SSL/TLS in a nutshell

State of the art and focus on the Record Protocol

Observation and analysis of the HTTPS ecosystem

Implementation aspects and focus on the parsing problem

Conclusion and perspectives
SSL/TLS in a nutshell
SSL/TLS: a security protocol providing
▶ server (and client) authentication
▶ data confidentiality and integrity

Two phases
▶ Handshake Protocol
   ▶ algorithm negotiation
   ▶ server authentication
   ▶ key exchange
▶ Record Protocol
   ▶ application data exchanges
SSL/TLS: a basic block of Internet security

A 20-year old protocol

- originally designed by Netscape to secure HTTP connections (SSL)
- maintained since 2001 by the IETF (TLS)
- now used for a broad spectrum of applications
  - to secure almost every cleartext protocols
  - to provide VPNs
  - to authenticate peers in an EAP exchange
The complexity of the protocol

The specifications (50+ RFCs) describe many variants

- 5 protocol versions
- 300+ ciphersuites
- 20+ extensions
- *interesting* features
  - compression
  - renegotiation
  - session resumption (2 methods)

A rich subject to study from different points of view
Part I
State of the art and focus on the Record Protocol
Overview

Many flaws and attacks devised since 1995

- it is hard to find relevant categories
- several issues may be considered in different categories

The proposed categories are:

- flaws affecting the Handshake Protocol
- attacks against the Record Protocol
- certificate-related issues
- implementation bugs

Publications describing the state of the art: [SSTIC 12, SSTIC 15]
Flaws affecting the Handshake Protocol

1994
SSLv2
1995
SSLv3
2001
TLS 1.0
2006
TLS 1.1
2008
TLS 1.2
2016?
TLS 1.3

Weak crypto parameters
- FREAK [BBD+15]
- LogJam [ABD+15]
- First MD5 collisions [WY05]
- SLOTH [BL16]

Specification flaws
- Bleichenbacher [Ble98]
- Insecure renegotiation
- 3Shake [BDF+14]
- KCI [HGFS15]

Cross-protocol attacks
- RSA/DHE confusion [WS96]
- DHE/ECDHE confusion [MVVP12]
- FREAK [BBD+15]
Attacks against the Record Protocol

CBC mode
- Rogaway [Rog95]
- Vaudenay [Vau02]
- BEAST [DR11]
- POODLE [MDK14]
- Lucky 13 [AP13]

Weak algorithms
- First RC4 biases
- TLS Plaintext Recovery with RC4 [ABP+13,IOWM13,GPdM15]
- Sweet32

Compression
- CRIME [RD12]
- TIME [BS13]
- BREACH [PHG13]
Description of the Record Protocol

Plaintext $P$ with $|P| < 2^{14}$

Compression (optional)

Compressed $C$ with $|C| < |P| + 1024$

MAC

$C$ | $MAC$

Encryption (XOR)

MAC’ed then Encrypted record

Stream cipher mode

MAC

$C$ | $MAC$

Padding

$C$ | $MAC$ | $Pad$

Encryption (CBC mode)

MAC’ed then Padded then Encrypted record

CBC mode

AEAD step

Authenticated and Encrypted record

AEAD mode
Proofs of concept against the Record Protocol

Considered attacks

- BEAST, exploiting CBC using implicit IV
- Lucky 13, a CBC padding oracle
- POODLE, an SSLv3-specific CBC padding oracle
- plaintext recovery using RC4 statistical biases
- CRIME and TIME, compression side-channel (client-side)
- TIME and BREACH, compression side-channel (server-side)
Proofs of concept against the Record Protocol

Considered attacks

- BEAST, exploiting CBC using implicit IV
- Lucky 13, a CBC padding oracle
- POODLE, an SSLv3-specific CBC padding oracle
- Plaintext recovery using RC4 statistical biases
- CRIME and TIME, compression side-channel (client-side)
- TIME and BREACH, compression side-channel (server-side)

All the attacks were illustrated by a PoC targeting HTTPS

- Powerful (but realistic) attacker
- Typical targets are authentication cookies
BEAST: CBC using implicit IV

Hypotheses:

- the connection uses CBC with implicit IV (TLS < 1.1)
- the attacker is able to observe encrypted packets
- the plaintext is partially controlled, adaptively
- multiple connections containing the secret can be triggered

Proposed countermeasures:

- use TLS 1.1
- use AEAD suites (requires TLS ≥ 1.2)
- use RC4
- split the records
RC4 statistical biases

Hypotheses:

▶ the connection uses RC4
▶ the attacker is able to observe encrypted packets
▶ multiple connections containing the secret can be triggered

Proposed countermeasures:

▶ use AEAD suites (requires TLS ≥ 1.2)
▶ use CBC mode
▶ use another streamcipher
▶ randomise the secret location
Record Protocol: the long-term solution

Plaintext $P$  
$|P| < 2^{14}$

Compression (optional)  
Compressed $C$  
$|C| < |P| + 1024$

MAC

$C$  
MAC

Encryption (XOR)

MAC’ed then Encrypted record

$RC4$  
Stream cipher mode

MAC

$C$  
MAC

Padding

MAC’ed then Padded then Encrypted record

$CBC$ mode

AEAD step

MAC

$C$  
MAC  
$Pad$

Authenticated and Encrypted record

$AEAD$ mode
Record Protocol: the long-term solution

Plaintext $P$ 

Compression (disabled)

Compressed $C$

MAC $C$  $MAC$

Encryption (XOR)

MAC’ed then Encrypted record

RC4 Stream cipher mode

MAC $C$  $MAC$  $Pad$

Padding

MAC’ed then Padded then Encrypted record

CBC mode

Authenticated and Encrypted record

AEAD mode
Record Protocol: when TLS 1.2/AEAD is not an option

In the absence of the long-term solution (e.g. for compatibility reasons)
  ▶ specific short-term fixes exist for most attacks
  ▶ we propose to avoid the repetition as a defense-in-depth mechanism

The masking principle (borrowed from the side-channel community):
  ▶ instead of sending a secret $s$
  ▶ draw a random string $m$ of the same length as $s$
  ▶ send $(m, s \oplus m)$

  ▶ the *intended* value remains the same
  ▶ but the representation is different each time

Publication describing MCookies and similar countermeasures: [ASIA-CCS 15]
Application to HTTP cookies: MCookies
Application to HTTP cookies: MCookies
Evaluation of MCookies

Security evaluation

- MCookies cover all first-order attacks...
- as long as the attacker does not tamper with packets

Performance impact

- MCookies used on secure httpOnly cookies
- 4 % overhead on overall HTTPS traffic
Evaluation of MCookies

Security evaluation

- MCookies cover all first-order attacks...
- as long as the attacker does not tamper with packets

Performance impact

- MCookies used on secure httpOnly cookies
- 4 % overhead on overall HTTPS traffic

MCookies with client-side support

- the overhead is reduced by half
- all attacks (including active ones) are thwarted
Part II
Observation and analysis of the HTTPS ecosystem
The motivation behind HTTPS campaigns

The main goal: get concrete data about SSL/TLS usage
- supported versions and features
- feature intolerance
- certificate quality
- at the time (2010-2011), no public datasets

Why choose HTTPS?
- the first and still the major use of SSL/TLS
- HTTPS servers expect to be contacted by strangers
- a diversified ecosystem
Available methodologies

Different ways to get SSL/TLS data:

- IPv4 SYN scan on 443/tcp, followed by SSL/TLS connections
- SSL/TLS connections towards a list of known domain names
- capture of real SSL/TLS traffic from consenting users

We chose the first method

- the active probing lets us choose the sent stimuli
- not relying on domain names gives access to a wide diversity of servers

Drawbacks

- distribution of the campaign over 3 weeks
- no support for SNI / virtual hosting
Big-picture data regarding our campaigns

About our 2011 campaigns:
- 26 M hosts with an open 443/tcp port
- 7 different stimuli sent
- 11 M answered at least once with SSL/TLS messages
- 140 GB of raw data

The article describing the methodology and the results on 2010-2011 campaigns: [ACSAC 12]
The motivation behind concerto

The tools used to produce the data for [ACSAC 12]

- parsifal to parse the answers
- (mostly undocumented or even not versionned) various scripts
The motivation behind **concerto**

The tools used to produce the data for [ACSAC 12]

- parsifal to parse the answers
- (mostly undocumented or even not versionned) various scripts

In 2015, we tried to run similar analyses on new campaigns

- problem: several criteria had to evolve
- how to compare the situation now and then?
The motivation behind **concerto**

The tools used to produce the data for [ACSAC 12]
- **parsifal** to parse the answers
- (mostly undocumented or even not versionned) various scripts

In 2015, we tried to run similar analyses on new campaigns
- problem: several criteria had to evolve
- how to compare the situation now and then?

