15-316: Software Foundations of Security and Privacy

Lecture Notes on Web Security: Application Model & Same-Origin Policy

Matt Fredrikson

Carnegie Mellon University
Lecture 20

1 Introduction

So far we have studied two very general types of security policies: safety and information flow. We have done so without referring to specific types of systems that are in common use, and for the most part without describing concrete vulnerabilities that these policies are meant to protect against. In today's lecture, we will begin looking at these policies in the context of web applications. We will first get some background on how web applications are structured and what the potential sources of vulnerability are, and then look at various forms of attack that exploit lapses in safety and information flow security. Throughout, we will discuss mitigation techniques and best practices. This unit will cover the next 2-3 lectures, starting with an overview of web application architectures and injection vulnerabilities today.

2 Web Applications

Before we can understand issues involving safety and information flow on the web, we need a bit of background on how web applications are typically developed, how they function, and the platform that they execute on. Web applications are *multi-tiered* applications that consist of code on a *client*, which is operated by the end-user who runs a browser, and a *server* that is operated by the web application owner. We'll first look at the client, and then the server, and finally see how the pieces fit together.

2.1 Hyptertext Transfer Protocol

The client and server components of a web application communicate with each other using *Hyptertext Transfer Protocol*, often referred to as HTTP. While we will focus on the situations where the client is a web browser, this need not be the case, and there are many different types *User Agents* that consume data obtained via HTTP – mobile apps, video game consoles, and Internet of Things devices such as "smart" thermostats and other home appliances are all examples of HTTP clients.

HTTP dates back to the 1980s, and allows the client to request a resource specified by a *Uniform Resource Location*, or URL. A URL consists of the following elements:

```
scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]
```

Scheme. A URI scheme specifies the protocol that should be used to look up the resource. The most common schemes on the web are `http` and `https`, but you have probably also encountered `ftp`, `mailto`, `file`, and `irc`.

User and password. Optionally, a URI can contain a username and password for resources that require password authentication.

Host. The host can be a registered name (e.g., `google.com` or `andrew.cmu.edu`) or an IP address (e.g., `28.2.42.10`).

Port. The network port number on which the resource can be accessed. Ports are given by 16-bit integers, and common numbers are 80 (for `http` resources), 443 (`https`), and 143 (IMAP email protocol).

Path. The path component is a hierarchical, slash-separated string that typically resembles a file path.

Query. Non-hierarchical data that is typically used to pass arguments to the server side of a web application in the form of key-value pairs. For example, the query string `q=cmu` is used by Google's search engine to specify the search query, so navigating to `google.com/search?q=cmu` will return the search results for query "cmu".

Fragment. The fragment portion of a URI can specify a secondary resource, such as a section heading within a page or a location somewhere in the middle of a file.

An HTTP *session* is a sequence of request-response transmissions between a client-server pair. The server listens for connections on a well-known TCP port, typically 80 or 443. On receiving a connection and subsequent request, the server responds with a status code, and barring an error, a message with the contents of the requested URL.

Requests and responses. There are several types of requests that a client can make of a server. Two that are most often encountered in typical browser sessions are **GET** and **POST**.

```
GET index.html HTTP/1.1
Accept: text/html image/gif, image/x-bitmap, image/jpeg
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Host: cmu.edu
Connection: keep-alive
```

Figure 1: Example HTTP **GET** request.

- A **GET** request asks the server for the contents of the specified URL, and nothing else. Importantly, this request should have no other effect on the requested URL.
- A **POST** request asks the server to accept some data contained in the request for processing or annotation of the URL. For example, these requests are commonly used to transmit the information entered by a user in a web form to the server.

Other types of requests include **HEAD** (return the header of a response without the body), **PUT** (store the request contents under a specified URL, or create the URL if it does not exist), **DELETE** (remove the specified resource), and **OPTIONS** (query the server for implemented methods). We will not discuss these types of requests further, as they are often not implemented, disabled, or irrelevant to the security issues we will focus on.

In addition to one of the methods listed above, and HTTP request may contain a number of headers that provide additional information. The Internet Engineering Task Force (IETF) has standardized a number of header fields, and there are many more that are commonly used and supported by modern browsers, but have not been standardized. Figure 1 shows an example of a typical **GET** request. The path of the requested URL is given after the method name, in this case `index.html`. This is followed by a list of content types that are accepted by the client, and a `User-Agent` string that identifies the server being used by the client to make the request. The `Host` field identifies the domain name of the server, which may be needed if, for example, the server hosts several domains at the same time. Finally, the `Connection` field specifies configuration directives desired by the client. In this case, the client tells the server to keep the underlying network connection alive, and await further requests.

