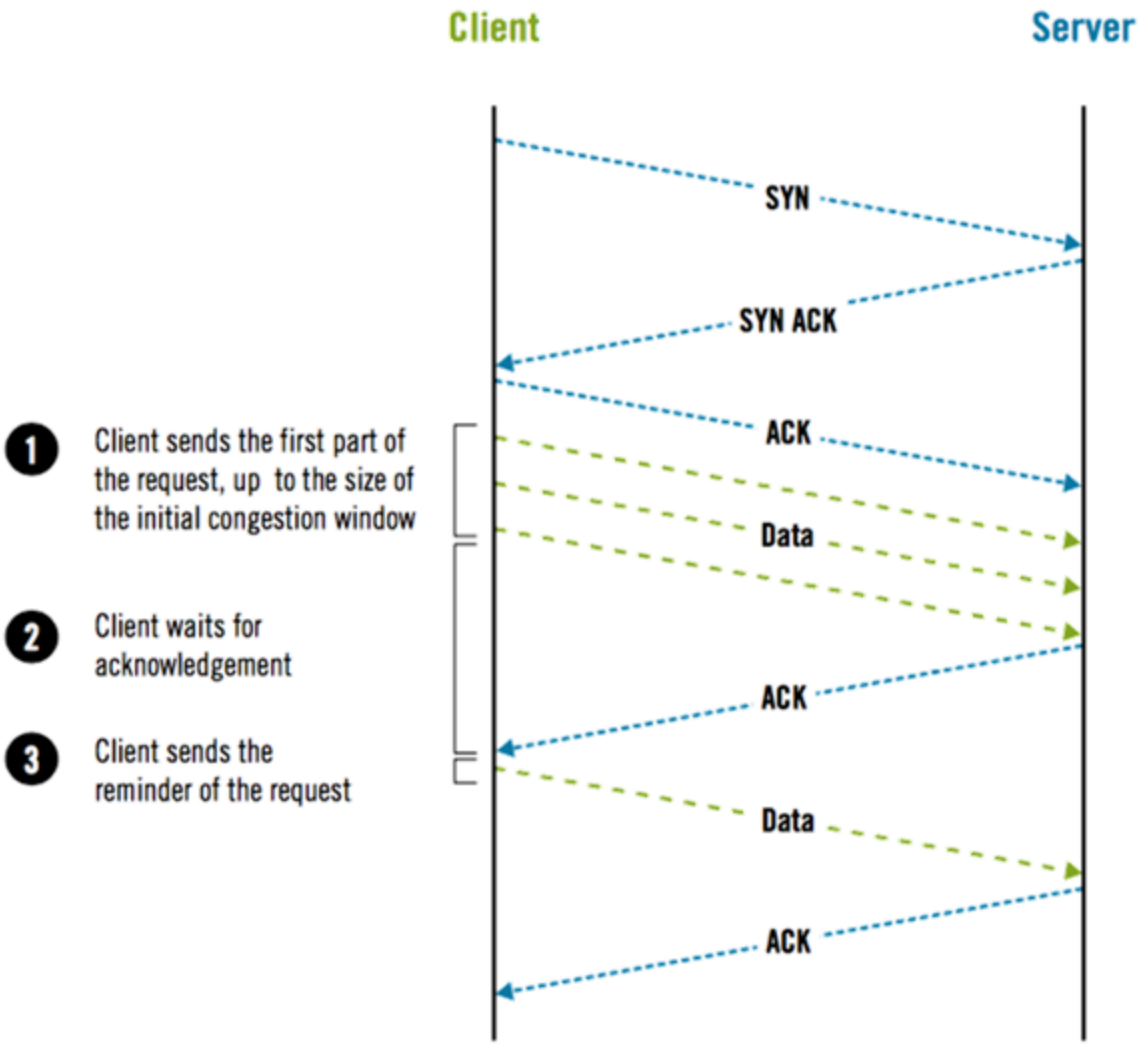
<td nowrap id="tdErrLgf"><a href="logoff.owa?
canary=d634cda866f14c73ac135ae858c0d894">Log Off</td>

