CHAPTER

1

Web Browser Security

A lot of responsibility is placed upon the broad shoulders of the humble web browser. The web browser is designed to request instructions from all over the Internet, and these instructions are then executed almost without question. The browser must faithfully assemble the remotely retrieved content into a standardized digestible form and support the rich feature set available in today's Web 2.0.

Remember, this is the same software with which you conduct your important affairs—from maintaining your social networks to online banking. This software is also expected to protect you even if you venture down the many figurative dark alleys of the Internet. It is expected to support venturing down such an alleyway while making a simultaneous secure purchase in another tab or window. Many assume their browser to be like an armored car, providing a secure and comfortable environment to observe the outside world, protecting all aspects of one's personal interests and deflecting anything dangerous. By the end of this book, you will have the information to decide if this is a sound assumption.

The development team of this "all singing and all dancing" software has to ensure that each of its numerous nooks and crannies don't provide an avenue for a hacker. Whether or not you consciously know it, every time you use a browser, you are trusting a team of people you have probably never met (and likely never will) to protect your important information from the attackers on the Internet.

This chapter introduces a methodology for web browser hacking that can be employed for offensive engagements. You explore the web browser's role in the web ecosystem, including delving into the interplay between it and the web server. You also examine some browser security fundamentals that will provide a bedrock for the remaining chapters of this book.

A Principal Principle

We invite you to forget about the web browser for a moment and reflect on a blank security canvas. Picture yourself in this situation: You are in charge of maintaining the security of an organization, and you have a decision to make. Do you deploy a piece of software based on the level of risk it will pose? The software will be installed on the Standard Operating Environment (SOE) for almost every machine in an organization. It will be used to access the most sensitive data and conduct the most sensitive operations. This software will be a staple tool for virtually all staff including the CEO, Board, System Administrators, Finance, Human Resources, and even customers. With all this control and access to business-critical data, it certainly sounds like the hacker's dream target and a high-risk proposition.

The general specifications of the software are as follows:

- It will request instructions from the Internet and execute them.
  - The defender will not be in control of these instructions.
  - Some instructions tell the software to get more instructions from:
    - Other places on the Internet
    - Other places on the intranet
    - Non-standard HTTP and HTTPS TCP ports
- Some instructions tell the software to send data over TCP. This can result in attacks on other networked devices.
- It will encrypt communication to arbitrary locations on the Internet. The defender will not be able to view the communication.
- It will continually increase what attackers can target. It will update in the background without notifying you.
- It often depends on plugins to allow effective use. There is no centralized method to update the plugins.

In addition, field research into the software reveals:

- The plugins are generally considered to be less secure than the core software itself.
- Every variant of the software has a history of documented vulnerabilities.
- A Security Intelligence Report[1] that summarizes attacks on this software to be the greatest threat to the enterprise.[2]

You have no doubt worked out we are referencing a web browser. Forgetting this and the events of history once again and going back to our blank security canvas, it would be mad not to question the wisdom of deploying this software. Even without the benefit of data from the field, its specifications do appear extremely alarming from a security perspective.

However, this entire discussion is, of course, purely conceptual in the real world. We're well past the point of no return and, given the critical mass of websites, nobody can decree that a web browser is a potentially substantial security risk and as such will not be supplied to every staff member. As you already know, literally billions of web browsers are deployed. Not rolling out a web browser to the employees of an organization will almost certainly impact their productivity negatively. Not to mention it would be considered a rather draconian or backward measure.

The web browser has ever-increasing uses and presents different hacking and security challenges depending on the context of use. The browser is so ubiquitous that a lot of the non-technical population views it as "The Internet." They have limited exposure to other manifestations of data the Internet Protocol can conjure. In the Internet age, this gives the browser an undeniably dominant position in everyday life, and therefore the Information Technology industry is tethered to it as well.

The web browser is almost everywhere in the network—within your user network zone, your guest zones, even your *secure* DMZ zones. Don't forget that in a lot of cases, user administrators have to manage their network appliances using web browsers. Manufacturers have jumped on the web bandwagon and capitalized on the browsers' availability, rather than reinvent the wheel.

The reliance on this piece of web browsing software is nothing short of absolute. In today's world it is more efficient to ask where the web browser *is not* in your network, rather than where it *is*.

Exploring the Browser

When you touch the web, the web touches you right back. In fact, whether or not you consciously realize it, you invite it to touch you back. You ask it to reach through the various security measures put in place to protect your network and execute instructions that you have only high-level control over, all in the name of rendering the page and delivering onto your screen the hitherto unknown/untrusted content.

The browser runs with a set of privileges provided to it by the operating system, identical to any other program in user space. These privileges are equivalent to those that you, the user, have been assigned! Let us not forget that user input is at all times nothing more than a set of instructions to a currently running program—even if that program is Windows Explorer or a UNIX shell. The only difference between user input and input received from any other source is the differentiations imposed by the program receiving the input!

When you apply this understanding to the web browser, whose primary function is to receive and execute instructions from arbitrary locations in the outside world, the potential risks associated with it become more obvious.

Symbiosis with the Web Application

The web employs a widespread networking approach called the *client-server model*, which was developed in the 1970s.[3] It communicates using a request-response[4] process in which the web browser conducts the request and the web server answers with a response.

Neither web server nor web client can really fulfill their potential without the other. They are almost entirely codependent; the web browser would have almost nothing to view and the web server would have no purpose in serving its content. This essential symbiosis creates the countless dynamic intertwined strands of the web.

The bond between these two key components also extends to the security posture. The security of the web browser can affect the web application and vice versa. Some controls can be secured in isolation, but many depend on their counterpart. In a lot of instances it is the relationship between the browser and the application that needs to be fortified or, from a hacker's perspective, attacked. For example, when the web server sets a cookie to a specific origin, it is expected that the web browser will honor that directive and not divulge the (potentially sensitive) cookie to other origins.

The security of the web browser's involvement with the web application needs to be understood in context. In many instances, discussions will delve into the interactions between these two components. Exploiting the relationship between these two entities is discussed in the following chapters.

Further research into web application vulnerabilities is strongly encouraged. A great resource for beginners and experienced security professionals alike is the *Web Application Hacker's Handbook*, by Dafydd Stuttard and Marcus Pinto, Wiley, 2011. which at the time of writing, is in its second edition.

Same Origin Policy

The most important security control within the web browser is the *Same Origin Policy*, which is also known as SOP. This control restricts resources from one origin interacting with other origins.

The SOP deems pages having the same hostname, scheme, and port as residing at the same-origin. If any of these three attributes varies, the resource is in a different origin. Hence, provided resources come from the same hostname, scheme, and port, they can interact without restriction.

The SOP initially was defined only for external resources, but was extended to include other types of origins. This included access to local files using the `file://` scheme and browser-related resources using the `chrome://` scheme. A number of other schemes are supported by today's browsers.

