Your Browser Wears No Clothes

Why Fully Patched Browsers Remain Vulnerable

By Michael Sutton

Shifts in technology and attack patterns are changing the rules such that it is now common for fully secured machines to become compromised.

As users of technology, we have been taught that the Internet is not always a safe place but that we can protect ourselves by patching and hardening our systems. While patch management and system hardening have long been the basics for enterprise security, shifts in technology and attack patterns are changing the rules. Today, it is not just possible, but common for a user with a fully secured machine to become compromised. At times, this occurs due to increasingly sophisticated social engineering attacks or newly discovered (so called zero-day) vulnerabilities. However, it is increasingly resulting from exploitation, which does not target a specific vulnerability on an individual platform, but instead is *abusing the functionality and structure of the Internet* itself. This fundamental shift to "naked browser attacks" changes everything. Just as attackers have continually adjusted their tactics, enterprises must adapt their approach to security if they wish to stay a step ahead in the never-ending arms race of Web security.

We are now firmly entrenched in the era of Web applications. It is no longer desirable but expected that the majority of enterprise development be architected as Web applications, whether they are to be used internally or externally. This shift has brought a degree of uniformity to the IT landscape. No matter what operating system you choose and regardless of the hardware that you select – mobile or otherwise – it will have a Web browser and that browser will adhere to at least a basic set of standards. By 2009, a JavaScript engine has become the norm on even mobile phone browsers and the majority of platforms can handle at least the most popular Rich Internet Application technologies such as Adobe Flash. This subtle and voluntary standardization has not only made possible Web 2.0 technologies such as AJAX but also created a broad attack base for those looking to do harm. If attacks can be leveraged that abuse JavaScript, for example, as opposed to a specific version of Internet Explorer, the potential population for attack has then risen from X% of Internet users to virtually 100%.

Attackers once viewed browsers as targets for attack. Now, browsers are becoming facilitators of attacks. Browsers are simply a door which permits access to the data that the attacker is after. The difference here is that a vulnerability does not have to be identified and exploited on the browser itself. Today, many attacks work cross-browser and cross-platform because they do not target the browser; they target functionality that is the same regardless of the browser platform. The Web was designed to be open – not secure. This fact was not lost on attackers and they are spending much of their time bending the rules of the Web to work in their favor. This has led to a surge in attacks which succeed against fully patched machines. We deem such attacks to be naked browser attacks as no patch to protect end users is forthcoming. In this new world order, we must revisit our approach to Web security if we are to fight back.

History

The attack cycle used to be simple – attackers would uncover a vulnerability within an application or operating system, exploit it, and continue to do so until the appropriate vendor released a patch to address the problem. This is, of course, an over simplification of the multitude of variables that could be involved in any specific case, but it does capture the basis of what has driven much of the security industry for some time. One encouraging evolution has been the shrinking window of time during which an attacker could take advantage of a given vulnerability. A decade ago it was not uncommon for enterprises to spend weeks or even months conducting regression testing before gaining the comfort necessary to apply vendor patches. This allowed for a substantial period of time during which public knowledge of a vulnerability was available and millions of machines remained open to attack. Fortunately, as enterprises have better understood the risks of exposure and vendors have improved processes for disseminating patches and communicating the risks associated with individual cases, the window of opportunity for attack

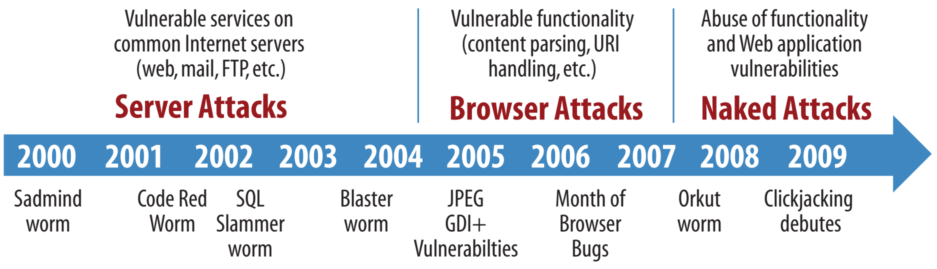


Figure 1 - Evolution of Attacks

has now shrunk to days and even hours following the release of public knowledge for a given vulnerability.

