Web Security

CSU CS557 - Fall 2017

Instructor: Lorenzo De Carli
Network Security

“Traditional” focus of network security
Network Security - II

What about stuff running on these guys?

“Traditional” focus of network security
What is network security?

• Traditionally, the focus of the networking community has been on security **IN** the network

  • Isolation between network segments
  
  • Network intrusion detection
  
  • DDoS detection/mitigation
  
  • ...

• But security of networked endpoints is just as critical!
What is this lecture about

• Security of network applications
  • In particular, of web services
  • Many other security issues, but this one is rather prevalent
Why worry about network application security?

- DDoS, malware etc. are just part of the picture.
- Lots of issues come from misuse/exploit of network application functionality.
  - E.g. making benign webservers/client perform malicious actions.
- Hard to detect these attacks purely at the network level.
  - Especially without understanding of application structure and goals.
Web applications

• Generally, a client-server application where part of the computation and UI run on a client (web browser), and part of the application logic and data storage run on a server (web server)

• Some variations possible (e.g. some storage on client)

• In this lecture we are going to stretch the definition to include most dynamic web sites (e.g. facebook.com, cnn.com, etc.)

• We focus on web applications due to their prevalence
Web application structure

- **Core application logic** on server (Node.js, Python, Java, Ruby running on a web application engine)

- **Presentation logic** on client (DHTML, CSS, JavaScript in a web browser)

- Persistent storage (user data) on server side, typically using SQL or NoSQL database

- Note: this is an oversimplification! Typically server side consists of more components, can be replicated/distributed, run in a cloud etc.
Web application standards

• Server-side logic uses a myriad of frameworks, languages, deployment styles, platforms, etc.

• Client-side: HTML+JavaScript to define application logic & presentation in a web browser

• Data exchange between client and server: HTTP, oftentimes using REST-style APIs

  • Modify/access resources using HTTP and textual descriptions (e.g. JSON data)
Is this the only model for applications on the web?

• No, but it is rapidly eating everything else

• Purpose-specific protocols (e.g. FTP, IRC) rapidly being replaced by equivalent services provided by web applications (Dropbox, Facebook chat, etc.)

• Side note: even streaming services now run over HTTP! (DASH - dynamic adaptive streaming over HTTP)
Web security issues

• In principle, there was the static web...
Web security issues - II

• …but things have changed a lot in the last 20 years!
Web security issues - III

• We now use the web for “grown-up” stuff
  • No, not *that* grown up stuff :-) 
  • I am talking about banking, remote control of home and industrial devices, storage and sharing of personal information, etc.
  • Sensitive tasks make protection of communication and data critical!
Web security issues - IV

- Two main problems:
  - Securing network communication between web client and server
  - Securing integrity of the (client-side and server-side) applications themselves!
Security of network communications

- In principle, there was **HTTP**

- HyperText Transfer Protocols (most of the stuff in the early Internet was text pages with link) - established 1989

- Application-level protocol to **request** and **modify** resources on a networked server

  - GET <URI> to request a resource

  - POST <URI> <DATA> to submit data to a resource

  - ...
Security of HTTP

- Basically, none
- HTTP consists of request and response messages sent on top of a TCP connection
- No provision for secrecy (encryption), integrity (hashes/checksums), etc.
- Limited provisions for authentication
  - Nothing much beyond obfuscated usernames/passwords
Security of HTTP - II

• Using basic HTTP means establishing an unencrypted, unauthenticated, unverified channel between client and server

• What could possibly go wrong?
Security of HTTP - III

Client

GET mybankingdata.html

200 OK <banking data>

HTTP channel

POST API/transferlotsofmoney

Server

Attacker

• Snooping

• Forgery/reply

• ...
Solution: SSL/TLS

• SSL stands for Secure Socket Layer
  • Originally proposed by Netscape in 1995
  • Transitioned to a universal standard (RFC) in 1999 as TLS 1.0
  • End-to-end security implemented on top of TCP
  • Implemented at the application level as shared library - gives apps the abstraction of secure transport layer (TLS stands for “Transport Layer Security” :-) )

• Typically uses dedicated TCP port (e.g. 443 for HTTP over TLS, or HTTPS)
What does TLS do?

