Chair for Network Architectures and Services
Department of Informatics
TU München – Prof. Carle

Network Security

Chapter 12

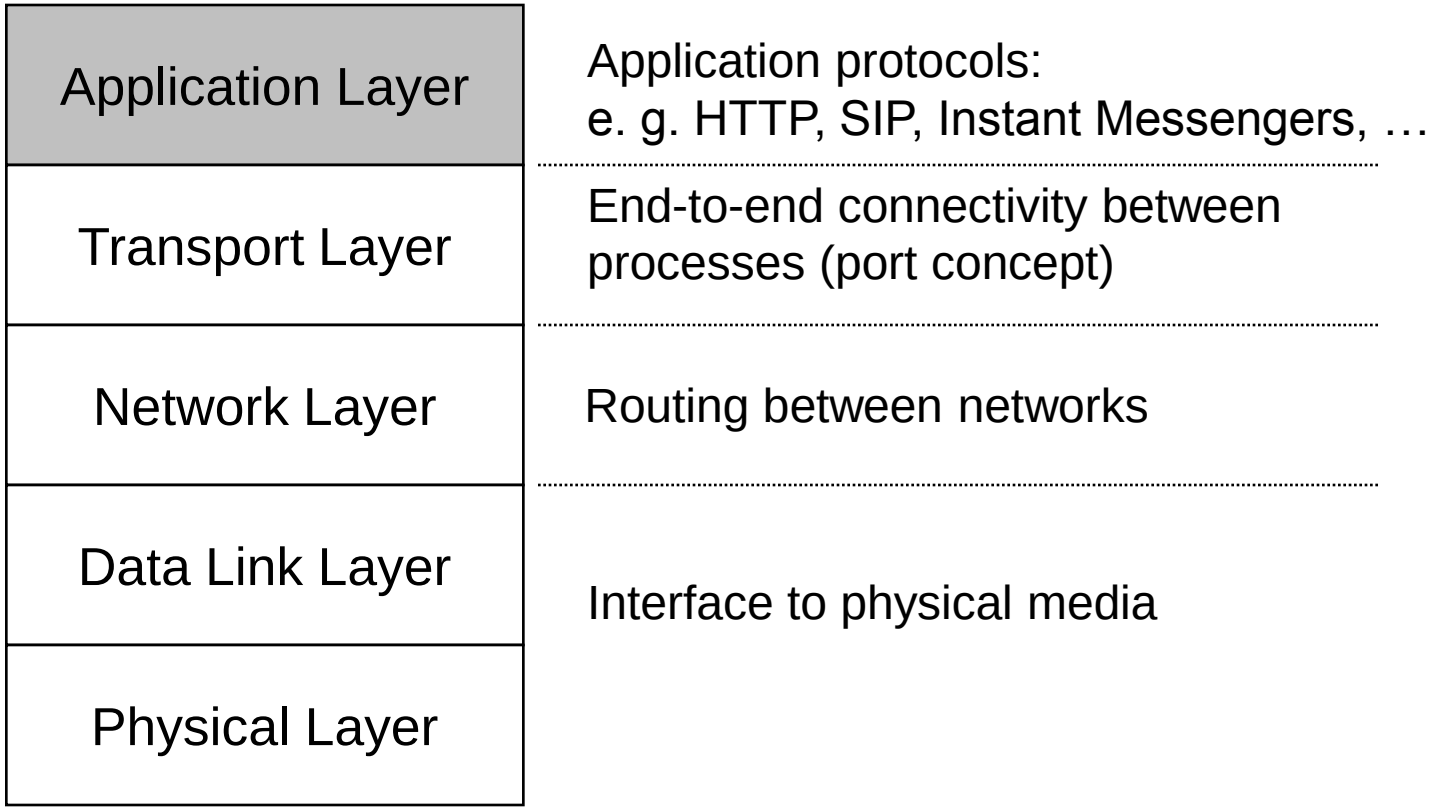
Application Layer Security

with friendly support by
P. Laskov, Ph.D.,
University of Tübingen

Application Layer Security

- 10.1: WWW Security
- 10.2: Web Service Security

Recap: Internet Protocol Suite



- TCP/IP stack has no specific representation for OSI layers 5, 6, 7 („session“, „representation“, „application“):
  the Application Layer is responsible for all three

Why Application Layer Security?

- So far, we were concerned with layers below the application layer:
  - Cryptography (mathematics)
  - Link Layer security
  - Crypto protocols: IPSec, SSL, Kerberos…
  - Firewalls
  - Intrusion Detection
- There are attacks where these defenses do not work:
  - Cross-Site Scripting, Buffer Overflows, …
- Possible because
  - These attacks are not detectable on lower layers (→ cf. WWW Security), or
  - The mechanisms do not secure the correct communication end-points (→ cf. Web Service Security, next lecture)
- In general, many applications need to provide their own security mechanisms
  - E. g. authentication, authorization

Part I: Introduction to the WWW

- Part I: Introduction to the WWW and Security Aspects
- Part II: Internet Crime
- Part III: Vulnerabilities and Attacks

Most important Web technologies

Name	Used for	Comment
HTML	Document structure	…
CSS	Document rendering	…
HTTP	Carrier protocol	…
Cookies	Session state keeping	…
URL/URI	Document location	…
JavaScript	Client-side computation and interaction	…
Flash	Client-side code execution	…
…	…	…

Introduction to the World Wide Web

- You all know it – but what is it exactly?
- Conceived in 1989/90 by Tim Berners-Lee at CERN

- Hypermedia-based extension to the Internet on the Application Layer
  - Any information (chunk) or data item can be referenced by a Uniform Resource Identifier (URI)
  - URI syntax (defined in RFCs) :
    `<scheme>://<authority><path>?<query>#<fragment>`
  - Special case: URL (“Locator”)
    `http://www.net.in.tum.de/de/startseite/`
  - Special case: URN (“Name”)
    `urn:oasis:names:specification:docbook:dtd:xml:4.1.2`

- Probably the best-known application of the Internet
- Currently, most vulnerabilities are found in Web applications

HTML and Content Generation

- HTML is the *lingua franca* of the Web
  - Content representation: structured hypertext documents
  - HTML documents – i. e. Web pages – may include:
    - JavaScript: script that is executed in browser
    - Java Applets: Java program, executed by Java VM
    - Flash: multimedia application, executed (played) by Flash player
- Today, much (if not most) content is created dynamically by server-side programs
  - (Fast-)CGI: interface between Web server and such a server-side program
  - Possible: include programs directly as modules in Web server (e.g. Apache)
- Often, dynamic Web pages also interact with the user
  - Examples: searches, input forms → think of online banking
- Examples of server-side technology/languages:
  - PHP, Python, Perl, Ruby, …
  - Java (several technologies), ASP.NET
  - Possible, but rare: C++ based programs

HTTP

- HTTP is the carrier protocol for HTML
  - Conceived to be state-less: server does not keep state information about connection to client
  - Mostly simple `GET`/`POST` semantics (`PUT` is possible)
  - HTML-specific encoding options
- OK for the beginnings – but the Web became the most important medium for all kinds of purposes (e. g. e-commerce, forums, etc.)
  → today: real work flows implemented with HTTP/HTML
  → need to keep state between different pages
  → **sessions**

Sessions Over HTTP

- Sessions: many work-arounds around the state-less property
  - Cookies (later)
  - Session-IDs (passed in HTTP header)
  - Parameters in URL
  - Hidden variables in input forms (HTML-only solution)
- Session information is a valuable target
  - E. g., online banking: credit card or account information
- Session IDs in the URL can also be a weakness
  - Can be guessed or involuntarily compromised (e. g. sending a link) → “session hijacking”
- **`GET`** command may encode parameters in the URL
  - Can be a weakness
  - Some URLs are used to trigger an action, e.g. `http://www.example.org/update.php?insert=user`
  - Attacker can craft certain URLs (→ Cross-Site Request Forgery)

HTTP Authentication

- HTTP Authentication
  - Basic Authentication: not intended for security
    - Server requests username + password
    - Browser answers in plain text → relies on underlying SSL for security
    - No logout! Browser keeps username and password in cache
  - Digest Authentication: protects username + password
    - Server also sends a nonce
    - Browser reply is MD5 hash: md5(username,password,nonce)
    - No mutual authentication – only client authentication
    - More secure and avoids replay attacks, but MD5 is known to have weaknesses
    - SIP uses a similar method
- HTTP authentication often replaced with other methods
  - Requires session management
  - Complex task

Cookies

- Small text files that the server asks the browser store
  - Client authenticates to server, receives cookie with a secret value
  - Uses this value to keep the session alive when transmitting HTTP(s)
- Problematic: which cookies is a site allowed to access?
  - `abc.com` must not access cookies for `xyz.com`
- Cookies come with a security policy implemented in the browser
- Problematic:
  - Cookie scope can be set via `domain` parameter, but need for care
  - Some browsers allow to restrict scope to a host name; others only to a domain
- Example: cookie set at foo.example.com

value of `domain`	Non-IE	Internet Explorer
(omitted)	`foo.example.com`	`*.foo.example.com`
`bar.example.com`	`not set, domain mismatch`	`not set, domain mismatch`
`example.com`	`*.example.com`	`*.example.com`

Cookies: a few more aspects

- Cookies can be exploited to work against privacy
  - User tracking: identify user and store information about browsing habits
  - 3rd party cookies: cookies that are not downloaded from the site you are visiting, but from another one
    - Can be used to track users across sites
  - Cookies can be set without the user knowing (there are reasonably safe standard settings)
  - Security trade-off: many Web pages require cookies to work, disabling them completely may not be an option
- Cookies may also contain confidential session information
  - Attacker may try to get at such information (→ Cross-Site Scripting)

JavaScript

- Script language that is executed on client-side (not only in browsers!)
  - Originally developed by Netscape; today more or less a standard
  - Object-oriented with C-like syntax, but multi-paradigm
  - Allows dynamic content for the WWW → AJAX etc.
  - Allows a Web site to execute programs in the browser
- The Web is less attractive without JavaScript – but anything that is downloaded and executed by a client may be a security risk

JavaScript

- Security Issues:
  - Allows authors to write malicious code
  - Allows cross-site attacks (we look at these a bit later in this lecture)
- Defenses:
  - Sandboxing of JavaScript execution
    - Difficult to implement
  - Same-origin policy (SOP)

Same-Origin Policy (SOP)

- One of the stronger defences for JavaScript
  - One JavaScript context should not be able to modify the context of another
  - Such access is otherwise possible with the Document Object Model API
- All browsers have a SOP – with OK consistency (IE is a bit different)
- Original idea (Netscape, 1995!):
  - Two JavaScript contexts are allowed access to each other if and only if protocols, host names and ports associated with the documents in question match **exactly**

Originating doc	Accessed doc	Non-IE	Internet Explorer
`http://abc.com/a/`	`http://abc.com/b/`	`Access OK`	`Access OK`
`http://ab.com/`	`http://www.abc.com`	`Host mismatch`	`Host mismatch`
`http://abc.com/`	`https://abc.com/`	`Protocol mismatch`	`Protocol mismatch`
`http://abc.com:81/`	`http://abc.com/`	`Port mismatch`	`Access OK (!)`

Same-Origin Policy (SOP)

- Critique of SOP:
  - Sometimes too restrictive: two co-operating Web sites `abc.com` and `xyz.com` cannot exchange information
  - Sometimes too broad: SOP can be violated
- Trying to make the SOP less restrictive is dangerous:
  - Common way: use JS property `document.domain`
  - Two sites sharing the same top-level-domain can agree to share context
  - Symmetric: both sites must opt-in and define the property
  - Critique: too broad
    - Assume `login.abc.com` wants to share with `payments.abc.com` and set `document.domain` to `abc.com` – suddenly all sub-domains are included, even `mallory.abc.com`
- Better method: use `postMessage()` as defined for HTML 5
  - Based on notion of JavaScript handle – allows to send to another window for which senders holds a handle

Same-Origin Policy

- The SOP only refers to JavaScript interactions
- It does not cover any other interactions and credentials, like:
  - State of SSL connection – good authentication or not
  - IP connectivity – SOP matches via host names
  - Information in cookies (they have their own kind of SOP)
- Example:
  - Assume two windows A and B in a browser, co-operating within SOP
  - A is a site with login, and user is logged in as „Alice“
  - A and B will now remain same-origin even if the user logs out as Alice and logs in again as Bob
  - Here, SOP provides no notion at all of „identity in a session“
- Interesting fact:
  - The XMLHttpRequest mechanism used in AJAX (Web 2.0) has a tweaked SOP
  - `document.domain` does not work
  - And IE supports ports, too

Most important Web technologies

Name	Used for	Comment
HTML	Document structure	Often abused for representation; theoretically XML/SGML-based; requires diligent parsing
CSS	Document rendering	Some parts in the standard can be exploited
HTTP	Carrier protocol	Several versions; stateless
Cookies	Session state keeping	Need to prevent cross-domain interactions
URL/URI	Document location	Inconsistent interpretation of RFCs; requires diligent parsing
JavaScript	Client-side computation and interaction	Requires safe execution environment
Flash	Client-side code execution	Requires safe execution environment
…	…	… there‘s much more …

Part II: Internet Crime

- Part I: Introduction to the WWW and Security Aspects
- Part II: Internet Crime
- Part III: Vulnerabilities and Attacks

