Web Security

Adam C. Champion and Dong Xuan
CSE 4471: Information Security
Outline

• Web Basics
• Web Threats and Attacks
• Countermeasures
Introduction

• Average user spends 16 h/month online (32 h/month in U.S.) [1]
  – People spend much time interacting with Web, Web applications (apps)
  – Their (lack of) security has major impact

• Interaction via Web browser

• We’ll first review some Web basics

Source: [2], [3]
The Web

- **Web page:**
  - Consists of “objects”
  - Addressed by a URL

- **Most Web pages consist of:**
  - Base HTML page, and
  - Several referenced objects.

- **URL has two components:** host name and path name

- **User agent for Web is called a browser:**
  - MS Internet Explorer
  - Netscape Communicator

- **Server for Web is called Web server:**
  - Apache (public domain)
  - MS Internet Information Server
The Web: the HTTP Protocol

HTTP: HyperText Transfer Protocol

- Web’s application layer protocol
- Client/server model
  - **Client**: browser that requests, receives, “displays” Web objects
  - **Server**: Web server sends objects in response to requests
- HTTP 1.0: RFC 1945
- HTTP 1.1: RFC 2068
The HTTP Protocol (more)

HTTP: TCP transport service:
• Client initiates TCP connection (creates socket) to server, port 80
• Server accepts TCP connection from client
• HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
• TCP connection closed

HTTP is “stateless”
• Server maintains no information about past client requests

Aside

Protocols that maintain “state” are complex!
☐ Past history (state) must be maintained
☐ If server/client crashes, their views of “state” may be inconsistent, must be reconciled
HTTP Example

Suppose user enters URL http://www.someschool.edu/aDepartment/index.html

1a. HTTP client initiates TCP connection to http server (process) at www.someschool.edu. Port 80 is default for HTTP server.

1b. HTTP server at host www.someschool.edu waiting for TCP connection at port 80. “Accepts” connection, notifies client.

2. HTTP client sends http request message (containing URL) into TCP connection socket.

3. HTTP server receives request message, forms response message containing requested object (aDepartment/index.html), sends message into socket.

(time contains text, references to 10 JPEG images)
HTTP Example (Cont.)

5. HTTP client receives response message containing HTML file, displays HTML. Parsing HTML file, finds 10 referenced JPEG objects

4. HTTP server closes TCP connection.

6. Steps 1-5 repeated for each of 10 JPEG objects
## Non-Persistent and Persistent Connections

<table>
<thead>
<tr>
<th>Non-persistent</th>
<th>Persistent</th>
</tr>
</thead>
<tbody>
<tr>
<td>• HTTP/1.0</td>
<td>• Default for HTTP/1.1</td>
</tr>
<tr>
<td>• Server parses request, responds, and closes TCP connection</td>
<td>• On same TCP connection: server, parses request, responds, parses new request, …</td>
</tr>
<tr>
<td>• 2 RTTs to fetch each object</td>
<td>• Client sends requests for all referenced objects as soon as it receives base HTML.</td>
</tr>
<tr>
<td>• Each object transfer suffers from slow start</td>
<td>• Fewer RTTs and less slow start.</td>
</tr>
</tbody>
</table>

But most browsers use parallel TCP connections.
HTTP Message Format: Request

- Two types of HTTP messages: request, response
- HTTP request message:
  - ASCII (human-readable format)

```
GET /somedir/page.html HTTP/1.0
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
Accept-language: fr
```
(extra carriage return, line feed)

Carriage return, line feed indicates end of message
HTTP Request Message: General Format

```
+----------------+----------------+----------------+----------------+
| method          | sp              | URL            | sp              |
|                 |                 |                |                 |
| header field name | :               | value          | cr              |
|                 |                 |                | if              |
|                 |                 |                |                 |
| entity body     |                 |                |                 |
+----------------+----------------+----------------+----------------+
```

request line

header lines

Entity Body
HTTP Message Format: Response

status line
(protocol
status code
status phrase)

HTTP/1.0 200 OK
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 ......
Content-Length: 6821
Content-Type: text/html

data data data data data data data...

data, e.g., requested html file
HTTP Response Status Codes

In first line in server→client response message.
A few sample codes:

200 OK
   – request succeeded, requested object later in this message

301 Moved Permanently
   – requested object moved, new location specified later in this message
      (Location:)

400 Bad Request
   – request message not understood by server

404 Not Found
   – requested document not found on this server

505 HTTP Version Not Supported
Try HTTP (Client Side) for Yourself

1. Telnet to your favorite Web server:

   telnet www.cse.ohio-state.edu/ 80

   Opens TCP connection to port 80 (default HTTP server port) at www.cse.ohio-state.edu.
   Anything typed in sent to port 80 at www.cse.ohio-state.edu.

