Application Security (apsi)

Lecture at FHNW

Lecture 6, 2020

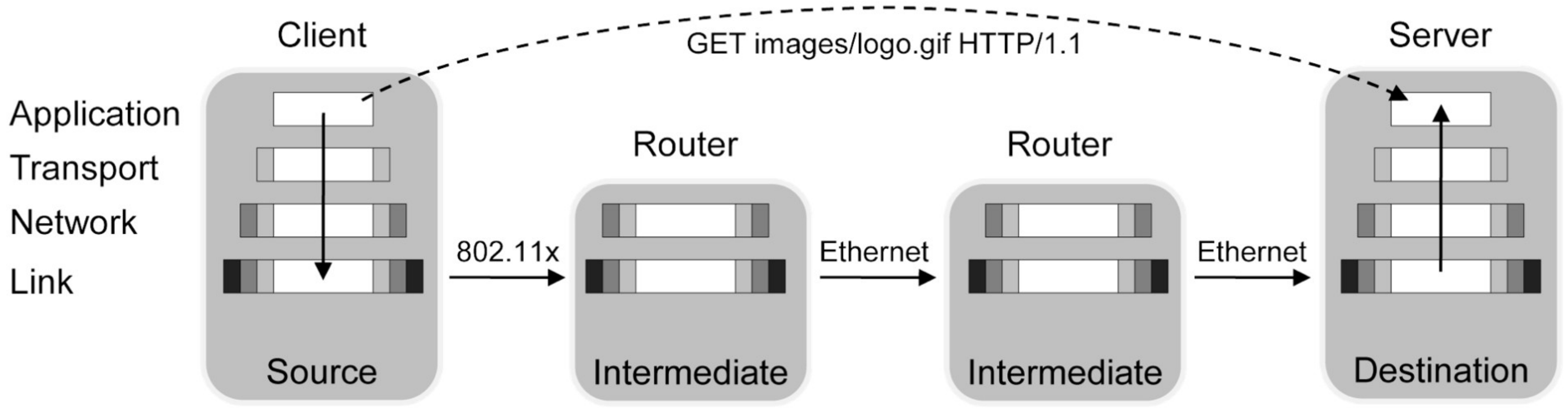
Arno Wagner, Michael Schläpfer, Rolf Wagner

<arno@wagner.name>, <{michael.schlaepfer, rolf.wagner}@fort-it.ch>

Agenda

- HTTP: Request: query, headers, body -- Response: headers, body
- HTTP: GET, POST and exotics (PUT, DELETE, ...)
- HTTP Authentication: Basic, Digest, NTLM
- Cookies: Setting, use with CGI, access on client via JavaScript
- Session management: Cookie, parameter, token
- SSL/TLS
- CGI interface

Example Request in the TCP/IP Model



HTTP

- The Hypertext Transfer Protocol
- First standardization: RFC 1945 (HTTP 1.0, 1996)
- Standardized today in RFC 7230, …, RFC 7235 (HTTP 1.1)
- Used for queries to and responses from web-servers via TCP

Properties:

- Plain-text protocol: In principle, telnet can be used as "web-browser"
- Standard payload for responses: HTML, many more supported
- Query consists of: 1. Query 2. Headers 3. Body (optional)
- Response consists of: 1. Headers 2. Body (optional)

HTTP Request Example: GET

Slightly shortened:

```
GET / HTTP/1.1
Host: www.arnowagner.info
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/59.0.3071.90 Safari/537.36 Vivaldi/1.91.867.38
Accept: text/html,application/xhtml+xml,image/webp,image/apng,*/*;q=0.8
DNT: 1
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8
```