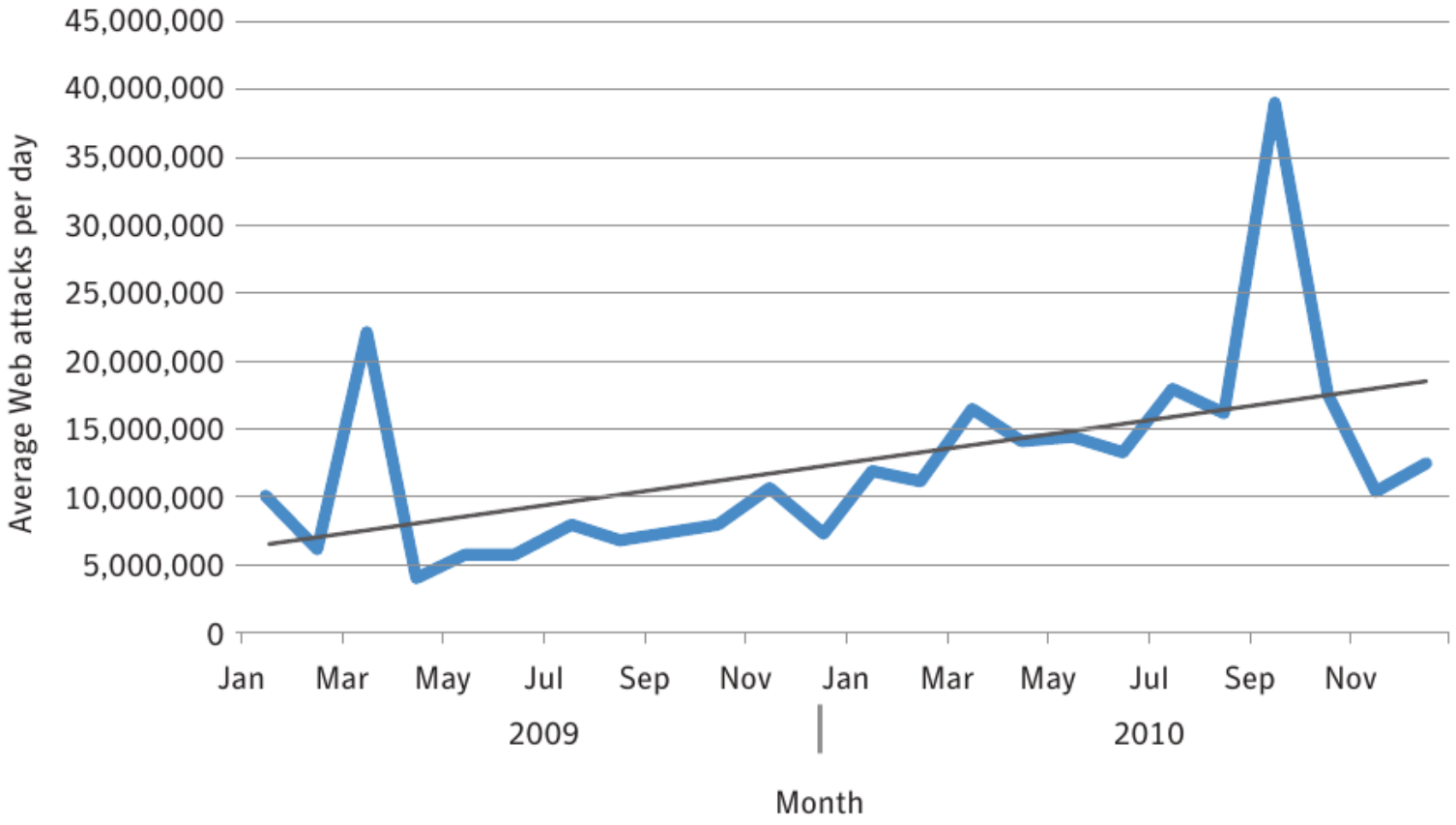
Vulnerabilities: some numbers

- 3,462 vs 2,029 web/non-web application vulnerabilities were discovered by Symantec in 2008
- Average exposure time: 60 days
- 12,885 site-specific XSS vulnerabilities submitted to XSSed in 2008 alone
- Only 3% of site-specific vulnerabilities were fixed by the end of 2008

- The bad guys are not some hackers who “want to know how it works”
- These days, it’s a business!
- “Symantec Underground Economy Report 2008”:

  “Moreover, considerable evidence exists that organized crime is involved in many cases …“
  [ed.: referring to cooperation between groups]

From the Symantec Report 2011



Average Web-based attacks per day, by month, 2009–2010

Source: Symantec Corporation

From the Symantec Report 2011

Overall Rank		Item	Percentage		2010 Price Ranges
2010	2009		2010	2009	
1	1	Credit card information	22%	19%	$0.07–$100
2	2	Bank account credentials	16%	19%	$10–$900
3	3	Email accounts	10%	7%	$1–$18
4	13	Attack tools	7%	2%	$5–$650
5	4	Email addresses	5%	7%	$1/MB–$20/MB
6	7	Credit card dumps	5%	5%	$0.50–$120
7	6	Full identities	5%	5%	$0.50–$20
8	14	Scam hosting	4%	2%	$10–$150
9	5	Shell scripts	4%	6%	$2–$7
10	9	Cash-out services	3%	4%	$200–$500 or 50%–70% of total value

From the Symantec Report 2011

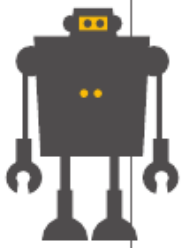
Item	Bulk Prices Observed	Unit Price
Credit card information	10 credit cards for $17	$1.70
	100 credit cards for $100	$1.00
	1000 credit cards for $300	$0.30
	750 credit cards for $50	$0.07
Credit card dumps	101 dumps for $50	$0.50
Full identities	30 full identities for $20	$0.67
	100 full identities for $50	$0.50

From the Symantec Report 2011

1M+

Bots

Rustock, the largest botnet observed in 2010, had well over 1 million bots under its control. Grum and Cutwail followed, each with many hundreds of thousands of bots.



$15

per 10,000 Bots

Symantec observed an underground economy advertisement in 2010 promoting 10,000 bots for $15. Bots are typically used for spam or rogueware campaigns, but are increasingly also used for Distributed Denial of Service attacks.

$0.07 to $100

per Credit Card

This was the range of prices seen advertised in the underground economy for each “stolen” credit card number, and, as in the real economy, bulk buying usually gets the buyer a significant discount.

Just FYI: from the Symantec Report 2008

Rank for Sale	Rank Requested	Category	Percentage for Sale	Percentage Requested
1	1	Credit card information	31%	24%
2	3	Financial accounts	20%	18%
3	2	Spam and phishing information	19%	21%
4	4	Withdrawal service	7%	13%
5	5	Identity theft information	7%	10%
6	7	Server accounts	5%	4%
7	6	Compromised computers	4%	4%
8	9	Website accounts	3%	2%
9	8	Malicious applications	2%	2%
10	10	Retail accounts	1%	1%

Table 1. Goods and services available for sale, by category[56]
Source: Symantec Corporation

Just FYI: from the Symantec Report 2008

Rank for Sale	Rank Requested	Goods and Services	Percentage for Sale	Percentage Requested	Range of Prices
1	1	Bank account credentials	18%	14%	$10–$1,000
2	2	Credit cards with CVV2 numbers	16%	13%	$0.50–$12
3	5	Credit cards	13%	8%	$0.10–$25
4	6	Email addresses	6%	7%	$0.30/MB–$40/MB
5	14	Email passwords	6%	2%	$4–$30
6	3	Full identities	5%	9%	$0.90–$25
7	4	Cash-out services	5%	8%	8%–50% of total value
8	12	Proxies	4%	3%	$0.30–$20
9	8	Scams	3%	6%	$2.50–$100/week for hosting; $5–$20 for design
10	7	Mailers	3%	6%	$1–$25

Table 2. Breakdown of goods and services available for sale and requested[64]

Just FYI: from the Symantec Report 2008

Exploit Type	Average Price	Price Range
Site-specific vulnerability (financial site)	$740	$100–$2,999
Remote file include exploit (500 links)	$200	$150–$250
Shopadmin (50 exploitable shops)	$150	$100–$200
Browser exploit	$37	$5–$60
Remote file include exploit (100 links)	$34	$20–$50
Remote file include exploit (200 links)	$70	$50–$80
Remote operating system exploit	$9	$8–$10

Table 8. Exploit prices

Source: Symantec Corporation

Just FYI: from the Symantec Report 2008

Attack Kit Type	Average Price	Price Range
Botnet	$225	$150–$300
Autorooter	$70	$40–$100
SQL injection tools	$63	$15–$150
Shopadmin exploiter	$33	$20–$45
RFI scanner	$26	$5–$100
LFI scanner	$23	$15–$30
XSS scanner	$20	$10–$30

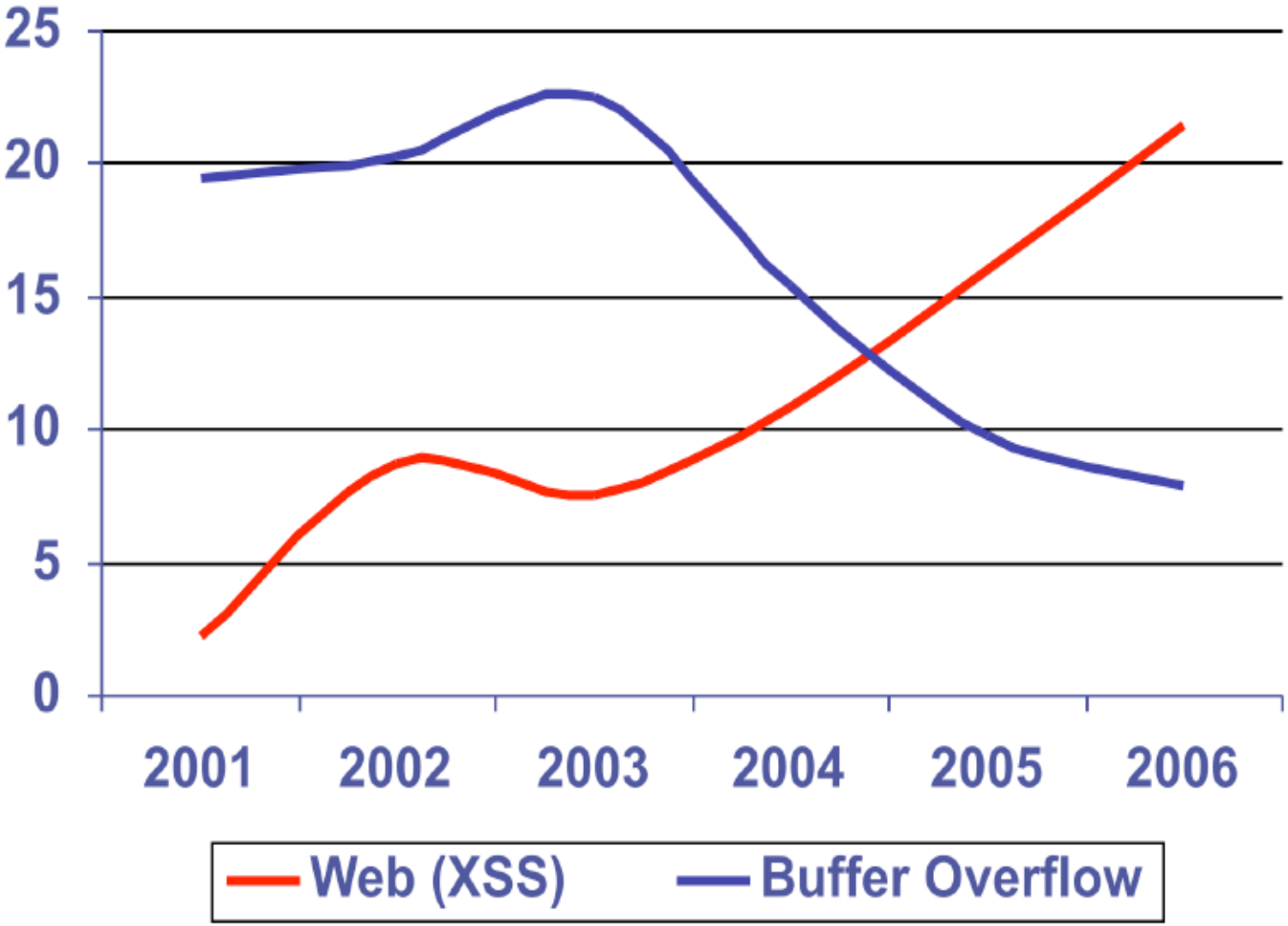
Table 5. Attack kit prices

Source: Symantec Corporation

Part III: Vulnerabilities and Attacks

- Part I: Introduction to the WWW and Security Aspects
- Part II: Internet Crime
- Part III: Vulnerabilities and Attacks

Comparison: two classic vulnerabilities



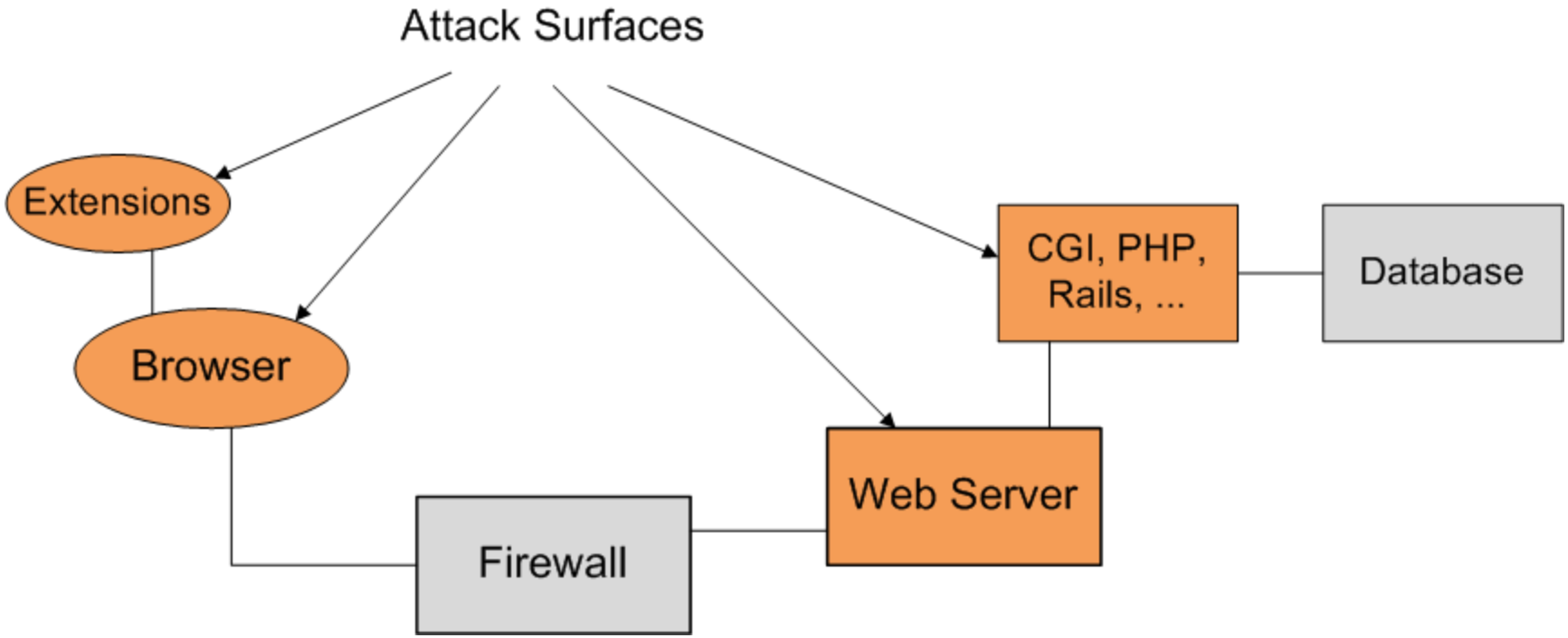
Source: MITRE CVE trends

Classification of Attacks (incomplete)

	Client-side	Server-side
Common implementation languages	❑ C++ (e. g. Firefox) ❑ XULRunner ❑ Java	❑ Web Server: C++, Java ❑ Script languages
Common attack types	❑ Drive-by downloads ❑ Buffer overflows	❑ Cross-Site scripting ❑ Code Injection ❑ SQL Injection ❑ (DoS and the like)
Result of attack	❑ Malware installation ❑ Computer manipulation ❑ Loss of private data	❑ Defacement ❑ Loss of private data ❑ Loss of corporate secrets