HTTP Headers

You can think of HTTP headers as the address and other instructions written on an envelope, which dictate where the package should go and how the contents of the package should be handled.

Some examples might be "Fragile: Handle with Care" or "Keep Flat" or "Danger: Explosives!" They are the prime directives the HTTP protocol uses to dictate what to do with the content that follows. Web clients supply HTTP headers at the start of all requests to the web server, and web servers respond with HTTP headers as the first item in any response.

The content of the headers determines how the content that follows is processed either by the web server or by the web browser. Some headers are required in order that the interaction can function; others are optional and some may be used purely for informational purposes.

Markup Languages

Markup languages are a way of specifying how to display content. Specifically, they define a standardized way of creating placeholders for data and placeholders for annotation related to the data within the same document. Every web page you have seen in your life is likely to have used a markup language to give the web browser instructions for displaying the page to you.

Different kinds of markup languages exist. Some markup languages are more popular than others, and each has its strengths and weaknesses. As you probably already know, HTML is the web browser markup language of choice.

HTML

HyperText Markup Language, or HTML, is the primary programmatic language in use for displaying web pages. Though initially extended from the Standard Generalized Markup Language (SGML), current HTML has gone through numerous changes since then.

The absolute dependence upon markup (coexistence of data and annotation or instructions) is the underlying cause of several important, persistent, and systemic security issues. You learn more about HTML and its innumerable features throughout this book.

XML

XML is a close relation to HTML. If you are familiar with HTML, you won't find XML too much of a challenge. Although neither is particularly pleasing to the human eye, they both provide a very rich way to represent complex data. You will encounter XML in frequent use on the web, normally as a transport for web services or other remote procedure call (RPC) interactions.

Cascading Style Sheets

Cascading style sheets (CSS) is the main method web browsers use to specify the style of the web page content (not to be confused with XSS, which is an acronym for the Cross-site Scripting security vulnerability).

CSS provides a way to separate the content from its style. A very basic example of this is displaying a sentence as bold. Of course, CSS is much more powerful than this simple example and extends itself to the complexity seen on the web.

Scripting

Web scripting languages are an art worth learning! If you interact with the web at a technical level, you are going to run headfirst into them at one point or another. Scripting in general is a prerequisite to working in Information Technology that somehow snuck in and took up a very prominent position in the browser.

You learn in later chapters that scripting in the browser is used by attackers to launch some of the most common exploits, including XSS. You'll need this knowledge in your arsenal.

JavaScript

JavaScript supports functional and object-oriented programming concepts. Unlike Java, which is a strongly typed language, JavaScript is loosely typed. The language has a dominant position in the web ecosystem and will be around for the foreseeable future. It runs in every browser by default.

An understanding of JavaScript is essential for you as a reader because the majority of the code in this book uses this language. Attacks written in JavaScript (not withstanding browser quirks) are compatible across browsers. This makes it a fantastic base language for browser hacking.

VBScript

VBScript is supported only in Microsoft browsers and is rarely used in serious web development. This is because it doesn't have cross-browser support. It is Microsoft's alternative to Netscape's JavaScript and its origin dates back to the early browser wars.

A lot of its functions can be achieved in JavaScript. Obviously, this raises the question of whether VBScript is needed at all. If anything, it seems like a throwback to the days when Internet Explorer had total dominance in this space.

Document Object Model

The document object model (more commonly referred to as the DOM) is a fundamental web browser concept. It is an API for interacting with objects within HTML or XML documents. The DOM provides a method for scripting languages to interact with the rendering engine by providing references to HTML elements in the form of objects.

The DOM is effectively conjoined with JavaScript (or other scripting languages of choice). It was created to allow a defined method of access to the living rendered document, such that scripts running in the browser could read and/or write to it dynamically. This allowed the page to change without making new requests to the web server and without the necessity of user interaction.

Rendering Engines

Rendering engines have been called various names in the context of web browsers, including *layout engines* and *web browser engines.*[5] These names are used interchangeably throughout the book.

These components play an essential role in the browser ecosystem. They are responsible for converting data into a format useful for presentation to the user on the screen. The web browser will likely use HTML and images in combination with CSS to create the final graphical product users see in their web browser. It is these engines that provide the user with the graphical experience. Though usually referred to in the graphical sense, text-based rendering engines exist too, such as Lynx and W3M.

Numerous rendering engines are used on the web.[6] The common graphical rendering engines discussed in this book include WebKit, Blink, Trident, and Gecko.

WebKit

WebKit is the most popular rendering engine and is employed within many web browsers. The most notable browser using the engine is Apple Safari, and in the past Google Chrome used it as well. It is one of the more popular rendering engines in use today.[7]

A goal of this open source project is for WebKit to become a general-purpose interaction and presentation engine[8] for software applications. Apart from its use in web browsers, the engine is used in various kinds of software including e-mail clients and instant messengers.

Trident

Microsoft's rendering engine is called MSHTML or, more commonly, Trident. It isn't a surprise that Trident is a closed source engine found within Internet Explorer. It is the second most popular rendering engine.

Like WebKit, Trident is also used in software other than web browsers. One example is Google Talk. Software can use the engine by using the `mshtml.dll` library, which comes with Windows.

Trident made its first appearance in version 4 of the web browser and has been a staple on the Internet ever since. Microsoft's latest version of Internet Explorer still employs Trident as the core rendering engine.

Gecko

Firefox is the most prominent software program that uses the open source Gecko rendering engine. It is probably the third most popular rendering engine behind WebKit and Trident.

Gecko is the open source rendering engine initially developed by Netscape in the 90s for its Netscape Navigator web browser. Modern versions of Gecko are mainly found in applications developed by the Mozilla Foundation and Mozilla Corporation, most notably the Firefox web browser.

Presto

Presto is (at the time of writing this book) the rendering engine for Opera. However, in 2013 the Opera team publicly announced that it would soon be dropping its in-house Presto rendering engine and migrating to the WebKit Chromium package.[9] The WebKit Chromium package was subsequently renamed to Blink (discussed in the next section).

At no other time has a major browser changed course with its rendering engine in such a dramatic fashion. This will almost certainly spell the extinction of Presto, and it will become one of the latest casualties of the browser wars.

Blink

In 2013 Google announced that it was forking WebKit to create the *new* Blink rendering engine. Blink's initial aim is to better support Chrome's multi-process architecture and reduce complexity in its browser. Time will tell if this engine will perform as well as WebKit, but the suggestion that Google will strip unessential functionality is a good start.

Geolocation

The Geolocation API provides mobile devices and desktops access to the geographical location of the web browser. It achieves this via various methods including GPS, cellular site triangulation, IP Geolocation, and local Wi-Fi access points.