2. Type in a GET HTTP request:

   GET /~xuan/index.html HTTP/1.0

   By typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server.

3. Look at response message sent by HTTP server!
HTTP Versions 2, 3

- Max. webpage latency: 250–300 msec (lower is better!)
- HTTP 2, 3 designed for security, performance
  - HTTP 2:
    - Supports encryption as “first-class citizen”
  - HTTP 3:
    - Uses Google’s QUIC transport protocol (UDP) for lower latency
Outline

• Web Basics

• Web Threats and Attacks
  – Information Leakage
  – Misleading Websites
  – Malicious Code

• Countermeasures
Information Leakage

• Sensitive information can be leaked via Web:
  – All files accessible under a Web directory can be downloaded via GET requests
  – Example 1:
    • [http://www.website.com/secret.jpg](http://www.website.com/secret.jpg) publicly accessible
    • [http://www.website.com/index.html](http://www.website.com/index.html) has no link to secret.jpg
    • Attacker can still download secret.jpg via GET request!
  – Example 2: searching online for “proprietary confidential” information
Misleading Websites

• Cybersquatters can register domain names similar to (trademarked) company, individual names

• Example: http://www.google.com vs. http://gogle.com vs. ...

• Practice is illegal if done “in bad faith”

• Arbitration procedures available for name reassignment (ICANN)
XSS and CSRF

- Cross-site scripting (XSS): inject JavaScript from external source into insecure websites
  - Example: input `<script type="text/javascript"><!--evil code--></script>`

- Cross-site request forgery (CSRF): force victim browser to send request to external website → performs task on browser’s behalf
SQL Injection

• Common vulnerability (~71 attacks/hour [18])
• Exploits Web apps that [17, 19]
  – Poorly validate user input for SQL string literal escape characters, e.g., '
• Example: [19]
  – "SELECT * FROM users WHERE name = '' + userName + '';"
    • If userName is set to ' or '1'='1, the resulting SQL is
      SELECT * FROM users WHERE name = '' OR '1'='1';
    • This evaluates to SELECT * FROM users ⇒ displays all users
Malicious Shellcode

• Shellcode is non-self-contained binary executable code
  – Distinct from malware that executes on its own
  – Shellcode can only execute after injection into a running process’s virtual address space

• Most shellcode written in Intel IA-32 assembly language (x86)

• When injected into JS code, shellcode executes
  – Hijacks browser process
  – Can totally control target process or system

• Shellcode: attack vector for malicious code execution on target systems (e.g., Conficker worm)
  – Usually, browser downloads JS code containing shellcode
  – JS code executes, controls target process/system
A Toy Shellcode

- Shellcode for `exit()` system call
  - Store 0 into register ebx
  - Store 1 into register eax
  - Execute instruction `int 0x80`
- Assembled shellcode injected into JS code

<table>
<thead>
<tr>
<th>JS code</th>
<th>...3caab00000000b801000000cd80ad46...</th>
<th>more JS code</th>
</tr>
</thead>
</table>

Disguised as normal data; injected into target processes’ address spaces; compromises target processes’ security
Outline

• Web Basics
• Web Threats and Attacks
• Countermeasures
  – HTTPS
  – Blacklist Filtering
  – Malicious Code Detection
HTTPS (HTTP Secure)

• HTTPS uses cryptography with HTTP [8]
  – Alice, Bob have public, private keys; public keys accessible via certificate authority (CA)
  – Alice encrypts message with Bob’s public key, signs message with her private key
  – Bob decrypts message with his private key, verifies message using Alice’s public key
  – Once they “know” each other, they can communicate via symmetric crypto keys

• HTTPS provides greater assurance than HTTP
TLS/SSL

- HTTPS uses Transport Layer Security (TLS), Secure Sockets Layer (SSL), for secure data transport [8]
  - Data transmitted via client-server “tunnel”
  - Much harder to compromise than HTTP