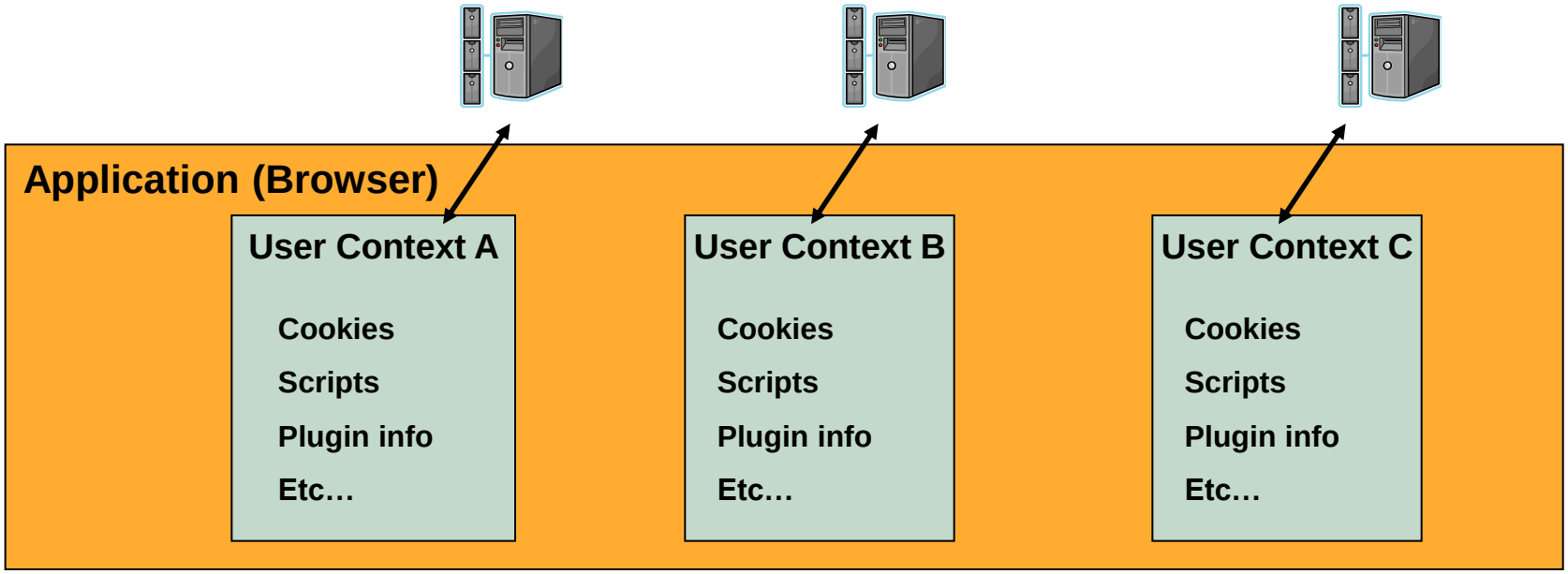
One Step Back: why is the WWW so vulnerable?

- Many important business transactions take place
- Much functionality, much complexity in software
  → many attack vectors, huge attack surface
- Even though we may implement protocols like TCP/IP really well, any (Web) application that interacts with the outside world must be open by definition and reachable even across a firewall



Informal Definition: Contexts

- Context (in general): collection of information that belongs to a particular session or process
  - Useful abstraction that helps us to classify the target of an attack
  - Here: not a formal definition, nor a model of actual implementation
- User Context (in a browser):
  - Collection of all information that “belongs” to a given session
  - Cookies, session state variables, plugin-specific information…
  - JavaScripts: downloaded and executed → obey same-origin policy!
  - Information from session A should not be accessible from Session B
  - Client and server must remain synchronized w.r.t. state information



Attack 1: Session Variables

- **Target of attack:**
  Synchronization of state information between client and server
  (in other words: the session management is attacked)
- **Typical scenario:**
  Exchange between client and server that takes
  several steps to complete
- **Typical approach of attack:**
  Swap state information during one step
- **Cause of vulnerability:**
  Server (or client) relies on information sent by the other party
  instead of storing it itself

- Best explained by example. Here:
  Server: a CA that can issue X.509 certificates
  Client: a Web browser that wants to acquire such a certificate

Attack 1: How the Work-Flow Should Be

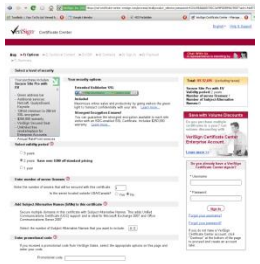
Browser = client

CA = server

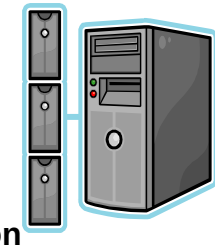
State:

A→CA

xyz.de



A: Request cert for domain `xyz.de`



State:

A→CA

xyz.de

2) Background:
Ownership verification

A owns xyz.de

Offer for xyz.de by CA

CA: Offer cert for domain `xyz.de`

A: Acknowledge request: cert for domain `xyz.de`

(A pays for xyz.de)

CA: Issue cert for domain `xyz.de`

Has cert

Question: where do you keep the session information?

If your answer is “in the cookie”: serious mistake.

In fact, the CA must NOT trust information by the browser. We show you why now.

Attack 1:
How to Attack the Synchronization of State Information

In this example, all state information is stored on client-side and retransmitted in each step (e. g. by reading from a cookie). The server does not store state.

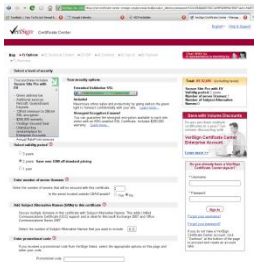
Browser = client

CA = server

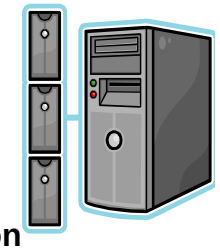
State:

A→CA

xyz.de



A: Request cert for domain `xyz.de`



State:

A→CA

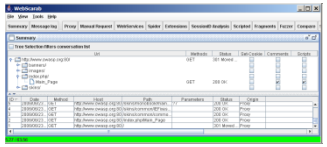
xyz.de

2) Background: Ownership verification

A owns xyz.de

Offer for xyz.de by CA

CA: Offer cert for domain `xyz.de`



Swap variables on the fly

A: Acknowledge request: cert for domain `mozilla.com`

(A pays for xyz.de)

CA: Issue cert for domain `mozilla.com`

Has cert!!!

Why Was the Attack Possible?

- In our example, all state information was kept on client-side in a cookie
- All the attacker did was to swap `mozilla.com` for `xyz.de` in the second HTTP request
- The server issued a cert for the wrong domain because it failed to notice that the *domain name in the first request was not the same as the name in the second request*.
- That was possible because the relevant information was not stored on server-side
- Do you think this is too easy and will not happen “in the real world”?
  - In fact, something like this ***may*** have happened in the beginning of 2009 to a CA that is included in Firefox’s root store.
  - Background info:
    - The attack did not succeed – because there was a second line of defense: all “high-value” domain names are double-checked by *human personnel*.
  - The CA publicly acknowledged there was an intrusion.
    - The CA described an attack pattern that hinted at what we have just seen.
    - The CA contacted the attacker – it was a White Hat

Defense / Mitigation

- Guideline 1: For each entity in the protocol:
  - Everything that is relevant for the correct outcome must be stored *locally*
  - It can be difficult to identify this information if you have complex work-flows…
- Guideline 2: All Input Is Evil
  - Always treat all input as untrusted
  - Never use it without verification

- Nota bene: what if the server uses Javascript/Java to “force” browser to behave correctly? → just use a HTTP proxy → NOT a defense!

- This was just a simple attack because an entity failed to obey these rules.
- In particular, Guideline 1 was violated.
- However, in the following, we show you that attacks are possible even if state is stored correctly and only Guideline 2 is violated.

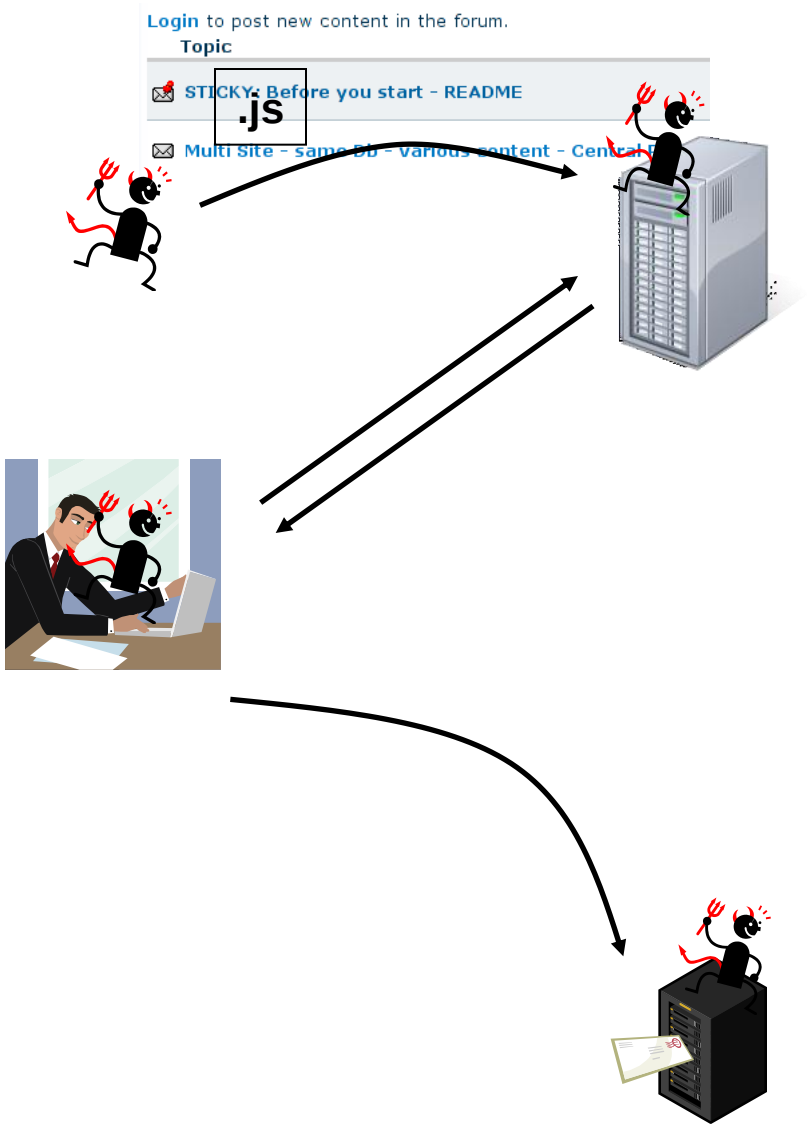
Cross-Site Scripting (XSS)

- **Target of attack:**
  Attempt to access user context from outside the session
  Goal is to obtain confidential information from the user context
- **Typical scenario:**
  User surfing the Web and accessing a Web site
  while having (Java)script enabled
- **Typical approach to attack:**
  Attacker plants a malicious script on a Web page;
  the script is then executed by the user's browser
- **Cause of vulnerability:** two-fold
  1) Attacker is able to plant malicious script on a Web page
  → flaw in Web software needed
  2) User browser executes script from a Web page
  → user's "trust" in Web site is exploited

- XSS is one of the most common attacks today

Cross-Site Scripting: Typical Attack

- Stage 1: Attacker injects malicious script
  - Here: in a Web forum where you can post messages
  - In addition to normal text, the attacker writes: `<script>[malicious function]</script>`
  - The server accepts and stores this input
- Stage 2: Unaware user accesses Web forum
  - Here: reads poisoned message from attacker
  - User receives:
    `<p>Hello, this is a harmless message`
    `<script>[malicious function]</script>`
    `</p>`
  - Everything within `<script>` is executed by browser *in the user's context*
- Possible Consequences:
  - Script reads information from cookies etc. and sends it to attacker's server
  - Script redirects to other site
    → download trojan etc.



Cross-Site Scripting: Why Does it Work?

- Why was the attack possible?
- Reason 1: The Web application did not **sanitize** input it received
  - Remember: all input is evil; and the attacker can *choose* his input
  - If the Web app had just dropped all HTML input, there would be no script uploaded
    → and none executed in the browser
  - Unfortunately, many Web sites allow users to post at least some HTML
    → a nice feature, but dangerous
- Reason 2:
  The user had trusted the Web site and did not assume
  malicious content could be downloaded and executed
  → **abuse of trust**

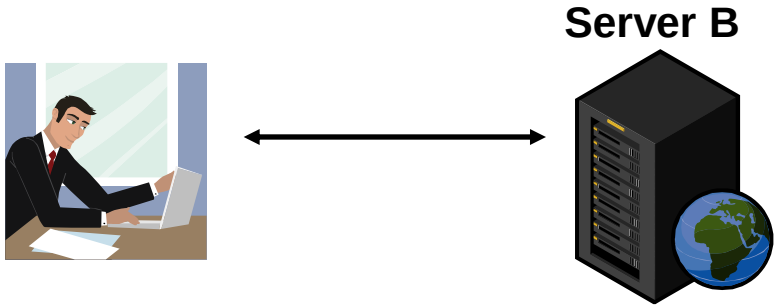
- Nota bene: none of the mechanisms you know so far is a defense!
  - Crypto protocols: encrypting/signing does not help here
  - Firewalls: work on TCP/IP level
  - XSS is a particularly useful example to show why there is a need
    for ***application layer security***

Cross-Site Request Forgery (XSRF)

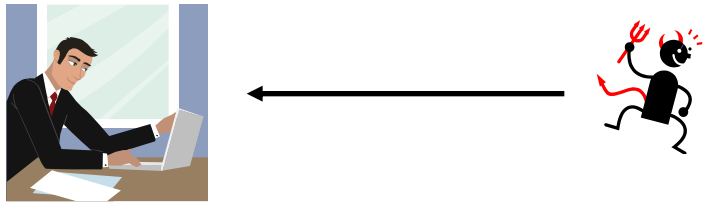
- **Target of attack:**
  User-Server context: session of client A with a server B
- **Typical scenario:**
  Authenticated user on a Web page on B which is OK and trusted; then the user surfs to server M which is malicious
- **Typical approach to attack:**
  - Attacker knows that user is logged in
    → crafts a URL to server B that executes an action
  - Attacker causes victim to call that URL
- **Cause of vulnerability:**
  - Attacker URL is called by user; within his user context
    → **abuse of server's trust** into requests from client
  - Browser **cannot recognise that request to the URL is malicious**
    → it seems to be in the correct context
    → instance of **"Confused Deputy"** problem (browser is deputy): authority of deputy (login to B) is abused

Cross-Site Request Forgery (XSRF)

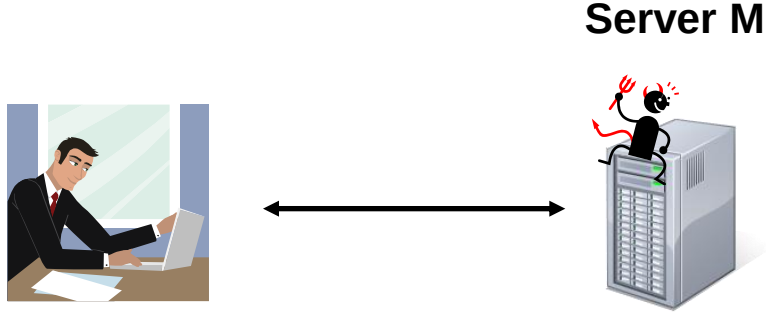
- Stage 1: user logs into Web site
  - Authenticated user
  - Session with server B
  - User keeps this session open



- Stage 2: attacker tricks user to surf to his own site, server M. Methods:
  - Phishing
  - XSS



- Stage 3: user surfs to malicious server M
  - In the HTML he receives, a malicious link is embedded
    harmless text
    ****
    more harmless text



→ undesired action executed

Defence against XSRF

- Particularly good defence against XSRF: Secret Tokens
- I.e. a Web site requires that the client (browser) proves knowledge of a secret value before acting on a URL
- Requires: server needs to transmit this value first, can be done via hidden field in input form etc.

