WYSISNWIV: What You Scan Is Not What I Visit

Qilang Yang, Dimitrios Damopoulos, and Georgios Portokalidis

Stevens Institute of Technology
Hoboken, NJ, USA
$\{qyang5, ddamopou, gportoka\}$@stevens.edu

Abstract. A variety of attacks, including remote-code execution exploits, malware, and phishing, are delivered to users over the web. Users are lured to malicious websites in various ways, including through spam delivered over email and instant messages, and by links injected in search engines and popular benign websites. In response to such attacks, many initiatives, such as Google's Safe Browsing, are trying to make the web a safer place by scanning URLs to automatically detect and blacklist malicious pages. Such blacklists are then used to block dangerous content, take down domains hosting malware, and warn users that have clicked on suspicious links. However, they are only useful, when scanners and browsers address the web the same way. This paper presents a study that exposes differences on how browsers and scanners parse URLs. These differences leave users vulnerable to malicious web content, because the same URL leads the browser to one page, while the scanner follows the URL to scan another page. We experimentally test all major browsers and URL scanners, as well as various applications that parse URLs, and discover multiple discrepancies. In particular, we discover that pairing Firefox with the blacklist produced by Google's Safe Browsing, leaves Firefox users exposed to malicious content hosted under URLs including the backslash character. The problem is a general one and affects various applications and URL scanners. Even though, the solution is technically straightforward, it requires that multiple parties follow the same standard when parsing URLs. Currently, the standard followed by an application, seems to be unconsciously dictated by the URL parser implementation it is using, while most browsers have strayed from the URL RFC.

1 Introduction

The popularity of the web has made it the prime vehicle for delivering malicious content to users, including browser exploits, malware, phishing, and web attacks, like cross-site scripting (XSS) [32] and cross-site request forgery (CSRF) [17] attacks. Such attacks are prevalent; Microsoft alone reported that more than 3.5 million computers visited a website containing a web-based exploit in the first quarter of 2012 [35]. The prominence of such attacks has lead to the development of many approaches [20, 22, 27, 30, 37] that automatically detect pages containing malicious content, leading to free and commercial tools [3–6, 8–13, 26, 36] that can scan URLs and routinely crawl the web to identify and filter, quarantine, warn, or take down malicious sites.

Users can reach malicious content by clicking on URLs, which have been injected by attackers into legitimate sites or the results of search engines and spread through spam sent over email and messages. Services which scan pages for malicious content, i.e., URL scanners, follow the same URLs to fetch content from servers and classify it as malicious or benign. Thus, it is essential that when a scanner follows a URL, it visits the same page that the user would visit through his browser or client application.

This paper presents an experimental study on how browsers and URL scanners parse URLs. Our experiments reveal discrepancies on how URLs are parsed, with browsers and URL scanners frequently following different standards and introducing their own rules. As a result, including a character like the backslash in a URL can lead a browser to one web page, while the scanner visits another. Essentially, attackers can hide their malicious content from the scanner, while users can still access it. This constitutes a new evasion strategy for attackers that want to avoid detection from URL scanners. While it may not always be available to them, as certain scanner-browser pairs will treat URLs the same way, this evasion strategy is powerful because it is not based on obfuscating content, but simply requires the inclusion of a character in their URLs.

Looking at Google Safe Browsing, in particular, we show that it transforms backslashes contained in URLs to forward slashes before it accesses a URL, a behavior which has been also noted by web developers in the past [2, 28]. On the other hand, Firefox, which uses its malicious-URL database to warn users that are about to accessing malicious sites, does not. Instead, it encodes the backslash character using percent-encoding (aka URL-encoding). As such, an attacker targeting Firefox browsers could essentially hide his exploit from initiatives like Google's Safe Browsing. We have disclosed the issue to both Google and Firefox, who are working on a solution.

