A Solution for the Automated Detection of Clickjacking Attacks

Marco Balduzzi
Institute Eurecom
Sophia-Antipolis
balduzzi@eurecom.fr

Manuel Egele
Technical University Vienna
pizzaman@seclab.tuwien.ac.at

Engin Kirda
Institute Eurecom
Sophia-Antipolis
kirda@eurecom.fr

Davide Balzarotti
Institute Eurecom
Sophia-Antipolis
balzarotti@eurecom.fr

Christopher Kruegel
University of California
Santa Barbara
chris@cs.ucsb.edu

ABSTRACT

Clickjacking is a web-based attack that has recently received a wide media coverage. In a clickjacking attack, a malicious page is constructed such that it tricks victims into clicking on an element of a different page that is only barely (or not at all) visible. By stealing the victim's clicks, an attacker could force the user to perform an unintended action that is advantageous for the attacker (e.g., initiate an online money transaction). Although clickjacking has been the subject of many discussions and alarming reports, it is currently unclear to what extent clickjacking is being used by attackers in the wild, and how significant the attack is for the security of Internet users.

In this paper, we propose a novel solution for the automated and efficient detection of clickjacking attacks. We describe the system that we designed, implemented and deployed to analyze over a million unique web pages. The experiments show that our approach is feasible in practice. Also, the empirical study that we conducted on a large number of popular websites suggests that clickjacking has not yet been largely adopted by attackers on the Internet.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security, Design, Experimentation

Keywords

Clickjacking, Web Security, *ClickIDS*, HTML `IFRAME`, CSS, Javascript, Browser Plug-In

ASIACCS'10 April 13–16, 2010, Beijing, China.

1. INTRODUCTION

Web applications have evolved from simple collections of static HTML documents to complex, full-fledged applications containing hundreds of dynamically generated pages. The combined use of client and server-side scripting allows developers to provide highly sophisticated user interfaces with the look-and-feel and functionalities that were previously only reserved to traditional desktop applications. At the same time, many web applications have evolved into mesh-ups and aggregation sites that are dynamically constructed by combining together content and functionalities provided by different sources. This rapid evolution, combined with the increasing amount of sensitive information that is now accessible through the web, has brought a corresponding increase in the number and sophistication of web-based attacks and vulnerabilities.

The *same origin policy*, first introduced in Netscape Navigator 2.0 [31], is still the keystone around which the entire security of cross-domain applications is built. The main idea is to implement a set of mechanisms in the browser that enforce a strict separation between different sources. This separation is achieved by preventing the interaction between pages that are from different origins (where the origin of a page is usually defined as a combination of the domain name, the application layer protocol, and the TCP port number). The same origin policy, hence, can guarantee that cookies and JavaScript code from different websites can safely co-exist in the user's browser.

Unfortunately, attackers, often driven by a flourishing underground economy, are constantly looking for exceptions, browser bugs, or corner cases to circumvent the same origin checks with the aim of stealing or modifying sensitive user information. One of these techniques, previously known as *UI Redress* [40], has recently gained increasing attention under the new, appealing name of *clickjacking*. Since Robert Hansen and Jeremiah Grossman announced a talk on the topic at OWASP AppSec 2008 [20], there has been a flood of news, discussions, and demonstrations on clickjacking.

The idea behind a clickjacking attack is simple: A malicious page is constructed such that it tricks users into clicking on an element of a different page that is only barely, or not at all noticeable. Thus, the victim's click causes unintentional actions in the context of a legitimate website. Since it is the victim who actually, but unknowingly, clicks on the element of the legitimate page, the action looks "safe" from the browser's point of view; that is, the same origin policy is not violated.

Clickjacking attacks have been reported to be usable in practice to trick users into initiating money transfers, clicking on banner ads that are part of an advertising click fraud, posting blog or forum messages, or, in general, to perform any action that can be triggered by a mouse click.

Beside several proof-of-concept clickjacking examples that have been posted on security-related blogs, it is not clear to what extent clickjacking is used by attackers in practice. To the best of our knowledge, there has only been a single, large-scale real-world clickjacking attack, where the attack was used to spread a message on the Twitter network [24]. We describe this attack in more detail in Section 2.

In this paper, we present a novel approach to detect clickjacking attempts. By using a real browser, we designed and developed an automated system that is able to analyze web pages for clickjacking attacks. We conducted an empirical study on over one million unique Internet web pages. The experiments we conducted show that our system works well in practice.

Our solution can be adopted by security experts to automatically test a large number of websites for clickjacking. Moreover, the clickjacking plug-in we developed can be integrated into a standard browser configuration in order to protect normal users from clickjacking during their daily Internet use.

To the best of our knowledge, we are the first to conduct a large-scale study of the clickjacking problem, and to propose a novel system for the automated testing, and detection of the attack. Our paper provides a first insight into the current prevalence of clickjacking attempts on the Internet.

The main contributions of this paper are as follows:

- We present our automated approach to detect clickjacking attacks.
- We describe the *ClickIDS* browser plug-in we developed, and the system we deployed to analyze more than a million unique Internet web pages.
- We present a first, large-scale attempt to estimate the prevalence of clickjacking attacks on the Internet.
- We assess to what extent clickjacking defense techniques have been adopted by examining thousands of popular websites.

