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

Network Security

Chapter 10

WWW and Application Layer Security

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

Recap: Internet Protocol Suite

Application Layer	Application protocols: e. g. HTTP, SIP, Instant Messengers, …
Transport Layer	End-to-end connectivity between processes (port concept)
Network Layer	Routing between networks
Data Link Layer	Interface to physical media
Physical Layer	

- 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

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: small text files that the server makes the browser store
    - Client authenticates to server → receives cookie with a "secret" value → use this value to keep the session alive (re-transmit)
  - 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

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)

A Few More Aspects

- 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

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: script may only access other resources on the Web if it comes from the same origin
  - Same-origin policy can be violated with Cross-Site Scripting

---

Part II: Internet Crime

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

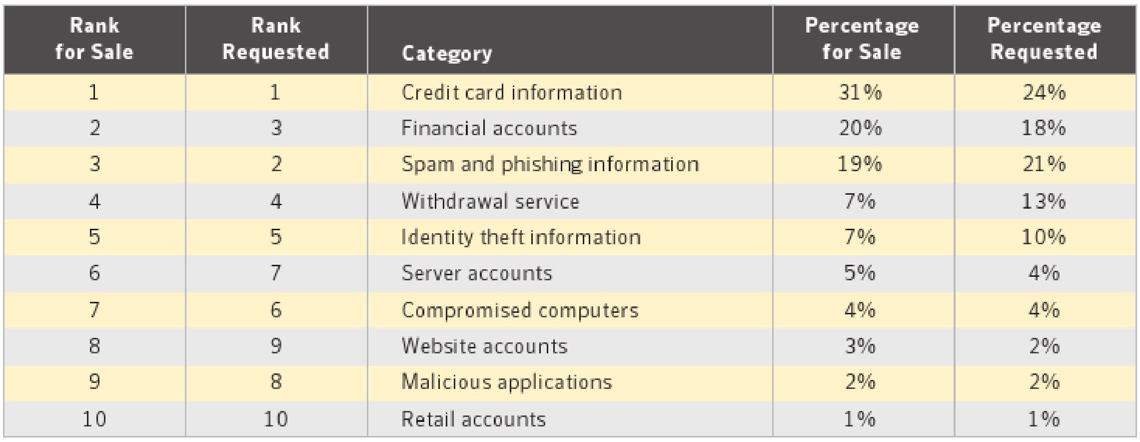
Vulnerabilities: some numbers

- 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]

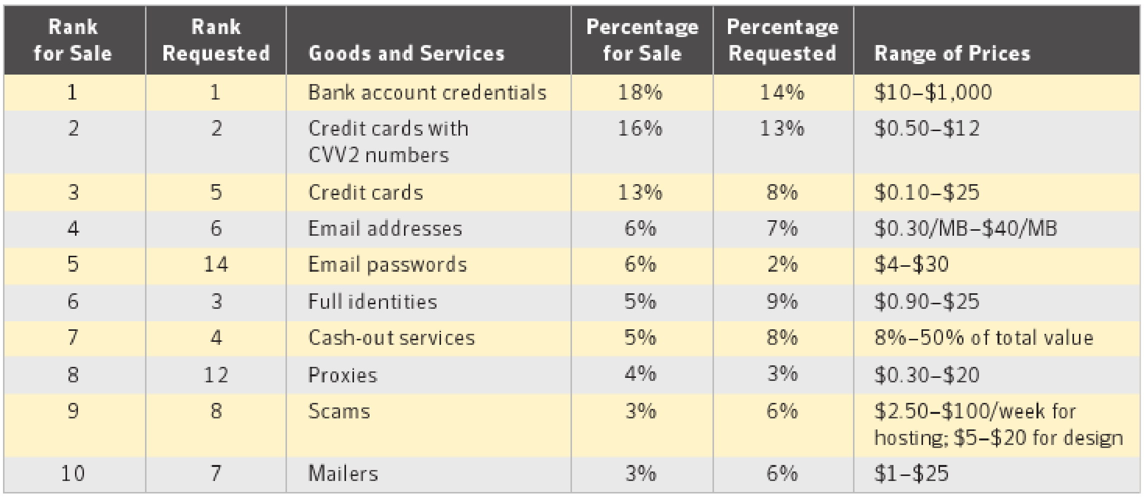
From the Symantec Report 2008 (1/4)