Many obvious instances exist where this information could be abused in real-world scenarios. Rigorous browser security protections have been put in place to reduce exploitation, leaving the main vector of attack as social engineering. This is discussed further in future chapters.

Web Storage

Web storage, sometimes referred to as DOM storage, was part of the HTML5 specification but it no longer is. It may be helpful for you to view web storage as supercharged cookies.

Like cookies, two main types of storage exist: one that persists locally and one that is available during the session. With web storage, *local storage* persists over multiple visits from the user and *session storage* is only available in the tab that created it.

One of the major differences between cookies and web storage is that web storage is created only by JavaScript, not by HTTP headers, nor are they transmitted to the server in every request. Web storage permits much greater sizes than conventional cookies. The size is browser dependent, but is generally at least 5 megabytes. Another important difference is that there is no concept of path restrictions with local storage.

SESSION STORAGE

Here is a simple example of using the web storage API. Run the following commands in the web browser JavaScript console. They will set the `"BHH"` value in the current tab's session store:

```
sessionStorage.setItem("BHH", "http://browserhacker.com");
sessionStorage.getItem("BHH");
```

The SOP applies to local storage with each origin being compartmentalized. Other origins cannot access the local storage, nor can subdomains access it.[10]

Cross-origin Resource Sharing

Cross-origin Resource sharing, or CORS, is a specification that provides a method for an origin to ignore the SOP. In its most lenient configuration, a web application can allow a cross-origin `XMLHttpRequest` to access all of its resources from any origin. The HTTP headers inform the browser whether it is permitted access.

A fundamental component of CORS is the addition of the following HTTP response headers to the web server:

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: POST, GET
```

When a browser sends a cross-origin `XMLHttpRequest` to a server that doesn't respond with these headers, no access will be given to the response content. This is aligned with expected SOP behavior. However, if the web server does return the preceding headers, modern browsers will honor the CORS specification and permit access to content of the response of the origin.

HTML5

HTML5 is the future. Well, not really… it is the present. Although the standard has not been completed, modern browsers are already implementing the core functionality. It is highly likely that the browser you use now supports numerous items from the HTML5 specification.

HTML5 is the next standard for HTML. It defines additions to the specification that augment the functionality, and, in turn, the user experience of the web.

The obvious change from a security perspective is the increase in attack surface. It provides many more methods that haven't had the exposure of the previous HTML4 generation. It also increases the permutations by which functionality can be used. Both of these combined increase the risk of successful attacks. This is true with most steps forward in technology, and in itself should not be a reason not to progress.

This book covers some of the additions, but not all. The ones used in attacks later in the book are discussed briefly in the following section.

WebSocket

WebSocket is a browser technology that enables you to open an interactive and very responsive, full-duplex communication channel between a browser and server. This behavior allows you to have stringent event-driven actions without the explicit need to poll the server.

WebSocket is a replacement for other AJAX-Push technologies such as Comet.[11] Whereas Comet requires additional client libraries, the WebSocket API is implemented natively in modern browsers. All the latest browsers, including Internet Explorer 10, support WebSocket natively. The only exceptions are some mobile browsers like Opera Mini and Android's native browser.

Web Workers

Before web workers, JavaScript in the browser was a single-threaded environment. Developers would use `setTimeout()` and `setInterval()` to achieve concurrency-like execution.

HTML5 introduces web workers, which could be seen as browser threads because they run in the background. There are two types: one is shared across anything that runs in the origin, and the other communicates only back to the function that created it.

The API has various other restrictions, but web workers do provide more flexibility to the developer. This same flexibility is provided to attackers by giving them more options to deploy their attack in the web browser.

History Manipulation

Various attacks covered in this book target the history functionality of the web browser. The capability of the history continues to change as the demand on the web browser changes.

In the past, it was sufficient to track the history when users clicked a link that took them to another page. Today, clicking a link may use scripting to render the page, and this is counted as a milestone in the user's experience.

HTML5 offers methods to manipulate the history stack. Scripts can add or remove locations using the history object, and they can also move the current page forward or backward in the history chain.

WebRTC

The Web Real-Time Communication (WebRTC) API is a significant development that uses HTML5 capabilities and JavaScript. It allows browsers to communicate with each other with the low latency and high bandwidth necessary to support real-time, media-rich communication.

At the time of this writing, WebRTC is supported in the latest Chrome, Firefox, and Opera browsers and is incorporated into them natively. It exposes features such as direct access to camera and audio equipment (to support video conferencing). The potential security implications for this type of high-utility yet invasive technology are obvious. Fortunately, WebRTC is open source so it is not beyond the easy reach of transparent analysis.

Vulnerabilities

The term "vulnerabilities" is an abstract collective and therefore a complex topic. It could be inferred that this book's existence is solely due to the existence of so-called "vulnerabilities." However, the definition of what is and what is not a vulnerability is not always clear. Sometimes a vulnerability is really a piece of functionality as it was originally intended, but which is later proven to be *excessively permissive*.

To make matters worse, some vulnerability classes go by multiple names. As a whole, the entire situation can be confusing. Throughout this book, the vulnerabilities are explained in the context of the attack for the purpose of clarity.

Many books have been written in the area of exploiting vulnerabilities in compiled code. This book does not aim to replicate this area of research. It covers many facets of browser security, but this field is too large for a single book or probably even a single bookshelf!

If you yearn for a greater in-depth understanding of exploiting compiled vulnerabilities, *The Shellcoder's Handbook* (second edition) is a recommended resource. Anyone interested in this topic is encouraged to pursue learning native code exploitation techniques, because it is an interesting and involved area.

Evolutionary Pressures

Web browsers have had one of the most dramatic and exciting evolutions in the Information Technology industry. Today, web browsers use cutting-edge techniques in performance, security, and development. They survive or die on an extremely aggressive battlefield.

Web browsers were once much less sophisticated pieces of software. The first web browser manifestations had a simple purpose—they were the display and followed hyperlinks across the embryonic web. Now they have support for add-ons, plugins, cameras, microphones, and geolocation. Needless to say, this is a long way from where they started.

The landscape of the web browser market share has been far from constant throughout the browser's colorful history. There have been winners and losers, niche and mainstream browsers, and reputations have risen and fallen. Netscape was an early casualty of the browser battles, but its demise gave birth to the Mozilla Organization and ultimately to Firefox. The old-timer, Internet Explorer, once dominant in the browser marketplace and vanquisher of the late Netscape, has been steadily losing ground to the open source browsers and, in recent years, to commercial offerings such as Google's Chrome and Apple's Safari. Yet, due to its continual development and the bedrock of the financial giant Microsoft, it continues to survive and evolve. It is fair to say that this tale of war is far from over.

The battlefield has changed and the browsers have evolved to tackle the new terrain. The important outcome of this arms race is that browser vendors understand that security is important to their users and are continually making browser exploitation more difficult. This has resulted in various advances in defensive technologies.

The following sections describe some of the major browser security features present in today's defense-heavy software.

HTTP Headers

A large chunk of the browser security evolution has occurred in the HTTP headers. Because directives in the scope of the entire request or response are placed in HTTP headers, they provide a natural mechanism for the server to instruct the browser to introduce additional security controls.

Content Security Policy

XSS is discussed in Chapter 2, but is raised briefly here to put the Content Security Policy (CSP) in context. CSP has been designed to mitigate XSS vulnerabilities by defining a distinction between instructions and content.

The CSP HTTP header `Content-Security-Policy` or `X-Content-Security-Policy` is sent from the server to stipulate the locations where scripts can be loaded. It also stipulates the restrictions on those scripts; for example, whether the `eval()` JavaScript function can be used.

Secure Cookie Flag

Historically, cookies were sent over both HTTP and HTTPS without discriminating between the two origins. This can impact the security of a session established with the web browser. A session token securely established between the server and browser over HTTPS can be divulged to an attacker via a standard HTTP request.

This is where the `secure` cookie flag leaps tall buildings in a single bound. The primary purpose of this flag is to instruct the browser to never send the cookie over any unsecured channel. This way, the sensitive session token can remain encased in an encrypted barrier whenever it is in transit.

HttpOnly Cookie Flag

The `HttpOnly` flag is another option that can be applied to cookies, and all modern browsers honor this directive. The `HttpOnly` flag instructs the browser to disallow access to the cookie content from any scripts. This has the security benefit of mitigating cookie theft resulting from XSS with JavaScript (discussed in Chapter 2).

X-Content-Type-Options

Browsers can employ a variety of content-sniffing methods to make a guess at what type of content has been returned from the web server. Based on this, the browser will perform the appropriate action that is mapped to that content type. The `nosniff` directive exists to disable this functionality and force the browser to render the content in accordance to the content-type header.

For example, if the server sends a `nosniff` directive in a response to a `script` tag, the browser will ignore the response unless the MIME type matches `application/javascript` (and a few others). On a site such as Wikipedia (which permits uploads), this could be of particular concern.

