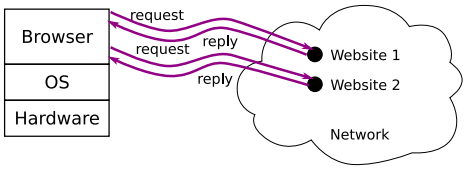
CSci 4271W
Development of Secure Software Systems
Day 22: Web security part 3: spoofing, tampering, and information disclosure

Stephen McCamant (he/him)
University of Minnesota, Computer Science & Engineering

Based in large part on slides originally by Prof. Nick Hopper
Licensed under Creative Commons Attribution-ShareAlike 4.0

Web applications



- The browser sends a request to a server
- The server processes this request, and sends a reply
- The browser receives data and code
- This may result in the need to send additional requests

Hypertext Transport Protocol

- HTTP is a stateless request/response protocol
- Clients send resource requests (usually GET, POST or PUT)
- Servers send responses (info/success/redirect/error)
- Response bodies can reference additional resources
- Most applications build stateful sessions on top of HTTP

Embedded content

HTML documents can reference many other resources:

- Style sheets – influence display of elements
- Scripts – `<script src="nextslide.js" />`
- Frames – include other pages
- Images – loaded and displayed with separate requests

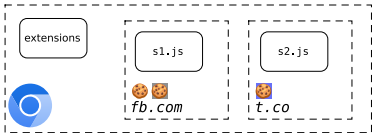
Scripts

JavaScript embedded in a page runs in a sandbox but can:

- Manipulate page's Document Object Model (DOM), adding or removing elements
- Make additional HTTP requests
- Open windows, capture user input
- Access page's local storage
- Interact with browser API

Security goals

Like OSes, browsers provide uniform resource access and attempt to protect applications from each other.
The unit of protection is the "origin" (informally, "domain")



Data associated with a page originating from domain A should not be leaked to or altered by a page originating from domain B. (The "same-origin policy".)

Securing web sites

Securing web sites can be challenging for many reasons:

- Sessions store (some) state at the client
- Inputs come from unknown, untrusted sources
- The statelessness of HTTP potentially allows replays, modification, and injection of requests
- Mutually untrusted applications may use the same physical server

Outline

Spoofing: server to users

How does the user know they've reached `www.bank.com`?

Spoofing: server to users

How does the user know they've reached `www.bank.com`?

TLS! Certificates, UI like lock icons, ...

Spoofing: server to users

How does the user know they've reached `www.bank.com`?

TLS! Certificates, UI like lock icons, ...

Potential problems: mal-in-the-middle/`sslstrip`, stored or reflected XSS, third-party embedded content

Spoofing: users to server

Many web sites store login cookies in browsers. Here are some mistakes:

Spoofing: users to server

Many web sites store login cookies in browsers. Here are some mistakes:

- isn't protected against forgery, e.g.
  `Set-cookie: user=admin&secret=456789`

Spoofing: users to server

Many web sites store login cookies in browsers. Here are some mistakes:

- isn't protected against forgery, e.g.
  `Set-cookie: user=admin&secret=456789`
- is protected with "roll-your-own" crypto:
  `Set-cookie: user=admin&tag=<username^0x5ec4e7>`

Spoofing: users to server

Many web sites store login cookies in browsers. Here are some mistakes:

- isn't protected against forgery, e.g.
  `Set-cookie: user=admin&secret=456789`
- is protected with "roll-your-own" crypto:
  `Set-cookie: user=admin&tag=<username^0x5ec4e7>`
- doesn't expire or is not tied to the current session:
  `Set-cookie: user=me&crypto=0xdeadbeef`

Spoofing: users to server

Many web sites store login cookies in browsers. Here are some mistakes:

- isn't protected against forgery, e.g.
  `Set-cookie: user=admin&secret=456789`
- is protected with "roll-your-own" crypto:
  `Set-cookie: user=admin&tag=<username^0x5ec4e7>`
- doesn't expire or is not tied to the current session:
  `Set-cookie: user=me&crypto=0xdeadbeef`
- contains only a counter-based session identifier:
  `Set-cookie: sessionid=12345`

More spoofing risks: passwords

- Password entry: unencrypted, digest auth., mixed-TLS

More spoofing risks: passwords

- Password entry: unencrypted, digest auth., mixed-TLS
- Password storage: cleartext, unsalted, encrypted vs. hashed

More spoofing risks: passwords

- Password entry: unencrypted, digest auth., mixed-TLS
- Password storage: cleartext, unsalted, encrypted vs. hashed
- Password recovery: "personal questions" are not. Reset asks for email, or sends old password...

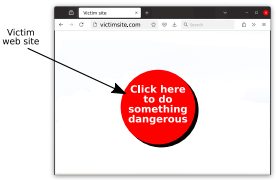
More spoofing risks: passwords

- Password entry: unencrypted, digest auth., mixed-TLS
- Password storage: cleartext, unsalted, encrypted vs. hashed
- Password recovery: "personal questions" are not. Reset asks for email, or sends old password...
- Password rate limits: attackers can try many leaked pairs of username/password

Spoofing mitigation

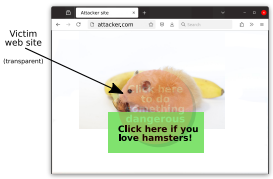
- Protect all site pages with TLS, no external includes on login; Use HSTS to pin TLS usage, and SCT for cert transparency
- Use secure session manager software (long random session ids, automatic expiration, weak password checks)
- All 🍪 should have issue or expire date, and secure crypto authentication (HMAC, not hash) applied to full cookie.
- No clear-text passwords. Password recovery via expiring, crypto-protected email link.