The response sent by the server contains a status code, followed by a list of header fields, an empty line, and lastly the contents of the URL. An example is shown in Figure 2.

Maintaining state. Most web applications are stateful, and account for the user's interaction with a sequence of URLs across the span of a session. Examples of stateful behavior include authenticating users, maintaining shopping carts, remembering preferences, and tracking user behavior. However, the basic HTTP protocol we have discussed so far is stateless, so some additional data in the form of a *cookie* is needed to associate requests with sessions.

```
HTTP/1.1 200 OK
Date: Tue, 10 April 2018 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Encoding: UTF-8
Content-Length: 138
Last-Modified: Mon, 09 April 2018 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)

<html>
<body>
  Hello World
</body>
</html>
```

Figure 2: Example HTTP response.

A cookie is a small piece of data that is sent by a user's browser to a web server. Cookies are associated with a domain and path, so that all requests that match these elements result in the browser sending the cookie along with a request. Cookies can be set in HTTP responses, or from JavaScript code running in the browser. Although the data stored in cookies can be arbitrary (but limited in size), most of the time they contain unique identifiers that tell the server who the user is. Figure 3 shows the sequence of messages that result in the establishment and use of a cookie. Notice that because the server did not specify an expiration date on the cookie, the browser will treat it as a session cookie in this example.

There are two types of cookies: persistent cookies and session cookies. Persistent cookies are stored in the browser permanently (although most browsers allow users to delete them whenever they want), and are used to track users across multiple sessions including ones that span browser shutdown. Persistent cookies have a specified expiration date, after which the browser will automatically delete them. Session cookies are ephemeral, and reside in the browser's memory only for as long as the user navigates the website. Session cookies have no expiration date, and disappear once the user closes the browser tab or navigates away from the website.

Cookies enable applications that first authenticate users with a login challenge (e.g., apps that request a username and password). When the user visits the login site, they complete a form containing username and password. These are sent to the server-side portion of the app (e.g., using URL parameters over an encrypted connection), which checks the provided credentials against a back-end database. If the correct credentials were provided, then the server responds by sending a response containing a session cookie that the server-side app associates with the successfully-authenticated user. They can then navigate the website without having to provide credentials each time they need to view protected content.

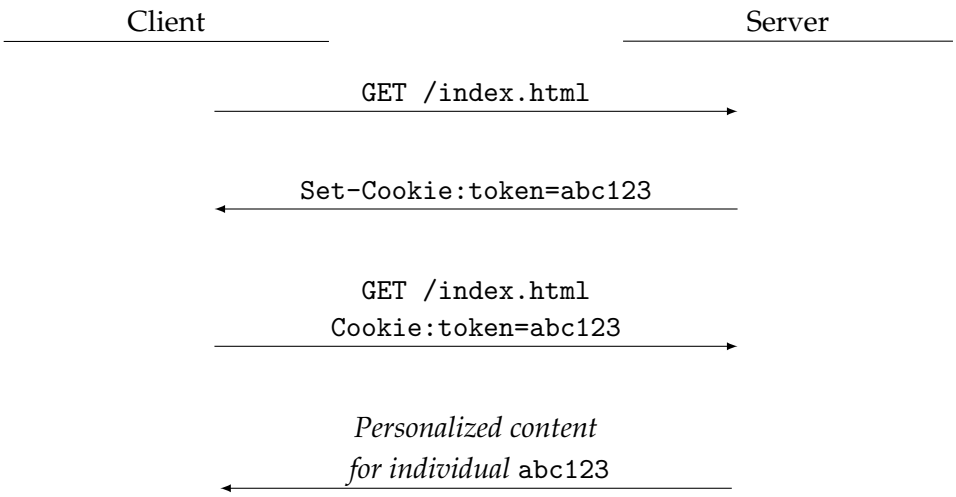

Figure 3: Simplified sequence of requests and responses to establish and utilize a cookie. If an HTTP request does not contain a cookie header, then the server responds by setting a cookie that is remembered by the browser and included in future requests.

2.2 Web apps on the client

The client side of a web application is run inside of a browser, which is a native program that makes requests to remote servers, downloads the code and associated data of the web app, and renders it in a graphical interface displayed to the user. Although the internal architecture of browsers varies from vendor to vendor, the architecture of the Chromium engine shown in Figure 4 is representative enough for our purposes in this lecture.