The problem is a general one, as every URL scanner tested has exhibited a behavior that creates opportunities for attackers. Technically, the solution to the problem is not a hard one, however, it requires coordination and agreement among the involved parties (i.e., browser and URL scanner developers). Unfortunately, it is also exacerbated by the fact that various applications parse text and automatically create links, when they identify URL patterns. We conducted tests with various applications and libraries, and we discovered that there also discrepancies on what they consider as acceptable URL patterns, leading to another instance of the same problem.

In summary, the contributions of this paper are the following:

- We identify a new evasion strategy made possible because browsers and URL scanners do not parse URLs consistently
- We develop an experimental methodology to reveal discrepancies on how browsers and scanners transform URLs
- We test all major browsers and URL scanners and show that the problem is general
- We test a variety of popular applications that dynamically create links for URL-like text and also discover discrepancies
- We examine popular libraries used for parsing URLs and discover that they follow different RFCs

2 Background

2.1 URL Encoding and Canonicalization

A uniform resource locator (URL) is a generic way to access a resource over the Internet and is most commonly used to access a service or page over the web. A URL is a uniform resource identifier (URI) and it is an Internet standard with the latest RFC describing it being RFC-3986 [1]. Its syntax is familiar and follows the format shown below.

$$\text{scheme://}\underbrace{\text{[user:password@]domain:port}}_{\text{authority}}\text{/path?query\#fragment}$$

URLs aim to be generic so that they can be used for a variety of protocols and by a variety of applications. However, as the web has increased in popularity, URLs are used by an increasing number of applications and have been extended with new features (e.g., internationalization), causing some contemporary implementations to stray from the RFC. The web Hypertext Application Technology Working Group (WHATWG), in an attempt to provide a more current standard, has defined the *URL Living Standard* [45]. Below, we discuss some basic aspects of URLs and URL parsing.

Delimiter A generic URL consists of a hierarchical sequence of components referred to as the scheme, authority, path, query, and fragment. Each component corresponds to a piece of information that is necessary to locate a unique resource. Hence, identifying components correctly when parsing a URL is critical for both browsers and servers. Several delimiters are applied in the URL syntax to help separate components. These are the colon (:), the at sign (@), the slash (/), the question mark (?), and the number sign (#).

URL-Encoding or Percent-encoding is a mechanism for encoding information in a URI to represent a data octet in a component, when the corresponding character of the the octet is outside the allowed character set or is being used for a special purpose such as the delimiter of, or within, the component. A percent-encoded character is a character triplet, which consists of the percent character (%) and two hexadecimal digits representing the octet's numeric value. For example, $\%3F$ is the percent-encoding of the question mark character (?). In percent-encoding format, the uppercase hexadecimal digits and the corresponding lowercase digits are equivalent and exchangeable.

Canonicalization or normalization refers to the process of converting data from one representation to a "standard" or canonical form. Generally, this is done to correctly compare data for equivalence, enumerate distinct data values, improve various algorithms, etc. On URLs, it is mainly done to determine, if two URLs are equivalent and it can include operations such as removing the default HTTP port (i.e., 80), converting the domain to lowercase, and resolving a path that contains a dot or double dot. Occasionally, applications introduce their own canonicalization rules, such as removing duplicate slashes (// $\rightarrow$ /), automatically completing incomplete IP address, deleting extra leading dots in the authority part, etc.

2.2 URL Scanners

Because of the importance of web browsers and the multitude of attacks targeting them or being delivered through them, several approaches [3–13, 20, 26, 36] have developed URL scanners. A URL scanner is a service that analyzes a URL, enabling the identification of viruses, worms, trojans, phishing and other kinds of malicious content detected by antivirus engines or website scanners. URL scanner services are commonly accessed through an online web service, a browser extension, a third party library, or a public web-based API.

Scanners have two main interfaces. The first, allows users to submit or report a URL for immediate or later scanning [3, 4, 8–11, 13, 20]. The scanner will then retrieve the content and scan it to determine maliciousness. Some scanners [4, 10] also consult multiple third-party scanners to determine if the content is malicious. The second interface, enables users to query whether a URL has been found to be malicious using the scanner's malicious-URLs database (blacklist). The database is queried looking for an exact or partial match. Regarding the ownership of the blacklist, some scanners maintain their own blacklist [5, 6, 26, 36], while others use third-party blacklists [12]. Finally, scanners can be divided into two categories: the ones that only check the content of submitted URLs and the ones that follow links within the submitted page [3, 8, 11, 13, 26].