Note: "`q=0.8`" is a "quality factor", i.e. a preference. Default is 1.0

HTTP Request Example: POST

Slightly shortened:

```
POST /files/htm-form-tutorial/html-form-tutorial-example-1.html HTTP/1.1
Host: javascript-coder.com
Connection: keep-alive
Content-Length: 45
Cache-Control: max-age=0
Origin: http://javascript-coder.com
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/59.0.3071.90 Safari/537.36 Vivaldi/1.91.867.38
Content-Type: application/x-www-form-urlencoded
Accept: text/html,application/xhtml+xml,image/webp,image/apng,*/*;q=0.8
DNT: 1
Referer: http://javascript-coder.com/files/html-form-tutorial-example-1.html
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8

Name=test&Email=test%40test.org&Submit=Submit
```

HTTP Response Example

```
HTTP/1.1 200 OK
Date: Sun, 12 Nov 2017 22:27:20 GMT
Server: Apache/2.4.10 (Debian)
Last-Modified: Sat, 19 Mar 2016 18:34:46 GMT
ETag: "116e-52e6b1e2f4ce0-gzip"
Accept-Ranges: bytes
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 2161
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>
<HEAD>
   <TITLE>Arno Wagner's Homepage</TITLE>
</HEAD>
<body text="#000000"  link="#0000EE" vlink="#551A8B" alink="#FF0000"
bgcolor="#FFFFFF">
<H1>Homepage of Dr. Arno Wagner </H1>
...
</BODY>
</HTML>
```

Exotic Requests: PUT, DELETE, ...

These are valid HTTP requests, but

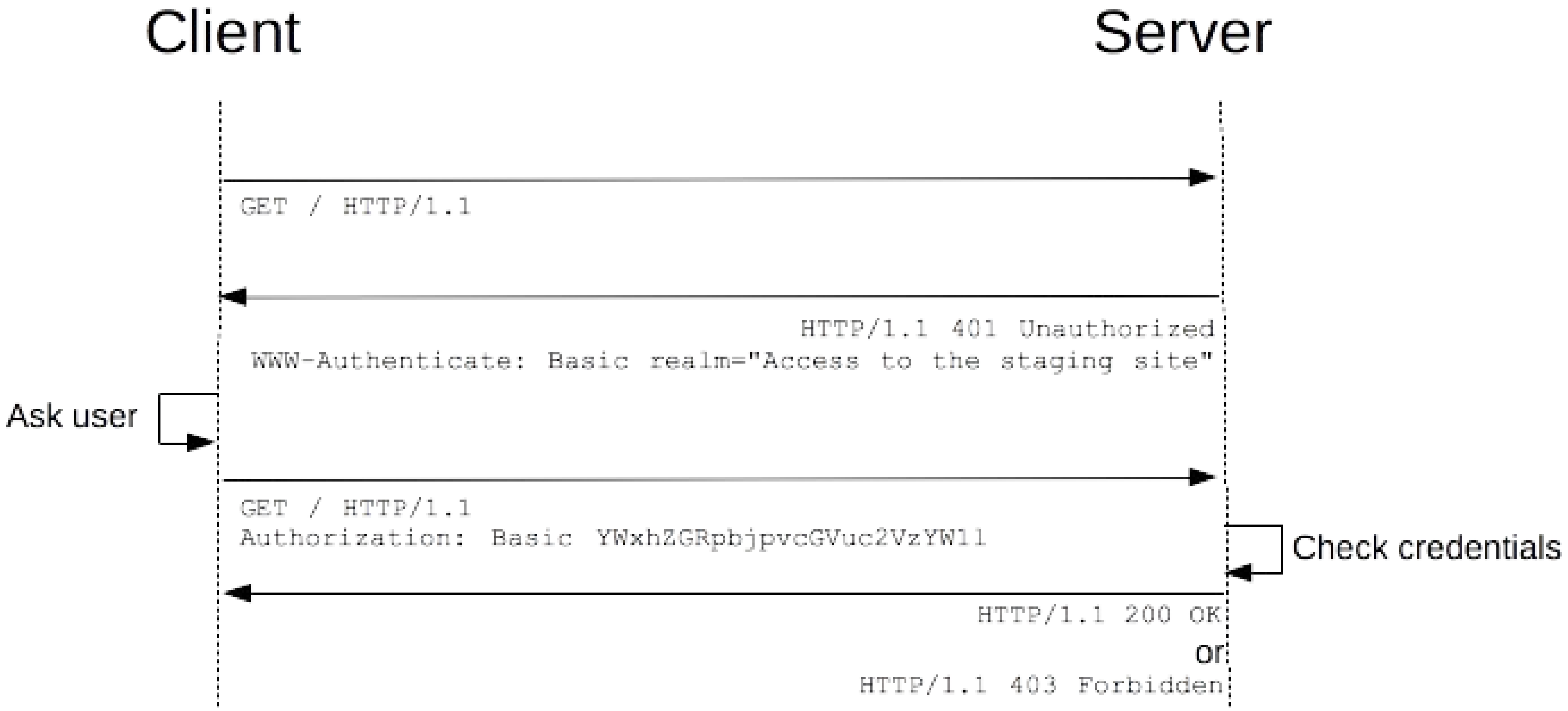
- Not widely used. Most things are done via GET and POST.
- May not work or not work right with some setups
- Should generally be avoided
  Exception: REST interfaces. These only look like a classic web interface...