TIME

- Choose plaintext attack



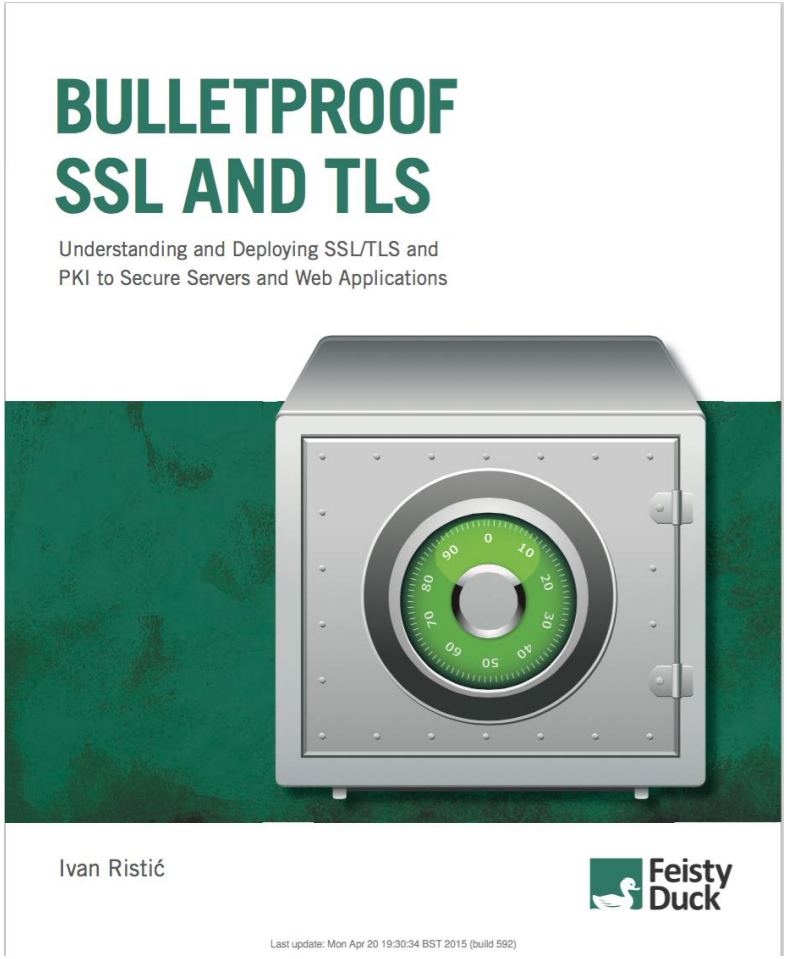
TIME: TCP Congestion Window as a timing oracle



Reading

- Ivan Ristic: Bulletproof SSL and TLS
  - **Chapter 6: Implementation Issues**
  - Chapter 7 Protocol attacks

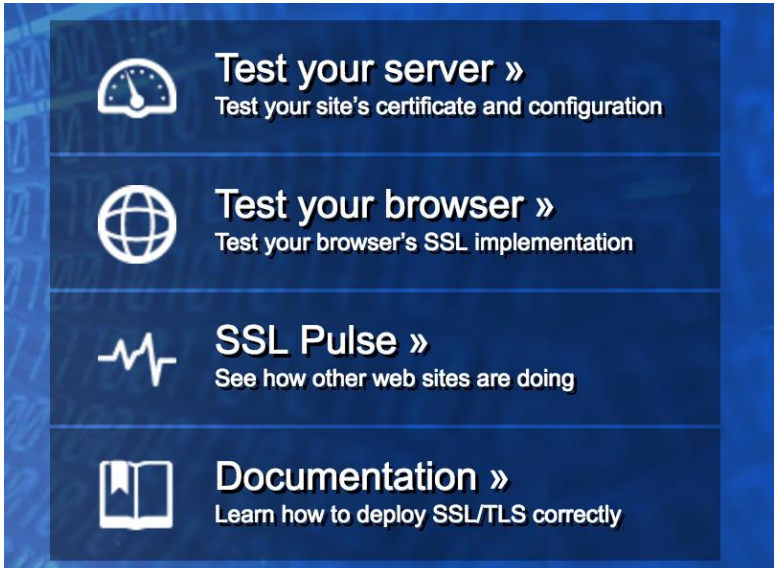
Reference

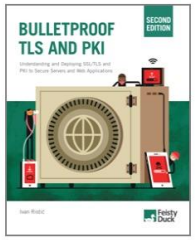


Bulletproof SSL and TLS
by Ivan Ristić
Feisty Duck

SSL Labs

https://www.ssllabs.com/





BULLETPROOF TLS AND PKI

Understanding and deploying SSL/TLS and PKI to secure servers and web applications, by Ivan Ristić