Among many URL scanners, there are two that are widely used in daily life, though sometimes users may not be aware of them. The first one is Google Safe Browsing [26], a Google service that helps applications check URLs against Google's constantly updated lists of suspected phishing, malware, and unwanted software pages. It is available through a series of web-based APIs. Google Safe Browsing as a scanner service is integrated into Chrome and Firefox, and even Safari uses its database. The second one is Microsoft's SmartScreen filter [36], a malware and phishing filter that is integrated in several Microsoft products including Internet Explorer and Hotmail. Any time a user gets a warning when visiting a web page in these browsers, it means that the URL is in one of the two scanners' blacklist or both.

3 The Problem

Figure 1 depicts the process, where URLs, which have been submitted by users or obtained by crawling the web, are scanned for malicious content. Links contained within pages are usually also followed and scanned, and when a page is found to contain malicious content, it is inserted into a database. That database can be later used by the browser to prevent users from accessing malicious content. For example, before fetching any page, the browser first checks the URL of that page in the database. If an entry does not exist, it proceeds to load and display the page to the user. If, however, an entry for the URL is found in the database, the browser redirects the user to a page warning him that he is about to visit a page containing malicious content. Even though the user can ignore the warning, research has shown that such warnings are effective in protecting users [14, 23].

This process through which the user is protected from visiting malicious pages can be undermined when scanners and browsers do not parse URLs consistently. There

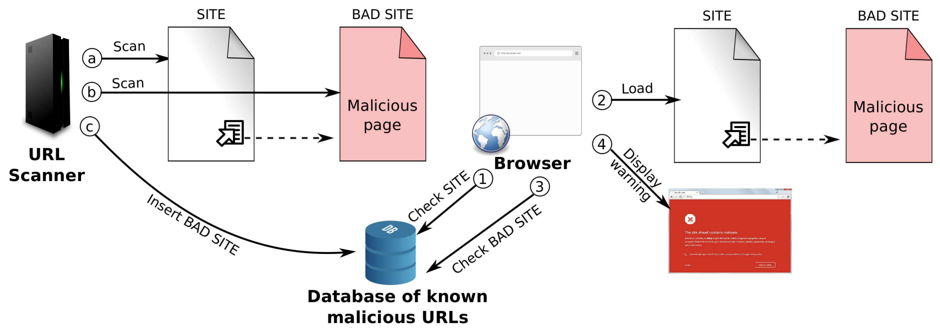

Fig. 1. Modern browsers utilize databases of known malicious URLs, populated by offline URL scanners, to warn and protect users.

may be many reasons that two programs do not parse a URL the same way. It may be consciously, because their developers chose to support a different standard, or because one of them has adopted additional standards and guidelines. It can also be because of program bugs that cause inconsistent behavior when parsing certain, otherwise legitimate, URLs. Independently of the reason, if the database utilized by a browser was produced by a scanner that treats certain URLs differently than the browser, the user is left exposed to malicious content, which would be otherwise detected and filtered.

The mechanics of such an attack are shown in Fig. 2. An attacker aware of discrepancies in URL parsing can place his malicious content under a URL that brings them about, e.g., $BADURL$. When the scanner processes a page containing this URL, the sought after behavior is triggered causing the scanner to transform the URL to $BADURL'$ before accessing it and scanning it. The attacker is essentially able to hide his malicious page from the scanner, so it can never be entered in the database, later used by the browser. It is interesting to note that even if the attacker for some reason placed malicious content under $BADURL'$, causing it to be logged in the database, the browser would actually check $BADURL$ instead, which would not match any of the logged entries. The problem is symmetric, in the sense that if the browser is the entity transforming the URL before accessing it, then the attacker can still hide malicious content by placing it under $BADURL'$ while putting innocuous content under $BADURL$.