The browser consists of a main process that manages the user interface, the (potentially) multiple tabs, and and plugins (also called extensions). This is referred to as the "browser process" or sometimes simply the browser. Each open tab corresponds to a separate *renderer* process that is ultimately responsible for running the web application code and deciding how the results should be displayed in the user interface. Finally, each tab can contain resources fetched from different sources, called *frames*. For example, websites often include frames for advertisements, where the contents of an advertising frame come from a different domain than that of the main content.

Chromium implements each process in the architecture as its own operating system process. There are several good reasons for doing this [BRJI08], mostly having to do with robustness to programming bugs and security. Because OS processes have hardware-enforced memory isolation boundaries, bugs, vulnerabilities, and exceptions that arise in one process can have limited effect on others that are running concurrently. So keeping the rendering engines separated from eachother and the main browser process ensures that bugs and vulnerabilities that arise when rendering content remain somewhat isolated in their effect on the rest of the tabs and browser. The downside

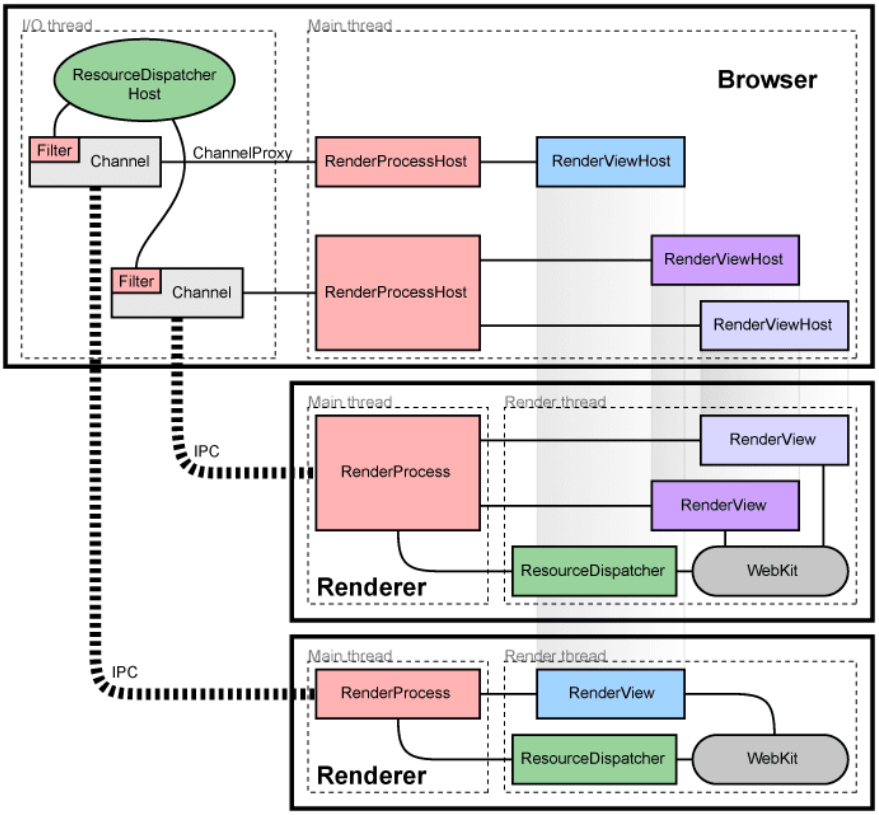

Figure 4: The multi-process architecture of the Chromium browser engine. Image source: https://www.chromium.org/developers/design-documents/multi-process-architecture

to this architecture is the overhead created by running a potentially large number of distinct processes that make frequent use of inter-process communication, leading to decreased system-wide performance and additional pressure on memory resources.

HTML & CSS. In the early days of the web ("Web 1.0"), the content rendered by browsers was static in the sense that it did not change and for the most part did not respond to user input. Web applications, which were nothing more than collections of web pages, consisted entirely of *Hypertext Markup Language* (HTML) documents that browsers used to interpret and compose text, images, and other content visible in the browser window.