- **Authentication based on asymmetric cryptography**
  - Server presents certificate w/ its public key and undergoes challenge/response authentication
  - Client authentication optional (reason: standard designed w/ client/server paradigm in mind)

- **Secrecy based on symmetric cryptography**
  - After authentication, client/server establish a symmetric key for encrypting channel data

- **Integrity**
  - MAC (message authentication code)
SSL/TLS in a nutshell

- Note: this is a significant simplification
- The actual TLS standard is far more complicated!
- Take CS556 if you want to know more :-)

Client

GET webpage.html

Security parameter configuration

Server certificate

Server challenge/response

(Optional) client certificate

(Optional) client challenge/response

Secure channel established!

Server
TLS Adoption

- Slowly crawling towards 100%!

Higher-level security issues

- Breaking into communication channels is not the only way to compromise application security

- Breaking into the application itself is likely to be easier than breaking HTTPS

- Note that those are distributed applications:
  - The client can be vulnerable
  - The server can be vulnerable
Browser security

- In principle, content was fully static
  - Pages built using HTML and stored on the server
  - Browser would fetch page and render it
Browser security - II

- Then, the first web applications arrive

- No real intelligence on the client - all logic run on server

- Performing any operation would require sending a request to server and reloading the page
• Browsers started to supporting some dynamic aspects in web page very early

• JavaScript introduced by Netscape in 1995

• However, for a long time it remained very limited, then things started to change

• Microsoft introduce Dynamic HTML in 1997 - enabling JavaScript to change the structure and content of a page without reloading it

• Still not enough to implement distributed applications

• What is missing?
The real game changer: XMLHttpRequest

JavaScript API call allowing JavaScript code running in a webpage to talk to a server

Introduced by Microsoft around 2000

The missing piece: enabled web pages that can perform computations, dynamically update their content, and communicate with remote servers

Basically, distributed applications
Browser security - V

• The rest is history
The browser as an OS

- Modern browsers are fundamentally acting as Operating Systems running multiple applications.

- Each browser tab runs presentation code (HTML) which defines how UI (web page) looks like, and application logic (JavaScript) which performs computation, fetch and send data, and update the UI.

- Differently from traditional applications, the program being run is not monolithic but dynamically constructed.
Browser-based applications

• When loading a web page, content is dynamically pulled from many sources
  • HTML, JavaScript code, images, etc.
  • Not only static content - active content (programs) too!
    • E.g. imported JavaScript libraries
• Security nightmare! A program which:
  • …consists of multiple concurrently running snippets of JavaScript code…
  • …which have full access to the structure of the web page and can receive and send data from wherever…
  • …and are dynamically imported from external website
• What could possibly go wrong?
The same-origin policy (SOP)

• Cornerstone of browser security

• Restricts certain operations on a page to scripts that come from a webpage with the same origin (host/protocol)

• Typical example:
Other issues

• The same-origin policy prevents obvious unwanted interactions between scripts and pages

• Many other issues remain! Mostly related to:
  
  • Dynamic nature to HTML pages
  
  • Lack of separation of presentation layer (HTML) and application logic (JavaScript)
  
  • Lack of sanitization of user input
Language (in)security

• Fundamental issue of client-side web code: tends to mix presentation (HTML) and application logic (JavaScript) in the same structure

• DOM tree:

• JavaScript can alter most aspects of the DOM, including generating and running new JavaScript code dynamically
Language (in)security - II

• The dynamic nature of web applications create many problems!