Figure 1 illustrates three distinct eras that have emerged for attackers over the past decade. Up until approximately 2004, many attacks focused on vulnerable Internet-facing services such as popular Web, mail and FTP servers. Over time, critical vulnerabilities in such services were exploited by attackers, which resulted in fast spreading worms. We then moved from server attacks to browser attacks. As servers became increasingly locked down, attackers shifted their focus to vulnerabilities in Web browsers. A plethora of vulnerabilities in all browsers led to attacks on end users. While such attacks generally required a social engineering component to convince a user to view a page or click on a link, these challenges were typically minimal. Today, we are entering an era of naked browser attacks in which there is no specific vulnerability in the browser itself, yet the attacks target end users.

Naked browser attacks

Naked browser attacks against secured browsers succeed either because they abuse trust established between the browser and a vulnerable Web application or simply because they abuse functionality of the Web itself, using it in an unintended manner. A variety of attacks fall into these categories, and it will not be possible to go into detail on all of them. We will instead introduce the concept by discussing cross-site scripting (XSS), which abuses browser/server trust, and clickjacking, which leverages intended functionality in an unintended way.

Cross-site scripting

XSS remains one of the most prevalent attacks that we face on the Web today despite having had a relatively high profile over the past several years. It has been a fixture in the OWASP Top Ten[1] list of common Web application vulnerabilities since the list was introduced in 2004. Additionally, the December 2008 WhiteHat Website Security Statistics Report[2] indicated that 67% of websites are likely to have XSS flaws. This is a truly frightening statistic that leaves Web users at risk each and every time they browse the Internet.

XSS illustrates one of the fundamental changes in security brought about by the interconnected nature of the Web. With XSS, the vulnerability which is abused resides not within a user's browser, but instead in a third-party Web application which the user accesses. Despite this fact, the user is the victim of the attack due to the fact that his browser responds to injected JavaScript as it should, by interpreting the code. In this attack, the browser has no way of distinguishing between user-supplied content which the user intended to include in a request and content which may have been injected through an XSS attack.

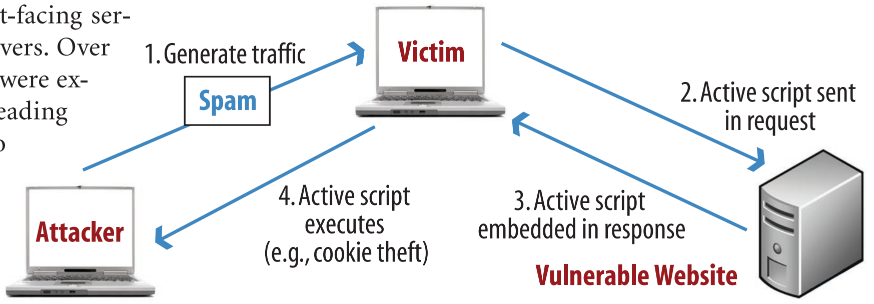

1. Generate traffic
Spam
Victim
2. Active script sent in request
Attacker
4. Active script executes (e.g., cookie theft)
3. Active script embedded in response
Vulnerable Website

Figure 2 - Typical XSS Attack Scenario

Typical attack

Figure 2 details a typical XSS attack scenario. The following walk-thru details why XSS succeeds without requiring a browser vulnerability.