HTML documents specify the structure of a webpage by specifying the grouping and relative layout of a set of HTML elements. The elements correspond to entities such as the title, header, paragraphs, and images of a page. Syntactically, elements are specified by *tags* given in angle brackets with tag names and attributes. For example, the following tag corresponds to an image element that will render `smiley.gif` with the specified height and width, and alternative text "Smiley face" in case the image for some reason cannot be rendered.

```
<img src="smiley.gif" alt="Smiley face" height="42" width="42">
```

Around the same time that HTML was proposed, *Cascading Style Sheets* (CSS) were proposed as a clean way to separate the content of a web page from its presentation. A CSS document specifies how the elements in an HTML document should be rendered, including aspects of layout, sizing, font, and coloring. An HTML document associates itself with a given CSS by specifying it in a tag. By separating content and presentation in this way, it is possible for a single HTML document to render appropriately on multiple types of devices or in several different modes. For example, most websites today specify different style sheets for desktop and mobile browsers to account for differences in form factor. CSS also makes it possible for multiple HTML documents to share the same presentation style, thus eliminating redundancy that would otherwise need to exist in the HTML tag attributes and simplifying the process of changing aspects of presentation.

Javascript. In 1995, the developers of Netscape wanted to add additional functionality to websites by allowing them to run scripts in the context of a rendered HTML document. The original goal was to embed the Scheme programming language into websites [Rau14], but due to a strategic collaboration with Oracle (the creators of Java) it was decided that the language should ultimately complement Java programs and use similar syntax. This led Brendan Eich, who Netscape had hired to develop this web scripting language, to create the initial prototype of the JavaScript language in ten days to preemptively defend the choice against competing proposals for other languages.

The result was a high-level interpreted language with no static typing discipline, prototype-based support for object-oriented programming, and essentially no resemblence to Java other than through its use of curly braces rather than parenthesis for lexical

grouping. It supports APIs for dealing with strings, arrays, dates, regular expressions, and rendered HTML content (more on this later), but contains only limited facilities for I/O. Although it has gone through several rounds of update and standardization, the same JavaScript that Eich developed in his early prototype remains universal on the web today.

JavaScript contains a number of unfortunate features that make it difficult to reason about both for developers and for automated tools that support safety and correctness. The most famous such feature is the `eval(str)` function, which takes a string argument and evaluates it as a JavaScript program. This allows programs to dynamically compute programs and run them, which causes obvious problems for static analysis. Another difficulty comes from JavaScripts *type coercion*, wherein data of a particular type is converted to a different type automatically. Consider the following example from Douglas Crockford [Cro08] where string, numeric, and Boolean data are coerced behind the scenes in unpredictable ways.

```
'' == '0'           // false
0 == ''             // true
0 == '0'            // true

false == 'false'    // false
false == '0'        // true

false == undefined  // false
false == null       // false
null == undefined   // true

' \t\r\n ' == 0     // true
```

We could fill several lectures with discussion of the many terrible features that remain supported in JavaScript, but for our purposes it suffices to say that conventional analysis techniques do not apply to JavaScript programs.

Document Object Model. One of the key motivations for incorporating a scripting language into web pages is to allow scripts to update the rendered content programmatically and in response to events such as user interaction. This is supported by the *Document Object Model* (DOM), which is an API for reading and manipulating parsed HTML content. The DOM is actually a language-independent API, but we will focus on its implementation in JavaScript as it is the most relevant to web app security.

The DOM maintains an internal representation of an HTML document as a tree structure. Each node corresponds to an element specified in the HTML, and the root of the tree (called the "document object") corresponds to the top-level document containing all of the elements. The JavaScript program running in the context of a page can make arbitrary changes to the DOM, which the browser will then render back on the UI. By extension, the DOM API allows JavaScript programs to register event handler callback functions on specified elements.

This allows websites to implement dynamic content in many ways resembling traditional desktop applications. As the user interacts with rendered elements, JavaScript

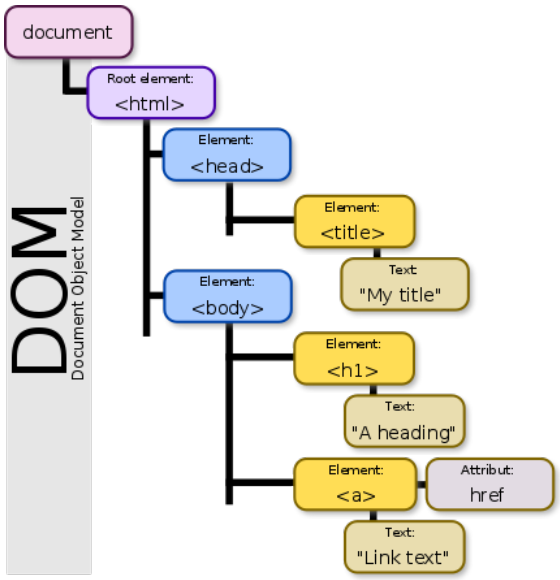

Figure 5: An example of the DOM hierarchy in a simple HTML document. Image source: https://commons.wikimedia.org/wiki/File:DOM-model.svg.

event handlers are invoked which can in turn change the layout and visual appearance of the page. The combination of these technologies—HTML, JavaScript, and the DOM—are the essential client-side elements of modern web applications.