This problem, which we name *What You Scan Is Not What I Visit* (WYSISNWIV), is not limited to browsers, but it can affect any application that creates links for displayed URLs. For example, instant messengers and various web applications, like web mail, do create links for URLs identified in text. Concurrently, there are various products that filter malicious URLs based on databases created by public and proprietary scanners [5, 24, 38, 42]. Each application-scanner pair, where the two do not process URLs the same way, can leave the user exposed to malicious content.

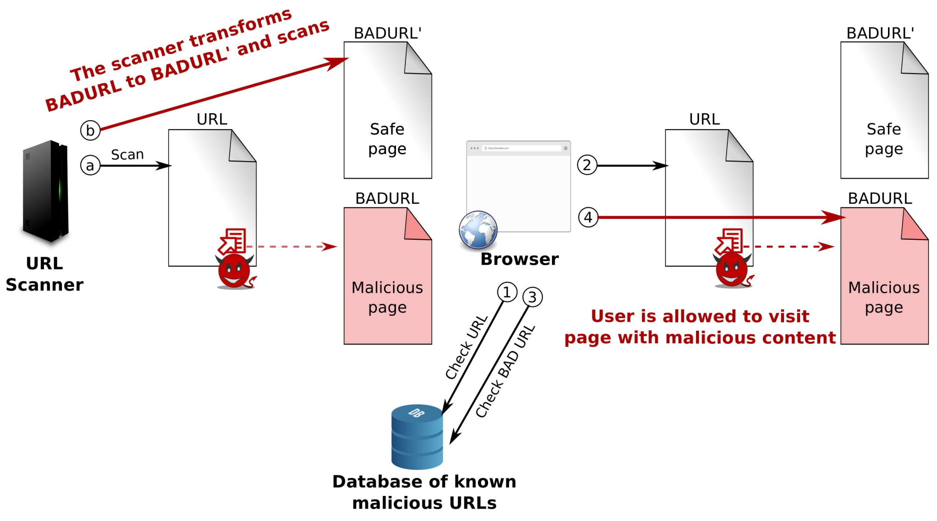
The scanner transforms BADURL to BADURL' and scans
BADURL'
Safe page
BADURL
Malicious page
URL
Scan
URL Scanner
Browser
Check URL
Check BAD URL
Database of known malicious URLs
User is allowed to visit page with malicious content
a
b
1
2
3
4

Fig. 2. A scanner parsing URLs differently from a browser allows hiding malicious content from the first, using a carefully crafted URL, while the latter follow it to malicious content.

4 Experimental Methodology

To detect discrepancies on how URLs are parsed, we design experiments that will drive browsers and URL scanners with various test inputs. This section describes how we generate the test inputs and the experiments run with browsers and URL scanners.

4.1 Generating Test Inputs

To generate the URL inputs used for testing, we follow a structured approach building on domain knowledge. In particular, we manually examine the following sources to identify high-level patterns of inputs, based on which we generate inputs for testing. The three following resources are used to identify high-level patterns for testing: i) the RFC 3986 document, ii) the code base of the Chromium and Mozilla Firefox web browsers, and iii) the unit tests that come with these browsers. The RFC 3986 specification broadly defines what is allowed, what may be allowed, but also what must be disallowed when a URL is constructed. However, it allows browsers to implement their own policies for “maybe allowed” characters. Thus, based on the study of the URL specification and two open-source web browsers, it would be possible to have discrepancies in some special cases, such as when encountering control characters, special Unicode characters, the backslash character, and encoded delimiters included in the URL path. Furthermore, unit tests provide key examples that web browsers and services should be able to successfully parse for compatibility reasons. Table 1 lists all the test inputs constructed based on the above sources.

Table 1. Inputs used for testing browsers and scanners.