The rest of the paper is structured as follows: Section 2 introduces the clickjacking attack, its impact, and discusses possible countermeasures. Section 3 describes our approach and the system we designed and implemented to test web pages for the presence of clickjacking attacks. In Section 4, we present the experiments we conducted, and discuss our findings. In Section 5, we provide an overview of the related work, and finally, we conclude the paper in Section 6.

2. CLICKJACKING

Despite extensive discussions and reports, clickjacking still lacks a formal and precise definition. Informally, it is a technique to lure the victim into clicking on a certain element of a page, while her intention is to interact with the content of a different site. That is, even though the victim is under the impression of clicking on a seemingly harmless page, she is actually clicking on an element of the attacker's choice. The typical scenario, as described by Grossman and Hansen [16], involves two different websites: A target site T, and a malicious site M.

T is a website accessible to the victim and important for the attacker. Such sites include, for example, online-banking portals, auction sites, and web mail services. The goal of the attacker is to lure the victim into unsuspectingly clicking on elements of the target page T.

M, on the other hand, is under control of the attacker. Commonly, this page is created in a way so that a transparent IFRAME containing T overlays the content of M. Since the victim is not aware of the invisible IFRAME, by correctly aligning T over M, an attacker can lure the victim into clicking elements in T, while she is under the impression of clicking on M. A successful clickjacking attack, for example, might result in the victim deleting all messages from her web mail inbox, or generating artificial clicks on advertisement banners.

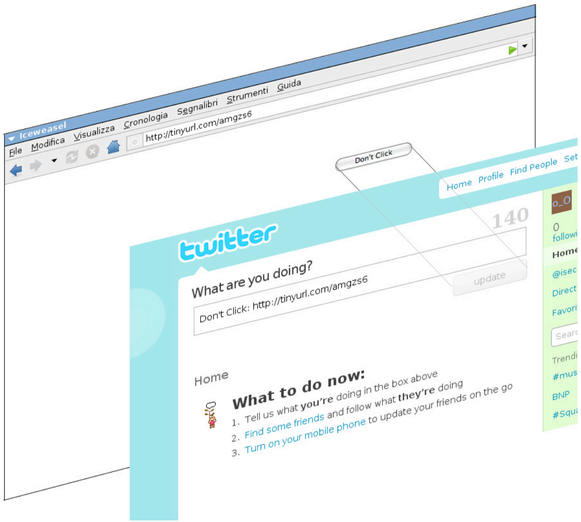
Figure 1 shows a real-world clickjacking attack that has been used to propagate a message among Twitter users [24]. In this attack, the malicious page embeds `Twitter.com` on a transparent IFRAME. The status-message field is initialized with the URL of the malicious page itself. To provoke the click, which is necessary to publish the entry, the malicious page displays a button labeled "*Don't Click*." This button is aligned with the invisible "*Update*" button of Twitter. Once the user performs the click, the status message (i.e., a link to the malicious page itself) is posted to her Twitter profile.

While clickjacking attacks can be performed in plain HTML, the use of JavaScript can be used to create more sophisticated attacks. For instance, JavaScript allows the attacker to dynamically align the framed content with the user's mouse cursor, thus making it possible to perform attacks that require multiple clicks.

Note that manipulating the frame's opacity level (e.g., making it transparent) is not the only way to mount a clickjacking attack. A click can also be "stolen" by covering the frame containing the victim page with opaque elements, and then leaving a small hole aligned with the target element on the underlying page. Another possible approach consists of resizing and/or moving the IFRAME in front of the mouse just before the user performs a click.

Unlike other common web vulnerabilities such as cross-site scripting and SQL injection, clickjacking is not a consequence of a bug in a web application (e.g., a failure to properly sanitize the user input). In contrast, it is a consequence of a misuse of some HTML/CSS features (e.g., the ability to create transparent IFRAMEs), combined with the way in which the browser allows the user to interact with invisible, or barely visible, elements.

A number of techniques to mitigate the clickjacking problem have been discussed on security-related blogs [41]. One approach proposes to extend the HTTP protocol with an optional, proprietary X-FRAME-OPTIONS header. This header, if evaluated by the browser, prevents the content to be rendered in a frame in cross-domain situations. A similar approach was proposed to enhance the CSS or HTML languages to allow a page to display different content when loaded inside a frame.



What are you doing?

Don't Click: http://tinyurl.com/amgzs6

Home

What to do now:

1. Tell us what you're doing in the box above
2. Find some friends and follow what they're doing
3. Turn on your mobile phone to update your friends on the go

```
<IFRAME style={
              width: 550px; height: 228px;
              top: -170px; left: -400px;
              position: absolute; z-index: 2;
              opacity: 0; filter: alpha(opacity=0);
              }
        scrolling="no"
        src="http://twitter.com/home?status=
           Don't Click: http://tinyurl.com/amgzs6">
</IFRAME>

<BUTTON style={
              width: 120px; top: 10px; left: 10px;
              position: absolute; z-index: 1;
              }>
        Don't Click
</BUTTON>
```

Figure 1: Clickjacking attack against Twitter: (a) Page rendering showing the two frames (b) HTML code of the malicious page

Some of the mentioned defenses are already implemented by browser vendors. For example, Microsoft's Internet Explorer 8 honors the X-FRAME-OPTIONS HTTP header and replaces a page whose header is set to `deny` with a warning message [27]. Additionally, the NoScript plug-in for Firefox [25] also evaluates this header and behaves accordingly [26].