Same-origin policy. Because websites can execute scripts, there is the possibility that malicious websites could use this functionality to interfere with or spy on the content and interactions of other websites. For example, you may click on an untrusted link at the same time that you have a website from your bank or healthcare provider open. It would be problematic if a script running on the untrusted site were able to use the DOM API to snoop on sensitive information displayed on either page, or worse yet, invoke event handlers that cause requests to your bank!

To protect the secrecy and integrity of web application content and data, browsers implement the *Same-origin Policy* (SOP). Roughly, the SOP requires that content loaded in a webpage (call it "website A") can not access data or functionality on another website ("website B") unless A and B have the same *origin*. An origin is defined as the combination of the URI scheme, host name, and port number from with the site was loaded.

Recalling the components of URLs discussed earlier, we can think of the policy as specifying security labels in terms of triples containing (*scheme*, *host*, *port*) from the URI corresponding to a page loaded in the browser. Scripts running under the label for a particular (*scheme*, *host*, *port*) triple are not allowed to access the DOM content or script state of pages running under different triples. This is a type of information flow policy that isolates content from different sources.

Website	*Outcome*
`http://cs.cmu.edu/~15312`	Allowed; same scheme, host, and port
`https://cs.cmu.edu/~15312`	Not allowed; different scheme
`http://www.cs.cmu.edu/~15312`	Not. allowed; different host
`http://andrew:pwd@cs.cmu.edu/~15312`	Allowed; same scheme, host, and port
`http://cs.cmu.edu:81/~15312`	Not allowed; different port
`http://cs.cmu.edu:80/~15312`	Depends on browser implementation

Figure 6: Example outcomes of the Same-origin Policy applied when a script on `http://cs.cmu.edu/~15316` attempts to access the data of pages from different URIs. In the bottom example, the scheme, host, and port all match because the default port for `http` is 80. Regardless, some browsers will not allow this because the port is made explicit in the latter but not the former.

It may help to see a few examples. Consider a script running on the page loaded from `http://cs.cmu.edu/~15316`. Figure 6 shows the SOP result when the script attempts to access the data of other websites. Note that different browser implementations may differ slightly in their implementation of the SOP, as shown in the last example.

Cross-origin access and embedded content. Websites often consist of content from multiple different domains. One common example is images, which are sometimes stored on severs from different origins. Similarly, JavaScript code and CSS documents are frequently provided as libraries and there are good performance reasons for a website to use the source stored on the library vendor's servers. Finally, frames subdivide the browser window into segments with content loaded from possibly unrelated URIs; this is one common way of displaying advertisements.

Although these do not necessarily have anything to do with JavaScript code or DOM access, these forms of embedded content seem to violate the intent of the Same-origin Policy in that they allow information to flow between different origins. For example, when a page from origin A loads an image from origin B, information can leak from A to B both through the simple fact that a request is being made, as well as through the pathname of the requested image. Likewise, information can leak from B to A through any error messages that may occur (e.g., if B returns that the content is inaccessible), and through attributes such as the width and height of the returned image.

Regardless, embedding is typically allowed as it is considered essential to supporting fully-functional web applications. A few case-specific details are with noting though.

- For scripts that are embedded from outside origins with a `<script src=...>` tag, the code is retrieved, parsed, and then run in the context of the origin that requested the script (*not* the origin that it was retrieved from). This supports cross-origin third-party libraries while partially avoiding explicit cross-origin flows. Moreover error messages are only returned for scripts from the same origin as the embedding page.