The absence of the directive becomes an issue when a specially crafted file is uploaded and then subsequently downloaded. The browser may be tricked into incorrectly interpreting the data MIME type and interpret a JPEG as a script, for example. This has obvious issues when considering the browser security controls; it may be possible for a user to gain control over a browser through a public web application. One way would be by uploading files of a permitted (and seemingly safe) content type, which are subsequently interpreted in another, more dangerous and volatile way.

Strict-Transport-Security

This HTTP header instructs the browser that communication to the website must occur over a valid HTTPS tunnel. It will not be possible for the user to accept any HTTPS errors and proceed over an insecure connection. Instead, the browser will explain the error without allowing the user to continue browsing.

X-Frame-Options

The `X-Frame-Options` HTTP header is used to prevent framing of the page in the web browser. When the browser sees the header, it should ensure that the page sent would not be displayed within an IFrame.

This header was developed to prevent UI redressing attacks, one of which is Clickjacking. This attack consists of framing the victim page in a foreground window that is 100-percent transparent. Users believe they are interacting with the opaque background (attacker) page, but they are, in fact, clicking the invisible foreground (victim) page.

The `X-Frame-Options` HTTP header prevents the successful execution of a subset of UI redressing attacks. These attacks are discussed in depth in Chapter 4.

Reflected XSS Filtering

This is a web browser security feature that attempts to detect, sanitize, and block Reflected XSS (covered in Chapter 2). The web browser attempts to passively discover successful Reflected XSS exploitation. It then attempts to sanitize the scripts delivered in the response and, in most instances, prevents them from executing.

Sandboxing

Sandboxing is an attempted real-world solution to a real-world problem. The base assumption is the browser *will* get compromised and come under the control of the attacker. Never have truer words been spoken! The fundamental (and pragmatic) position is that developers will inevitably write vulnerable code.

Many believe that vulnerable code will inevitably appear somewhere within a software product. Let's face it, even those in the security community who point their fingers at developers are susceptible. The sandbox is a good attempt at addressing this universal problem.

Obviously, the degree to which developers will conform to this premise (that is, write vulnerable code) will vary depending on many complex factors, such as lack of sleep or coffee bean quality. The sandbox is simply a mitigating control. It attempts to encapsulate a high-probability area of browser compromise in a protective wall. It allows for an increased focus on a smaller attack surface. This provides a good risk-versus-reward investment of resources for the browser security team.

Sandboxing is not a new solution; variations have been seen in other areas of computing. For example, Sun used compartmentalization on Trusted Solaris, and FreeBSD used Jails. This restricted access to resources depending on the process permissions.

Browser Sandboxing

A sandbox can be applied at many levels. It could, for example, be applied at the kernel level to separate one user from another user. It could be applied at the hardware level to achieve privilege separation between kernel and user space.

The browser sandbox is the highest-level sandbox possible for a user-space program. It is the barrier between the privileges given to the browser by the operating system, and the privileges of a subprocess running within the browser.

To completely compromise the browser, you will need to take at least two steps. The first one is to find a vulnerability in the browser functionality. The next step is to break through the sandbox. The latter is known as a *sandbox bypass*.

Some browser sandboxing strategies open up every website in separate processes, making it difficult for a malicious website to cause further impacts against other currently visited sites, or to the operating system itself. This sandboxing also applies to plugins and extensions, such as separate processing for PDF rendering.

Sandbox-bypass vulnerabilities are normally of the *compiled code* variety and attempt to completely subvert the functionality of the running process. At this stage the effectiveness of the sandbox is tested: can it prevent the subverted execution path from achieving full process privileges?

IFrame Sandboxing

IFrames can be used as a mechanism to include potentially untrusted content from cross-origin resources, and in some cases untrusted content from same-origin ones. For example, one popular inclusion in websites is Facebook's social media widget.[12] The possibility of an IFrame becoming hostile is not a new idea, and browser vendors have long offered various ways to mitigate the collateral damage from a rogue IFrame.

The HTML5 specification has put forward an IFrame sandboxing proposal that has been embraced by modern browsers. This provides developers a way to employ least privilege. Sandboxed IFrames is a method to include an HTML5 attribute that adds additional restrictions to the inline frame.

These restrictions include preventing the use of forms, stopping script execution, not allowing top-navigation, and trapping it with an origin. These restrictions extend from each parent frame, ensuring that any nested IFrames automatically inherit the restrictions upon creation.

Anti-phishing and Anti-malware

Forging entities online (including e-mails) in an effort to steal personal information such as credentials is traditionally called *phishing*. Numerous organizations have services cataloging known phishing websites, and modern browsers can make use of this information.

The browser checks each site visited against a known list of malicious sites. If it detects that the requested site is actually a phishing site, the browser will take action. This is explored further in Chapter 2.

Similarly, web servers may become infected without their owner's consent, or are created specifically for the purpose of hosting content that may attempt to compromise the browser by exploiting known vulnerabilities. These sites may also encourage the user to manually download and execute software that will bypass the browser's defenses and be launched directly.

Various organizations maintain active blacklists of sites proven to be hosting malicious code, and can be linked directly into the browser to provide real-time protection.[13]

Mixed Content

Websites with *mixed content* vulnerabilities have an origin using the HTTPS scheme and then request content via HTTP. That is, everything that goes in to creating the page is not delivered via HTTPS.

Data not transferred over HTTPS is at risk of being modified and could negate any advantage of employing the encryption of some data. In the instance of a script being transmitted over the unencrypted channel, an attacker could inject instructions into the data stream that compromise the interaction between the web browser and the web application.

Core Security Problems

The evolution of the ever-expanding feature set of browser security controls underpins a bigger and more fundamental picture. Traditional network security used to rely on the deployment and maintenance of external or perimeter defenses, such as firewalls. Over time, these devices have been seen to block all but the essential traffic not only into, but also out of, your organization.

Although the network is getting tighter, businesses still require access to their information, and the increase in the use of web technology (pretty much anything travelling over TCP port 80 or 443) has been growing at an accelerating rate. In fact, firewalls have been so successful at reducing the open floodgates of traffic that all we often have left is a shining beam of HTTP traffic. Good examples of this can be seen in the growth in popularity of SSL VPN technology over traditional IPSEC VPNs.

Arguably, all firewalls have effectively done is reduce network traffic down to two ports: 80 and 443. This transfers extreme reliance to the web browser security model.

The following subsections explore the general picture surrounding browser security and why the contradictory forces at play create a complex playground of attack and defense. We converge on why, in general, the laser beam of web traffic has failed to isolate the network perimeter and instead has created a prism of attack possibilities.

Attack Surface

The meaning of *attack surface* will probably not be new to you. The attack surface is the region of the browser that is vulnerable to influence from untrusted sources. Given this is, in the smallest case, the entire rendering engine, the extent of the problem becomes clear. The web browser has a large and ever-increasing attack surface. There is a vast array of APIs and numerous abstractions to store and recall data.

Conversely, the attack surface of the network at large is by now able to be kept under tight control. Access points and permitted traffic flows are well understood and change control processes can account for alterations. Access to different ports on the firewall, for example, can be trivially verified and restricted via well-known methods.