1. **Generate traffic** – XSS requires that a user send a specially crafted request to a vulnerable Web server. The request contains embedded active content (usually JavaScript), which is designed to perform an action of the attacker's choosing. Often the script will attempt to send the user's cookie contents for the targeted website to the attacker. Sending spam email is a common way to get victims to send the predefined request. Typically, the link is embedded within an HTML formatted email message, which includes a message to entice the victim to click on the link.
2. **Active script sent in request** – Should the user click on the link in the spam email message, he will be sending a request to the vulnerable Web server. However, rather than just a simple request for the URL of a Web page, the request will also include the injected JavaScript, either as parameters in the URL itself (GET request) or within the body of the request (POST request).
3. **Active script embedded in response** – The vulnerable Web page includes functionality which will accept user supplied input and include it in the dynamically generated page returned to the user. Such behavior is fine, so long as user supplied input is appropriately sanitized to ensure that the content received was the content expected. The absence of such controls is what permits XSS attacks. For example, we could assume that the page allowed a user to

1 http://www.owasp.org/index.php/OWASP_Top_Ten_Project.

2 http://www.whitehatsec.com/home/resource/stats.html.

input his name so that the resulting page displayed a "hello [name]" message. While a string was expected, without proper sanitization, that same input vector could be used to inject malicious JavaScript.

4. **Active script executes** – When the browser receives the response, the page content will include the injected malicious JavaScript, which will be interpreted and executed by the browser.

Impact

While this simple attack scenario involves a single attacker and victim, XSS is commonly used in more complex attack scenarios. In January 2008, it was revealed that attackers had used an XSS vulnerability on the login page of Banca Fideuram, an Italian bank to inject a fake login form within an IFRAME.[3] The attack would send a user's authentication credentials to an attacker-controlled server and was a particularly dangerous attack as it was hosted on a trusted, SSL-protected webpage. XSS attacks are quickly evolving and are no longer static in nature. XSS worms have now started to appear on popular social networking sites such as Orkut[4] and MySpace.[5]

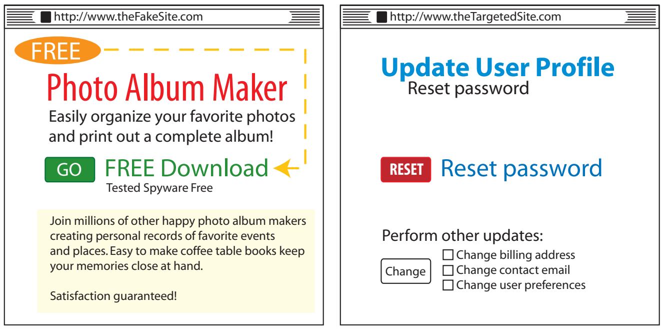
It is important to remember that XSS attacks will succeed regardless of whether or not users have applied all outstanding security patches. XSS succeeds because browsers are designed to interpret JavaScript. With XSS, the attacker is abusing an input validation vulnerability on a Web application, yet the user viewing the page becomes the victim. In addition to this, the old adage that users need to surf only "reputable" pages to remain safe, simply does not apply. Virtually all major websites have experienced XSS vulnerabilities, and given the social engineering component of an XSS attack, sites with high volumes of traffic, tend to be included in the most successful attacks.

Clickjacking

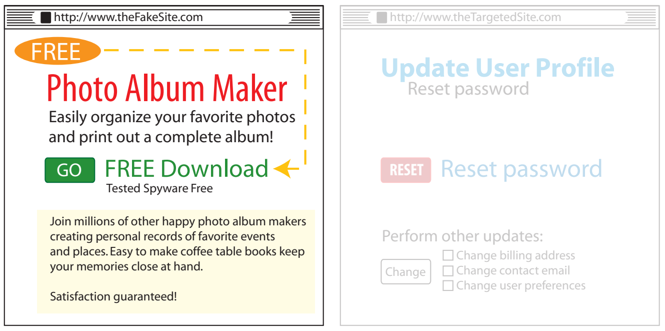
Clickjacking leapt into the media spotlight this past summer when researchers were asked by Adobe to pull a talk on the subject just days before it was to be delivered. This attack layers good content over bad, sprinkled with a little social engineering in order to trick a user into performing an action that he did not intend to execute.