In the meanwhile, web developers who do not wish their content to be displayed as frames in other pages have been adopting so-called *frame-busting* techniques. An example of frame-busting is the JavaScript snippet shown in Figure 2. The code compares the origin of the content with the currently displayed resource in the browser and, upon a mismatch, it redirects the browser, thus, "busting" the frame.

Interestingly, an attacker who specifies the IFRAME's attribute `security="restricted"` [28] can force the Internet Explorer to treat the frame's content in the security context of *restricted sites*, where, by default, active scripting is turned off. Therefore, if the embedded page does not make use of JavaScript beyond frame-busting, this protection can be thwarted by an attacker.

```
<script type="text/javascript">
 if ( top.location.hostname !=
      self.location.hostname )
 top.location.replace(self.location.href);
</script>
```

Figure 2: Example of JavaScript Frame-busting code

An alternative solution, not relying on JavaScript, requires the user to re-authenticate (e.g., by re-typing the password or by solving a CAPTCHA) in order to perform any sensitive actions. However, frequent re-authentications degrade the user experience, and thus, cannot be used extensively.

Finally, a possible way to mitigate the problem consists of detecting and stopping clickjacking attempts in the browser. The *ClearClick* extension, recently introduced into the NoScript plug-in, offers some degree of protection. To this end, ClearClick attempts to detect if a mouse click event reaches an invisible, or partially obstructed element. The click event is put on hold, and the user is informed about the true origin of the clicked element. Only if the user agrees, the event propagation continues as usual. Note that NoScript's ClearClick exhibited a large number of false positives in our experiments (i.e., see Section 4).

3. OUR CLICKJACKING DETECTION APPROACH

In this section, we present our approach to simulate user clicks on all the elements of the page under analysis, and to detect the consequences of these clicks in terms of clickjacking attacks. Our technique relies on a real browser to load and render a web page. When the page has been rendered, we extract the coordinates of all the clickable elements. In addition, we programmatically control the mouse and the keyboard to properly scroll the web page and click on each of those elements.

Figure 3 shows the architecture of our system, which consists of two main parts: A testing unit is in charge of performing the clicks, and a detection unit is responsible for identifying possible clickjacking attempts on the web page under analysis.

The detection unit combines two browser plug-ins that operate in parallel to analyze the automated clicks. The first plug-in consists of code that we developed in order to detect overlapping clickable elements. To complement this solution, we also adopted the NoScript tool, that has recently introduced an anti-clickjacking feature. Our experimental results (see Section 4) show that the combination of the two different detection techniques greatly reduces the number of false positives.

The testing unit contains a plug-in that extracts the coordinates of the clickable elements rendered on the page, and a browser-independent component that moves the mouse to the coordinates, and simulates the user's clicks. In addition, the testing unit is responsible for navigating the browser by typing into the address bar the address of the web page to visit.

In the following, we explain the two units in more detail.

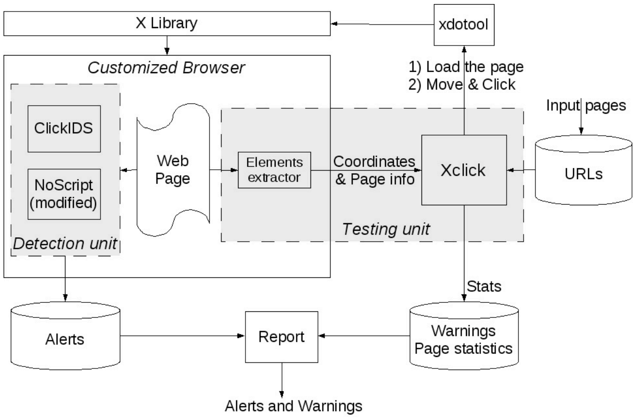


Figure 3: System architecture

3.1 Detection unit

This unit is responsible for detecting and logging any clickjacking attacks that are contained in the web page under analysis.

The detection is handled by two browser plug-ins. The first component is a solution that we developed to detect when multiple clickable elements co-exist and overlay in the region of the page where the user has clicked. We call our detection solution *ClickIDS*. The second plug-in is the modified version of the NoScript open-source tool that saves the generated alerts into a database instead of displaying pop-ups to the user. In the following, we describe ClickIDS and NoScript in more detail.

3.1.1 ClickIDS

ClickIDS is the browser plug-in that we implemented. It intercepts the mouse click events, checks the interactions with the elements of a web page, and detects clickjacking attacks.

The basic idea behind ClickIDS is simple. A suspicious behavior is reported when two or more clickable elements of different pages overlap at the coordinates of the mouse click. As clickable elements, we consider links (or, more precisely, the area enclosed between HTML `<A>` tags), buttons, and form inputs fields such as checkboxes, radio buttons, menu, and text fields. In addition, we also take into account Adobe Flash content, embedded in HTML with `<EMBED>` tags and associated with the application-type *x-shockwave-flash*.

We motivate the consideration of Flash content in two ways. First, when clickjacking was first reported in October 2008, it gained interest fast mainly because a clickjacking exploitation technique against the Adobe Flash Player Setting Manager would have permitted to modify the web-cam and microphone security settings [34]. Basically this exploit allowed an attacker to remotely turn the user's computer into an eavesdropping device. Second, for some advanced attacks to be successful, an attacker would need to steal multiple user-clicks, and therefore, would prefer to overlay the clickjacked site with flash content (e.g., a game) that persuades the user to perform several clicks.