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

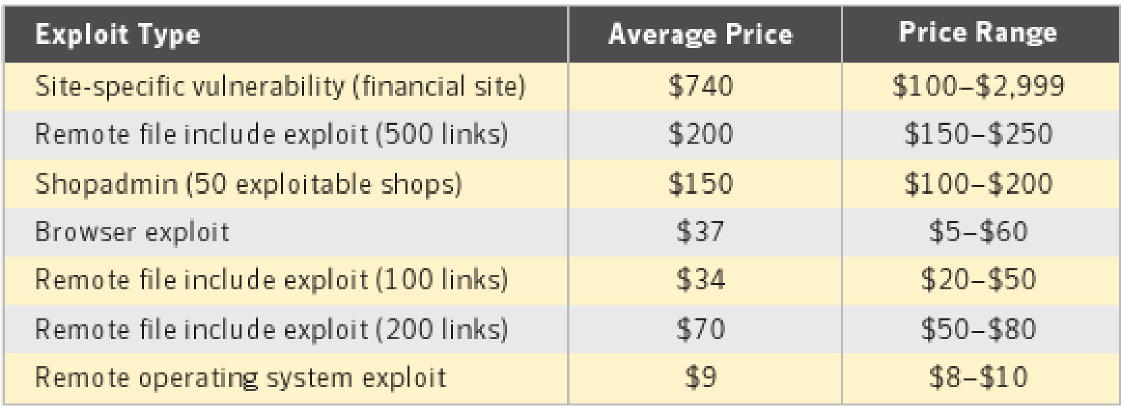
From the Symantec Report 2008 (2/4)



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]

From the Symantec Report 2008 (3/4)

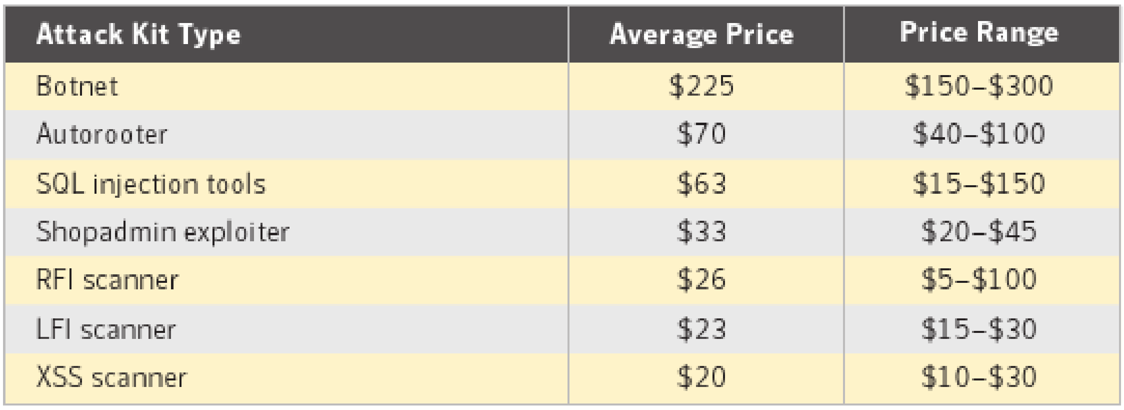


Exploit Type	Average Price	Price Range
Site-specific vulnerability (financial site)	$740	$100–$2,999
Remote file include exploit (500 links)	$200	$150–$250
Shopadmin (50 exploitable shops)	$150	$100–$200
Browser exploit	$37	$5–$60
Remote file include exploit (100 links)	$34	$20–$50
Remote file include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