The **concerto** way, towards reproducible analyses
- keep the raw data and the associated metadata
- automate the analysis process
- run it from scratch when needed
concerto, step by step

Context preparation
- NSS certificate store extraction from source code
- metadata injection (stimuli, certificate store)

Answer injection
- answer type analysis
- raw certificate extraction

Certificate analysis
- certificate parsing
- building of all* possible chains

Statistics production
- TLS parameters, certificate chain quality, server behavior
Implementation choices

Design rationale

▶ store enriched data in CSV tables
▶ split data processing into simple tools
▶ avoid tools requiring a global view when possible

*Challenges

▶ X.509v1 certificates generated by appliances
  ▶ 140,000 self-signed *distinct* certificates
  ▶ containing the *same* subject (and issuer)
  ▶ 20 billion signatures to check
▶ the max-transvalid option

concerto is an open-source project available on GitHub
Dataset selection

<table>
<thead>
<tr>
<th>Campaign type</th>
<th>Date</th>
<th>Available</th>
<th>Retained</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFF</td>
<td>IP</td>
<td>2010</td>
<td>yes</td>
</tr>
<tr>
<td>Our campaigns</td>
<td>IP</td>
<td>2010-2014</td>
<td>yes</td>
</tr>
<tr>
<td>[HBKC11]</td>
<td>IP + DN + PO</td>
<td>2011</td>
<td>partially</td>
</tr>
<tr>
<td>SSLPulse</td>
<td>DN</td>
<td>recurring since 2012</td>
<td>no</td>
</tr>
<tr>
<td>Internet Census</td>
<td>?</td>
<td>2012</td>
<td>yes</td>
</tr>
<tr>
<td>[DWH13]</td>
<td>IP + DN</td>
<td>recurring since 2013</td>
<td>yes</td>
</tr>
</tbody>
</table>

IP IPv4 SYN scan followed by active probing  
DN Active probing on a list of Domain Names  
PO Passive Observation

certoo offers a portable way to study these different datasets

The results allow us to study trends from 2010, 2011, 2014 and 2015
Big picture

Number of servers by category

- Open 443/tcp
- TLS hosts
- Trusted hosts
- EV hosts


Number of servers:
- Open 443/tcp: 0 M, 10 M, 20 M, 30 M, 40 M, 50 M
- TLS hosts: 0 M, 10 M, 20 M, 30 M, 40 M, 50 M
- Trusted hosts: 0 M, 10 M, 20 M, 30 M, 40 M, 50 M
- EV hosts: 0 M, 10 M, 20 M, 30 M, 40 M, 50 M
Evolution of TLS parameters

- **TLS 1.0**: 96% in 2010, 96% in 2011, 97% in 2014, 49% in 2015
- **TLS 1.1**: 0% in 2010, 0% in 2011, 0% in 2014, 0% in 2015
- **TLS 1.2**: 30% in 2010, 67% in 2011, 47% in 2014, 47% in 2015
- **SSLv3**: 0% in 2010, 0% in 2011, 0% in 2014, 0% in 2015
Certificate chain quality (1/2)

- RFC Compliant
- Unordered
- Transvalid
- Incomplete

<table>
<thead>
<tr>
<th>Year</th>
<th>RFC Compliant</th>
<th>Unordered</th>
<th>Transvalid</th>
<th>Incomplete</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td>58%</td>
<td></td>
<td></td>
<td>37%</td>
</tr>
<tr>
<td>2011</td>
<td>55%</td>
<td></td>
<td></td>
<td>39%</td>
</tr>
<tr>
<td>2014</td>
<td>51%</td>
<td></td>
<td>11%</td>
<td>35%</td>
</tr>
<tr>
<td>2015</td>
<td>50%</td>
<td>13%</td>
<td></td>
<td>34%</td>
</tr>
</tbody>
</table>
Certificate chain quality (2/2)