It is uncommon for a browser vendor to remove functionality from the software. It is more often that the vendors are adding the latest bells and whistles. Like most products, there is rarely a visible reward for reducing capability while backward compatibility is maintained. As the feature set is extended, so is the size of the potential attack surface.

Modern browsers update automatically and silently in the background, sometimes changing the attack surface without the defender's knowledge. In some instances this can be a good thing. However, for a mature and capable security team, this may pose more challenges than advantages.

However, when it comes to the common web browser it is rare to find members of an organization's security team with substantial experience in defending it. Even though this single piece of software is one of the most trusted, it potentially presents the largest attack surface to the Internet.

Rate of Change

Browser security teams may not be working on a time line that aligns with the organization. Often, it is out of the control of the organization to implement browser fixes that might be wanted to bolster the security posture.

Web browser bugs relating to security are often given a lower priority by developers than some in the security community would prefer. With the January 2013 release of Firefox 18.0, Mozilla boasted[14] that one of the fixes was mixed content vulnerability prevention. That is, disabling loading HTTP content when the origin has a scheme of HTTPS. You may be surprised to learn that this bug was first reported in December 2000.[15] This is probably a worst-case example, but it does serve to demonstrate the lag that can occur.

The lack of end-user control over web browser security updates is not dissimilar to other pieces of software. It is also unlikely that an organization would be in a position to halt the use of every browser while waiting for a critical fix. If that assumption holds, then most organizations will be vulnerable to attacks on the browser in the time window between the release of a public exploit and the vendor issuing a fix.

Silent Updating

Silent background updates, while offering a potential avenue for attack, also provide arguably a greater value to users. The necessity to ensure available updates are applied rapidly has driven some developers to implement their own silent mechanisms.

Google for example implemented a silent update feature for its Chrome browser.[16] The user was not given the option to disable the feature, thereby ensuring all updates were applied in a timely manner without user intervention.

One notable example was when silent updating was leveraged by Google to deploy its own PDF rendering engine in Chrome to replace the Adobe Reader software. This ensured every self-updating instance of Chrome was no longer beholden to the update process of this third-party plugin.

Now herein lies the rub. Browsers updating and adding functionality in the background potentially increases the attack surface of every browser if done incorrectly. It also necessitates that any organization's security team outsource a degree of dependency to the browser's developers. When coupled with the fact that areas given to the browser developer's attention may not be aligned with the needs of a given end-user organization, this dependency can be frustrating.

Extensions

Extensions provide a method to augment the browser behavior without using a standalone piece of software. They can influence every page that the browser loads and the inverse—every page can potentially influence them.

Every extension adds a place a hacker can target and thereby increases the attack surface of the browser. In some instances, universal XSS vulnerabilities can even be introduced for that browser. You delve into extensions and their vulnerabilities further in Chapter 7.

Plugins

A plugin is generally a piece of software that can run independently of the browser. Unlike extensions, the browser only runs plugins if the web application includes them in the page via an object tag or, in some cases, the content-type header.

Some of the Internet can't be accessed without the correct plugins and this is why browsers provide the capability to augment their functionality. For example, Java applets are used in some VPN gateways like Juniper.

A lot of the mainstream browser plugins are required for standard business practices, and some of these plugins have a history of containing security vulnerabilities. This means the defender is faced with the decision of using vulnerable software or disabling a subset of business activity.

The majority of plugins don't have a central update mechanism. This means security has to be applied manually in some instances. Obviously, this creates an overhead and complexity to defending an infrastructure.

Plugins tend to receive a lot of negative coverage in the security media. Many of these applications have had substantial vulnerabilities and, in some cases, are proven so insecure it has resulted in security professionals advising organizations to remove them altogether. Operating system vendors have also acted independently, deactivating vulnerable plugins through their own automatic update schemes, indefinitely or until a solution is found.

Plugins can add considerable attack surface. They expose additional functionality and targets for a hacker. You examine plugins in more depth in Chapter 8.

Surrendering Control

The browser requests instructions from arbitrary locations on the Internet. Its primary function is to render content to the screen, and provide a user interface for that content, in precisely the way that the author intended. As a by-product of this core function, it is necessary to surrender a significant degree of control to the web server. The browser must execute the supplied commands or risk failing to render the page properly. On the modern web, it is common for a web application to include numerous resources and scripts from other origins. These too must be executed if the page is to display as intended.

Traditionally these instructions may have been as simple as, "Where should I position this text, and where does this image go?" Modern web applications and browsers, on the other hand, may request, "I'm going to turn on your microphone now, and send this data asynchronously to a server over there."

This type of invasive functionality immediately raises the question of whether all users are guaranteed to be browsing only non-malicious websites. The answer is in almost all circumstances, of course not! The inability to guarantee the sanctity of content sourced from remote locations in real time is the fundamental basis of all browser insecurities and their exploitation.

TCP Protocol Control

It is not common for the server-client model to provide so much flexibility over which port the client communicates on, or which IP addresses the client can use during data exchange.

This functionality can be very useful to an attacker. It means there is almost no restriction to only attacking HTTP protocols or particular systems. Other factors come into play here that set the stage for a whole new class of attacks. You explore these Inter-protocol attacks in Chapter 10.

Encrypted Communication

SSL and TLS can be used to communicate with trusted organizations over the Internet, protecting the integrity and confidentiality of your messages with encryption. Conversely, the exact same technology can also be used to communicate securely with attackers.

The aim of encrypted communication between the browser and the server is to protect the data between those two endpoints. This creates substantial complications for defenders. They do not get the opportunity to spot malicious data. This browser-supported encrypted tunnel works in favor of the attackers as they smuggle in their commands and smuggle out their spoils.

Same Origin Policy

SOP is applied inconsistently across browser technologies and is possibly one of the most confusing concepts. As previously mentioned, the SOP was created in an attempt to isolate resources manifested in a browser, to prevent items from one location from interacting with other, non-related resources sourced from other locations also running in the same browser. It is, essentially, a sandbox.

This particular sandbox is of paramount importance to browser security. Given the browser's prime position at the center stage of network activity, the browser effectively interconnects disparate zones of trust as standard—and is responsible for maintaining the peace. To support the needs of each zone, the autonomous functions that are permitted to interact with the origin are quite extensive. If these functions can breach the SOP, legitimate functions become hostile because they may now traverse security zones.

Understanding the SOP doesn't end with grasping its implementation within the browser alone. SOP implementations are often substantially different between browsers, their versions and even in plugins. Chapter 4 dives very deeply into the SOP in all its incarnations, and offers a plethora of ways in which to bypass this control. These bypasses are available thanks to SOP quirks in Java, Adobe Reader, Silverlight and the various different browser implementations.

Fallacies

A lot of rules of thumb that worked in the past no longer apply in the current global threat landscape. The following fallacies are easy traps to fall into. Unfortunately, a lot of these fallacies continue to be propagated by people who have good intentions.

Robustness Principle Fallacy

The Robustness Principle,[17] which is also known as Postel's Law, instructs programmers to "be conservative in what you do, be *liberal* in what you accept from others." This does not go hand in hand with practical security.