- Advantage:
  - Reliable if secret values cannot be guessed
- Disadvantage:
  - State-keeping on server-side necessary

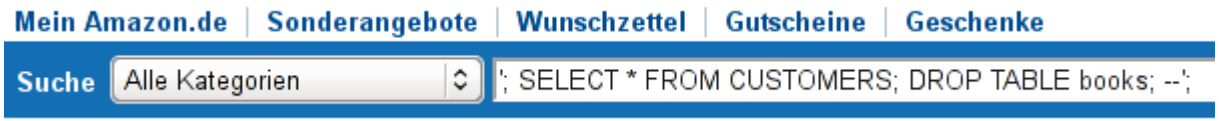
SQL Injection

- **Target of attack:**
  Server context
- **Typical scenario:**
  Web server runs with an SQL database in the background; attacker wants to extract or inject information to/from the database
- **Typical approach to attack:**
  Attacker writes SQL code into an input form, which is then passed to the SQL database; evaluated and output returned
- **Cause of vulnerability:**
  Web server does not sanitize the input and accepts SQL code

- SQL Injection is a real classic attack

SQL Injection

- Attacker injects SQL into search form:



- The author of the Web page may have intended to execute:

```
SELECT author,book FROM books WHERE book = '$title';
```

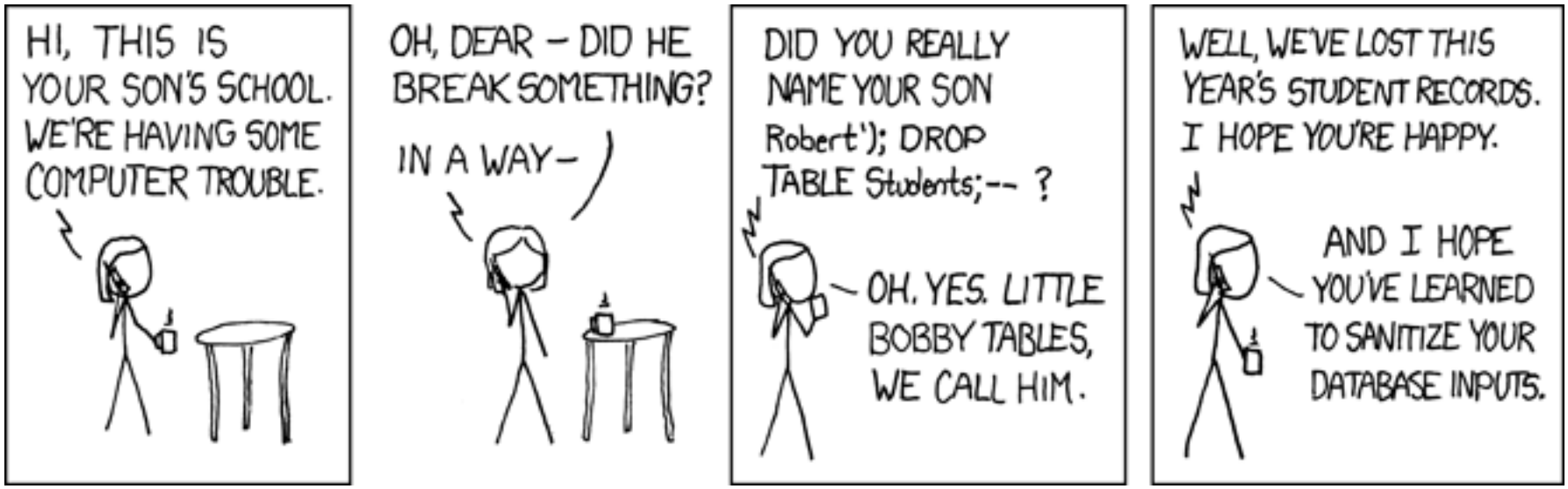
- Through the SQL injection, this has become something like:

```
SELECT author,book FROM books
WHERE book = ''; SELECT * FROM CUSTOMERS; DROP TABLE
books;
```

- You just lost your catalogue and compromised your customers data
- Amazon, of course, is too clever not too sanitize their input – but it is amazing how many other Web sites fail to do so!

- Fortunately, this exact example won't work anymore

Sanitize or Be Sorry



General defences for XSS, XSRF, SQL Injection

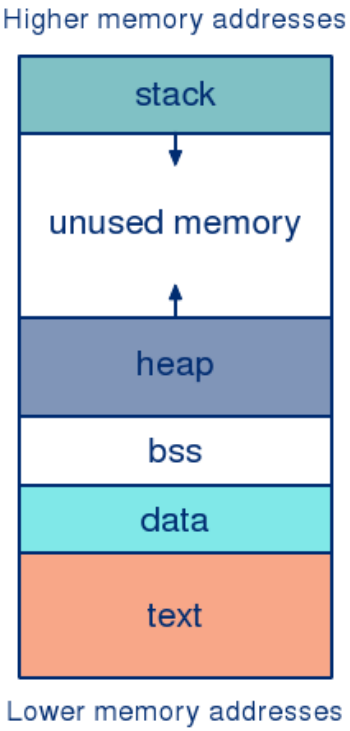
- Some options on **client-side** against XSS/XSRF:
  - JavaScript is often a must for many “good” Web pages
    → turning it off is not an option
    → better sandboxing? → very complex
  - Turning on some security settings can provide some security
    → unfortunately, these are often not activated by default
- Better protection can be achieved on **server-side**:
  - Treat all input as **untrusted**
  - **Sanitize** your input and output: proper **escaping**
    - Escape (certain) HTML tags and JavaScript
    - Exceedingly difficult and complex task!
    - Whitelisting is better than blacklisting – the black list may grow
- Do not write your own escaping routines
  - Modern script languages offer this functionality

Buffer Overflows

- **Target of attack:**
  Running process on a server (process has a context!)
- **Typical scenario:**
  An application that is accessible on the Internet and has a certain built-in flaw
  Vulnerable C(++)-based application on the Internet
- **Typical approach to attack:**
  - Attacker sends byte stream to vulnerable application; either causing it to crash or to execute attacker code in the process context of the application
- **Cause of vulnerability:** two-fold
  - Buffer overflow in application → serious programming mistake (root cause: von Neumann machine)
  - Application does not check its input

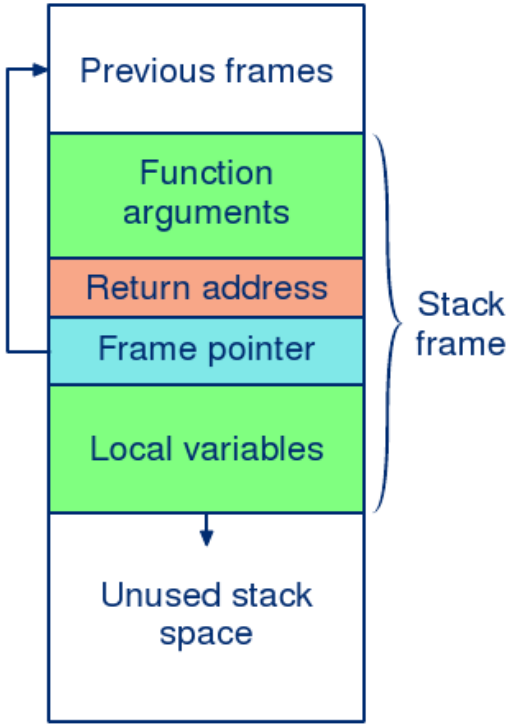
Buffer Overflows

- von Neumann machine:
  program and data share memory
- Applies to all kinds of software
- Memory segments:
  - text – program code
  - data – initialized static data
  - bss – unitialized static data
  - heap – dynamically allocated memory
  - stack – program call stack
- The vulnerability is in the code:
  - Programmer creates buffer on the stack and does not check its size when writing to it
    `char* buffer; readFromInput(buffer);`
- Exploit:
  - Because of the way the stack is handled, you can overwrite the return address



Buffer Overflows

- Stack is composed of frames
  - Pushed on the stack during function invocation, and popped back after returning
- Each frame comprises
  - functions arguments
  - return address
  - frame pointer to start address of previous frame
  - local variables
- Stack grows to bottom; local variables are written towards top
- Without proper boundary checking, a buffer content can overspill into adjacent area
- Attacker:
  - Find out the offset to the return address
  - Write data to the buffer: overwrite return address, add your own code
  - Application continues to run from the new address, executing the new code



Simple Code Example

```
#include <stdio.h>
#include <string.h>
int vulnerable(char* param)
{
   char buffer[100];
   strcpy(buffer, param);
}

int main(int argc, char* argv[] )
{
   vulnerable(argv[1]);
   printf("Everything's fine\n");
}
```

(from [ISec2010])

Buffer Overflows

- Buffer overflows are mostly a problem for applications written in languages with direct control over memory (like C/C++)
- These are becoming less frequent on Web servers, and checks have become better: correspondingly, we observe a switch to other attacks

- Mitigation of this kind of exploit:
  - Data execution protection:
    mark certain areas in memory as non-executable
  - Address space layout randomization: choose stack memory allocation at random (“hardened kernels” do this)
    → Support by operating system helps
  - The latter two are very effective together
  - Canaries: preceed the return value with a special value:
    before following the return value, check if is still the same
- Be careful when writing in C/C++, use up-to-date compilers, use the defences the hardware and OS offers

Summary

- **Web applications** have a **natural attack surface**:
  they must accept input from outside
- **Very complex interactions** between protocols, client+server:
  - Difficult to find all weaknesses in advance
  - In part due to the many mechanisms for session management

- **Typical attacks:**
  - Cross-Site Scripting (XSS): violation of user context, abuse of user trust
  - Cross-Site Request Forgery: confused deputy
  - SQL injection
  - Buffer overflows

- **Defenses:**
  - Most important defense is to **sanitize** and **validate** input data
  - **All input is evil**
  - Also, be aware of your **{user,server,process} contexts**
  - Conventional defenses like cryptography or firewalls are no protection

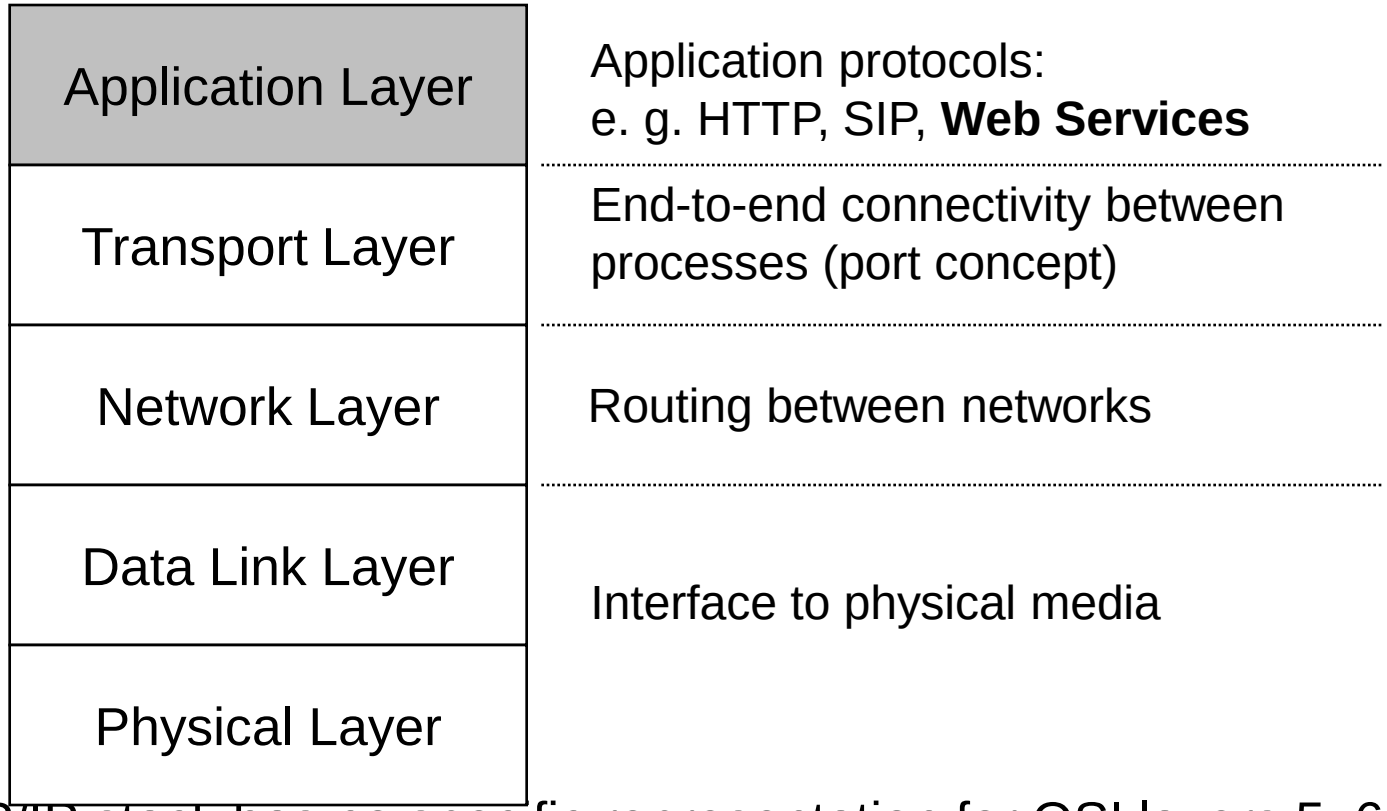
Application Layer Security

- 10.1: WWW Security
- 10.2: Web Service Security

Part I: Introduction to Web Services

- Part I: Introduction to XML and Web Services
- Part II: Securing Web Services
- Part III: Identity Federation

Recap: Internet Protocol Suite



- TCP/IP stack has no specific representation for OSI layers 5, 6, 7 („session“, „representation“, „application“):
  the Application Layer is responsible for all three

Web Services: loose definition

- No consensus on a precise definition “in the community”
- **Loose definition: a collection of technologies that employ HTTP technology and enable application interoperation over the Internet.**
- Examples:
  - Web APIs (e. g. Google Maps, …)
    → often used for **meshups**: Web application that combines data from different sources
  - XML-driven Web Services using a variety of XML-based protocols like SAML, WSDL, UDDI
    → often used for Service-Oriented Architectures
  - RESTful services (recent development, out of scope)
- The distinguishing trademarks seem to be:
  - Use of HTTP
  - Application interoperation – not human users

Web Services: Contributors

- We will mostly (but not exclusively) focus on the latter: XML-driven technologies.
- These technologies have been defined in a (large) number of standards and by several committees
- Standardization Committees:
  - OASIS: Organization for the Advancement of Structured Information Standards
    - Large number of members of different membership classes, including many global players like IBM, Microsoft, SUN
    - Responsible for, e.g., many WS-* standards, SAML, UDDI
    - Offers an Identity Federation standard
  - W3C: World Wide Web Consortium
    - Defines WWW standards: “W3C Recommendations”
    - Responsible for, e.g., HTML, XML, XSL-*, SOAP, WSDL
  - Liberty Alliance
    - Offers an Identity Federation Standard
- Many standards were first developed by companies and then brought to the attention of a standardization committee.