When our plug-in is loaded, it registers to the document event *load*. Each time a new page is loaded, the page-handler routine is executed. This routine registers the loaded page and attaches a second click-handler to it. At this point, every click of the user in the context of the web page is intercepted, and handled by the click-handler routine.

If the clicked element is clickable (according to our previous definition), we register the current mouse coordinates. Then, we scan the main page and the contained FRAMEs and IFRAMEs to check if they contain clickable elements at the same position. If there exists at least one element that overlays the clicked one, we generate an alert. ClickIDS is more precise in identifying attacks based on overlapping elements. However, unlike NoScript, it is not able to detect attacks based on partially obstructed pages. Nevertheless, the combination of the two different techniques can effectively reduce the number of false positives generated by the tools individually.

Note that we also developed a third component that we call *Stopper*, which drops mouse events after all the registered listeners have been successfully executed. This prevents the browser from actually opening a new page, submitting a form, or downloading a file in response to the mouse clicks.

3.1.2 NoScript

NoScript is a Firefox add-on that provides protection against common security vulnerabilities such as cross-site scripting. It also features a URL access-control mechanism that filters browser-side executable contents such as Java, Adobe Flash, and Microsoft Silverlight.

In October 2008, an anti-clickjacking feature was integrated into NoScript. This feature protects users against transparent IFRAME-based attacks. Starting from version 1.8.2, the protection has been extended to cover also partially obstructed and disguised elements. The implemented technique, denoted ClearClick, resembles one proposed by Zalewski [41], and is based on the analysis of the click's neighborhood region. An alert is triggered when a mouse click is detected in a region where elements from different origins overlap.

For our prototype, we modified NoScript version 1.9.0.5 and we replace the visual alerts that are normally generated for the user with a logging capability toward an external SQL database. Our customized version produces an entry for each clickjacking attempt, containing a reference to the website that has generated the alert, the URL of the (I)FRAME that has been clickjacked, and the element of the page that has been clicked (tag name, type, href, and coordinates).

3.2 Testing unit

The testing unit simulates the behavior of a human user that interacts with the content of a web page. It is responsible for instructing the browser to visit a certain URL, and then to iteratively click on the clickable elements contained on the page.

We designed our testing unit to overcome the limitations of existing systems for web application testing. Solutions such as Selenium [7] and Watir [8] simulate the mouse actions from inside the browser, sending events to the element that should be clicked. Although this approach is convenient for testing the functional requirements of web applications (such as the correctness of links and form references), it is not suitable for our purposes. The reason is that we do not know on which element the user intended to click on (this is, in fact, the premise for a clickjacking attack). Hence, we did not wish to "simulate" a user click inside the browser, but to control the mouse at the window level, and to actually move it over the interesting element, and click the left button on it. By doing this, we can also be sure that every JavaScript code in the page (such as the ones registered to `OnMouseOver` or `OnMouseUp` events) are executed exactly in the same way as they would if the user was controlling the mouse herself.

Our tool utilizes *xdotool* [9], a wrapper around the X11 testing library, to move the mouse on the screen and to generate keyboard and mouse events. The testing unit can place the mouse cursor at the screen coordinates where the web page's clickable elements are rendered. Since the clicks are generated from the graphical interface itself, from the browser's point of view, they are identical to those of a real user.

The main component of the testing unit is the *Xclick* script. It receives the list of URLs to visit, and it feeds them, one by one, to the browser. Once the page is successfully loaded, the script positions the mouse over each element, and clicks on them. If the element coordinates are outside the browser window, Xclick properly scrolls down the page to show the required content. In addition, Xclick properly manages special elements such as form drop down boxes which can be rolled up pressing the escape button. For large elements (e.g., images and flash contents), it is more difficult to predict the exact position where the clickjacking may occur. Therefore, Xclick performs multiple clicks at fixed intervals to cover the entire element area, in such a way to raise any possible clickjacking attacks. Finally, to improve the reliability of the system, Xclick is able to detect and close any windows, popups, or browser tabs that are opened by the web page as a consequence of the mouse clicks. A pseudocode of the script is detailed in Figure 4.

The coordinates of the web page's clickable elements are received from the *element extractor*, a custom extension that we installed in the browser. This component is registered to the page-open event such that each time a page is loaded, a callback function is called to parse the page's DOM, and to extract all the information about the clickable elements. The plug-in also extracts information concerning all the FRAMEs and IFRAMEs included in the visited page, including their URL and opacity values. The opacity is a CSS3 property that specifies the transparency of an HTML element to a value varying from 0.0 (fully transparent) to 1.0 (completely opaque).

3.3 Limitations

The main limitation of our current implementation to detect clickjacking attempts is that the testing unit interacts only with the clickable elements of the page. In general, this is not a requirement for mounting a clickjacking attack because, at least in theory, it is possible for an attacker to build a page in which a transparent IFRAME containing the target site is placed on top of an area containing normal text.