- Content embedded in an `iframe` can come from arbitrary domains, but is run in the context of the origin that the content is loaded from. So if a webpage in origin A embeds and `iframe` with an advertisement from origin B, the ad will run in the context of B and the Same-origin Policy will apply more or less as though the content were loaded in a separate tab.
- Images from different domains can be loaded and displayed on a webpage, but the contents of the image itself, i.e. its pixels, cannot be read by the page loading the image.
- Cookies use a separate definition of origins. A page can set a cookie for its own domain or any parent domain, as long as the parent domain is not a *public suffix* (there are only a small number of these). The browser will make a cookie available to a page in the cookie's domain, including any subdomains, no matter which protocol or port is used. Cookies can be further scoped by setting the path, thus limiting the pages to which the cookie is sent to be within specified path.

As you might imagine, exceptions to the SOP due to cross-origin access create subtlety and complexity that is sometimes difficult to manage. In brief, you can understand most of these SOP "exceptions" as being governed by the principle that the SOP limits the ability of pages to *read* content from other domains, but not their ability to *send* data to other domains.

2.3 Web apps on the server

We mentioned earlier that web applications are "tiered" into portions consisting of the client side and server side. On the server tier, a common architecture divides the application into components that are responsible for the "application logic" and data storage separately. The application logic component takes care of network communications with the client, doing the core work of receiving requests, computing responses, and sending them back to the client. The data storage component is typically a traiditional relational database backend that contains any data needed to service client requests.

For traditional "static" websites, the database may not play much of a role if HTML documents are pre-generated and stored on the server's filesystem. In this case, the server just parses the URI recieved from the client and uses the path component to locate an appropriate HTML file to send back. But it should come as no surprise that the vast majority of web applications require more than this.

Server-side scripting. Just as client-side scripting supports dynamic content, server-side scripting languages are commonly used to generate dynamic content to send to the client in response to the details of a request. While on the client side JavaScript is almost exclusively used, there is no de-facto standard server-side scripting and there are several common approaches for dynamically generating content on this tier. Among the most common are the *Common Gateway Interface* (CGI) and PHP.

You are already familiar with CGI from its inclusion in Lab 0. In its simplest incarnation, a server that uses CGI executes a program as though it were a console command and sends any output that it produces back to the client. The URI sent by the client for a CGI script might look like the following:

```
https://retailer.com/cgi-bin/purchase?arg1=credit&arg2=5105105105105100
```

In this example, the URI specifies a CGI script called `purchase`, and provides two arguments in the query portion of the URI. The first `arg1` is set to the value `"purchase"`, and the second `arg2` is set to the string `"5105105105105100"`. It is the web server's job to locate an executable program mapped to the path `cgi-bin/purchase`, and execute it with command-line arguments corresponding to those provided in the URI.

The CGI script itself can do whatever the permissions on the server allow. It can perform arbitrary computations, read and write files allowed by the filesystem permissions, contact a back-end database, make network requests, and so on. The point is to generate a response to send back to the client that can be rendered by the browser. So in most cases, the script will generate an HTML document containing information specific to the query it received, and print it to standard output so that the web server can send it to the client.

CGI is still used in web applications, but it is much more common to implement dynamic server-side functionality in a language like PHP. PHP in some ways simplifies the dynamic creation of HTML by allowing developers to write a template HTML document with portions that contain embedded PHP code. When the client requests a PHP file, the web server invokes the PHP interpreter passing the client's arguments, and the embedded PHP elements are evaluated into strings within the HTML template. The end result is an HTML document corresponding to the original template, with specific portions having content that was dynamically computed based on the client's request.

Figure 7 shows an example PHP script and the HTML document that it produces when requested with a particular argument. Notice that the script is embedded within an HTML document (i.e., template). The PHP interpreter runs each embedded script with the arguments given in the request, and whatever output the script produces is placed in the same location within the template as the script itself.

References

[BRJI08] Adam Barth, Charles Reis, Collin Jackson, and Google Chrome Team Google Inc. The security architecture of the Chromium browser, January 2008. URL: http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf.

[Cro08] Douglas Crockford. Appendix B: The Bad Parts. In *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[Rau14] Axel Rauschmayer. *Speaking JavaScript*. O'Reilly Media, Inc., 1st edition, 2014.

```
<!DOCTYPE html>
<html>
  <body>
    <?php
      echo '<p>Hello, $_GET["name"]</p>';
    ?>
  </body>
</html>
```

```
<!DOCTYPE html>
<html>
  <body>
    <p>Hello, Andrew</p>
  </body>
</html>
```

Figure 7: Example PHP script (top) that emits an HTML document (bottom) with a <p> element that is dynamically-computed from the `name` argument provided in the URI `http://ex.com/test.php?name=Andrew`.