Recap: XML

- XML = Extensible Markup Language
- A generic “meta-language”, designed as a set of syntax rules to encode documents. Ideas:
  - Separate document content from its representation
  - Machine-readable, but accessible for humans
  - XML is practically a subset of SGML (Standard Generalized Markup Language) from the 1980s
- Representation rules are stored in different documents
  → allows to define different representations for all kinds of output formats (HTML browsers, PDF, audio…)
- XML is used to define many markup languages you know:
  - HTML → called XHTML
  - XSL-T: transformation into other (markup) languages
  - XML Schema: used to define a markup language (!)
- Many related standards:
  - XPath: access parts of documents
  - XSL-FO: representation for a rendering device, e.g. PDF renderer (ironically, defined in prose: XML Schema is not powerful enough…)
- XML is used in the definition of practically all Web Services Standards!
  - E.g. SOAP, WSDL

Example: XML Fragment

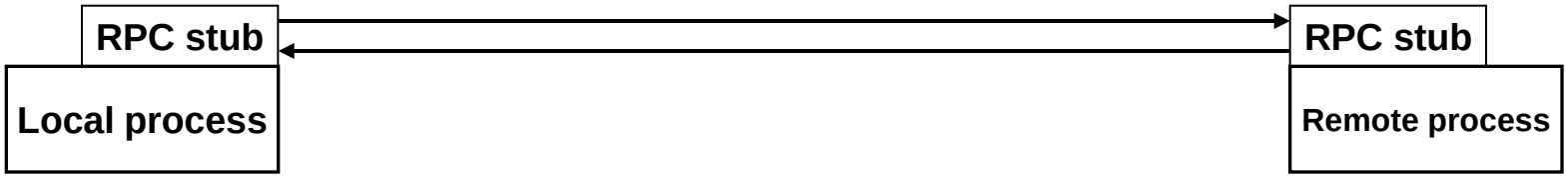
- XML documents can be **highly complex**:
  - Tree structured
  - Can be deeply nested
  - White spaces etc. have no semantics → only structure counts
- This makes **parsing XML computationally very intensive**
- Example (a kind of policy document that we wrote once):

```
<accessControl>
      <policy id="0">
         <requirement>
            <dataItem name="CERTIFICATE_VERIFICATION" />
         </requirement>
         <control>
            <action id="VERIFY_CERT" retType="BOOL"/>
            <argument position="4" type="LIST">
               <item>7831fd24756d1c645843c9016c6bae2d20f476bc</item>
               <item>ff3fa9d8c509b254c3fa185bd4b85b3c9eb84a3d</item>
               <item>6dbd654eae4bd8b8a1151d8b4b5d197a275efdf3</item>
            </argument>
         </control>
         <nextStateOnPositiveEvaluation id="1"/>
         <nextStateOnNegativeEvaluation id="NACK"/>
      </policy>
</accessControl>
```

Important concepts:
- **Element ordering does not matter!**
- **White spaces do not matter!**

Web Services as a Middleware

- In many respects, **Web Services are similar to Remote Procedure Calls (RPC)**
  - Used like a local function:
    - Parameter marshalling
    - Call to remote process with parameters
    - result is returned by remote process
  - Middleware can abstract over the particularities of the communication over the network
  - Loose coupling (asynchronous)
  - Web Services are generally more complex than a simple RPC
  - But there is also a standard for RPC: XML-RPC ☺
- Web Services are **realized with with HTTP and XML**



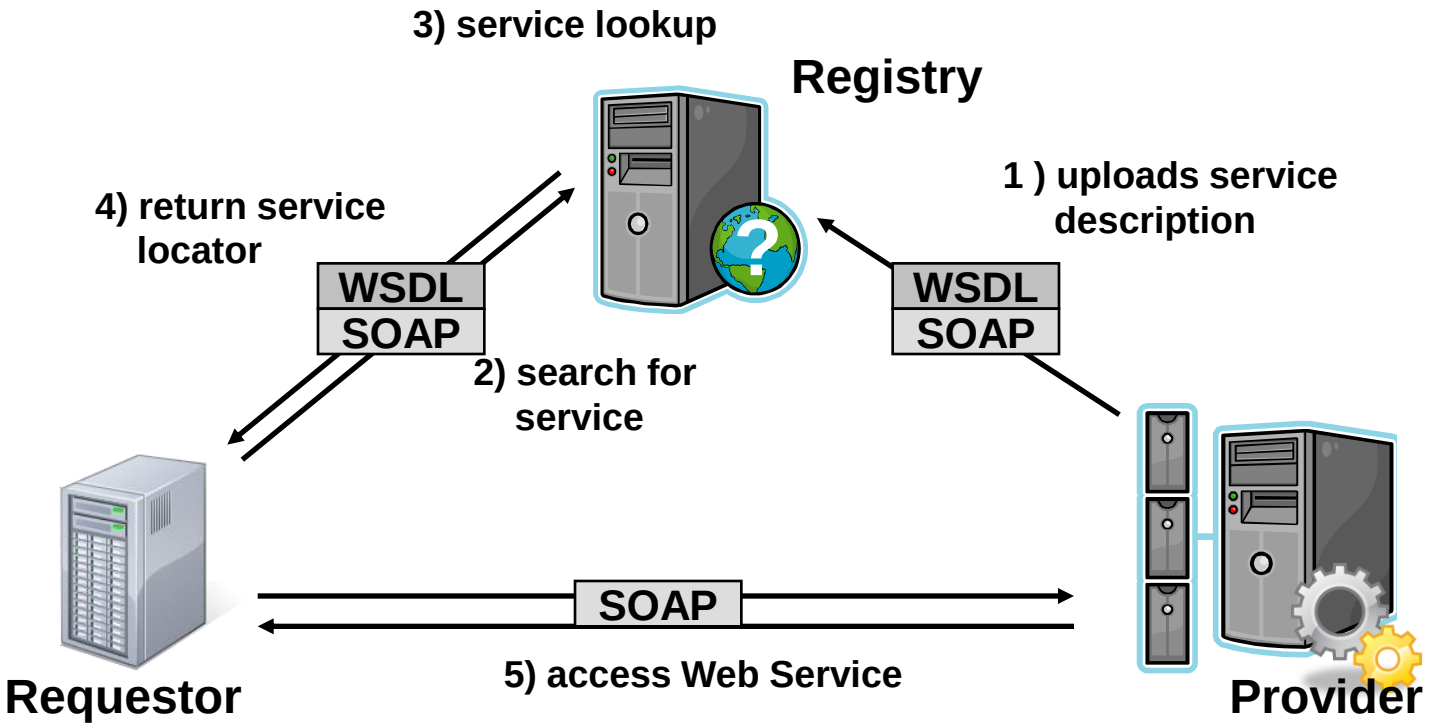
Simplified RPC example

Web Services as a Middleware

- **Why HTTP?**
  - Because HTTP technology is around, well-supported and well-accepted → easy to win support
  - But state-less property is not in favour of HTTP as Web Services realize complex work flows
- **Why XML?**
  - Already well-accepted: easy to win support from vendors
  - XML is often already a company-internal format: no conversion necessary
  - Easy to define your own, domain-specific language (e. g. B2B)
  - Relatively easy to define service composition and orchestration
- **Why not XML?**
  - Parsing very slow
  - White spaces and element ordering: have impact on encryption and signing
- On the whole, the advantages of HTTP/XML outweighed the drawbacks
  → Web Services are, after all, a very industry-relevant concept

Original Vision of Web Services

- **Building blocks: a classic architecture**
  - **Service description**
    WSDL: Web Services Description Language
  - **Service discovery: via a registry**
    (possibly UDDI = Universal Description, Discovery and Integration)
  - **Interaction: carrier protocol: SOAP**
    (SOAP used to be an acronym, now it's just a name)



More Web Services Visions

- **Service composition:**
  - A service is composed of many sub-services, all defined in WSDL
- **Business-process orchestration:**
  - Model business processes with Web Services
  - WS-BPEL: Business Process Execution Language
- **Service-Oriented Architecture:**
  - Paradigm of software architecture
  - Realize functionality as a composition of services
  - Can be done with Web Services (other frameworks possible)
- We will not go into more detail here
- The idea you should get is: **Web Services can be very complex**
- **Complexity means attack surface and room for mistakes**
  … especially if:
  - Self-designed interactions are possible
  - Self-designed security mechanisms are possible

Web Services: Our Focus

- There are a **large** number of Web Service standards
- Many complement each other, some are competitors
  - Just browse oasis-open.org
- We will mostly discuss:
  - SOAP + XML Encryption + XML Signature
  - SAML: Security Assertion Markup Language
  - Identity Federation standards

- It is almost impossible to discuss all aspects of Web Services and their security
  - There are whole lectures just on this topic (use Google)

- *“The nice thing about standards is that there are so many to choose from. And if you really don't like all the standards you just have to wait another year until the one arises you are looking for.”*

  - A. Tanenbaum

Part II: Securing Web Services

- Part I: Introduction to XML and Web Services
- Part II: Securing Web Services
- Part III: Identity Federation

Securing Web Services

- **Security Challenges**
  - Securing Identities
  - Securing Messages
    (Web Service communication is always message-based)
  - Securing multi-hop message flows
    - In particular important for Service Oriented Architectures

- **Web Service security and other protocols** are not mutually exclusive:
  - You can use SSL to create a secure pipe **between hosts**
  - Interoperate well with, e. g.
    - X.509 certificates (if you have a PKI)
    - Kerberos tokens

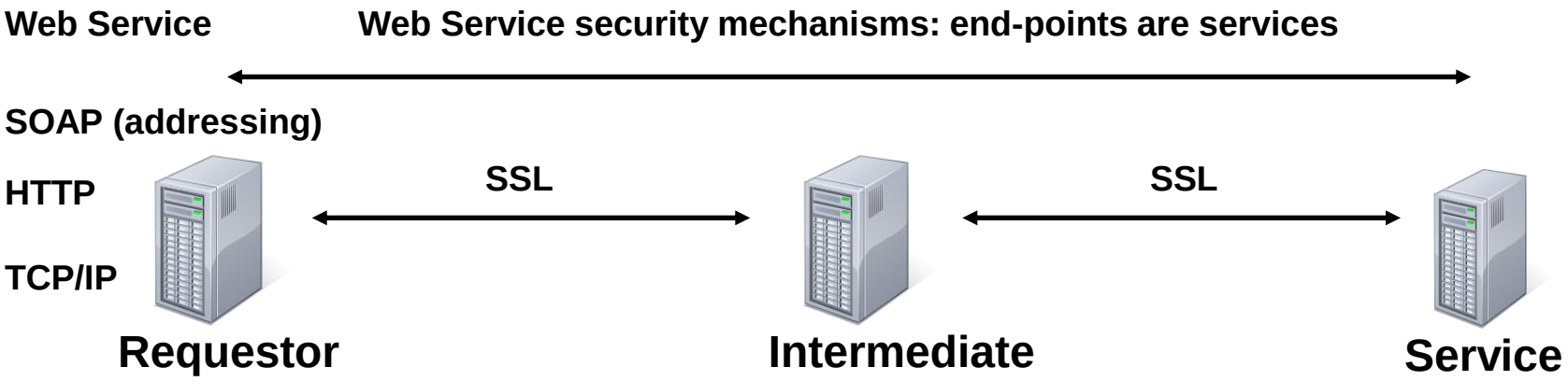
- **Question: why not just use SSL?**
  - SSL secures an underlying TCP/IP connection (“bit pipe”)
    → SSL is point-to-point between hosts
  - The identity of a Web Service is not the identity of the bit pipe: different end-point (service, not host)
  - SSL is no help in multi-hop scenarios

Securing Web Services

- Web Service documents may be business-relevant:
  may need **legally binding signatures**
  → SSL would secure the wrong endpoint
  → Same reason why you encrypt e-mails:
  the endpoint is not the host, but the (human) user

- Web Service documents may pass intermediaries
  (e. g. SOA, service orchestration):
  these may inspect & change documents en route

  → SSL provides only end-to-end semantics
  → Need for cryptographics mechanisms that allow such modifications



Security-relevant Standards

- A **number of standards** address security for Web Services
- **XML Encryption (XML-Enc, W3C)**
  - Defines how to encrypt XML content
- **XML Digital Signature (XML DSig, W3C)**
  - Defines how to sign XML content
- **WS-Security (OASIS)**:
  - Describes how to use XML Encryption and XML Digital Signature to secure SOAP messages
- **SAML: Security Assertion Markup Language (OASIS)**
  - Describes how to create and exchange authentication and authorization tokens (“assertion”)
  - Designed for interaction between an Identity Provider and an Service Provider
  - Uses XML-Enc and XML Dsig

- We will first look at XML Dsig

XML Signature (XML DSig)

- **Idea: sign an XML document**
- **Stated goals**:
  - Message Integrity
  - Origin Authenticity (sender)
  - Non-repudiation
    - This is remarkable as meaningful non-repudiation needs secure time-stamping
- Supports the **usual cryptographic mechanisms**:
  - HMACs (shared key cryptography)
  - Signatures with public-key cryptography
- An **XML signature is an XML fragment** itself
- However, the signature mechanism has been modified to work with **XML's pecularities**

XML Signature

- **Designed to**:
  - **Sign** anything that can be **referenced by a URI**
  - Even if the URI points to a location **outside** of the XML document (!)
  - Thus, can be applied to a part of the document or the whole document, or also some external document
  - Multiple signatures on a document are allowed (business uses, multi-party agreement etc.)
- **Pitfalls**: because you sign XML, you need to take care of:
  - White spaces (tabs vs. spaces)
  - Line endings (Windows world vs. UNIX world…)
  - Character-set encoding (UTF-8? ISO-8859-1?)
  - Escape sequences
  - etc.
- XML Signature thus needs to **canonicalize the document prior to signing**