In order to cope with this additional set of attacks, we combine the alerts produced by our plug-ins with the warnings generated by the Xclick tool for the web pages that contain cross-domain transparent IFRAMEs. In particular, our approach generates a final report containing both the *alert messages* for the pages where a clickjacking attempt is detected, and the *warning messages* that are raised when no attacks are detected, but the page contains transparent IFRAMEs that partially overlap the rest of the page content. As explained in the Section 4, by analyzing the warning messages, it is possible to detect the clickjacking attacks that do not make use of clickable elements.

```
start browser
for url in input:
   check the browser functionalities, else:
      restart it
   feed the browser with the url and instruct it
                                 to load the page
   wait for the page to be loaded
   if a timeout occurs:
      continue
   check the elements extractor's logfile, else:
      continue
   parse the logfile for the list_of_elements and
                               the page statistics
   record the page statistics in the database

   for element in list_of_elements:
      if element > 50x50px:
         crop it (multi click)
      if element.coordinates are in the next page:
         scroll the browser page
      check the element.coordinates validity else:
         continue
      move the mouse on the element.coordinates
      click
      if element.type == select:
         press 'esc' to close the menu
```

Figure 4: *Xclick* Pseudocode

4. EVALUATION

To test the effectiveness of our prototype tool in detecting clickjacking attacks, we first created five different test pages, based on the examples published on the Internet, that contained clickjacking attacks,. In all cases, the system correctly raised an alert message to report the attack.

Having initially validated our approach on these test pages, we set out to test the effectiveness of our system in identifying real-world websites containing similar, previously-unknown clickjacking attacks. We combined different sources to obtain an initial list of URLs that is representative of what an average user may encounter in her everyday web browsing experience. More precisely, we included the top 1000 most popular websites published by Alexa [2], over 20,000 profiles of the MySpace [4] social network, and the results of ad-hoc queries on popular search engines. In particular, we queried Google and Yahoo with various combinations of terms such as "porn," "free download," "warez," "online game," "ringtones," and "torrents." We ran each query in different languages including English, German, French, Italian, and Turkish.

To increase the chances of finding attacks, we also included sources that were more likely to contain malicious content. For this purpose, we included domains from `malwaredomains.com` [3], lists of phishing URLs published by PhishTank [6], and domains that were queried by malware samples analyzed by the Anubis [19] online malware analysis tool.

Combining all these sources, we generated an initial seed list of around 70,000 URLs. Our crawler then visited these URLs, and continued to crawl through the links embedded in the pages. Overall, we visited 1,065,482 pages on 830,000 unique domains.

For our crawling experiments, we installed our tool on ten virtual machines executed in parallel on VMWare Server 3. Since a clickjacking attack is likely to exploit a transparent IFRAME, to speedup the analysis, we decided to dedicate half of the machines to click only on pages containing transparent IFRAMEs. Each machine was running Debian Linux Squeeze and Mozilla Firefox 3, equipped with our detection and testing plug-ins. The browser was customized for being suitable for running in the background, and enabling automated access to its interface. We also disabled any user interfaces that required user interaction, blocked pop-ups and video content, and disabled document caching.

4.1 Results

We ran our experiments for about two months, visiting a total of 1,065,482 unique web pages. We analyzed those pages in "online mode" and we performed an average of 15,000 pages per day. Around 7% of the pages did not contain any clickable element – usually a sign that a page has been taken down, or it is still under construction. The remaining pages contained a total of 143.7 million clickable elements (i.e., an average of 146.8 elements per page).

37.3% of the visited pages contained at least one IFRAME, while only 3.3% of the pages included a FRAME. However, only 930 pages contained completely transparent IFRAMEs, and 627 pages contained IFRAMEs that were partially transparent. This suggests that while IFRAMEs are commonly used in a large fraction of Internet sites, the use of transparency is still quite rare, accounting for only 0.16% of the visited pages. Table 1 summarizes these statistics.

	Value	Rate
Visited Pages	1,065,482	100%
Unreachable or Empty	86,799	8.15%
Valid Pages	978,683	91.85%
With IFRAMEs	368,963	37.70%
With FRAMES	32,296	3.30%
Transparent (I)FRAMEs	1,557	0.16%
Clickable Elements	143,701,194	146.83 el./page
Speed Performance	71 days	15,006 pages/day

Table 1: Statistics on the visited pages

Table 2 shows the number of pages on which our tool generated an alert. The results indicate that the two plug-ins raised a total of 672 (137 for ClickIDS and 535 for NoScript) alerts. That is, on average, one alert was raised every 1,470 pages. This value drops down to a mere 6 alerts (one every 163,000 pages) if we only consider the cases where both plug-ins reported a clickjacking attack. Note that NoScript was responsible for most of the alerts, and, interestingly, 97% of these alerts were raised on websites containing no transparent elements at all.

To better understand which alerts corresponded to real attacks, and which ones were false positives, we manually analyzed all alerts by visiting the corresponding web pages. The results of our analysis are reported in the last three columns of Table 2, and are discussed in the following section.

	Total	True Positives	Borderlines	False Positives
ClickIDS	137	2	5	130
NoScript	535	2	31	502
Both	6	2	0	4

Table 2: Results

4.2 Discussion

Around 5% of the alerts raised during our experiments involved a frame pointing to the same domain of the main page. Since it is very unlikely that websites would try to trick the user into clicking on a hidden element of the site itself, we marked all these messages as being false positives. However, we decided to manually visit some of these pages to have an insight into what kind of conditions tend to cause a false positive in the two plug-ins.

We then carefully analyzed the pages containing cross-domain frames. In this set, we identified a number of interesting cases that, even though not corresponding to real attacks, matched our definition of clickjacking. We decided to divide these cases in two categories: The true positives contain real clickjacking attempts, while the borderline cases contain pages that were difficult to classify as being clickjacking.