Several results about chain validity periods:

- for trusted hosts most chains are valid between 1 and 5 years...
- yet some of them were valid for 20 years

- for TLS hosts in general, 10-year certificates are common
- the record is a 1000-year validity period

RSA is still the most common public key algorithm used:

- we encountered 16,384-bit keys...
- the standard for trusted hosts went from 1024-bit in 2010 to 2048-bit keys in 2015
Server behavior

Beyond the use of different certificate stores, the contribution of our approach in [ACSAC 12] is the use of multiple stimuli:

▶ using different versions
▶ including extensions or not
▶ proposing restricted sets of ciphersuites

Results:

▶ EC- and TLS 1.2-intolerance has regressed between 2011 and 2014
▶ The proportion of HTTPS servers accepting SSLv2 is still important in 2014 (40 %)
  ▶ all vulnerable to DROWN attack
  ▶ the situation is worse in practice (SMTPS servers in particular)
Part III
Implementation aspects and focus on the parsing problem
The motivation behind our parsers

How to handle SSL/TLS data and the embedded X.509 certificates?

▶ reuse existing stacks
  ▶ limited scope (we don’t want to reject unknown options)
  ▶ liberal code (we want to see invalid parameters)
  ▶ fragile implementations (the input might be challenging)

▶ write many parsers in different languages

▶ develop a framework in OCaml called parsifal
  ▶ the idea: automate tedious parts via code generation
  ▶ result: a solution to quickly write robust and efficient parsers
parsifal

Robustness of the code

- OCaml is a statically-typed language
- automatic memory management
- exhaustive pattern matching as a reliable safeguard

Efficiency

- writing concise code, even to describe complex structures
- the result is rather fast

Limitations

- mostly suited for standalone analysis tools
- integration within existing projects might be hard

parsifal led to several publications: [CRiSIS 13, SSTIC 13, SPW 14a]

parsifal is an open-source project available on GitHub
In parsifal we trust

Many unparsed certificates with our early parsers
- we added support for corner cases
- even illegitimate, but popular, ones (with a warning)

What are the remaining files?
- corrupted files
- private keys...
In *parsifal* we trust

Many unparsed certificates with our early parsers

- we added support for corner cases
- even illegitimate, but popular, ones (with a warning)

What are the remaining files?

- corrupted files
- private keys...

Similarly, we encountered interesting invalid certificate signatures:

- $C$ and $C'$, differing only on extensions...
- with the same signature

Anomalies signaled by our tools are usually something worth investigating...
2014: a tough year for TLS implementations

In 2014, all major TLS stacks were affected by a critical vulnerability

- February: goto fail in Apple
- February: goto fail in GnuTLS
- April: Heartbleed in OpenSSL
- June: Early CCS in OpenSSL
- August: Bleichenbacher revival attack in JSSE
- September: Universal signature forgery in NSS, CyaSSL and PolarSSL
- November: remote code execution in SChannel (MS)

A thorough analysis of implementation flaws has been submitted to CT-RSA 17
Classical programming errors

Bugs in this category:
- memory management errors (Heartbleed)
- trivial mistakes in the logic (goto fail)
- missing checks (BasicConstraints)

Lessons to learn:
- some mistakes are repeated in different independent code bases
- it may be time to use better languages / tools
- negative and non-regression tests should be improved and shared
Parsing bugs

Bugs in this category:
- ASN.1 DER encoding (null chars, signature forgery)
- TLS record splitting (OpenSSL downgrade attack, *Heartbleed*)

Lessons to learn:
- parsing is often overlooked
- simple specs are beautiful... and more secure
The real impact of obsolete cryptography on security

Bugs in this category:
- MAC-then-Encrypt is hard to implement safely
- similarly, RSA encryption using PKCS#1 v1.5 is still a problem

Lessons to learn:
- obsolete and dangerous cryptographic schemes must be removed...
- including in the code base...
- without any delay (TLS 1.1 should have included EtM)
The consequences of complex state machines

Bugs in this category:

- automata are not properly implemented

Lessons to learn:

- an implementation should only parse expected messages
- simple (and well-specified) state machines are beautiful
Conclusions and perspectives
Conclusion

SSL/TLS is a rich protocol with a troubled history

▶ an important corpus of specifications, with many features
▶ a diversified ecosystem, with a slow evolution
▶ many implementations facing interesting challenges

TLS 1.3: a new hope?