Canonicalization (C14N) rules

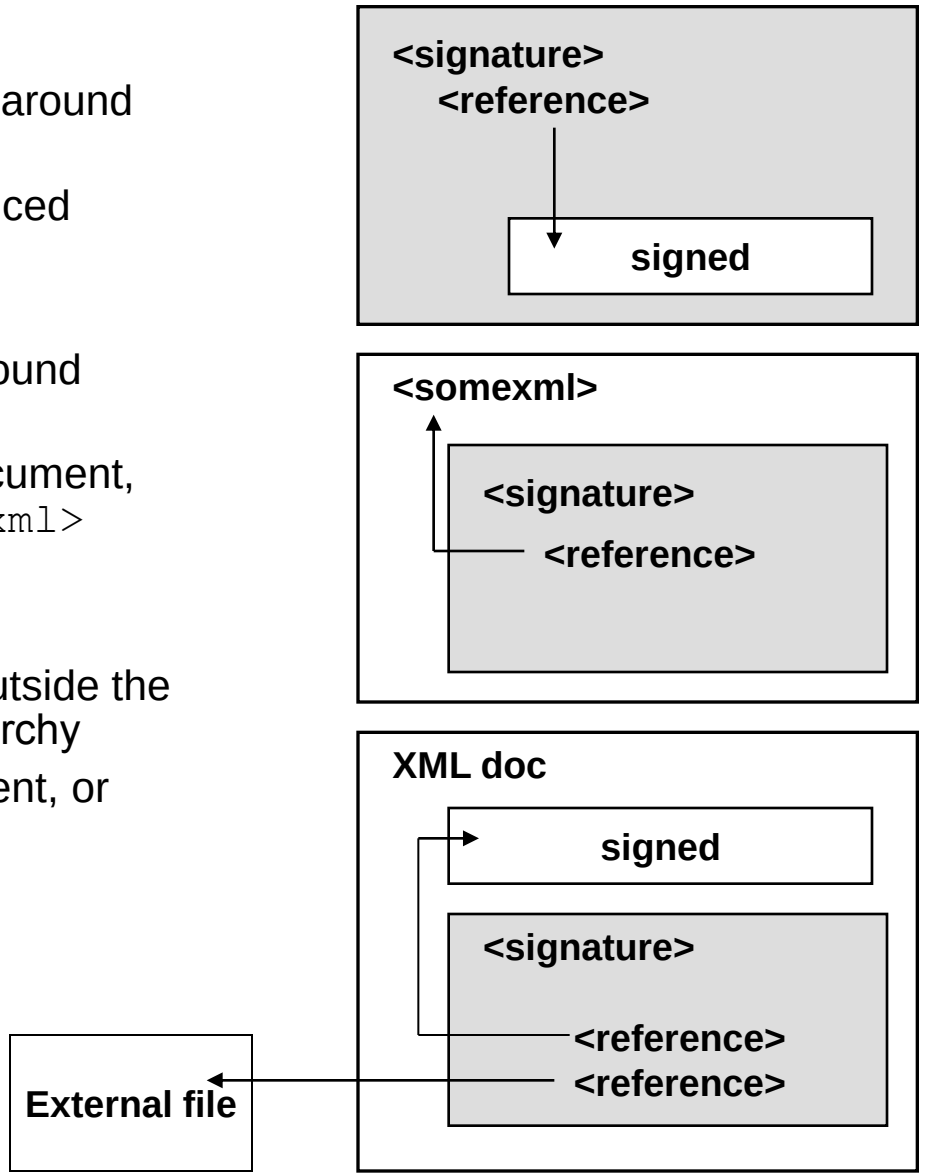
- Steps to take:
  - Encode in UTF-8
  - Normalize line breaks
  - Normalize attribute values
  - Replace character and parsed entities
  - Replace CDATA sections
  - Remove XML declaration and DTD definition
  - Normalize <element /> to <element></element>
  - Normalize whitespaces outside the document and outside elements
  - But retain it within character content
  - Set attribute value delimiters to double quotes
  - Replace special characters in attribute values and character content with character references
  - Remove superfluous namespace declarations
  - Add default attributes to each element
  - Order lexicographically on namespace declarations and attributes of each element
- Question: how fast, do you think, did programmers come up with interoperable implementations? (it took a while)

Transformations

- XML DSig allows to apply five different types of **transformations** when signing
  - Base64 (not a concern)
  - XPath Filtering (selection of elements)
  - Enveloped Signature transform (→ type of signature)
  - XSL-T transform (→ change document tree)
  - Canonicalization
- The transformations are referenced from within the signature
- They need to be reversed before the signature can be validated

Three Kinds of Signature

- **Enveloping signature**
  - `<signature>` element wraps around whole document
  - Signature is stored and referenced inside the document
- **Enveloped signature**
  - `<somexml>` element wraps around document
  - Signature stored inside the document, but reference points to `<somexml>` (parent element)
- **Detached signature**
  - Reference points to element outside the `<signature>` element's hierarchy
  - Can be inside the XML document, or outside



Code Example

- Simplified enveloping signature (not a correct document)

```
<Signature xmlns=http://www.w3.org/2000/09/xmldsig#>
  <SignedInfo>
    <CanonicalizationMethod
      Algorithm=http://www.w3.org/TR/2000...” />
    <SignatureMethod Algorithm=http://www.w3.org...” />
    <Reference URI=“#important”>
      <DigestMethod Algorithm=http://www.w3.org/...” />
      <DigestValue>60nvZ+TB7...</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>ae5fb6fc3e…</SignatureValue>
<KeyInfo>FAE6C...
  <KeyValue>
    <RSAKeyValue>
      <Modulus>uCiu...</Modulus>
    </RSAKeyValue>
  </KeyValue>
</KeyInfo>
 [...]
<Object>
  <SignedItem id=“important”>Secret stuff</SignedItem>
</Object>
</Signature>
```

Discussion of XML Signature

- **Performance:**
  - Signed documents become very large
  - Parsing, canonicalization and transformation are slow
  - Inclusion of external documents allows to design malicious documents that keep referencing more documents (DoS on the parser)
- **Complexity:**
  - Three different kinds of signature
  - Five kinds of transformations that can be applied before signing
  - Complex canonicalisation rules
  - Nonsensical possibilities to signed data and signatures are not explicitly forbidden: signature before signed data etc.
  - Makes analysis of the standard very difficult
- **Correct and comprehensive implementation is difficult**
  → interoperability is easily threatened

Discussion of XML Signature

- If **applied in a sane manner**, the **standard does provide security**
- But you need to be very careful what transformations etc. you allow
- If you want to preserve all XML features, you probably need such a complex mechanism
  → XML Signature is often called a semantic signatures, because it does not just read in the whole document and sign that

- It should be noted that other mechanisms for signatures have been developed:
  - XML-RSig: Really Simple XML Signature:
    "read the BLOB and sign it"
  - XMPP (Jabber): MIME body parts → easier
- Complex overhead is a trade-off usefulness vs. feasibility

Some opinions on XML Signature

- XML Signature has drawn both praise and severe criticism for its flexibility

- Some quotes:
  - "*XML Digital Signature is the latest and greatest technology for you to ensure integrity and non-repudiation. Its remarkable flexibility allows you to sign parts or all of XML documents as well as binary and remote objects.*"

    - J. Rosenberg, D. Remy in [RoRe2004]

  - "*They reinvented the wheel in XML, but made it square to avoid accusations that they'd just reinvented the wheel*."
  - "*Secure XML, the definitive reference on the topic, spends fully half of its 500-odd pages trying to come to grips with XML and its canonicalisation problems*."

    - P. Gutmann, University of Auckland [Gu2004]

XML Encryption (XML-Enc)

- Idea: **encrypt XML content**
- **Goal: Confidentiality**
- Just like XML Digital Signature, XML-Enc is agnostic to crypto algorithms
  - You can use it with, e. g.
    - Shared-key cryptography (3DES, AES, …)
    - Public-key cryptography (RSA, …)
  - Usual pattern with public-key cryptography:
    - Generate a symmetric key K
    - Encrypt only K with public key
    - Use K to encrypt the real content
- XML Encryption shares many features with XML Signature

XML Encryption

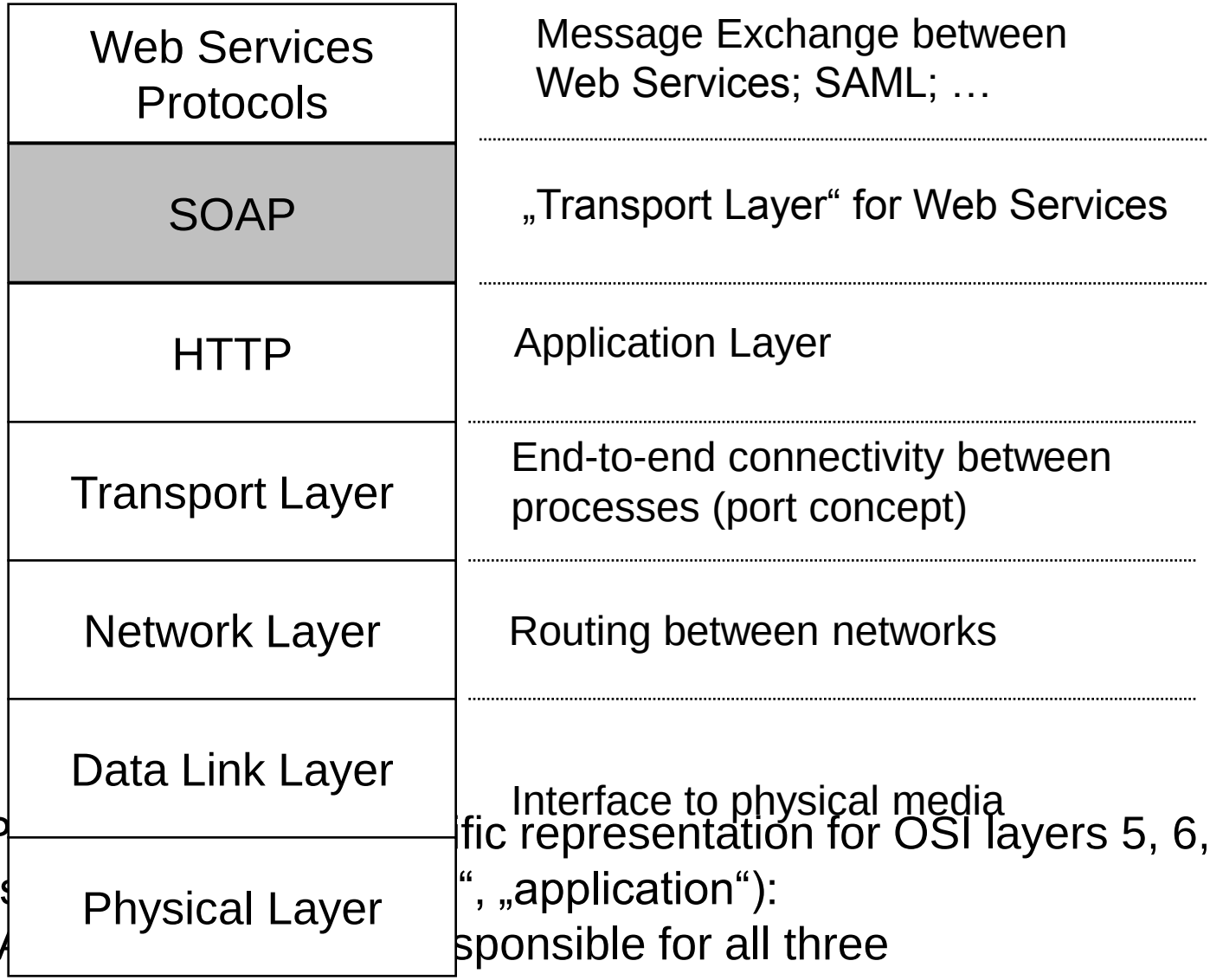
- Uses `<EncryptedData>` element
  - Either points to encrypted content
  - Or replaces unencrypted content
    (if content is within same document)
- XML-Enc and XML DSig are designed to be used together

Code Example

- Simplified again

```
<MyDoc>
<EncryptedData Id="encdata" xmlns="http://…">
  <EncryptionMethod Algorithm=http://... />
  <CipherData>
    <CipherValue>…</CipherValue>
  </CipherData>
  <EncryptionProperties>
    <EncryptionProperty Target="encdata">
      <EncryptionDate>2010-01-01</EncryptionDate>
    </EncryptionProperty>
  <Object id="encdata">...</Object>
  <Signature>
    …
  </Signature>
</MyDoc>
```

Towards a Web Services “Stack”



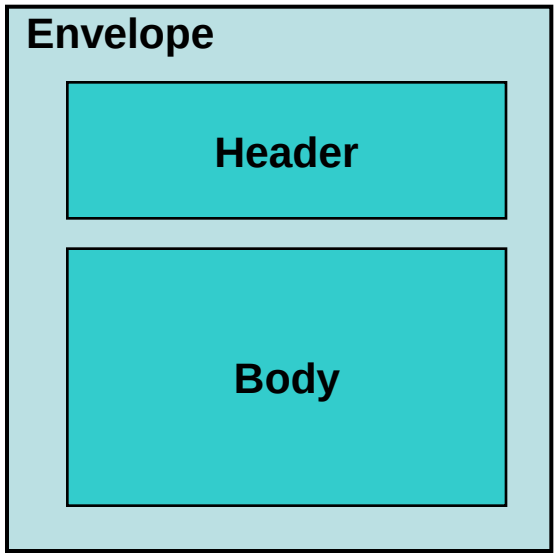
- TCP
fic representation for OSI layers 5, 6, 7
(„ses
“, „application“):
the A
sponsible for all three

Use Cases For XML DSig/XML Encryption

- You know XML Digital Signature and XML Encrytion now
- These standards **form the foundation of many Web Service security protocols**:
  - SOAP
  - WS-Security
  - SAML
  - WS-Federation
  - ID-FF (Identity Federation by Liberty Alliance)
  - …

A Closer Look At SOAP

- **Defines how to send structured XML over a network**
  - Follows paradigm of state-less, one-way messages
  - But applications can create complex communication patterns from this by supplying additional application-specific information
  - Thus, SOAP is agnostic to what it conveys
  - Used as a foundation layer for Web Service protocols
- **Simple message format:**




```
<soap:Envelope xmlns:soap="http://...">

     <soap:Header>
         <app-specific:requestor id="…" />
     </soap:Header>


     <soap:Body>
         <app-specific:request item="…" />
     </soap:Body>



</soap:Envelope>
```

SOAP

- SOAP defines **bindings**: important specifications how to use SOAP with underlying protocols
  - HTTP + (SSL +) TCP
  - SMTP

- Some criticism:
  - May lead to abuse of HTTP semantics
  - Firewalls are often configured to accept HTTP → must now inspect XML content → increases attack surface
  - However, HTTP is a core element in Web Services anyway

- **Information in SOAP can be cryptographically secured with XML Signature and Encryption**
- However, there are many ways to get this wrong!
  → self-designed crypto protocols are often flawed

Security Issues To Think About

- Web Services are a **valuable target for attackers**
  → business-relevant data = money

- We have seen that **XML Signature and XML Encryption can provide security, but at the price of high complexity**
- **Designing a crypto protocol** and protocol handlers must thus be done with **extra great care here**
  - Simple example: ***first*** verify that the signature is from a known key, ***then*** do the signature check
  - Otherwise, you leave yourself open to complexity or DoS attacks