The web browser is extremely liberal with what it will render. This is one of the main reasons that XSS has been so difficult to stamp out. The browser makes development of secure filters and encoders difficult, because the web browser will permit instructions being executed in many ways.

To encourage secure coding practices among developers, the Robustness Principle should be replaced with "be conservative in what you do, be *ultra conservative* in what you accept from others." If this was instilled in the next generation of developers, hackers would have a much more difficult time!

External Security Perimeter Fallacy

A lot of organizations like to abstract their security boundaries to concoct a customized castle-and-moat model. Their defenses, they will reason, have rings of walls to protect their critical assets. The incorrect assumption is that a layered onion–style approach provides the most secure results. Unfortunately, this is not medieval Europe. It is a complex network!

The fundamental problem with that defense pattern is that it assumes attackers enter from the most external layer and, in a *Braveheart* manner, battle their way sequentially through each wall. This notion diverges from reality almost as much as Hollywood films might from the true events of history.

The organization's intranet is a constantly evolving environment with attackers appearing in *Whac-A-Mole* style throughout the infrastructure. The reality is that the web browser is prolific and, in some instances, performs like a portal straight through the external perimeter.

Defense perimeters have therefore been indirectly compromised and cannot defend against attacks ricocheting off the web browser. Defensive resources need to be invested into the Micro Security Perimeter that needs to encompass critical assets. Today's networks must defend against devices changing from ally to foe when least expected.

In the real world, security is a finite resource that should be allocated where it will bolster the defenses of the most valuable assets.

Browser Hacking Methodology

At this point in the chapter, we hope you appreciate the complexity of the challenges facing the browser. Securing the web is no easy task, and arguably a lot of the responsibility for doing so falls upon the browser. It is the first and last line of defense.

In a hypothetical, post-apocalyptic, high-tech world, where every website was compromised and malicious, the ideal browser would still keep your computer safe. We are very far from this security utopia.

It is time to deconstruct the vague notion of browser hacking and turn it into a staged approach that can persist beyond the elimination of present weaknesses and survive redaction. We have defined a method that we hope can maintain relevance regardless of the present security terrain.

This section introduces our methodology and its proposed chronology for hacking the web browser. It is shown in Figure 1-1 and examines a process flow of attack paths and decisions to achieve compromise.

This methodology aims to be effective in directing your browser hacking engagements. The chapters in this book have been organized to map directly to the major phases of the methodology. Each chapter focuses on the practical stages involved and delves into technical specifics. As you master each chapter, your wider understanding of the methodology will increase.

Depending on the target, some paths in the methodology may be trivial because freely available security tools will have automated the process. Other parts will present more of a challenge.

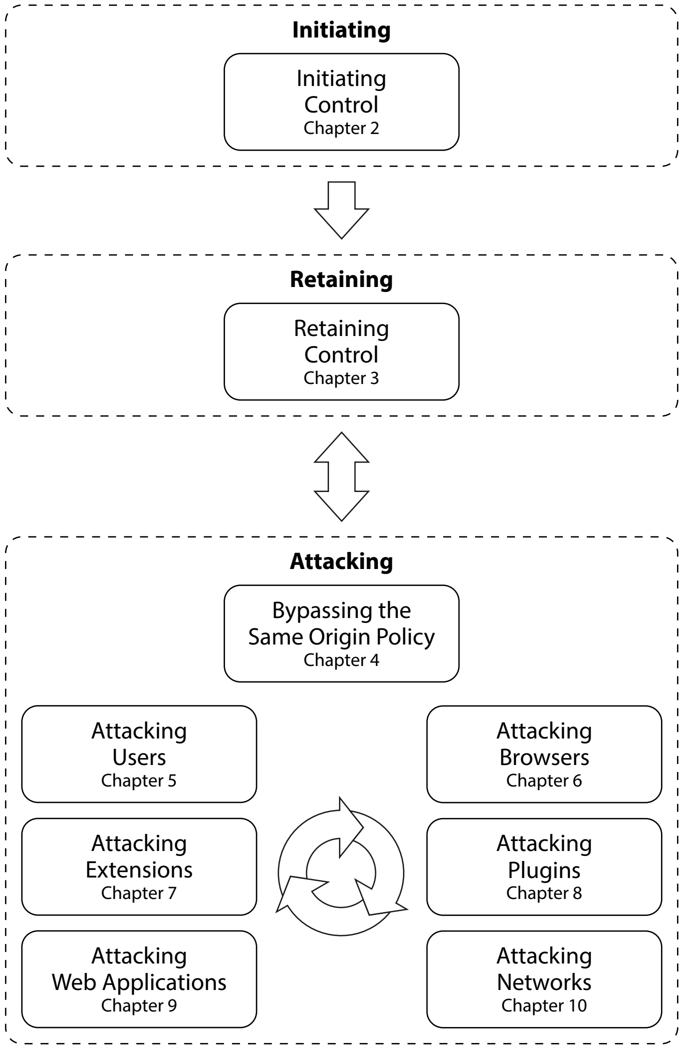
Initiating
Initiating Control Chapter 2
Retaining
Retaining Control Chapter 3
Attacking
Bypassing the Same Origin Policy Chapter 4
Attacking Users Chapter 5
Attacking Browsers Chapter 6
Attacking Extensions Chapter 7
Attacking Plugins Chapter 8
Attacking Web Applications Chapter 9
Attacking Networks Chapter 10

Figure 1-1: Browser Hacker's Handbook methodology

The browser hacking methodology comprises three main sections, and these encapsulate the high-level hacking steps. They are represented as dotted lines surrounding the various phases in the diagram. This grouping of stages provides an overview of the methodology progression starting at Initiating, moving to Retaining, and then to Attacking.

The first encapsulation is Initiating, which is the setup for the entire process. The Retaining encapsulation follows next, and is where the maintenance of your grasp of the browser happens. This is the creation of a beachhead within the target browser or on the device on which the browser resides; it is the initialization of browser compromise.

The real action comes in the next grouping. The Attacking encapsulation contains seven attacking options that are covered in the following sections and more in depth later in the book. During these phases, different facets of the browser will be targeted and exploited. Some of the Attacking techniques covered can result in additional Initiating phases on other browser instances, resulting in cyclical expansion of attack and extent of compromise.

Initiating

The Initiating encapsulation has one phase within it. This seemingly innocuous phase is the first and most important step in hacking the web browser. Without this phase, no other attacks are possible and the target browser is out of range.

Initiating Control

Every attack sequence permutation starts by running instructions within the web browser. For that to happen, the browser must encounter (and execute) instructions under your control.

This is the topic for Chapter 2, which discusses methods by which you can trick, entice, fool or force a browser into both encountering and, most importantly, executing some arbitrary code.

Retaining

Now that you have successfully attacked, how do you increase your control over the target? You need to maintain control of the browser in a manner that facilitates further launching of attacks.

Retaining Control

Let's consider the genie and the three wishes fantasy; a genie appears and grants three wishes. The cunning recipient may well attempt to extend his or her good fortune by wishing for more wishes with the last wish—thus stress testing the genie's exclusion policy!

Well, in the case of maintaining communication with a compromised web browser, the initial code instructs the browser to repeatedly ask you for your next wish. You unleash the genie in the hooking stage and enslave the browser from that point on to keep granting wishes.

Just as the genie may disappear in a puff of smoke, this state of affairs may not long continue. The position of endless wishes is contingent upon the actions the user subsequently takes. He might close the tab in which the initial exploitation took place, or use it to browse to another site, thus terminating the JavaScript payload and therefore the communication channel.

Before getting carried away launching additional attacks, it is wise to be patient and instead consider methods of increasing influence over the browser. In this phase of the methodology, you are attempting to reduce the potential of losing control of the browser by the user surfing away from the origin or even closing down the web browser.

You can achieve this in several ways at different levels of persistence. It is important to be patient and leverage this phase as completely as possible before proceeding to the next, because the longer you can keep a browser hooked, the more of the attack surface you can interrogate and the more controlled your attack will be.

It is also worth noting that sometimes, during the subsequent Attacking encapsulation, successful attacks will reveal methods to increase the strength of the beachhead, improving the degree of control. For this reason there is a bi-directional arrow between the two phases in the methodology. Experience will help determine where efforts into enhancing the resilience of the control channel should supersede attacking efforts, and where attacking efforts have fed back into the control channel's flexibility and persistence.

Attacking

At this stage in the methodology, you leverage the control gained over the browser and explore the attack possibilities from the present position. Attacks can take many forms, ranging from "local" attacks against the browser instance or the operating system on which it resides, to attacks on remote disparate systems in arbitrary locations.

The observant reader will have noticed that Bypassing the Same Origin Policy sits atop and apart from the other elements of the Attacking encapsulation. Why is that so? Because it fits within all the attacking steps. It is a security control that will be circumvented or utilized during the other exploitation phases.

Something else that should become quickly evident is the cyclical arrow at the center of the Attacking encapsulation. Far from being just revolutionary, it is likely that any one of the attacking phases will reveal details that could lead to a successful attack in any one of the other phases. From this position, you will probably jump between the different categories, depending on which one is most likely to produce the most efficient rewards.

Seven core classes of attacks are defined that can be launched from the web browser. Various factors come into play when deciding what path should be taken here. The main influences will be the scope of the engagement, the desired target, and the capabilities of the hooked browser.

Bypassing the Same Origin Policy