Description of transformation	**Generated Tests**
Convert the scheme and host to lower case	Equivalence of *HTTP://WWW.EXAMPLE.ORG/PATH* and *http://www.example.org/path*
Decode percent-encoded octets of unreserved characters	URL with sampled character from range %41-%5A ('A'-'Z') and %61-%7A ('a'-'z')
Remove default port	Equivalence of *http://www.example.org:80/path* and *http://www.example.org/path*
Add trailing '/'	Equivalence of *http://www.example.org/path/* and *http://www.example.org/path*
Removing dot-segments ('.')	Equivalence of *http://www.example.org/path/././index.html* and *http://www.example.org/path/index.html* (Number of '.' in the range of 1-5)
Removing the end fragment '#frag'	Equivalence of *http://www.example.org/path/index.html#frag* and *http://www.example.org/path/index.html*
Replacing IP with domain name	Equivalence of *http://www.example.org/path* and *http://10.0.0.1/path*
Limiting protocols	Equivalence of *https://www.example.org/path* and *http://example.org/path*
Removing duplicate slashes	Equivalence of *https://www.example.org/path//index.html* and *http://example.org/path/index.html*
Unicode character handling	Multiple URLs with characters sampled from \x00-\xffffff
Printable characters that need to be percent-encoded	Double quote character *http://www.example.org/path"path*
Backslash character ('\')	*https://www.example.org/path\index.html*
Non–ASCII characters that neeed to be percent-encoded	Multiple URLs with character sampled from range %C0 - %FF
Control characters	URLs including characters \a, \b, \e, \n, \t, \0, \v, \f, \r
Encoded delimiters	URLs including percent-encoded delimiters '#', '?', and '/' in the path
Leading dots	http://...www.example.org (Number of '.' in the range of 1-5)
Whitespace/Tab	*http://www.example.org/pa th* and *http://www.example.org/path%a0path*

4.2 The Experiments

Our first experiment aims to discover differences on how browsers parse and transform URLs, before submitting a request to the server. We developed a browser driver, as a bash script, which launches a browser and requests a URL from the set of test inputs. The browser then performs canonicalization and transformations on the URL and establishes a connection to our server, where it sends the request including the transformed URL. The server was developed using Python on top of the *werkzeug* library and accepts every URL request, logs it, and responds with a default web page. After the page is loaded at the browser, we also extract the URL that was requested from the browser's history database. The URLs requested, received by the server, and stored in the history database are compared to identify discrepancies. To facilitate comparison, we use a unique path prefix on each request that allows us to compare the appropriate URLs.

The second experiment means to evaluate how online URL scanners deal with URLs reported as being malicious, and for scanners that also follow links within the reported pages, discover how they treat the URLs contained within those pages. The latter test serves to establish how a scanner's internal algorithms parse and transform URLs, which reveals how it operates when or if it is used to crawl the web for malicious content. For the first part of this experiment, we manually submit URLs pointing to malicious content to the scanners using the interface provided, most commonly an HTML form. For the latter, we submit URLs pointing to benign pages, which do not directly contain malicious content, but do include URLs pointing to malicious content. All URLs point to our own server that logs information like the remote IP address, other information like the user-agent included in the request, and the timestamp of the request. We also use unique paths in each case to differentiate between scanners.

In the our final experiment, we focus on browsers and URL scanners that work in synergy, such as Chrome and Firefox using Google's Safe Browsing malicious-URLs database. In this test, we submit both benign and malicious websites located in different URL paths, using characters and patterns discovered in the previous experiment to "hide" malicious content from the scanner. The aim is to confirm that we can construct URLs that will point the scanner to safe content, while a browser following the URL will visit malicious content instead.

5 Results

5.1 Browsers

We tested four browser families on three desktop operating systems (OS). We tested Firefox v35.0.1, Chrome v40.0.2214.115, and Opera v27.0.1689.69 on Ubuntu v14.04 LTS, Mac OS X v10.10.2, and Windows 7 SP1. We also tested Safari v8.0.3 on Mac OS X, and Safari v5.1.7 and Internet Explorer (IE) 8.0. 7601.17514 on Windows 7. Our results show that Firefox URL-encodes backslashes to ($\backslash \rightarrow \%5C$), while every other browser canonicalizes the URL replacing backlashes with forward slashes ($\backslash \rightarrow /$). We also tested three mobile OS: Android v4.4.2 with Firefox v38.0.5, Chrome v43.0.2357.78, and Opera Mini v29.1, iOS 8 with Chrome v43.0.2357.51, Safari v8.3,

Table 2. The URL scanners considered during testing. All, except Wepawet, scan for both Phishing and malware sites. Some of the scanners, such as VirusTotal, also use third-party databases and scanners.