- Some further attacks to think of:
  - SQL injection
  - XPath and XQuery injection
  - Complexity and DoS attacks on parsers
  - More are listed on owasp.org

Example of Parser DoS: Entity Expansion

- The following may expand to 2 GB when parsed
  (note: we did not try it; it probably depends on the parser)

```
<!DOCTYPE foo [
<!ENTITY a "1234567890" >
<!ENTITY b "&a;&a;&a;&a;&a;&a;&a;&a;" >
<!ENTITY c "&b;&b;&b;&b;&b;&b;&b;&b;" >
<!ENTITY d "&c;&c;&c;&c;&c;&c;&c;&c;" >
<!ENTITY e "&d;&d;&d;&d;&d;&d;&d;&d;" >
<!ENTITY f "&e;&e;&e;&e;&e;&e;&e;&e;" >
<!ENTITY g "&f;&f;&f;&f;&f;&f;&f;&f;" >
<!ENTITY h "&g;&g;&g;&g;&g;&g;&g;&g;" >
<!ENTITY i "&h;&h;&h;&h;&h;&h;&h;&h;" >
<!ENTITY j "&i;&i;&i;&i;&i;&i;&i;&i;" >
<!ENTITY k "&j;&j;&j;&j;&j;&j;&j;&j;" >
<!ENTITY l "&k;&k;&k;&k;&k;&k;&k;&k;" >
<!ENTITY m "&l;&l;&l;&l;&l;&l;&l;&l;" >
]>
<foo> fooo &m; bar </foo>
```

Source: [iSec2010]

Securing SOAP: WS-Security

- Framework that **defines how XML Signature and XML Encryption can be employed safely for SOAP and XML-based application protocols**
- **WS-Security does not define new mechanisms**
  → "standardizing the standards"
- Some WS-Security Features:
  - **Signatures** with XML Signature (sane methods)
  - **Encryption** with XML Encryption (sane methods)
  - **Transports Security Tokens**:
    - X.509 certificates
    - Kerberos Tokens
    - SAML Tokens (more about SAML shortly)
    - Passwords
    - Password digests
  - **Timestamps**
- Also describes alternatives for use cases where only host-to-host security is required: simpler, uses SSL/TLS

WS-I BSP

- **WS-Interoperability Basic Security Profile**
  - A standard by the Web Services Interoperability Organization
  - Defines comprehensively how to use the mechanisms in Web Services security ***safely***
- Intent is **clarification** → improve ease of use
- **Some remarkable points:**
  - Prohibits the use of some protocols with flaws, like older SSL versions (SSL 2.0 disallowed!)
  - Defines ciphersuites to use
  - Restrictions on SOAP envelope, header and processing
  - Enveloping XML Signature disallowed, enveloped signature discouraged → emphasis on detached signature!
  - Rules for transforms
  - Rules to facilitate encryption processing

Security Assertion Markup Language (SAML)

- **Motivation for SAML**:
  - Web Services may cross organisational boundaries
    → need for authentication and authorization for access control
    → convey “security attributes” between organisations
  - Portable (shared) “identities” with attributes between organisations
- **SAML works with assertions.** We speak of:
  - Subject: an entity that is asserting its identity
  - Assertion: a claim about a subject that must be proved
- **SAML can be used to exchange assertions between organisations**
- **SAML consists of three parts:**
  - Assertions
  - Protocol: XML schema and request/response protocol
  - Bindings: e. g. to SOAP/HTTP
- So-called **SAML Profiles** specify **use patterns** for SAML, i.e. how assertions are embedded, extracted and processed
  - E. g. a profile for use with Web Browsers

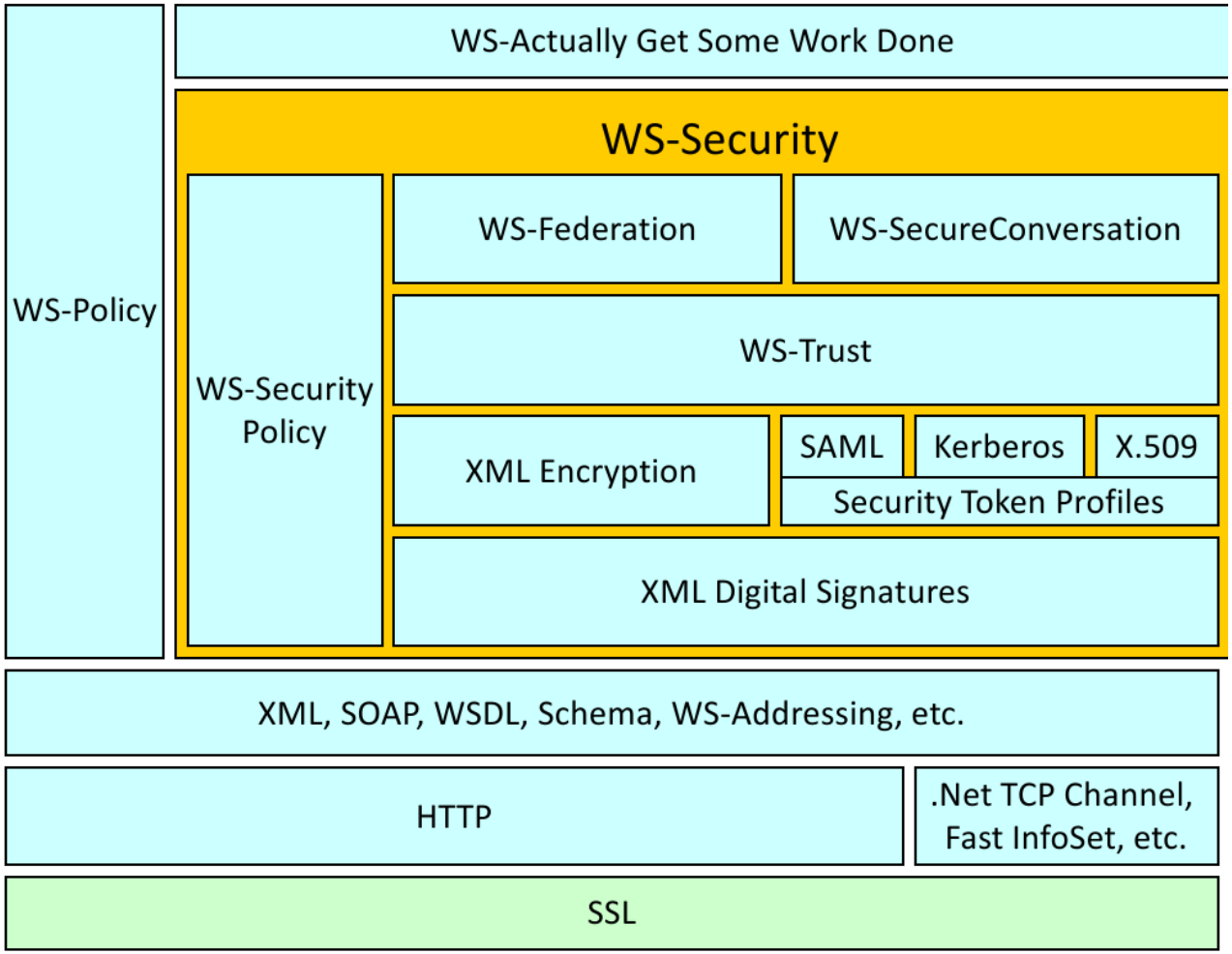
SAML Assertions

- **Three types of assertions**:
  - **Authentication**: states that an authority has authenticated the subject of the assertion
  - **Authorization**: states that an authority has granted or denied access to the subject of the assertion
  - **Attributes**: qualifying information about an authentication or authorization
- Some elements that are **common to all assertions**:
  - Issuer
  - Timestamp
  - Subject
  - Conditions on assertion (e. g. “not valid after…”)
  - Intended audience
  - Signatures

Example: SAML Authentication Response

```
<samlp:Response xmlns:samlp="urn:..." InResponseTo="..." Version="2.0" IssueInstant="2007-12-10T11:39:48Z"
        Destination="...">
    <saml:Issuer>the-issuer</saml:Issuer>
    <samlp:Status xmlns:samlp="...">
        <samlp:StatusCode xmlns:samlp="..." Value="urn:oasis:names:tc:SAML:2.0:status:Success" />
    </samlp:Status>
    <saml:Assertion xmlns:saml="urn:..." Version="2.0" ID="..." IssueInstant="2007-12-10T11:39:48Z">
        <saml:Issuer>the-issuer</saml:Issuer>
        <Signature xmlns="...">
          ...
        </Signature>
        <saml:Subject>
            <saml:NameID>...</saml:NameID>
            <saml:SubjectConfirmation Method="...">
                <saml:SubjectConfirmationData>...</saml:SubjectConfirmationData>
            </saml:SubjectConfirmation>
        </saml:Subject>
        <saml:Conditions NotBefore="2007-12-10T11:29:48Z" NotOnOrAfter="2007-12-10T19:39:48Z">
          ... e. g. audience restrictions
        </saml:Conditions>
        <saml:AuthnStatement AuthnInstant="2007-12-10T11:39:48Z" SessionIndex="...">
            <saml:AuthnContext>
                <saml:AuthnContextClassRef>urn:...Password</saml:AuthnContextClassRef>
            </saml:AuthnContext>
        </saml:AuthnStatement>
        <saml:AttributeStatement>
            <saml:Attribute Name="givenName">
                <saml:AttributeValue xmlns:saml="...">...</saml:AttributeValue>
            </saml:Attribute>
              ... more attributes ...
        </saml:AttributeStatement>
    </saml:Assertion>
</samlp:Response>
```

Bringing It All Together



Source: [iSec2010]

Recap: Security Guidelines for Web Services

- **Recommendations in several standards**
  - WS-Security
  - WS-I Basic Security Profile
  - Following these recommendations is strongly encouraged
- **Decrease attack surface**:
  - Always use SSL/TLS for host-to-host communication
  - Complexity is (one) enemy of security
  - Where you can, reduce the complexity of your protocol
- **Do not create/use protocols that you do not actually need**
  - Even SAML Profiles have been found to have weaknesses
- **Do not forget attacks outside cryptography**:
  - DoS
  - Injection attacks
- **Conclusion**: Security for Web Services can be much work and should be addressed with great care.

Further Pointers

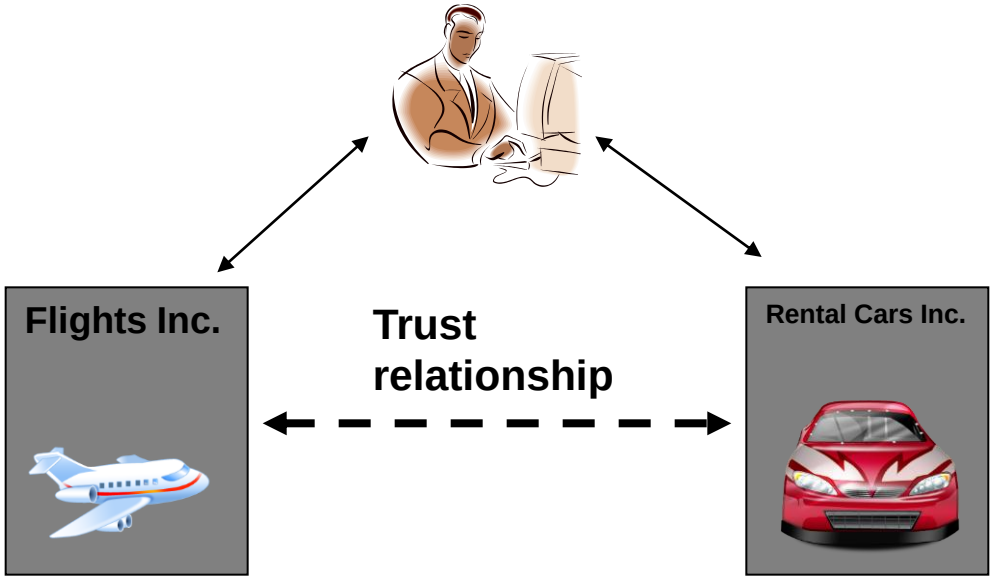
- There are more security-relevant standards, which we will not discuss further here
- Have a look yourself, if you want, at:
  - WS-SecureConversation
    → establishes security contexts, SSL-like pattern
  - WS-Reliability
    - Reliable communication for, e.g., transactions
  - WS-Trust
  - WS-Policy
  - WS-Interoperability

Part III: Identity Federation

- Part I: Introduction to XML and Web Services
- Part II: Securing Web Services
- Part III: Identity Federation

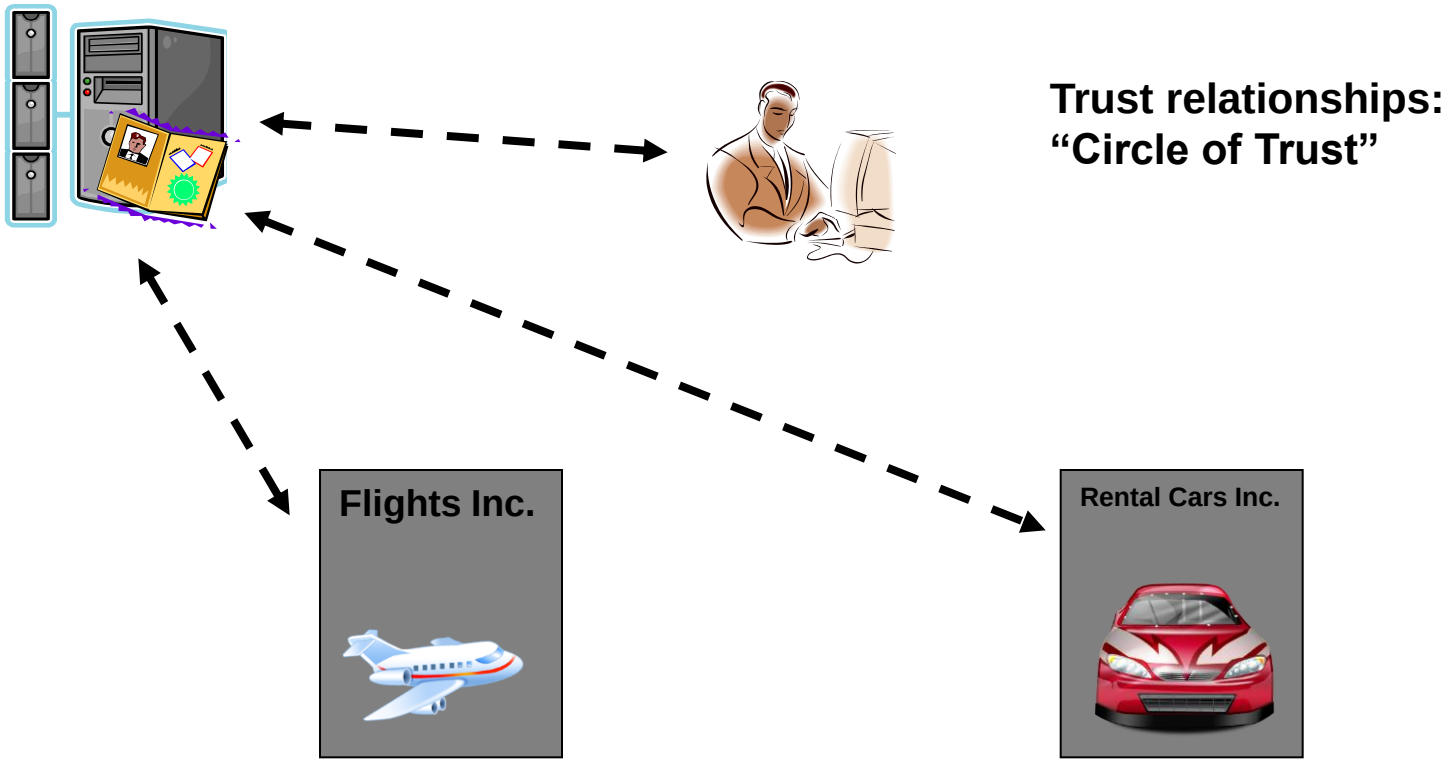
Identity Federation As Shared Authentication