• Most significant issue consists of various types of code injection attacks

• Basic problem: unsanitized user input (e.g. from a form) includes code which ends up being executed by the browser or the server

• Let’s seem some examples
Cross-site scripting (XSS)

- Probably the most well-understood and well-known web app vulnerability
- Still one of the most widespread ones
- Involves causing a benign webservice to serve malicious JavaScript code to victim clients
- Various types; we are going to consider reflected and persistent XSS
Reflected XSS

• Basic idea:

  • A benign web server receives user input which is used to generate part of the web page sent back to the client (e.g. by displaying the same input as a string in the page)

  • The application does not sanitize the input, so JavaScript code in the input may become part of the page and be executed

  • Attacker tricks user to click on a click which uses vulnerability to inject malicious code in the page

  • User browser loads the page and executes malicious code
Reflected XSS - example

(From Excess XSS by Jakob Kallin and Irene Lobo Valbuena, Chalmers University of Technology)
Persistent XSS

• Reflected XSS is bad enough…

• But **persistent XSS** is worse!

• In persistent XSS, the attacker somehow injections malicious JavaScript code on the server (e.g. as a DB entry)

• After the injections, all clients visiting the compromised page may receive and execute the malicious code!

• Classic example: post a content containing malicious code in a forum posts

  • All users browsing the forum will execute the code!
Persistent XSS - example

(From Excess XSS by Jakob Kallin and Irene Lobo Valbuena, Chalmers University of Technology)
Can we do something about this?

- Good programming practices are a start
- Always check user input, e.g. ensure that it does not contain `<script>` tags
- Still not enough! JavaScript offers many ways to sneak malicious code as innocent-looking text
- Sanitization of web input is an active research area:
  - Homeijer et al., Beck, USENIX Security 2011
  - Saxena et al., Scriptgard, CCS 2011
Can we do something about this? - II


- No sanitization system truly fits the requirement of web applications
  - Legitimate code is sometime blocked
  - Existing sanitizers do not protect against all attacks
XSS is going away right?

Source: https://snyk.io/blog/xss-attacks-the-next-wave/
XSS containment

• XSS is here to stay

  • JavaScript + DOM is too complex and dynamic to secure properly

  • People will always make mistake when performing input sanitization

• Different angle: XSS containment
XSS containment

• General idea: identify and block scripts and resources which may have been introduced in a web page maliciously

• Two approaches:

1. Content security policies
2. Javascript templating
Content Security Policy

- **CSP** for friend and family

- General idea: bind a web page to a policy describing which resources can be legitimately loaded from that page
  - Source domain for acceptable external resources
  - Hashes of acceptable resources
  - Nonces (more on this later)
CSP location

- Typically sent by server as HTTP header field (outside HTML DOM - cannot be touched)
- Example (https://content-security-policy.com/browser-test/):

**HTTP Request:**

GET /browser-test HTTP/1.1
Host: content-security-policy.com
User-Agent: Mozilla/5.0 (Macintosh; Intel ...)
Accept: text/html
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://content-security-policy.com/
Connection: keep-alive
If-Modified-Since: Mon, 03 Apr 2017 17:39:49 GMT
Cache-Control: max-age=0

**HTTP Response:**

HTTP/1.1 200 OK
Server: nginx
Date: Wed, 18 Oct 2017 23:02:38 GMT
Last-Modified: Mon, 03 Apr 2017 17:39:49 GMT
Connection: keep-alive
Content-Security-Policy: default-src 'none'; script-src 'self' ssl.google-analytics.com 'sha256-xzi4zkCjuC8lZcD2UmnrqDG0vurmq12W/XKM5Vd0+MIQ='; style-src 'self'
maxcdn.bootstrapcdn.com fonts.googleapis.com;
font-src fonts.gstatic.com
maxcdn.bootstrapcdn.com; img-src 'self'
ssl.google-analytics.com;

<HTML content>
CSP directives

- CSP consists of a set of directives for the browser
- Most well-known ones involve controlling JavaScript code
  - Also possible to filter images, stylesheets, etc.
- Most relevant for our discussion is `script-src`
CSP directives - II

- **script-src**: specifies valid sources for JavaScript code. Valid arguments:
  - ‘self’: same origin as the current page
  - ‘none’: no source
  - `<hash-source>`: hash of safe script. Example: ‘sha256-cLuU6nVzrYJjo7rUa6TMrz3nylPFrPQrEUpOHllb5ic=’
  - `<nonce-source>`

Source: https://blog.mozilla.org/security/2014/10/04/csp-for-the-web-we-have/
What is this ‘nonce’ thing?