Examples:

- PUT: Body included should be stored under the file given as path in the URL
- DELETE: File given as path in the URL should be deleted
- HEAD: Same as GET, but do not deliver the body
- OPTIONS: CORS preflight requests
- Some more exist

HTTP Authentication

- Basic Authentication



```
GET / HTTP/1.1
```

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="Access to the staging site"
```

```
GET / HTTP/1.1
Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l
```

```
HTTP/1.1 200 OK
```

```
HTTP/1.1 403 Forbidden
```

HTTP Authentication

Basic Authentication

- Part of the HTTP specification (RFC 7617)
- Simple authentication scheme, supported by all Web Servers and Browsers
- But: usernames and passwords stored in a file on the Server and,
- Even worse: credentials transmitted in cleartext (Base64 encoded)

Digest Access Authentication

- Server sends a fresh nonce (random value)
- User computes a hash value of uname, pw, nonce, url, etc.
- Usually MD5 used as hash function

Cookies

Cookies (in this context) are a mechanism to store data in the client

- Server includes one or more "`Set-Cookie`" headers
  General form: `Set-Cookie: <cookie-name>=<cookie-value>`
- Cookie value is ASCII without control characters, spaces, tabs and
  ( ) < > @ , ; : \ " / [ ] ? = { }   Use quotes for fewer restrictions.
- Options can be given
  Examples:
  ```
  Set-Cookie: <cookie-name>=<cookie-value>; Secure
  Set-Cookie: <cookie-name>=<cookie-value>; Expires=<date>; HttpOnly
  ```
- The client (browser) sends all cookies back on subsequent requests to the same site, subject to some limitations (Same Origin Policy)
  Note: All cookies get packed into one header field in the HTTP request:
  ```
  Cookie: name=value; name2=value2; name3=value3
  ```

Cookie Options

Browsers are not required to support cookies or specific options!

- Expires=<date>: Time the cookie gets deleted
  If absent: Cookie gets deleted on browser shutdown ("session cookie")
- Max-Age=<number>: Cookie lifetime in seconds
- Domain=<domain-value>: Domains the cookie get sent to
  If absent, the same-origin policy applies (explained later)
- Path=<path-value>: Cookie will only be sent with queries that have a path starting with the given path. Note: Not useful to secure against access!
- Secure:Cookie will only be sent if SSL and HTTPS is used
- HttpOnly:Prevent client-side JavaScript access to the cookie
- SameSite=Strict: Prevent cookies sent with cross-site requests.
  Offers some protection against CSRF attacks

Cookies: Access via JavaScript

- Cookies can be set in the browser via JavaScript:

  ```
  document.cookie = "yummy_cookie=choco";
  document.cookie = "tasty_cookie=strawberry";
  ```

  sets two cookies. Normal cookie options can be used

- Cookies can be queried in JavaScript. The variable

  ```
  document.cookie
  ```

  contains the cookies in the format of the "Cookie" HTTP header

Note: This access is limited via the "Same Origin Policy" → Next lecture
Note: If done wrong, cookies can be stolen via XSS attacks in this way

Session Management

Purpose: Maintain user log-in

- Usually done by giving the client a token that indicates the user is logged in

This "token" usually contains:

- A session ID, which must be hard to guess
- The user ID
- Optional additional parameters, such as authorization level

Mechanisms used:

- GET or POST parameter, sent to server again, e.g., via hidden form field
- Cookie or other HTTP header (Bearer)

Security Problems with Session Management

Note: this is just a partial list

Session ID:

- May be guessed
- May get stolen by other sites accessed with the same browser-session
- May be sniffed on the network

=> Usually full session compromise ("Session Hijacking")

Other parameters:

- User ID, authorization level, etc. : May be manipulated on browser-side
  Example: User logged in as normal user, sets authorization to admin
  Fix: Must be cryptographically protected or stored only on server

Keeping State Securely on Client-Side

In general, the client (browser+user) <u>must not</u> have access to the full application state. This can include application data.

Example: The state can contain the user-name and privilege level

Example: Technical identifiers may need to stay hidden form users

Options (proper practices for secure encryption must be followed):

- Use encrypted parameters
- Use encrypted cookies

Alternative:

- Keep state on the server and give a non-guessable (!) session-ID to the client
- Feasible for cloud services (stateless architecture, autoscaling)?

Remark: Cookies and Application Paths vs. Proxies

- Cookies get overwritten if a new cookie with the same name and host is set (assuming no path setting)