4.2.1 *Analysis of false positives*

Most of the false alarms were generated by pop-ups that dynamically appear in response to particular events, or by banners that are placed on top of a scrollable page. In both cases, the content of the advertisement was visible to the user, but it confuses both NoScript (because the area around the mouse click is not the one that NoScript is expecting), and ClickIDS (because the banner can contain clickable elements that overlap other clickable elements on the main page). For similar reasons, the use of dynamic drop down menus can sometimes confuse both plug-ins.

NoScript also reported an alert when a page contained a transparent IFRAME positioned completely outside of the page margins. A manual examination of some of these cases revealed that they corresponded, most of the time, to compromised websites where an attacker included a hidden IFRAME pointing to a malicious page that attempts to infect the user's computer with malware. Even though these were obvious attacks, no attempts were done to intercept, or steal the user's clicks. Another common scenario that induced NoScript to generate false alarms are sites that contain IFRAMEs overlapping the page content in proximity, but not on top of, a clicked element. While ClickIDS did not report these pages, it raised several false alarms due to harmless overlapping of clickable elements even though the page and IFRAME contents were perfectly visible to the user.

Nevertheless, note that by combining together the two techniques (i.e., ClickIDS and NoScript), only four false positive messages were generated.

4.2.2 *Analysis of true positive and borderline cases*

In our experiments, we were able to identify two real-world clickjacking attacks. The first one used the transparent IFRAME technique to trick the user into clicking on a concealed advertisement banner in order to generate revenue for the attacker (i.e., click fraud). Both plug-ins raised an alert for this attack. The second attack contained an instance of the Twitter attack we already discussed in Section 2. We were able to detect this second case by analyzing the warnings generated by our system for the web pages that contain cross-domain transparent IFRAMEs. In fact, by the time we visited the page, Twitter had already implemented an anti-clickjacking defense (i.e., A javascript frame-busting code now substitutes the framed page with empty content).

Even though the pages containing these clickjacking attacks turned out to be examples posted on security-related websites, they were true positives and we detected them automatically. Hence, our system was able to detect these pages by automated analysis of real-world Internet pages.

Moreover, we also found a number of interesting cases that are difficult to accurately classify as either being real attacks, or false positives.

A first example of these borderline cases occurred when an IFRAME was encapsulated in a link tag. In this case, a cross-domain IFRAME was included in a page as link content (i.e., between `<A>` tags). We found that on certain browsers, such as the Internet Explorer, the user can interact with the framed page normally, but when she clicks somewhere on the content that is not a clickable element, the click is caught by the encapsulating link. The result is a kind of “reversed” clickjacking in the sense that the user believes that she is clicking on the framed page, but is instead clicking on a link in the main page. Even though it matches our attack definition, this setup cannot be used to deceive the user into interacting with a different web page. It is unclear to us why the developers of the site chose to use this technique, but we believe that it might have a usability purpose – no matter where the user would click on the page, she would be directed to a single URL chosen by the developer.

Another interesting case that we observed in our experiments occurs when the page to be included into an IFRAME is not available anymore, or has been heavily modified since the page was created. If the IFRAME is set with the CSS attributes `allowtransparency:true` and `background-color: transparent`, the content of the IFRAME is visible to the user, but the area that does not contain anything (e.g., the page background) is not. The obvious intention of the page authors was to display some content from another page (e.g., a small horizontal bar containing some news messages), but since the destination page was not found, and therefore returned a mostly empty page, the IFRAME was rendered as a transparent bar. If the area overlaps with clickable elements, the user could end up clicking on the transparent empty layer (containing a page from a different domain) instead of the main page elements.

4.2.3 *False negatives*

In order to estimate the false negative rate of our tool, we analyzed all pages for which warning messages were raised (i.e., the pages containing cross-domain transparent IFRAMEs, but in which no attack was reported by our plug-ins). Most of the 140 pages for which our tool raised a warning were pages that included the Blogger [10] navigation bar on the top of the page. This bar is implemented as a transparent IFRAME that is automatically set to be opaque when the mouse is moved to the top of the page. In this case, the transparency is used as a means to easily control the appearance and disappearance of the navigation bar.

4.3 Pages implementing frame-busting techniques

In our study, we conducted the following experiment to assess the prevalence of web sites that implement the so-called frame-busting technique (see Section 2).

First, we prepared a web page that accepts a single parameter denoting a URL that should be embedded in an IFRAME. Once the page and all contents (i.e., the IFRAME) finished loading and rendering, we verified that the IFRAME was still present. Pages that perform frame-busting would substitute the whole content in the browser window, thus removing the IFRAME. To automate this experiment, we implemented a Firefox extension that takes a list of URLs to be visited. Once a page is loaded, the extension waits for a few seconds and then verifies the presence of the IFRAME. If the IFRAME is not part of the document’s DOM-tree anymore, we conclude that the embedded page performed frame-busting.

Simultaneously, we analyzed the HTTP headers of the visited websites. The optional X-FRAME-OPTIONS header is intended to control whether a resource can be embedded in a frame or not. While it is known that Internet Explorer 8 and the NoScript plug-in for Firefox honor this header, we also wanted to find out how common the use of this header is among the sites on the Internet.