- Problems: [8]
  - Relies on CA infrastructure integrity
  - Users can make mistakes (blindly click “OK”)

Source: [8]
**HTTPS Example**

- User visits website via HTTPS, e.g., [https://gmail.com](https://gmail.com)
- Browser sends TLS/SSL request, public key, message authentication code (MAC) to gmail.com; gmail.com does likewise
  - TLS/SSL encrypt entire connection; HTTP layered atop it
  - Both parties verify each other’s identity, generate symmetric key for following communications
- Browser retrieves public key certificate from gmail.com signed by certificate authority (Equifax)
  - Certificate attests to site’s identity
  - If certificate is self-signed, browser shows warning
- Browser, gmail.com use symmetric key to encrypt/decrypt subsequent communications
Blacklist Filtering (1)

- Misleading websites: Register domain names similar to trademarks, e.g., www.google.com, gogle.com, etc.
- XSS:
  - Validate user input; reject invalid input
  - Blacklist offending IP addresses
- CSRF:
  - Use random “token” in web app forms
  - If token is replayed, reject form (blacklist IP addresses)
- SQL injection:
  - Validate user input to databases, reject invalid input
  - Blacklist IP addresses
Blacklist Filtering (2)

• Helpful browser extensions:
  – NoScript/NotScripts/… (stop XSS)
  – AdBlock (can stop malicious scripts in ads)
  – SSL Everywhere (force HTTPS)
  – Google Safe Browsing
  – etc.
Defending Against Shellcode

- Two main detection approaches:
  - Content Analysis
    - Checks objects’ contents before using them
    - Decodes content into instruction sequences, checks if malicious
  - Hijack Prevention
    - Focuses on preventing shellcode from being fully executed
    - Randomly inserts special bytes into objects’ contents, raises exception if executed
    - Can be thwarted using several short “connected” shellcodes
Content Analysis

• Two major types of content analysis:
  – Static Analysis
    • Uses signatures, code patterns to check for malicious instructions
    • Advantage: Fast
    • Disadvantages: Incomplete; can be thwarted by obfuscation techniques
  – Dynamic Analysis
    • Detects a malicious instruction sequence by emulating its execution
    • Advantages: Resistant to obfuscation; more complete than static analysis
    • Disadvantage: Slower

• Focus on dynamic analysis (greater completeness)
Dynamic Analysis

• Approaches assume self-contained shellcodes
• Analyses’ shellcode emulation:
  – Inefficiently uses JS code execution environment information
  – All memory reads/writes only go to emulated memory system
  – Detection uses GetPC code
• Current dynamic analysis approaches can be fooled:
  – Shellcode using JS code execution environment info
  – Shellcode using target process virtual memory info
  – Shellcode not using GetPC code
• To detect all malicious shellcodes, we need a better approach
• Our design rationale: [20]
  – Use dynamic analysis to detect malicious JS objects
    • Create a virtual execution environment for detection
      – Leveraging: (1) target processes’ virtual memory information; (2) target systems’ context information in detection
      – NOT a whole-system emulator
  – Facilitate multiple-level redundancy reduction
    • Stack frames: check origins of JS code being interpreted
    • Native methods: check if native methods to be called originate from JS interpreter or external components
    • Objects’ properties
• Assume: JS interpreter’s (native) methods have no memory errors
JSGuard (2)

• It’s hard to fool our method: [20]
  – Shellcode can use JS code execution environment information to fool other dynamic analysis approaches
    • Our design leverages system’s context information
  – Shellcode can use target process’s virtual memory information to fool other dynamic analysis approaches
    • Our design uses target processes’ virtual memory information
  – Shellcode can avoid GetPC code to fool other dynamic analysis approaches
    • Our method does not rely on GetPC code for detection. We leverage real virtual memory content to decode instructions and emulate their execution
JSGuard (3)

- JSGuard architecture shown in figure below [20]
- We mainly check JSString objects for shellcode injection (hard to inject shellcode in other JS objects)
- Architecture runs in client-side application’s address space (Firefox browser)
- JSString objects input to malicious JSString detector, which scans for shellcode using shellcode analyzer

Source: [20]
Summary

• Web based on plaintext HTTP protocol (stateless)
• Web security threats include information leakage, misleading websites, and malicious code
• Countermeasures include HTTPS, blacklist filtering mechanisms, and malicious code detection
References (1)

References (2)

References (3)