In Figure 3 we can see an administrative interface permitting a password reset being layered together with a fake site, which obfuscates everything. The content of the fake page is designed to hide what is really going on and convince an unsuspecting user to, in this case, reset his password, not download free software.

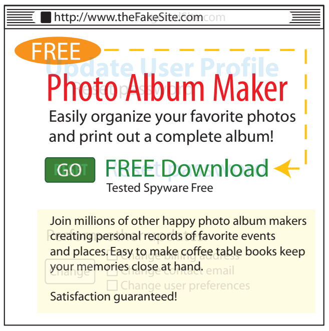
Clickjacking requires the following three components:


http://www.theFakeSite.com
FREE
Photo Album Maker
Easily organize your favorite photos and print out a complete album!
GO
FREE Download
Tested Spyware Free
Join millions of other happy photo album makers creating personal records of favorite events and places. Easy to make coffee table books keep your memories close at hand.
Satisfaction guaranteed!
http://www.theTargetedSite.com
Update User Profile
Reset password
RESET
Reset password
Perform other updates:
Change
Change billing address
Change contact email
Change user preferences

1. **Embedded content**: fake content (left) matches target (right) reset password layout, aligning the Go and Reset buttons.


http://www.theFakeSite.com
FREE
Photo Album Maker
Easily organize your favorite photos and print out a complete album!
GO
FREE Download
Tested Spyware Free
Join millions of other happy photo album makers creating personal records of favorite events and places. Easy to make coffee table books keep your memories close at hand.
Satisfaction guaranteed!
http://www.theTargetedSite.com
Update User Profile
Reset password
RESET
Reset password
Perform other updates:
Change
Change billing address
Change contact email
Change user preferences

2. **Obfuscation**: target opacity set to zero. (25% opacity here for demonstration purpose.)


http://www.theFakeSite.com
FREE
Update User Profile
Photo Album Maker
Easily organize your favorite photos and print out a complete album!
GO
FREE Download
Tested Spyware Free
Join millions of other happy photo album makers creating personal records of favorite events and places. Easy to make coffee table books keep your memories close at hand.
Satisfaction guaranteed!

3. **Layering**: target content exactly positioned with fake content. Target z-index property set to place it above the fake content. Opacity of target set to zero, making target content invisible while keeping Reset button active. (15% opacity for demonstration.)

Figure 3 – Clickjacking

1. **Embedded Content** – The targeted action (e.g., password reset), which is on a page not controlled by the attacker, is embedded within a page controlled by the attacker. This is typically accomplished by using an IFRAME on the attacker-controlled page. This is why "frame busting" code, which prevents content from being displayed in an IFRAME, is commonly recommended as a server-side defense against Clickjacking.
2. **Obfuscation** – The third-party content, including the password reset button, despite being on the page will not be visible as it has its opacity value set to zero. Opacity is used to adjust the transparency of an object.
3. **Layering** – Attacker-controlled content is actually layered below the third-party content and absolutely positioned to ensure that the two buttons lineup. However, the *Go* button for downloading software is visible instead of the *OK* button for the password reset due to the opacity settings (see Obfuscation). Layering is accomplished by leveraging z-index properties, which set the depth value of webpage elements. In this case, the third party content would have

3 http://news.netcraft.com/archives/2008/01/08/italian_banks_xss_opportunity_seized_by_fraudsters.html.

4 http://www.washingtonpost.com/wp-dyn/content/article/2007/12/19/AR2007121900781_pf.html.

5 http://en.wikipedia.org/wiki/Samy_(XSS).

a higher z-index value than the attacker controlled content. This way, although the victim sees the Go button, he is actually clicking the OK button.

Impact

Clickjacking is an enabler for social engineering attacks. As with XSS, a user will not fall victim simply by viewing a malicious webpage. Instead, he must click on a link to trigger the attack. By combining a variety of legitimate HTML formatting techniques, clickjacking facilitates the necessary social engineering by making it appear to the end user that he is clicking on a link other than that which the browser interacts with. Once an attacker can influence the mouse clicks made by a user, the potential attacks that can be conducted are virtually limitless. Having a user unknowingly upload data to an attacker-controlled location, for example, could compromise privacy. Authentication could be bypassed but adding a rogue user account or lowering predefined security settings. Complete system compromise would also be an option should a user's mouse clicks download a malicious binary. In short, the potential for damage in a successful clickjacking attack is limited only to the imagination of the attacker.