We constructed a list of popular URLs to visit by downloading the top 200 entries of each of the 17 Alexa categories [11]. Furthermore, we added the top 10,000 pages from the Alexa rankings to the list. Due to many pages that are present in both lists, we visited a total of 11,005 unique URLs. 1,967 pages did not finish rendering within a 30 seconds timeout, and were, thus, not evaluated.

Our experiment revealed that out of the remaining 9,038 pages, 352 websites (3,8%) already implement frame-busting techniques to prevent being loaded in a frame. Furthermore, only one of the visited pages[1] was using the X-FRAME-OPTIONS header.

5. RELATED WORK

The first report of a possible negative impact of transparent IFRAMEs is a bug report for the Mozilla Firefox browser from 2002 [30]. However, the term clickjacking, commonly referring to this behavior, was coined by Hansen and Grossman much later in 2008 [15]. While Hansen and Grossman elaborated on the involved techniques, we are, to the best of our knowledge, the first to conduct an empirical study on this topic.

The protection of web browsers from clickjacking attacks is the focus of the *ClearClick* component in the NoScript [25] Firefox plug-in. Additionally, Zalewski [41] suggests a se-

[1] http://flashgot.net/

ries of techniques that should help mitigate the clickjacking threat. These include, for example, an obstruction algorithm similar to the one adopted by NoScript, and a HTTP protocol enhancement to prevent a web page from being displayed in a frame. Note that the recently released Microsoft Internet Explorer 8 and NoScript already implement this approach.

Closely related to our automatic approach on detecting clickjacking attacks are existing automatic web application security scanners (e.g., [1, 5, 17, 23]). Although these scanners automatically test web applications just as we do, they do not focus on detecting clickjacking attempts.

SecuBat [23] by Kals et al. automatically detects SQL injection and cross-site scripting vulnerabilities on web pages. SecuBat crawls the web, and launches attacks against any HTML forms it encounters. By analyzing the server response, successful attacks can be detected. A similar approach is followed by Huang et al. [17]. In their work, the authors perform black box testing of web applications to detect possible SQL injection and XSS flaws. As the success of such security scanners relies on the test input that is provided to the tested web applications, Wassermann et al. [38] propose a system to automatically generate such inputs.

Jovanovic et al. [22] introduce Pixy to automatically detect web application vulnerabilities through static analysis. Pixy is able to identify flaws that lead to SQL injection, cross-site scripting, or command injection vulnerabilities. Huang et al. [18] also perform static source code analysis to automatically add runtime protection mechanisms to web applications. Similarly, Wassermann et al. [37] propose a static string analysis-based approach to automatically detect injection vulnerabilities in web applications. Detecting SQL injection vulnerabilities by statically analyzing a web application's source code is performed by Xie et al. in [39]. Egele et al. [13] infer the data types and possible value sets of input parameters to web applications by applying static analysis. This information can be leveraged to fine-tune application level firewalls and help protect web applications from injection attacks.

By combining dynamic data tainting with static analysis, Vogts et al. [35] created a system that effectively detects websites that perform cross-site scripting attacks. In contrast, Balzarotti et al. [12] leverage static and dynamic analysis techniques to automatically validate sanitization in web applications.

Software engineering researchers suggest new methodologies and tools for assess the quality of web applications. Ricca and Tonella in [33] proposes an UML model that help to understand the static structure of web application and to exploit semi-automatic white-box testing. Huang et al. [17] describes and deploys in real-world applications a number of software-testing techniques that addresses frequent coding faults that lays to unwanted vulnerabilities. Their work has produced the vulnerability assessment tool *WAVES*.

Different research projects have examined the prevalence of malicious websites on the Internet. Moshchuk et al. [29] crawled over 18 million URLs and looked for web pages that host spyware. In a later study, Provos et al. [32] discover websites that perform drive-by download attacks. This malware detection approach relies on virtual machine introspection [14] where a web browser in a VM visits potentially malicious pages. If the subsequent analysis of the VM's state indicates that an additional process was created during the page visit, the site is regarded as being malicious.

Wang et al. present Strider HoneyMonkey [36] where they visit known malware hosting pages with web browsers running on operating systems with different patch levels. The idea is that, by observing the successful infections, it is possible to learn the vulnerability that was exploited. In addition, a zero-day exploit can be detected if a fully patched system is infected.

In contrast to the approaches that examine the prevalence of malicious websites that host malware, the focus of our work is on the detection of those web pages that host clickjacking attacks. To the best of our knowledge, we are the first to investigate on the clickjacking problem, and to propose an efficient solution for the automatic detection of clickjacking attacks.

Note that our study provides a first insight into the current prevalence of clickjacking attempts on the Internet.

6. CONCLUSION

Clickjacking is a web attack that has recently received wide media coverage. There have been many news items, discussions, and blog postings on the topic. However, it is currently unclear to what extent clickjacking is being used by attackers in the wild, and how significant the attack is for the security of Internet users. In this paper, we presented our system that is able to automatically detect clickjacking attempts on web pages. We validated our tool and we conducted empirical experiments to estimate the prevalence of such attacks on the Internet by automatically testing more than one million web pages that are likely to contain malicious content and to be visited by Internet users. By distributing the analysis on multiple virtual machines we were able to scan up to 15,000 web pages per day. Furthermore, we developed a new detection technique, called *ClickIDS*, that complements the ClearClick defense provided by the NoScript plug-in. We integrated all components into an automated, web application testing system.