- **Nonce**: random number generated in a cryptographically secure way (== ‘hard to guess’) and only used once

- How are nonces used to protect web page integrity?
  - Every time a web page is returned, the web server generates a nonce $R$ and associates it to each JavaScript program used by the page
  - A directive of the form `script-src 'nonce-R'` is inserted in the page’s CSP
  - The browser rejects (i.e., does not execute) any JavaScript code whose nonce is missing or does not match $R$
Nonce example

- CSP in the HTTP header will contain:
  
  `content-security-policy: default-src 'self'; script-src 'nonce-2726c7f26c'`

- Script in the served HTML page will contain:
  
  `<script nonce="2726c7f26c">
  alert(123);
  </script>`

- (Source: https://blog.mozilla.org/security/2014/10/04/csp-for-the-web-we-have/)
Report vs Enforcement

- CSP can be run in **report mode** or **enforcement mode**

- **Report mode**: loads and executes resources violating the policy, but logs events to a server-provided URL

- **Enforcement mode**: blocks anything that violates the policy

- **Idea**: try report mode first, once the policy has been polished and tuned switch to enforcement mode
CSP sounds great...

...so everyone is using it, right?

Well...

- “Content Security Problems”, Calzavara et al., CCS 2016
- Crawl of Alexa top 1M site (roughly “the most popular sites on the Internet”)
- Results: 8133 found to use some form of CSP (adoption rate is a whopping 0.8% of crawled sites)
- Of those, only 246 websites enforce robust CSP policies (i.e., 0.02% of crawled sites)
Developers just don’t get it

• Common problems:

• Typos in policies (breaking them!)

• CSP in report-only mode without a report URL being specified (policy violations are lost!)

• In many cases, it seems that developers are using CSP without even knowing it (e.g., because it is implemented by the web application engine used for the website)
CSP problems

- Why are people so wary of CSP?

- Unless a policy is carefully developed, it risks breaking the web application
  - E.g. by forbidding legitimate scripts to run

- Resistance to adoption probably grounded in this issue

- New version of CSP standard enable more dynamism
CSP problems - II

- Is the future bleak then? Well, looks like despite its problems, adoption of CSP is growing...

![Figure 1: Committing and abdicating websites](https://example.com/figure1.png)

(Source: Calzavara et al., CCS 2016)
Another option

• CSP is a useful technology, but presents a number of limitations

• Another approach is to **block malicious scripts** before they are served to the client

• Can be done using **JavaScript templating/whitelisting**
1. Build a model of all benign JavaScript programs that can be served by the web application

2. Configure proxy or IDS to block any script that does not fit the model

Problem: web applications generate JavaScript dynamically - so in general, every time a page is returned the JavaScript code may be different!
JavaScript templating - II

• CSPAutoGen (Pan et al., CCS 2016)

• Approach:
  • Crawl website, collecting a large number of JavaScript samples
  • Generate **templates** describing sets of similar JavaScript programs
  • Configure the gateway to block every snippet of served JavaScript code which does not match a template
How are templates generated?

- Each snippet of JavaScript code is translated to a simplified representation (generalized ASTs) where fine details are discarded.
  - Values/names of individual variables are replaced by abstract variable types.
- Issue: web applications evolve quickly.
  - Templates become ineffective (i.e., start blocking benign scripts) after a few months.
That’s all for today!