Table 8. Exploit prices

Source: Symantec Corporation

From the Symantec Report 2008 (4/4)



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

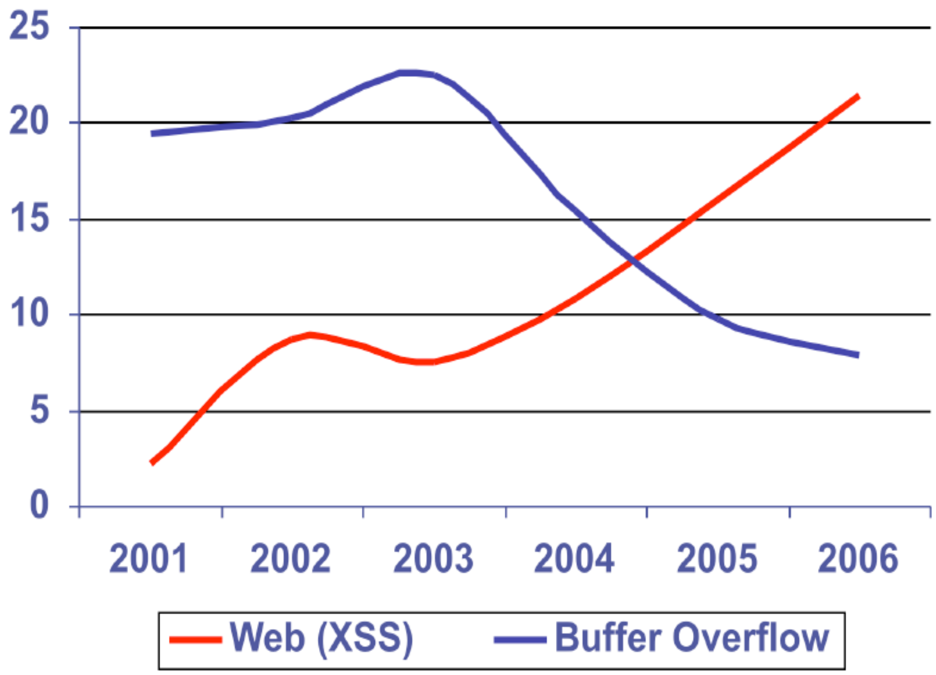
Table 5. Attack kit prices

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



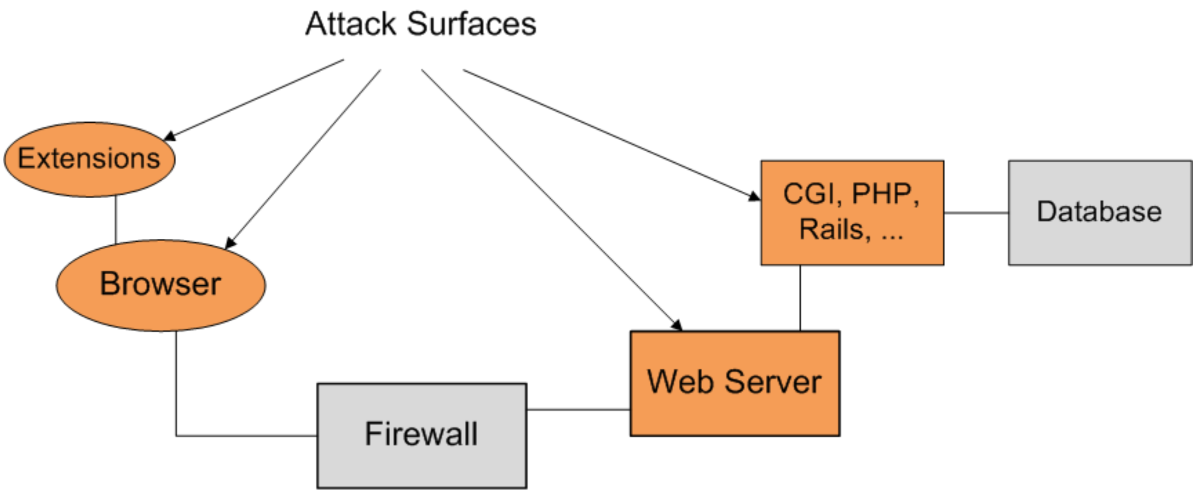
Source: MITRE CVE trends

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



Attack Surfaces

Extensions

Browser

Firewall

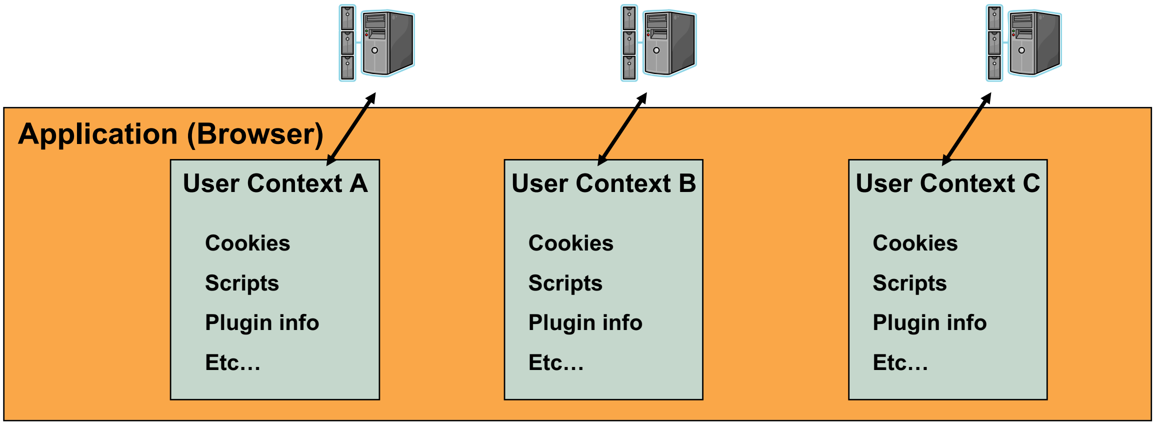
Web Server

CGI, PHP, Rails, ...

Database

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



Application (Browser)