- Entity Bob wishes to do business:
  - Bob wants to reserve a flight from Flights Inc.
  - Bob also wants to rent a car from Rental Cars Inc.
- On booking the flight, Bob consents to federate an identity
  - A pseudonym for use with Rental Cars Inc. is generated
  - Bob is redirected to Rental Cars Inc. with a security token that proves his membership with Flights Inc. (with the pseudonym!)
    Assertion: *"pseudo_bob is a member of domain Flights Inc."*
- Identity Federation: propagation of trust / authentication across organizational boundaries



Identity Provider

- Example may be extended by having a third party acting as the Identity Provider for Bob
- Bob authenticates with credential from Identity Provider



Identity Federation: Concepts

- **Concept is not new: sharing of Identities between organisations**
  - Portability of an identity
  - You know similar concepts, e. g. Kerberos
- **Use-cases**:
  - Allows users (or Web Services) to access services outside their own administrative domain
  - Most common example: Single Sign-On
- **Several standards implement Identity Federation,** also with Web Service technology, esp. SAML:
  - WS Federation (OASIS), part of the Web Services suite
  - ID-FF by Liberty Alliance: large consortium to establish open standards for Identity Federation
  - Shibboleth (Internet2)
  - OpenID: decentralized, more “community-oriented” and simpler standard

Identity Federation: Concepts

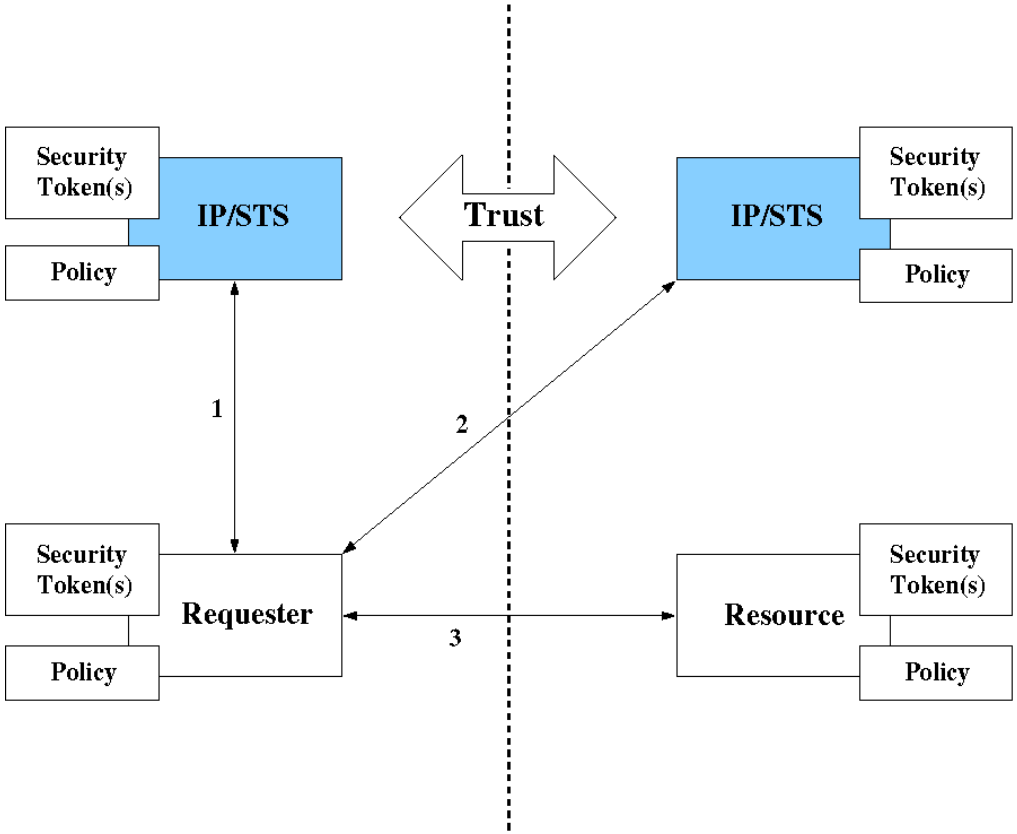
- The **basic schema** is always the same
  - An **entity has an Identity Provider (IdP) vouching for its identity**
  - In order to access a service, the entity **requests a credential from IdP**
    - May be explicitly for the service or generic
  - Entity **presents this credential to the Service Provider**

- Participants in an Identity Federation form a **"Circle of Trust"**
  - Within this circle of trust, an entity may use its federated identity to authenticate, access services etc.
  - Any organisation may act as an Identity Provider (if it is trusted by reyling participants)

- Nota bene: concepts like **Identity Management** that (may) build on Identity Federation **require much more than the pure security concepts** we present here
  - Validity between domains
  - Expiry
  - Secure administration
  - Roles & Access Control
  - Etc.

Identity Federation: Relationships 1

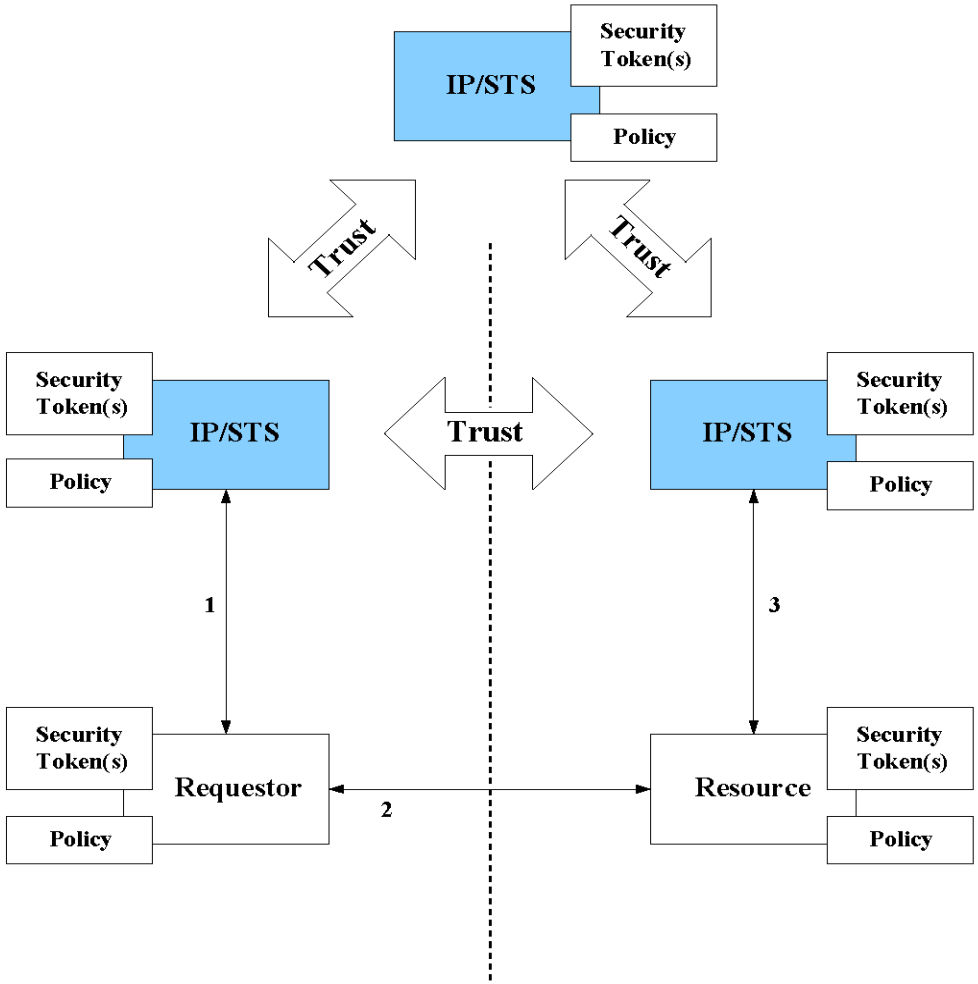
Note:
STS = Security Token Service



- **Simple model: direct trust between organisations**
  - Each organisation has an Identity Provider
  - Requester asks for a credential from his Identity Provider and presents it to the STS of the Service Provider he wishes to access
  - That STS may then grant access to the service
- Each participant may follow his own policies in this process

Identity Federation: Relationships 2

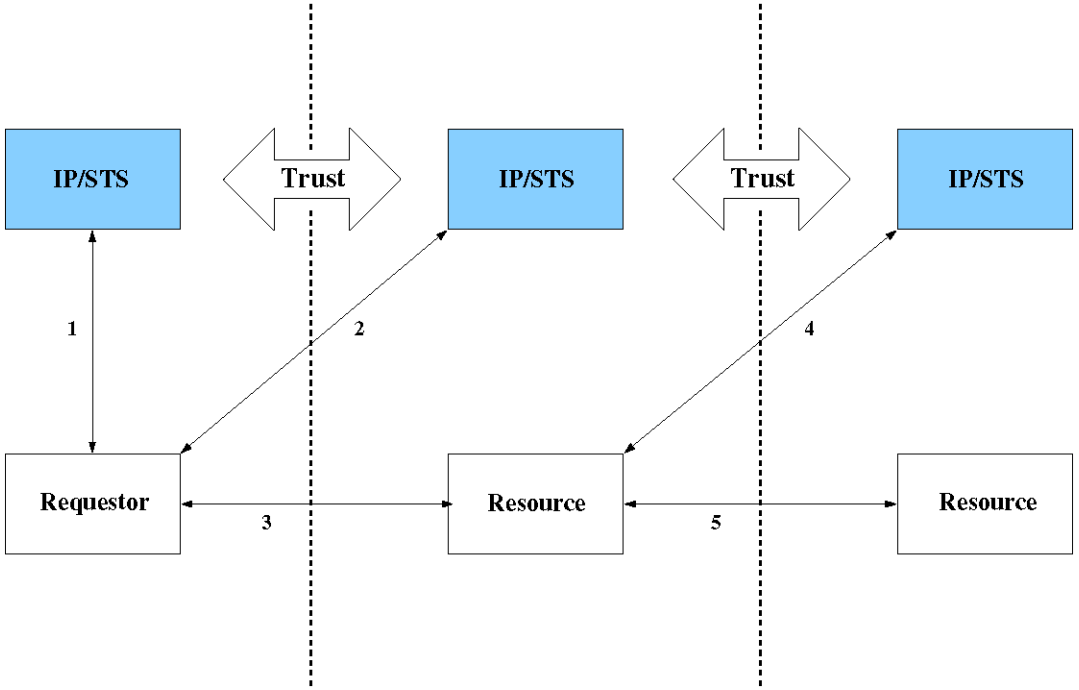
Note:
STS = Security Token Service



- **Extended model: trust between organisations is mediated by a Trusted Third Party**

Identity Federation: Relationships 3

Note:
STS = Security Token Service



- **Extended model with delegation**:
  - In order to fulfill a request, a resource accesses another (third-party) resource first
  - First resource acts “on behalf” of requestor

OpenID

- OpenID is a "**more decentralized"** system for Identity Federation
  - **No *a priori* trust** relationships envisaged → no Circles of Trust
  - Idea is that you **login with an identity you registered with an OpenID provider**
  - It is left to the Service Provider to decide whether to accept authentication with an unknown OpenID provider
- **Some features**:
  - **XML-based**
  - Supports **Discovery** mechanisms for OpenID providers
  - More **aimed at a Web scenario**: less comprehensive and generic in comparison with Web Services standards
  - Allows **delegation**: you can **host your own identity** and delegate each authentication process to your OpenID provider
  - OpenID is **well supported** on the Web

References WWW Security

[RFC3986] *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. http://tools.ietf.org/html/rfc3986

[RFC2965] *HTTP State Management Mechanism*. RFC 2965. http://tools.ietf.org/html/rfc2965

[ECMA262] *ECMAScript Language Specification*. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf

[Sym2009] Symantec. *Symantec Report on the Underground Economy*. Symantec. 2009. http://www.symantec.com

[HoEnFr2008] T. Holz, M. Engelberth, F. Freiling. *Learning More About the Underground Economy: a Case Study of Keyloggers and Dropzones*. Technical Report TR-2008-006. Universität Mannheim. 2008.

[Za2011] M. Zalewski. *The Tangled Web – a guide to securing modern Web applications*. No Starch Press. 2011.

[HoLe2002] M. Howard, D. LeBlanc. *Writing Secure Code*. Microsoft Press. 2002.

[Wil2009] T. Wilhelm. *Professional Penetration Testing*. Syngress Media. 2009.

[ISec2010] International Secure Systems Lab. http://www.iseclab.org. 2010.

[Mo2010] Timothy D. Morgan. *Weaning the Web off of Session Cookies: Making Digest Authentication Viable*. http://www.vsecurity.com/download/papers/WeaningTheWebOffOfSessionCookies.pdf

References Web Service Security

[XMLEnc] W3C. *XML Encryption*. http://www.w3.org/standards/techs/xmlenc.

[XMLDSig] W3C. *XML Signature*. http://www.w3.org/standards/techs/xmlsig

[Gu2004] P. Gutmann. *Why XML Security is Broken.* http://www.cs.auckland.ac.nz/~pgut001/pubs/xmlsec.txt. 2004.

[RoRe2004] J. Rosenberg, D. Remy. *Securing Web Services with WS-Security*. SAMS Publishing. 2004.

[XMPPSig] RFC 3923. *End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP)*.

[iSecAttack] iSEC Partners. Attacking XML Security. http://www.isecpartners.com/files/iSEC_HILL_AttackingXMLSecurity_bh07.pdf

[SAML2010] OASIS. *OASIS Security Services (SAML) TC*. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security

[OWASP] Open Web Application Security Project. 2010. http://www.owasp.org

[WSI] Web Services Interoperability Organization. *Basic Security Profile Version 1.0*. 2010. http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html

[OpenID] OpenID Foundation Web Site. http://openid.net/

[iSec2010] iSEC Partners. *Attacking XML Security*. http://www.isecpartners.com/files/iSEC_HILL_AttackingXMLSecurity_bh07.pdf