Of the web pages we visited, we could confirm two proof-of-concept instances of clickjacking attacks used for click fraud and message spamming. Even though the pages containing these clickjacking attacks have been posted as examples on security-related websites, we found them automatically. Furthermore, in our analysis, we also detected several other interesting cases that we call "borderline attacks". Such attacks are difficult to accurately classify as either being real attacks, or false positives.

Our findings suggest that clickjacking is currently not the preferred attack vector adopted by attackers on the Internet. In fact, after we had finished our study, Jeremiah Grossman posted in his blog that he only expects clickjacking to become a real problem in 5 to 6 years from now [21].

7. ACKNOWLEDGMENTS

This work has been supported by the European Commission through project FP7-ICT-216026-WOMBAT and FP7-ICT-216331-FORWARD, by the POLE de Competitivite SCS (France) through the MECANOS project, by FIT-IT through the Pathfinder project, by FWF through the Web-Defense project (No. P18764) and by Secure Business Austria.

8. REFERENCES

[1] Acutenix web security scanner. http://www.acunetix.com/.

[2] Alexa top sites. http://www.alexa.com/topsites.

[3] Malware domain blocklist. http://www.malwaredomains.com/.

[4] Myspace. http://www.myspace.com.

[5] Nikto. http://www.cirt.net/nikto2.

[6] Phishtank: Join the fight against phishing. http://www.phishtank.com/.

[7] Selenium web application testing system. http://seleniumhq.org/.

[8] Watir automated webbrowsers. http://wtr.rubyforge.org/.

[9] xdotool. http://www.semicomplete.com/projects/xdotool/.

[10] http://www.blogger.com, 2009.

[11] Alexa Internet, Inc. Alexa - top sites by category. http://www.alexa.com/topsites/category/Top/, 2009.

[12] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, pages 387–401, 2008.

[13] M. Egele, M. Szydlowski, E. Kirda, and C. Kruegel. Using static program analysis to aid intrusion detection. In *Detection of Intrusions and Malware & Vulnerability Assessment, Third International Conference, DIMVA 2006, Berlin, Germany, July 13-14, 2006, Proceedings*, pages 17–36, 2006.

[14] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*, 2003.

[15] R. Hansen. Clickjacking details. http://ha.ckers.org/blog/20081007/clickjacking-details/, 2008.

[16] R. Hansen and J. Grossman. Clickjacking. http://www.sectheory.com/clickjacking.htm, 09 2008.

[17] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web application security assessment by fault injection and behavior monitoring. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 148–159, New York, NY, USA, 2003. ACM.

[18] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM.

[19] International Secure Systems Lab. http://anubis.iseclab.org, 2009.

[20] Jeremiah Grossman. (Cancelled) Clickjacking - OWASP AppSec Talk. http://jeremiahgrossman.blogspot.com/2009/06/clickjacking-2017.html, September 2008.

[21] Jeremiah Grossman. Clickjacking 2017. http://jeremiahgrossman.blogspot.com/2009/06/clickjacking-2017.html, June 2009.

[22] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, pages 258–263, 2006.

[23] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: a web vulnerability scanner. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 247–256, New York, NY, USA, 2006. ACM.

[24] M. Mahemoff. Explaining the "Don't Click" Clickjacking Tweetbomb. http://softwareas.com/explaining-the-dont-click-clickjacking-tweetbomb, 2 2009.

[25] G. Maone. Hello ClearClick, Goodbye Clickjacking! http://hackademix.net/2008/10/08/hello-clearclick-goodbye-clickjacking/, 10 2008.

[26] G. Maone. X-frame-options in firefox. http://hackademix.net/2009/01/29/x-frame-options-in-firefox/, 2009.

[27] Microsoft. IE8 Clickjacking Defense. http://blogs.msdn.com/ie/archive/2009/01/27/ie8-security-part-vii-clickjacking-defenses.aspx, 01 2009.

[28] Microsoft Corporation. Security attribute (frame, iframe, htmldocument constructor). http://msdn.microsoft.com/en-us/library/ms534622(VS.85).aspx.

[29] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware in the web. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2006, San Diego, California, USA*, 2006.

[30] Mozilla Foundation. https://bugzilla.mozilla.org/show_bug.cgi?id=154957, 2002.

[31] J. Nielsen. The same origin policy. http://www.mozilla.org/projects/security/components/same-origin.html, 2001.

[32] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 4–4, Berkeley, CA, USA, 2007. USENIX Association.

[33] F. Ricca and P. Tonella. Analysis and testing of web applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society.

[34] US-CERT. CVE-2008-4503: Adobe Flash Player Clickjacking Vulnerability. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4503, 10 2008.

[35] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*, 2007.

[36] Y.-M. Wang, D. Beck, X. Jiang, and R. Roussev. Automated web patrol with strider honeymonkeys:

Finding web sites that exploit browser vulnerabilities. In *IN NDSS*, 2006.

[37] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. *SIGPLAN Not.*, 42(6):32–41, 2007.

[38] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 249–260, New York, NY, USA, 2008. ACM.

[39] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[40] M. Zalewski. Browser security handbook, part 2. `http://code.google.com/p/browsersec/wiki/Part2#Arbitrary_page_mashups_(UI_redressing)`, 2008.

[41] M. Zalewski. Dealing with UI redress vulnerabilities inherent to the current web. `http://lists.whatwg.org/htdig.cgi/whatwg-whatwg.org/2008-September/016284.html`, 09 2008.