Clickjacking works because IFRAMEs and properties such as z-index values and opacity are standards that are respected by most common browsers. Individually, they are powerful tools, which allow for the design of some truly impressive Web applications. However, when combined and used by a malicious attacker, they can be made into a viable attack vector. Once again, we are seeing attacks, which do not leverage individual vulnerabilities. Instead, they abuse intended functionality by using it in ways other than that which it was intended for.

Other Attacks

XSS and clickjacking are certainly not the only client-side attacks that do not rely on client-side vulnerabilities. They are, however, two of the better known examples of such attacks which is why they have been highlighted. There are numerous attacks which have similar characteristics. Attacks such as cross-site request forgery, content spoofing, URL redirection, HTTP response splitting, etc., all have elements of naked browser attacks. Moreover, many of these attacks represent emerging issues, which are on the rise and just starting to be seen in widespread attacks.

Challenges

You cannot stop what you do not know about. If an attack blends in with legitimate Web traffic, it will always be harder to detect. A browser exploit lends itself to signature-based detection, as an attack generally requires that anomalous traffic be sent. Take for example a buffer overflow in a Web browser. In general, Web content will need to be created which includes data to trigger the attack, shellcode to execute once control has been gained, and some data for padding to ensure that everything lands in the right place. None of this is standard content on a Web page, so we can look for it. Clickjacking by contrast would be harder to detect using signatures. None of the components of clickjacking are nefarious individually. All are legitimate properties available to Web application developers. Therefore, they will commonly be seen on a variety of webpages. It is simply the combination of a variety of legitimate attributes, which makes an attack possible. This fact alone means that an elusive silver bullet to prevent such attacks is not likely to be found.

Defense in depth

Defending against attacks, which succeed regardless of your diligence patching and hardening browsers, is an unnerving thought. We have been trained to tighten patch management procedures as a first line of defense and here are increasing volumes of attacks that bypass that entire process. Moreover, naked browser attacks typically involve elements of social engineering, and it is difficult, if not impossible, to prevent an attack which involves an employee serving as an unknowing accomplice.

Look at virtually any text discussing how to defend against attacks such as XSS or CSRF and the content will discuss how to secure the Web application, not how to protect the browser affected by the attack. We have to date, focused the majority of our security capital on defending servers, not browsers. However, typical enterprises have hundreds of browsers for every server, and the majority of browsers reside on laptops that leave the confines of the enterprise on a regular basis. Moreover, individuals that have limited security knowledge at best operate those browsers. When looking at enterprise security from that perspective, it is easy to see why we need to shift our priorities.

Existing solutions

Browser patches are not available to protect against these attacks because the browsers themselves are not vulnerable. Rather, they are behaving as intended. That said, there are certain client-side applications that can aid in protecting against them. NoScript, for example, is an excellent extension for Firefox and other Mozilla-based browsers, which permits granular control over the execution of active content such as JavaScript, Java, Flash, and other plugins. It also includes specific controls to identify and block XSS and clickjacking attacks. Administrators should, however, be cautioned that NoScript is designed for a more sophisticated user and many of the options may be confusing to an average employee. While some administrators tout disabling script engines altogether within browsers, this is no longer a viable alternative, given the heavy reliance on JavaScript by modern Web applications.

In the long run, it is hopeful that browser vendors will also begin to expand security functionality to combat against attacks despite the fact that the vulnerabilities leverage weaknesses in Web applications as opposed to the browsers themselves. Microsoft will be stepping up to the plate with the release of Internet Explorer 8, which will include functionality to detect reflected XSS attacks.[6]

Intrusion detection/prevention (IDS/IPS) systems often have signatures to detect attacks such as XSS, but they tend to identify exploitation of specific popular Web applications. As mentioned, signature-based detection of the attacks discussed in this paper is not trivial as the actual attacks can take many forms. Signatures that are too specific will miss the attacks, and those that are too generic will result in high false positive rates.