  Effect: If two applications are placed behind a proxy use the same cookie names, they will interfere with each others cookies

  Fix: Application specific names!   No "JSESSIONID" or the like.

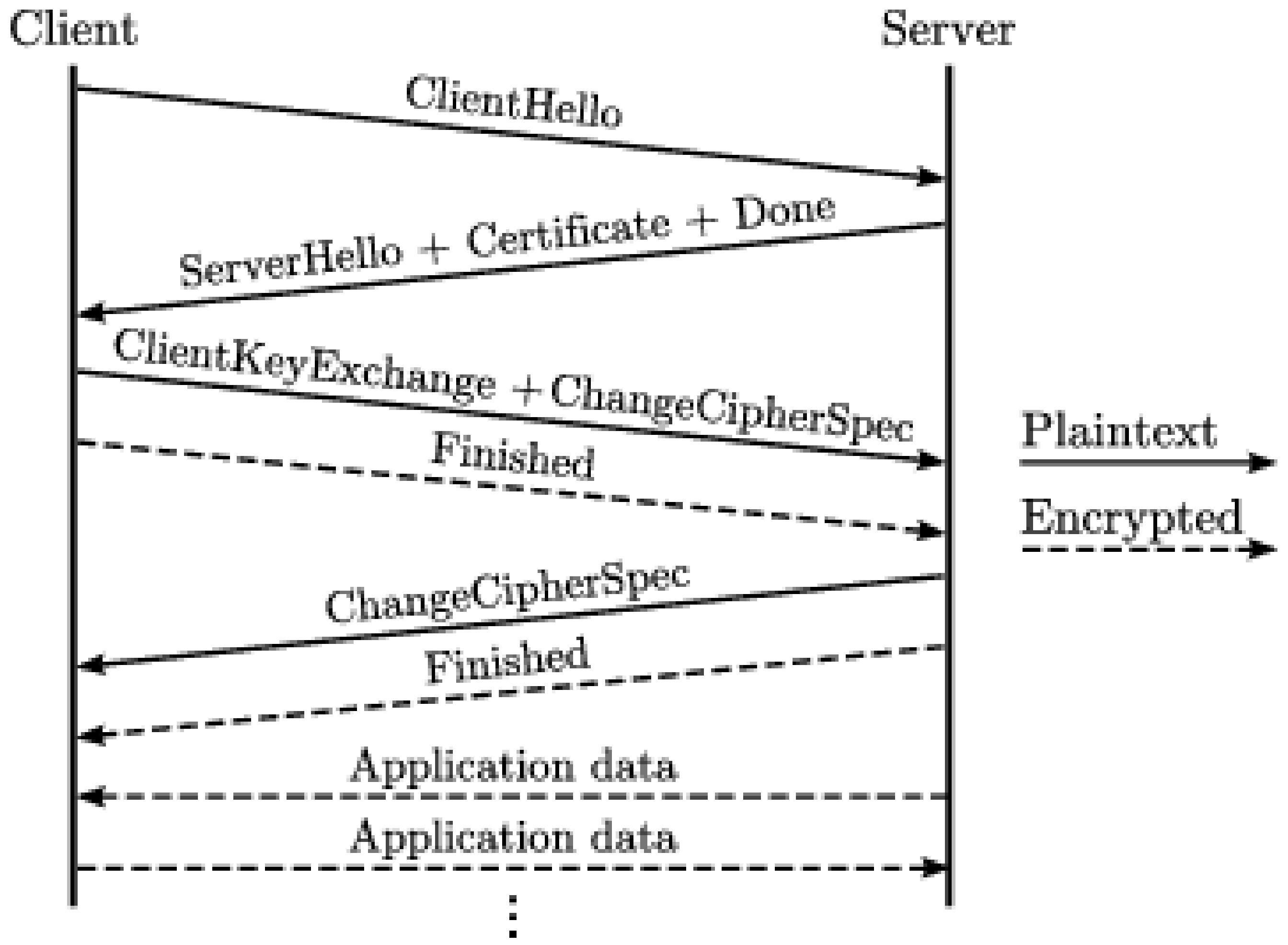
- Proxies decide the target to send a request to by looking at the path

  Problem: Paths like "/", "/app/", "/images/", "/css/" are not useful to distinguish between two (or more) applications.

  Fix: Use a first path component that identifies the application uniquely.

Especially in an enterprise scenario, you must always expect that your application needs to be capable to run over a proxy together with other applications. This can happen at a later time.

SSL/TLS


Client
Server
ClientHello
ServerHello + Certificate + Done
ClientKeyExchange + ChangeCipherSpec
Finished
ChangeCipherSpec
Finished
Application data
Application data
Plaintext
Encrypted

SSL/TLS

- Protocols providing authentication and encryption services
- HTTP with SSL/TLS = HTTPS
- Can be configured to be more or less secure (e.g., Cipher Suites)
- See BSI Technische Richtlinie TR-02102-2 for recommendations:
  - Use only TLS 1.2 and TLS 1.3 (are your users ready? Browsers?)
  - Use recommended Cipher Suites only (again, are your users ready?)
  - Many more configuration details
- Other use cases with SSL/TLS:
  - Mail Transfer (POP3S, SMTPS, IMAPS)
  - File Transfer (FTPS)
  - XMPPS
  - OpenVPN

CGI: General Idea

Request:

- Web-server calls executable program (self-contained)
- Headers go into environment variables
- Meta-Headers (query, client, client IP, etc.) also in environment variables
  Example: QUERY_STRING gives GET parameters
- POST-Body (from web forms using POST) can be read from STDIN
  Format: Same as GET parameters
  Example: name=wagner&role=lecturer&tasks=entertain%20students

Response:

- Result is written verbatim to STDOUT

1) Headers first (must be at least one): CONTENT_TYPE, STATUS, etc.
2) One empty line
3) Then body

CGI Scripts in Different Languages

Configuration used:

```
ScriptAlias /cgi-bin/ "/usr/local/apache2/cgi-bin/"
User daemon
Group daemon
```

This means:

- CGI scripts are placed into /cgi-bin/ in the Apache ServerRoot
- Scripts need to be executable for "daemon"
  => Potentially insecure, but easy to do: chmod 0755 <file>

Quick howto: https://httpd.apache.org/docs/2.4/howto/cgi.html

Note: Reading STDIN not demonstrated, works as usual.

Note: To report an error, use the "Status" header
Example: Print "`Status: 500 Internal Server Error`"

CGI in BASH

The most simple version, does almost nothing, text/plain output:

```
#!/bin/bash
set -f
echo "Content-type: text/plain; charset=iso-8859-1"
echo
echo CGI/1.0 test script report:
echo
echo "Script runs as"
echo `/usr/bin/id`
echo argc is $#. argv is "$*".
echo
echo SERVER_SOFTWARE = $SERVER_SOFTWARE
echo SERVER_NAME = $SERVER_NAME
echo GATEWAY_INTERFACE = $GATEWAY_INTERFACE
echo SERVER_PROTOCOL = $SERVER_PROTOCOL
echo SERVER_PORT = $SERVER_PORT
echo REQUEST_METHOD = $REQUEST_METHOD
echo HTTP_ACCEPT = "$HTTP_ACCEPT"
echo PATH_INFO = "$PATH_INFO"
echo PATH_TRANSLATED = "$PATH_TRANSLATED"
echo SCRIPT_NAME = "$SCRIPT_NAME"
echo QUERY_STRING = "$QUERY_STRING"
echo REMOTE_HOST = $REMOTE_HOST
echo REMOTE_ADDR = $REMOTE_ADDR
echo REMOTE_USER = $REMOTE_USER
echo AUTH_TYPE = $AUTH_TYPE
echo CONTENT_TYPE = $CONTENT_TYPE
echo CONTENT_LENGTH = $CONTENT_LENGTH
```

CGI in BASH: Output

```
CGI/1.0 test script report:

Script runs as
uid=1(daemon) gid=1(daemon) groups=1(daemon)
argc is 0. argv is .

SERVER_SOFTWARE = Apache/2.4.6 (Unix) OpenSSL/1.0.1t
SERVER_NAME = g
GATEWAY_INTERFACE = CGI/1.1
SERVER_PROTOCOL = HTTP/1.0
SERVER_PORT = 8080
REQUEST_METHOD = GET
HTTP_ACCEPT = text/html, text/plain, text/sgml, text/css,
application/xhtml+xml,
 */*;q=0.01
PATH_INFO =
PATH_TRANSLATED =
SCRIPT_NAME = /cgi-bin/test-cgi
QUERY_STRING =
REMOTE_HOST =
REMOTE_ADDR = 192.168.3.10
REMOTE_USER =
AUTH_TYPE =
CONTENT_TYPE =
CONTENT_LENGTH =
```

CGI in Perl

Minimal Perl cgi-script, writes HTML:

```
#!/usr/bin/perl
use strict;
use warnings;

print "Content-type: text/html\n\n";

print "<head>\n</head>\n<body>\n";
foreach my $key (keys %ENV) {
    print "$key --> $ENV{$key}<br>\n";
}
print "</body>\n";
```

CGI in Perl: Output