URL Scanners	**Available Actions**			**Uses Third-party Database/Scanner**
	Scan URL	Query URLs DB	Report URL	
Wepawet	✓			
Google Safe Browsing		✓	✓	
Virustotal	✓			✓
Sucuri Sitecheck	✓			
gred	✓			
Online Link Scan	✓			✓
urlQuery	✓			
PhishTank		✓	✓	
Scumware		✓	✓	
WebInspector	✓			
Zscaler Zulu	✓			
SmartScreen Filter		✓	✓	
ScanURL		✓		✓
stopbadware		✓	✓	✓

and Opera Mini , and Windows Mobile 8.1 with IE and Opera Mini. Once again, Firefox URL-encodes backslashes. Interestingly, Opera Mini on iOS leaves the backslash character unchanged, while every other browser replaces them with slashes. These modifications occur both when a user types a URL in the address bar and when clicking on a link. As a result, browsers doing canonicalization can never access pages hosted on URLs containing a backslash as a legitimate character.

5.2 URL Scanners

Table 2 lists all the URL scanners we considered in our experiments. We selected several state-of-the-art URL scanners, including products of academic research and freely-accessible production systems. For example, Wepawet [20] is a product of academic research, while VirusTotal [10], Sucuri SiteCheck [8], gred [3], Online Link Scan [4], urlQuery [9], ScanURL [12], PhishTank [5], Scumware [6], WebInspector [11], and Zscaler Zulu URL Risk Analyzer [13] are mature products. We focused our experiments on Google Safe Browsing [26] and Microsoft's SmartScreen Filter [36], as the first is being used by Chrome, Firefox, and Safari, and the latter from IE, for protecting users from malicious URLs. Most scanners permit us to submit URLs (e.g., through a web form) for scanning, returning a report on their state (e.g., whether it is malicious). Others, offer a way to check whether a URL is contained in their database of malicious URL, while, finally, some allow us to report URLs, which will be later checked.

From the scanners listed in Table 2, we tested all that allowed us to submit a URL for scanning or report URLs. We also tested ScanURL, which, after submitting a URL query, provides feedback on the actual URL being searched in the database, granting us

Table 3. The tested URL scanners handle certain characters differently from browsers. Pairs of browsers and scanners that have such discrepancies leave users exposed when the particular scanner is used to filter URLs, as the scanner does not process the same page the browser will visit (e.g., the pairs Chrome/Firefox and Google Safe Browsing).

Scanners	**Transformations**					
	Manual submission			Injection in submitted page		
	$\backslash$	$\%3F$ (?)	$\%23$ (#)	$\backslash$	$\%3F$ (?)	$\%23$ (#)
Wepawet	$\%5C^{\ddagger}$	$\%3F$	$\%23$	N/A		
VirusTotal	$\%5C^{\ddagger}$	$?^{\star}$	$\#^{\star}$	N/A		
gred	*deleted**	$\%3F$, $\%253F$	$\%23$, $\%2523$	*error**	$\%3F$	$\%23$
Online Link Scan	$\backslash\backslash^{\star}$	$\%3F$	$\%23$	N/A		
urlQuery	$\%5C^{\ddagger}$	$\%3F$	$\%23$	N/A		
ScanURL	*deleted**	$\%3F$	$\%23$	N/A		
PhishTank	$\backslash\backslash^{\star}$	$\%3F$	$\%23$	N/A		
Scumware	*deleted**	$\%3F$	$\%23$	N/A		
WebInspector	$/^{\dagger}$	$\%3F$	$\%23$	$\%5C^{\ddagger}$	$\%3F$	$\%23$
Zscaler Zulu	$/^{\dagger}$	$\%3F$	$\%23$	$/^{\dagger}$	$\%3F$	$\%23$
Google Safe Browsing	*varies*$^{\star\bullet}$	$?^{\star}$	$\%23$	$/^{\dagger}$	$\%3F$	$\%23$
Sucuri SiteCheck	*deleted**	*error**	*error**	*error**	*error**	$\%23$