User Context A

Cookies
Scripts
Plugin info
Etc…

User Context B

Cookies
Scripts
Plugin info
Etc…

User Context C

Cookies
Scripts
Plugin info
Etc…

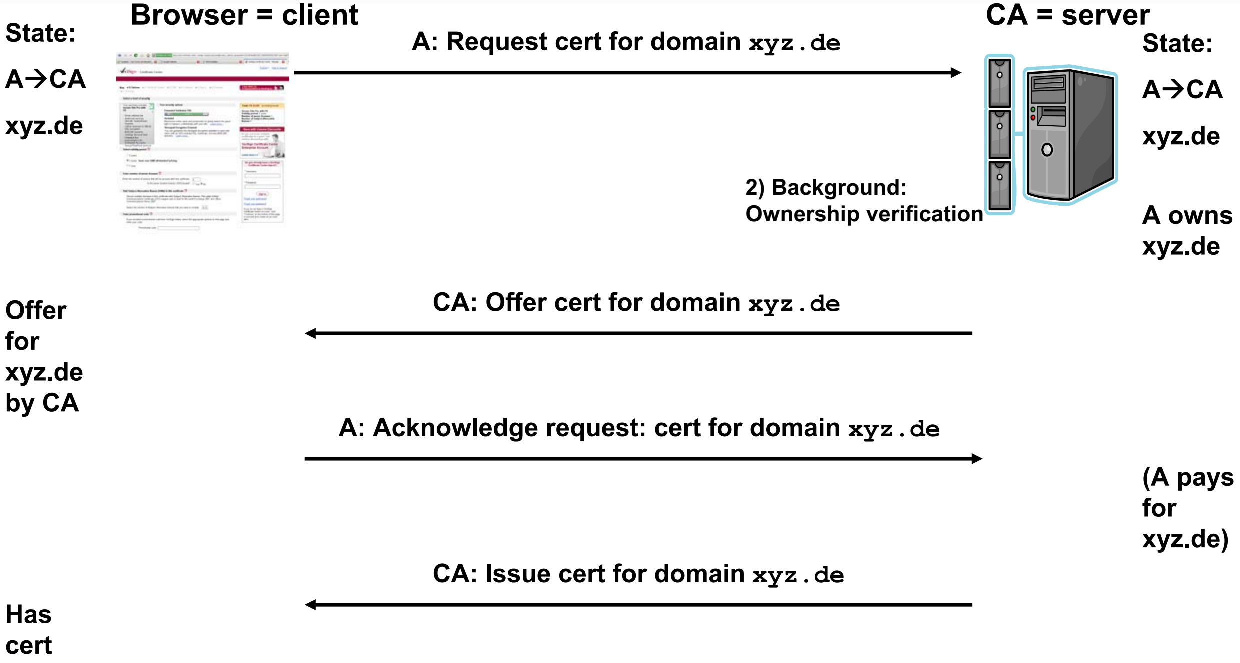
Attack 1: Session Variables

- ❑ **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: (client)
A→CA
xyz.de

A: Request cert for domain `xyz.de`

2) Background:
Ownership verification

State: (server)
A→CA
xyz.de

A owns xyz.de

CA: Offer cert for domain `xyz.de`

Offer for xyz.de by CA

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

(A pays for xyz.de)

CA: Issue cert for domain `xyz.de`

Has cert

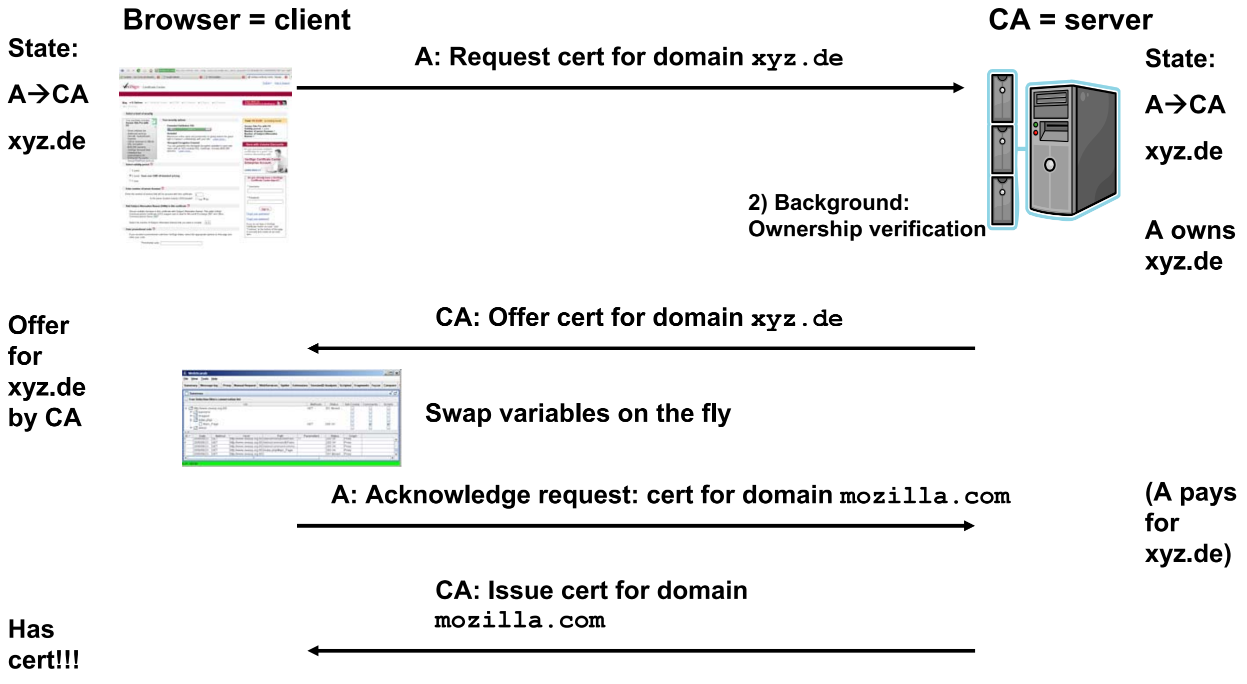
Question: where do you keep the session information?