The SOP can be thought of as the primary sandbox. If you are able to bypass it, you have created a successful attack automatically by being able to access another origin previously occluded by the browser. By bypassing the SOP, you can now attack that newly revealed origin with any other applicable technique in a potential chain reaction.

The varied interpretations of the SOP are explored in depth in Chapter 4. When you have a bypass there are many attacks you can conduct without interference. In this chapter you will examine some of the inconsistences and the ways to exploit this lapse in the browser's most fundamental security control.

Attacking Users

The first attacking option presented in the browser hacker methodology is Attacking Users, which is discussed in Chapter 5. This covers attacks involving the browser users and their potentially implicit trust of the attacker-controlled environment.

Using the leverage gained over the browser and your ability to control the rendered page, you are able to create an environment that may encourage the user to enter compromising information so it can be captured and used.

You can trick the user into unknowingly granting permission for an otherwise secured event to occur, such as running an arbitrary program or granting access to local resources. You can create hidden dialog boxes and transparent frames or control mouse events to help in this aim, belying the true function of the user interface and presenting a false impression to the user.

Attacking Browsers

The category for Attacking Browsers includes direct attacks on the core browser itself. You sink your teeth into this in Chapter 6, where you explore a range of areas from fingerprinting vendors to full exploitation.

The web browser is an attack surface mammoth. There is a vast array of APIs and abstractions to store and recall data. It is no wonder that web browsers have been plagued with vulnerabilities in one form or another for years. What is more surprising is that the developers of the web browser get it right as many times as they do.

Attacking Extensions

If you fail to successfully attack the core browser, the doorway is by no means shut. You can also attack its (probably numerous) optionally installed extras.

This is covered in Chapter 7 and has been classified in the methodology as Attacking Extensions. In this chapter you examine the differences between variants and specific extension implementations.

You will explore various classes of extension vulnerabilities. Extension vulnerabilities can be used to leverage functions resident therein to conduct cross-origin requests or even execute operating system commands.

Attacking Plugins

One of the most traditionally vulnerable areas of the web browser are the plugins. A plugin is notably different than an extension in that they are third-party components, which are initialized solely at the discretion of the served web page (as opposed to being persistently incorporated into the browser).

The Attacking Plugins category in the methodology is covered in Chapter 8. This includes attacks on pervasive plugins like Java and Flash. You explore how to discover what plugins are installed, reveal exploitable historical weaknesses discovered by various researchers in the field, and learn how certain security features designed to protect against plugin abuse can be bypassed.

Attacking Web Applications

The browser is built to use the web so it should be no surprise that the phase Attacking Web Applications exists in the methodology. This area includes attacking web applications using the standard functionality of the web browser. Chapter 9 delves into leveraging standard browser functionality to exploit web applications.

Imagine the wealth of Intranet-accessible applications commonly accessible only from within an organization's perimeter. What if an external website in another tab can browse those websites? You will learn the assumption that intranet sites are protected from external attack by the firewall is demonstrably false.

Attacking Networks

You may not have noticed your web browser connecting to non-standard ports, but this scenario is actually quite common. Applications often install a web server on an arbitrary port number, and some websites on the Internet even publish their content on ports other than 80 or 443.

What if your browser wasn't connecting to a web server at all? What if it was connecting to a service that fulfills a completely different purpose and uses a completely different protocol? This would not violate the SOP and in most cases, would be valid from the perspective of the browser's security controls. Repurposing these browser behaviors allows for sophisticated attack scenarios.

The Attacking Networks phase jumps to targeting the lower layers of the OSI model. In Chapter 10, you discover that all the techniques can equally apply to attacking any TCP/IP network.

Summary

Arguably, the web browser is the most important piece of software of this decade. Software vendors are rarely developing custom client software for their applications. They are more frequently developing application user interfaces with web technology; not just the traditional online web applications, but local and intranet applications too. The web browser is dominating the position of client in the server-client model.

The web browser is already exerting power in almost all networks, and even if the desire were to remove it from any organization, it is unlikely this could be achieved. An organization has no choice but to have web browsers in its network.

Hackers are typically attacking from a perspective of pretending to be a non-malicious web server sending valid communication to the web browser. In most cases, the web browser will not know that it is communicating with a rogue web server. The browser executes all instructions sent by the rogue web server in the allegedly safe haven inside the firewall perimeter.

In the remainder of this book, you master the methodology and learn techniques on how to exploit the web browser and the devices it can access.

Questions

1. What function does the DOM have within the web browser?
2. Why is having a secure browser important to a holistic security approach?
3. Name some of the differences between JavaScript and VBScript.
4. Name three ways the server can reduce the security of a web browser.
5. What is the web browser's attack surface?
6. Describe sandboxing.
7. When a browser is using HTTPS to communicate, can a proxy view the communication?
8. Name three security-related HTTP headers.
9. Why is the Robustness Principle not a security professional's friend?
10. Which scripting language is available in Internet Explorer and not the other modern browsers?

For answers to the questions please refer to the book's website at `https://browserhacker.com/answers` or the Wiley website at: `www.wiley.com/go/browserhackershandbook`.

Notes

1. Microsoft. (2013). *Security Intelligence Report (SIR) Vol. 15*. Retrieved December 12, 2013 from `http://www.microsoft.com/security/sir/default.aspx`
2. Antone Gonsalves. (2013). *Browsers pose the greatest threat to enterprise, Microsoft reports*. Retrieved December 12, 2013 from `http://www.networkworld.com/news/2013/041913-browsers-pose-the-greatest-threat-268914.html`
3. Wikipedia. (2013). *Client-server model*. Retrieved December 12, 2013 from `http://en.wikipedia.org/wiki/Client-server_model`
4. Wikipedia. (2013). *Request-response*. Retrieved December 12, 2013 from `http://en.wikipedia.org/wiki/Request-response`
5. Wikipedia. (2013). *Web browser engine*. Retrieved December 15, 2013 from `http://en.wikipedia.org/wiki/Layout_engine`
6. Wikipedia. (2013). *List of layout engines*. Retrieved December 15, 2013 from `http://en.wikipedia.org/wiki/List_of_web_browser_engines`
7. Wikipedia. (2013). *WebKit*. Retrieved December 15, 2013 from `http://en.wikipedia.org/wiki/WebKit`
8. WebKit Open Source Project. (2013). *The WebKit Open Source Project - WebKit Project Goals*. Retrieved December 15, 2013 from `http://www.webkit.org/projects/goals.html`
9. Bruce Lawson. (2013). *300 million users and move to WebKit*. Retrieved December 15, 2013 from `http://my.opera.com/ODIN/blog/300-million-users-and-move-to-webkit`
10. Doug DePerry. (2012). *HTML5 Security. The Modern Web Browser Perspective*. Retrieved December 15, 2013 from `https://www.isecpartners.com/media/18610/html5modernwebbrowserperspectivefinal.pdf`
11. Alex Russell. (2006). *Comet: Low Latency Data for the Browser*. Retrieved March 8, 2013 from `http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/`
12. Facebook. (2013). *Getting Started for Websites - Facebook developers*. Retrieved December 15, 2013 from `https://developers.facebook.com/docs/guides/web/`
13. StopBadware. (2013). *Firefox Website Warning | StopBadware*. Retrieved December 15, 2013 from `https://www.stopbadware.org/firefox`
14. Mozilla. (2013). *Firefox Notes - Desktop*. Retrieved December 15, 2013 from `http://www.mozilla.org/en-US/firefox/18.0/releasenotes/`

15. Mozilla. (2013). *62178 - implement mechanism to prevent sending insecure requests from a secure context*. Retrieved December 15, 2013 from `https://bugzilla.mozilla.org/show_bug.cgi?id=62178`

16. Thomas Duebendorfer and Stefan Frei. (2009). *Why Silent Updates Boost Security*. Retrieved December 15, 2013 from `http://research.google.com/pubs/pub35246.html`

17. Andrew Gregory. (2008). *Andrew Gregory - The Myth of the Robustness Principle*. Retrieved December 15, 2013 from `http://my.opera.com/AndrewGregoryScss/blog/2008/05/27/the-myth-of-the-robustness-principlehttp://my.opera.com/Andrew%20Gregory/blog/2008/05/27/the-myth-of-the-robustness-principle`