▶ most of the obsolete algorithms have been removed!
▶ without 0 RTT, the specification has been simplified
▶ 0 RTT mode(s) might revert all this benefit
▶ a long-awaited RFC, but the devil is in the detail
Conclusion and perspectives

Perspectives

- Propose MCookies standardization to the W3C
- Prove TLS 1.3 security properties
  - or propose a restricted profile if needed
- Extend the study to other protocols (IKEv2/IPsec, SSH)
- Study the interaction between TLS and the application protocol
SSL/TLS SoKs

MCookies and other defense-in-depth mechanisms for HTTP

Methodologies and tools to analyse the SSL/TLS ecosystem

Other contributions
[SPW 14b] Mind your Language(s), É. Jaeger, O. Levillain.
Backup slides
The attacker’s models
An example about the diversity of the TLS ecosystem

What can a TLS server answer to a client proposing the following ciphersuites: **AES128-SHA** and **ECDH-ECDSA-AES128-SHA**?
An example about the diversity of the TLS ecosystem

What can a TLS server answer to a client proposing the following ciphersuites: AES128-SHA and ECDH-ECDSA-AES128-SHA?

A AES128-SHA
B ECDH-ECDSA-AES128-SHA
C an alert
An example about the diversity of the TLS ecosystem

What can a TLS server answer to a client proposing the following ciphersuites: \texttt{AES128-SHA} and \texttt{ECDH-ECDSA-AES128-SHA}?

A \texttt{AES128-SHA}  
B \texttt{ECDH-ECDSA-AES128-SHA}  
C an alert  
D something else (\texttt{RC4_MD5})
An example about the diversity of the TLS ecosystem

What can a TLS server answer to a client proposing the following ciphersuites: **AES128-SHA** and **ECDH-ECDSA-AES128-SHA**?

A  AES128-SHA  
B  ECDH-ECDSA-AES128-SHA  
C  an alert  
D  something else (**RC4_MD5**)  

The explanation?

- a ciphersuite is a 16-bit integer  
- until (relatively) recently, all ciphersuites were of the form 00 XX  
- so why bother with the most significant byte?
Context preparation

NSS certificate store extraction

Note: the file used to extract EV does not exist anymore

Metadata injection
Answer injection

Typical figures for a full IPv4 campaign

<table>
<thead>
<tr>
<th>Table</th>
<th>N rows</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>answers.csv</td>
<td>40 M</td>
<td>4 GB</td>
</tr>
<tr>
<td>chains.csv</td>
<td>20 M</td>
<td>2 GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary contents</th>
<th>N</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>raw certificates</td>
<td>10 M</td>
<td>10 GB</td>
</tr>
</tbody>
</table>
Certificate analysis

Typical figures for a full IPv4 campaign

<table>
<thead>
<tr>
<th>Table</th>
<th>N rows</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>parsed_certs.csv</td>
<td>10 M</td>
<td>6 GB</td>
</tr>
<tr>
<td>unparsed_certs.csv</td>
<td>100</td>
<td>10 KB</td>
</tr>
<tr>
<td>links.csv</td>
<td>14 M</td>
<td>1 GB</td>
</tr>
<tr>
<td>built_chains.csv</td>
<td>120 M</td>
<td>12 GB</td>
</tr>
<tr>
<td>trusted_certs.csv</td>
<td>6 M</td>
<td>300 MB</td>
</tr>
<tr>
<td>trusted_chains.csv</td>
<td>9 M</td>
<td>450 MB</td>
</tr>
</tbody>
</table>
Statistics production

TLS parameters
- proportion of TLS answers
- negotiated versions
- chosen ciphersuites
- RFC 5746 support

Certificate chain quality
- RFC-compliance
- trusted chains w.r.t a given certificate store

Server behavior
- intolerance to a given stimulus
- comparison of answers to a duplicate stimulus
### Backup slides