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

CA: Offer cert for domain `xyz.de`

Offer for xyz.de by CA

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!

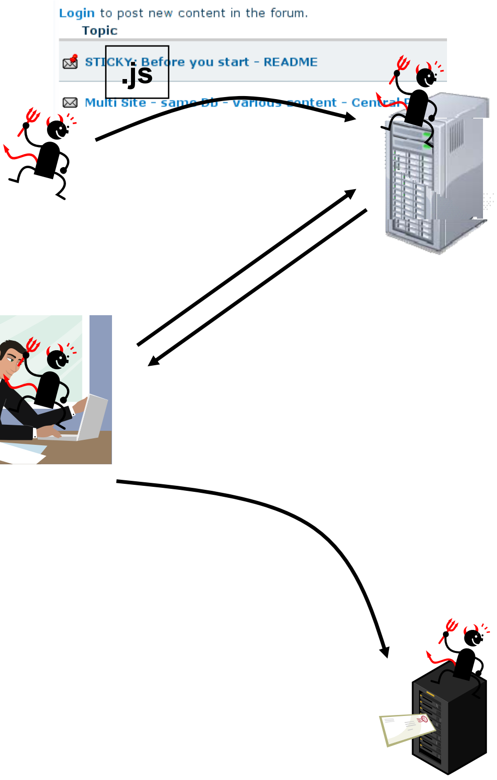
- This was just a simple attack because an entity failed to obey these rules.
- In particular, Guideline 1 was violated.
- However, in the following, we show you that attacks are possible even if state is stored correctly and only Guideline 2 is violated.

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



Login to post new content in the forum.

Topic

STICKY: Before you start - README

Multi Site - same DB - various content - Central

- 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***

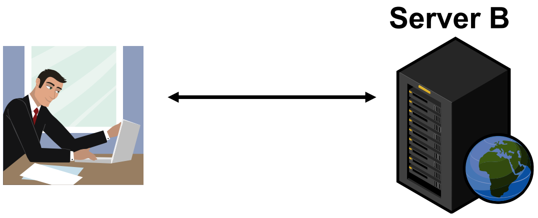
Cross-Site Request Forgery

- **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
  - 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

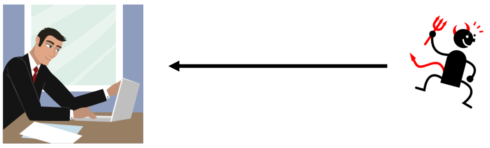
---

Cross-Site Request Forgery

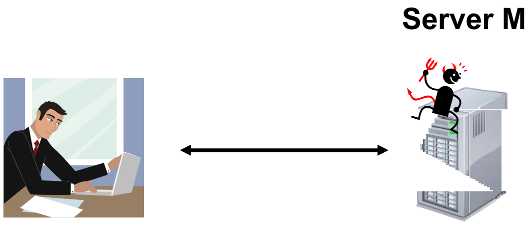
- Stage 1: user logs into Web site
  - Authenticated user
  - Session with server B
  - User keeps this session open



- 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
    <p>harmless text</p>
    ****
    <p>more harmless text</p>