Defending against naked browser attacks

Not surprisingly, there is no silver bullet to protect against naked browser attacks. Patches are not available to address the weaknesses that permit such attacks and with plenty of finger pointing going on to place blame elsewhere, quick fixes will not be forthcoming. With that in mind, it is important that enterprises implement a variety of detective and preventive controls to combat naked browser attacks.

Monitor

Although detective in nature, properly monitoring and logging activity on the network can ensure that naked browser attacks can be isolated and followed up on when they do occur. Logs should be consolidated and not just maintained separately at each individual Web gateway, so that incidents can be correlated across physical locations. Analyze Web logs for anomalous traffic patterns. This could include large spikes in traffic to a particular page as attackers are for example herding users to a particular location to be attacked. Sudden drops in expected traffic volume could also raise suspicion as infected machines may be blocked from going to particular sites. This often happens, for example, in order to prevent the downloading of new anti-virus signatures, which could identify an infected machine. Monitoring is not however sufficient on its own. Someone must own the process to ensure that reports are produced, analyzed, and escalated when necessary.

Manage

A common philosophy in security is that users should have only the appropriate level of access necessary in order to do their job. Why is it then that when it comes to Web access, enterprises typically let users do anything, with the possible exception of blocking objectionable content through URL filtering? Web applications have been given that name for a reason – they are applications and as such we can and should restrict access based on functionality, not just destination. For example, while it may be fine for users to view content on Facebook, perhaps you want to restrict some or all users from uploading content in order to protect against data leakage. Look for solutions that allow you to control not just where users are going, but also what they are doing.

Merge

Various commercial and free data feeds (e.g., Phishtank, Google Safe Browsing, OpenDNS, etc.) exist, which identify potentially malicious content. Such feeds can be incorporated into Web filtering solutions to block access to sites that may be involved in browser-based attacks such as phishing scams or botnet attacks. When leveraging such content it is important to also regularly review metrics to ensure that the lists are adding value and not creating unnecessary levels of false positives.

Educate

User education should not be overlooked. While diligent users will never replace technical controls, ensure that users have the knowledge to not just avoid attacks but also to escalate issues when needed. When establishing programs, ensure that education is delivered on a continual basis and in a variety of formats. People learn in different ways, but repetition is essential if the knowledge is to be retained.

Conclusion

Attackers once focused their efforts on targeting corporate servers, looking for gaping holes that would give them the keys to the kingdom. As enterprise servers became better locked down, corporate desktops became a target, especially Web browsers which have had a questionable security track record at best. Today, many of the attacks targeting browsers are naked attacks, requiring no browser vulnerabilities whatsoever. The interconnected reality of the Web ensures that risk is not isolated between Web browser and server.

Some of the attacks discussed in this paper expose user data due to vulnerabilities on the Web applications that they have been exposed to. Other attacks simply abuse intended functionality in an unintended way. Regardless, an increasing number of attacks on Web browsers will succeed even when the browser in question is fully patched and hardened. As a result, enterprises must take a step back and revisit how they view Web security. Patch management is no longer the silver bullet that we had once hoped for.

About the Author

Michael Sutton has spent more than a decade in the security industry conducting leading-edge research, building teams of world-class researchers and educating others on a variety of security topics. As VP of Security Research, Michael heads Zscaler Labs, the research and development arm of the company. Michael is a frequent speaker at major information security conferences; he is regularly quoted by the media on various information security topics, has authored numerous articles, and is the co-author of Fuzzing: Brute Force Vulnerabilities. He may be reached at michael.sutton@zscaler.com.

6 http://msdn.microsoft.com/en-us/library/cc994337(VS.85).aspx.