```
HTTP_ACCEPT_ENCODING --> gzip, compress, bzip2
REMOTE_PORT --> 48686
LD_LIBRARY_PATH --> /usr/local/apache2/lib
CONTEXT_PREFIX --> /cgi-bin/
HTTP_ACCEPT_LANGUAGE --> en
GATEWAY_INTERFACE --> CGI/1.1
SERVER_PORT --> 8080
QUERY_STRING -->
REQUEST_URI --> /cgi-bin/printenv2
SCRIPT_FILENAME --> /usr/local/apache2/cgi-bin/printenv2
SERVER_SIGNATURE -->
SERVER_SOFTWARE --> Apache/2.4.6 (Unix) OpenSSL/1.0.1t
HTTP_USER_AGENT --> Lynx/2.8.9dev.1 libwww-FM/2.14 SSL-MM/1.4.1 GNUTLS/3.3.8
HTTP_ACCEPT --> text/html, text/plain, text/sgml, text/css, application/xhtml+xml, */*;q=0.01
DOCUMENT_ROOT --> /usr/local/apache2/htdocs
REQUEST_METHOD --> GET
SCRIPT_NAME --> /cgi-bin/printenv2
SERVER_ADDR --> 192.168.3.10
HTTP_HOST --> g:8080
SERVER_NAME --> g
REMOTE_ADDR --> 192.168.3.10
CONTEXT_DOCUMENT_ROOT --> /usr/local/apache2/cgi-bin/
SERVER_ADMIN --> you@example.com
REQUEST_SCHEME --> http
SERVER_PROTOCOL --> HTTP/1.0
PATH --> /usr/local/apache2/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

CGI in C

Prints environment, output is text/plain:

```
#include <stdio.h>

int main(int argc, char *argv[], char *envp[]) {
  int i;
  char * line;

  printf("Content-type: text/plain; charset=iso-8859-1\n\n");
  for (i = 0; envp[i] != NULL; i++)  {
    line = envp[i];
    printf("%s\n", line);
  }
  return(0);
}
```

CGI in C: Output

```
HTTP_HOST=g:8080
HTTP_ACCEPT=text/html, text/plain, text/sgml, text/css, application/xhtml+xml, */*;q=0.01
HTTP_ACCEPT_ENCODING=gzip, compress, bzip2
HTTP_ACCEPT_LANGUAGE=en
HTTP_USER_AGENT=Lynx/2.8.9dev.1 libwww-FM/2.14 SSL-MM/1.4.1 GNUTLS/3.3.8
PATH=/usr/local/apache2/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
LD_LIBRARY_PATH=/usr/local/apache2/lib
SERVER_SIGNATURE=
SERVER_SOFTWARE=Apache/2.4.6 (Unix) OpenSSL/1.0.1t
SERVER_NAME=g
SERVER_ADDR=192.168.3.10
SERVER_PORT=8080
REMOTE_ADDR=192.168.3.10
DOCUMENT_ROOT=/usr/local/apache2/htdocs
REQUEST_SCHEME=http
CONTEXT_PREFIX=/cgi-bin/
CONTEXT_DOCUMENT_ROOT=/usr/local/apache2/cgi-bin/
SERVER_ADMIN=you@example.com
SCRIPT_FILENAME=/usr/local/apache2/cgi-bin/printenv_c
REMOTE_PORT=48690
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.0
REQUEST_METHOD=GET
QUERY_STRING=
REQUEST_URI=/cgi-bin/printenv_c
SCRIPT_NAME=/cgi-bin/printenv_c
```

CGI in Java

Java has a problem: It has no self-contained executables

What to do? → Use a call-wrapper!

- Wrapper, place in cgi-bin and make executable (see CGI in Bash)
  This gets called as CGI-Script. It then executes the Java code:

```
#!/bin/bash
/usr/bin/java Env_cgi
```

- Java Code, place Env_cgi.class with wrapper and make readable

```
class Env_cgi {
  public static void main( String ... args ) {
    System.out.println("Content-type: text/plain; charset=iso-8859-1\n\n");
    for( Object o : System.getenv().entrySet()  ){
      System.out.println( o );
    }
  }
}
```

Note: No attempt at good Java style was made...

CGI Security

A CGI-script is a gigantic backdoor into your server!

(The same is true for any web-application...)

=> Effective security is critical!

- Must avoid dangerous calls with client-supplied parameters
  → see, for example, "taint checking", Lecture 1
- Must avoid any execution of client-supplied code
  → This includes buffer-overflows and injection attacks for SQL, XML, etc.
- Must configure permissions in server correctly
- Should execute as limited as possible