Our current view of security

• until now, we have assumed:
  • valid user input
  • non-malicious users
  • nothing will ever go wrong
• this is unrealistic!
The real world

• in order to write secure code, we must assume:
  • invalid input
  • evil users
  • incompetent users
  • everything that can go wrong, will go wrong
  • everybody is out to get you
  • botnets, hackers, script kiddies, KGB, etc. are out there

• the security mindset: assume nothing; trust no one
Attackers' goals

- Why would an attacker target my site?
- **Read private data** (user names, passwords, credit card numbers, grades, prices)
- **Change data** (change a student's grades, prices of products, passwords)
- **Spoofing** (pretending to be someone they are not)
- **Damage or shut down the site**, so that it cannot be successfully used by others
- **Harm the reputation or credibility** of the organization running the site
- **Spread viruses** and other malware
Tools that attackers use

Assume that the attacker knows about web dev and has the same tools you have:

• **Firebug**
• extensions e.g. [Web Dev Toolbar](#)
• **port scanners**, e.g. [nmap](#)
• network sniffers, e.g. [Wireshark](#), [EtherDetect](#), [Firesheep](#)
Some kinds of attacks

- **Denial of Service (DoS)**: Making a server unavailable by bombarding it with requests.
- **Social Engineering**: Tricking a user into willingly compromising the security of a site (e.g. phishing).
- **Privilege Escalation**: Causing code to run as a "privileged" context (e.g. "root").
- **Information Leakage**: Allowing an attacker to look at data, files, etc. that he/she should not be allowed to see.
- **Man-in-the-Middle**: Placing a malicious machine in the network and using it to intercept traffic.
- **Session Hijacking**: Stealing another user's session cookie to masquerade as that user.
- **Cross-Site Scripting (XSS)** or HTML Injection: Inserting malicious HTML or JavaScript content into a web page.
- **SQL Injection**: Inserting malicious SQL query code to reveal or modify sensitive data.
Information leakage

when the attacker can look at data, files, etc. that he/she should not be allowed to see

- files on web server that should not be there
  - or have too generous of permissions (read/write to all)
- directories that list their contents (indexing)
  - can be disabled on web server
- guess the names of files, directories, resources
  - see loginfail.php, try loginsuccess.php
  - see user.php?id=123, try user.php?id=456
  - see /data/public, try /data/private
Man-in-the-middle attack

when the attacker listens on your network and reads and/or modifies your data

• works if attacker can access and compromise any server/router between you and your server
• also works if you are on the same local area network as the attacker
• often, the attacker still sends your info back and forth to/from the real server, but he silently logs or modifies some of it along the way to his own benefit
• e.g. listens for you to send your user name / password / credit card number / ...
Secure HTTP (HTTPS)

- **HTTPS**: encrypted version of HTTP protocol
- all messages between client and server are encrypted so men in the middle cannot easily read them
- servers can have certificates that verify their identity
Session hijacking

when the attacker gets a hold of your session ID and masquerades as you

• exploit sites that use HTTPS for only the initial login:
  • HTTPS: browser → server (POST login.php)
  • HTTPS: browser ← server (login.php + PHPSESSID cookie)
  • HTTP: browser → server (GET whatever.php + PHPSESSID cookie)
  • HTTP: browser ← server (whatever.php + PHPSESSID cookie)

• attacker can listen to the network, get your session ID cookie, and make requests to the same server with that same session ID cookie to masquerade as you!
• example: Firesheep
HTML injection

a flaw where a user is able to inject arbitrary HTML content into your page

• This flaw often exists when a page accepts user input and inserts it bare into the page.

• example: magic 8-ball ([8ball.html](http://example.com/8ball.html))

• What kinds of silly or malicious content can we inject into the page? Why is this bad?
Injecting HTML content

8ball.php?question=<em>lololol</em>

• injected content can lead to:
  • annoyance / confusion
  • damage to data on the server
  • exposure of private data on the server
  • financial gain/loss
  • end of the human race as we know it
• why is HTML injection bad? It allows others to:
  • disrupt the flow/layout of your site
  • put words into your mouth
  • possibly run malicious code on your users' computers
Cross-site scripting (XSS)

a flaw where a user is able to inject and execute arbitrary JavaScript code in your page

8ball.php?question=<script type='text/javascript'>alert('pwned');</script>

- JavaScript is often able to be injected because of a previous HTML injection
- Try submitting this as the 8-ball's question in Firefox:
  - `/* <![CDATA[ */ (function (cfg) {
    BrowserPonies.setBaseUrl(cfg.baseurl);
    BrowserPonies.loadConfig(BrowserPoniesBaseConfig);
    BrowserPonies.loadConfig(cfg);})({
      "baseurl":"http://panzi.github.com/Browser-Ponies/",
      "fadeDuration":500,
      "volume":1,
      "fps":25,
      "speed":3,
      "audioEnabled":false,
      "showFps":false,
      "showLoadProgress":true,
      "speakProbability":0.1,
      "spawn":{
        "applejack":1,
        "fluttershy":1,
        "pinkie pie":1,
        "rainbow dash":1,
        "rarity":1,
        "twilight sparkle":1,
        "autostart":true}); /* ]]> */ }}</script>`

- injected script code can:
  - masquerade as the original page and trick the user into entering sensitive data
  - steal the user's cookies
  - masquerade as the user and submit data on their behalf (submit forms, click buttons, etc.)

- ...
Securing against HTML injection / XSS

- one idea: disallow harmful characters
  - HTML injection is impossible without `< >`
  - can strip those characters from input, or reject the entire request if they are present
- another idea: allow them, but escape them

```php
$htmlspecialchars | returns an HTML-escaped version of a string

$text = "<p>hi 2 u & me</p>";
$text = htmlspecialchars($text);  # "&lt;p&gt;hi 2 u &amp; me&lt;/p&gt;"
```
Another XSS example

• example: Lab 4, Buy-a-Grade ([buyagrade.html](#))

• Recall that the user submits his name, section, and credit card number to the server, which are then displayed on the page.

• How can we inject HTML/JavaScript into the page? Be creative...

• What could we do to steal the user's sensitive information?
SQL injection

*a flaw where the user is able to inject arbitrary SQL into your query*

- This flaw often exists when a page accepts user input and inserts it bare into the query.
- example: simpsons grade lookup ([start.php](#))
- What kinds of SQL can we inject into the query? Why is this bad?
A SQL injection attack

• The query in the Simpsons PHP code is:

```sql
$query = "SELECT * FROM students
WHERE username = '$username' AND password = '$password'";
```

• Are there malicious values for the user name and password that we could enter?

• Password: `" OR '1'='1`

• This causes the query to be executed as:

```sql
$query = "SELECT * FROM students
WHERE username = '$username' AND password = "' OR '1'='1";
```

• What will the above query return? Why is this bad?
Too true...

- injected SQL can:
  - change the query to output others' data (revealing private information)
  - insert a query to modify existing data (increase bank account balance)
  - delete existing data (`; DROP TABLE students; --`)
  - bloat the query to slow down the server (`JOIN a JOIN b JOIN c ...`)
  - ...
Securing against SQL injection

• similar to securing against HTML injection, escape the string before you include it in your query

```
quote returns a SQL-escaped version of a string
```

```
$username = $db->quote($_POST["username"]);
$password = $db->quote($_POST["password"]);
$query = "SELECT name, ssn, dob FROM users
WHERE username = $username AND password = $password";
```

• replaces ' with \', etc., and surrounds with quotes