#### Typical figures for a full IPv4 campaign

<table>
<thead>
<tr>
<th>Table</th>
<th>N rows</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>answers.csv</td>
<td>40 M</td>
<td>4 GB</td>
</tr>
<tr>
<td>chains.csv</td>
<td>20 M</td>
<td>2 GB</td>
</tr>
<tr>
<td>parsed_cert.csv</td>
<td>10 M</td>
<td>6 GB</td>
</tr>
<tr>
<td>unparsed_cert.csv</td>
<td>100</td>
<td>10 KB</td>
</tr>
<tr>
<td>links.csv</td>
<td>14 M</td>
<td>1 GB</td>
</tr>
<tr>
<td>built_chain.csv</td>
<td>120 M</td>
<td>12 GB</td>
</tr>
<tr>
<td>trusted_cert.csv</td>
<td>6 M</td>
<td>300 MB</td>
</tr>
<tr>
<td>trusted_chain.csv</td>
<td>9 M</td>
<td>450 MB</td>
</tr>
</tbody>
</table>
Implementation choices

Design rationale
- store enriched data in CSV tables
- split data processing into simple tools
- avoid tools requiring a global view when possible

Challenges
- X.509v1 certificates generated by appliances
  - 140,000 self-signed distinct certificates
  - containing the same subject (and issuer)
  - 20 billion signatures to check
- the max-transvalid option

concerto is an open-source project available on GitHub
The main idea behind parsifal: \( \mathcal{P} \text{Types} \)

\( \mathcal{P} \text{Types} \): the basic blocks of a parsifal parser

- an OCaml type \( t \);
- a parse\(_t\) function (\( \text{bytes} \to t \))
- a dump\(_t\) function (\( t \to \text{bytes} \))
- a value\(_{of\_t}\) function (\( t \to \text{value} \))
The main idea behind parsifal: $\mathcal{P}Types$

$\mathcal{P}Types$: the basic blocks of a parsifal parser

- an OCaml type $t$
- a $\text{parse}_t$ function ($\text{bytes} \to t$)
- a $\text{dump}_t$ function ($t \to \text{bytes}$)
- a $\text{value}_t$ function ($t \to \text{value}$)

The goal: relieve the programmer from writing tedious code

To this aim, three kinds of $\mathcal{P}Types$:

- basic $\mathcal{P}Types$, provided by the standard library
- keyword-assisted $\mathcal{P}Types$
- custom $\mathcal{P}Types$
Implementing TLS records

```plaintext
enum tls_version (16, UnknownVal V_Unknown) =
| 0x0002 -> SSLv2        | 0x0302 -> TLSv1_1
| 0x0300 -> SSLv3        | 0x0303 -> TLSv1_2
| 0x0301 -> TLSv1

enum tls_content_type (8, Exception) =
| 0x14   -> ChangeCipherSpec | 0x16   -> Handshake
| 0x15   -> Alert            | 0x17   -> ApplicationData

struct tls_record = {
    content_type : tls_content_type;
    record_version : tls_version;
    content_length : uint16;
    record_content : binstring [content_length];
}
```
Perspectives on the specification front

MCookies development
- propose MCookies to the W3C
- propose MTokens to web application framework
- extend the concept to other secrets/protocols, when possible

TLS 1.3
- ensure the specification is as clear and simple as possible
- continue to model the protocol and to prove its security properties
- propose a secure restricted profile if needed

Other protocols
- IKEv2/IPsec
- SSH
Perspectives on the knowledge of the SSLiverse

Launch new campaigns

▶ multi-stimuli campaigns on IPv4 space are still rare
▶ explore more protocols
▶ extend existing efforts to publish dashboards such as SSL Labs

Relation to specification and deployment goals

▶ use campaigns as a laboratory to test the intolerance to new features
▶ use campaigns as a way to check when obsolete features can be safely removed
Perspectives on software improvement

Study TLS implementations using safe(r) languages

- miTLS in $F^*$
- nqsb-TLS in OCaml
- assess the security and the usability of such stacks

Analyse and test existing stacks

- static analysis tools
- protocol fuzzers (FlexTLS, tlsfuzzer)
- black-box state-machine inference using $L^*$
- assess the coverage of such methodologies