→ **undesired action executed**

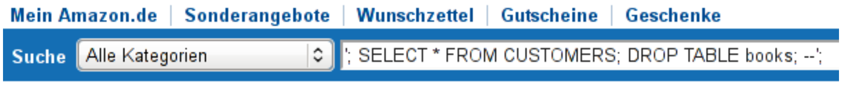
SQL Injection

- **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:



'; SELECT * FROM CUSTOMERS; DROP TABLE books; --';

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

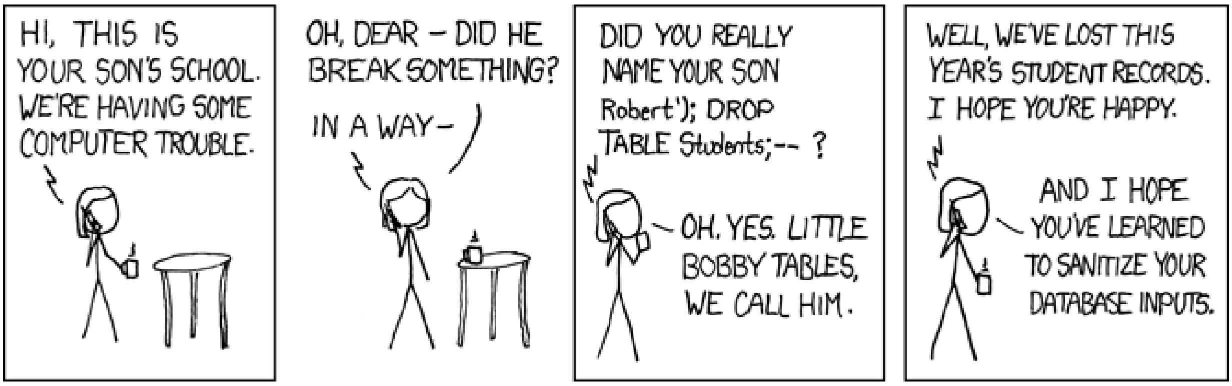
```
SELECT author,book FROM books WHERE book = ‘$title’;
```

- 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!

Sanitize or Be Sorry



Defenses For XSS, XSRF, SQL Injection

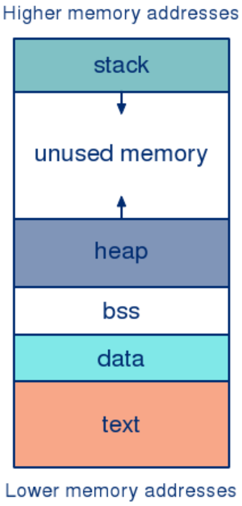
- ❑ Some options on **client-side** against XSS/XSRF:
  - JavaScript is often a must for many “good” Web pages
    → turning it off is not an option
    → better sandboxing? → very complex
  - Turning on some security settings can provide some security
    → unfortunately, these are often not activated 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

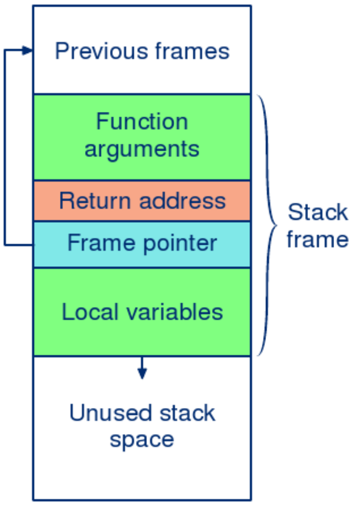
Buffer Overflows

- 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
  - Frames are pushed on the stack during function invocation, and popped back after returning
- Each frame comprises
  - functions arguments
  - return address
  - frame pointer: the address of the start of the previous frame
  - local variables
- Without proper bound 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
  - Essentially, you take over the control flow



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++)
- 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
  - 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++ etc. and/or do not trade (perceived) speed-ups for clean code

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

References

[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.

[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