Clickjacking



Attacker page includes victim page in frame (e.g. with style "`opacity:0`") to receive attracted user clicks.

Clickjacking



Attacker page includes victim page in frame (e.g. with style "`opacity:0`") to receive attracted user clicks.

Clickjacking defenses

Problematic: `Referer` headers

Also problematic: "frame busting"

```
<script>
if (self.location != top.location) {
    top.location = self.location;
}
</script>
```

Best: `Content-Security-Policy: frame-ancestors 'self';`

Outline

Review: web and security model

Spoofing attacks

Announcements intermission

Tampering and information disclosure

Project 2 status

- Grades and a modest amount of Project 1 feedback were posted just before class
  - For fairness, nothing more will be posted until after Monday's one-time-extended deadline
- Given the limits on the amount of feedback and the time to work with it, our expectations about using feedback in project 2 will be proportionately reduced
  - But still non-zero
- You will have another opportunity to incorporate feedback in project 3 (coming soon)

Outline

Review: web and security model

Spoofing attacks

Announcements intermission

Tampering and information disclosure

Parameter tampering

HTTP GET/POST params can be modified by attackers:

Parameter tampering

HTTP GET/POST params can be modified by attackers:

- Never rely on client-side validation

Parameter tampering

HTTP GET/POST params can be modified by attackers:

- Never rely on client-side validation
  - "Drop downs" can be filled arbitrarily
  - Numbers can be negative, out of range, etc.
  - Pre-filled values can be altered

Parameter tampering

HTTP GET/POST params can be modified by attackers:

- Never rely on client-side validation
  - "Drop downs" can be filled arbitrarily
  - Numbers can be negative, out of range, etc.
  - Pre-filled values can be altered
- Hidden fields are not hidden from attackers who know to look for them

Parameter tampering

HTTP GET/POST params can be modified by attackers:

- Never rely on client-side validation
  - "Drop downs" can be filled arbitrarily
  - Numbers can be negative, out of range, etc.
  - Pre-filled values can be altered
- Hidden fields are not hidden from attackers who know to look for them
- Watch for repeat names in requests:

```
http://v.com/s?user=ape&pin=12345&user=admin
```

Parameter pollution

HTTP GET/POST parameters can be initialized by attackers and reflected by server-side scripts

Parameter pollution

HTTP GET/POST parameters can be initialized by attackers and reflected by server-side scripts

```
<a href="http://y.co/show?Fid=0&Mid=0%26cmd=delete">
Check Mail</a>
```

Parameter pollution

HTTP GET/POST parameters can be initialized by attackers and reflected by server-side scripts

```
<a href="http://y.co/show?Fid=0&Mid=0%26cmd=delete">
Check Mail</a>
```

⇓

```
<a href="/show?Mid=0&cmd=delete&csrf=x">Subject 1</a>...
<a href="/show?Mid=1&cmd=delete&csrf=x">Subject 2</a>...
```

Parameter pollution

HTTP GET/POST parameters can be initialized by attackers and reflected by server-side scripts

```
<a href="http://y.co/show?Fid=0&Mid=0%26cmd=delete">
Check Mail</a>
```

⇓

```
<a href="/show?Mid=0&cmd=delete&csrf=x">Subject 1</a>...
<a href="/show?Mid=1&cmd=delete&csrf=x">Subject 2</a>...
```

All GET/POST parameters must be validated

- before and after conversions
- against known-good patterns
- And if possible, not passed to foreign parsers

Cookie tampering

Cookies can also be modified/set/unset by attackers:

- `Set-Cookie: articlesLeft=4&0xcrypto`
- `Set-Cookie: email="user@example.com"`
- `Set-Cookie: NoPremium=true&0xcrypto`

Store only data that can be validated or referenced on server

Information disclosure

Very common bug: system generates outputs (bills, test results, scores, teleconferences, price quotes,...) sequentially, with links:

```
From: <billing-no-reply@v.com>
Your bill is now viewable at:
https://billing.v.com/view/123456.pdf
```

Information disclosure

Very common bug: system generates outputs (bills, test results, scores, teleconferences, price quotes,...) sequentially, with links:

```
From: <billing-no-reply@v.com>
Your bill is now viewable at:
https://billing.v.com/view/123456.pdf
```

Also common: access control misconfigured, old pages left in production, debug statements left on, hidden fields

Information disclosure

Very common bug: system generates outputs (bills, test results, scores, teleconferences, price quotes,...) sequentially, with links:

```
From: <billing-no-reply@v.com>
Your bill is now viewable at:
https://billing.v.com/view/123456.pdf
```

Also common: access control misconfigured, old pages left in production, debug statements left on, hidden fields

Always require authentication for private data.

Directory traversal

Somewhat common pattern:

```
<img src="https://v.io/loadImg?file=314.png">
```

On server side, calls `open("314.png")`

Directory traversal

Somewhat common pattern:

```
<img src="https://v.io/loadImg?file=314.png">
```

On server side, calls `open("314.png")`

Possible attacks:

- `"...?file=../../../etc/passwd"`
- `"...?file=/web/images/../../../etc/passwd"`
- `"...?file=..%c0%af../../etc/passwd"`
- `"...?file=../../../etc/passwd%00.png"`