$\star$Affects pairing with all browsers.
$\dagger$Affects pairing with browsers that URL encode backslash.
$\ddagger$Affect pairing with browsers that transform backslash to forward slash.
$\bullet$ Handling of the backslash depends on the character following it.

this way an indication on the transformations it performs on URLs. There was no way to test stopbadware or SmartScreen Filter. The first did not provide a way to expose how it handles URLs, while the latter is integrated into IE, where the user can use the graphical interface to manually check and report URLs. Because both the URL submission process and filtering is handled by the browser, we cannot test for discrepancies in a meaningful way.

From all the tested URL patterns, we discovered three transformations that can cause problems when a scanner's database is coupled with a browser to filter malicious URLs (e.g., like Chrome and Firefox using the Google Safe Browsing database). The patterns are: the backslash character ($\backslash$), and the URL encoded characters ? ($\%3F$) and # ($\%23$). Interestingly, the handling of the backslash character is not well defined in RFC-3986, while ? and # are delimiters in the URL format. The results are summarized in Table 3 and further discussed below.

URL-encoded Delimiters The characters ? and # are delimiters for URLs and need to be URL-encoded or percent-encoded, if they are present in other parts of the URL, where they are allowed. This way the characters are *escaped*. Our results show that certain scanners unescape these characters, unintentionally transforming the URL, like in the example illustrated below with ? ($\%3F$):

http://www.example.org/$\underline{\textit{path\%3Fdistorted}}$ $\rightarrow$ *http://www.example.org/$\underline{\textit{path}}$?distorted*

Table 4. Examples of URL transformations caused by handling backslash (\) differently.

	Original URL	*http://www.example.org/path\distorted*
	URL-encoded	*http://www.example.org/path%5Cdistorted*
'\' **is**	canonicalized	*http://www.example.org/path/distorted*
	dropped	*http://www.example.org/pathdistorted*
	backslash escaped	*http://www.example.org/path\\distorted*

The underlined part is actually the path requested from the server at $www.example.org$ in each case. As indicated by Table 3, Google Safe Browsing and VirusTotal do such a transformation and, as a result, check a different path, than the one the browser visits. Even worse, Sucuri SiteCheck does not accept $\%3F$ at all, treating URLs including it as invalid. Similarly, for $\%23$, the encoded version of #, Sucuri SiteCheck does also not accept it, while VirusTotal unescapes it. Interestingly, when gred encounters either of the two percent-encoded delimiters, it checks two links: the original link, treating the encoded character as an encoded character, and a link where the percent character ($\%$) is itself escaped to $\%25$. gred seems to be very careful in handling form input in this case, accounting for both eventualities, even though no browser seems to treat the $\%$ character that way.

Backslash Handling Backslash (\) is the character handled in the most inconsistent way among different scanners. We have identified four different transformations that the backslash character is submitted to in our tests: it can be URL-encoded to $\%5C$, canonicalized to (i.e., replaced by) a forward slash (/), simply dropped from the URL, or escaped using another backslash (\\). Examples of these URL transformations are listed in Table 4. Three scanners, Wepawet, Virustotal, and urlQuery escape it by percent-encoding it to $\%5C$. This behavior is akin to the encoding done by Firefox, and as a result pairing any of these scanners with any browser, aside Firefox, would enable an attacker to hide malicious content from the scanner. The reverse happens with Zscaler Zulu that replaces the character with a forward slash, which makes it a bad fit for using with Firefox. ScanURL and Scumware will always completely drop the backslash URL, while Online Link Scan and PhishTank will escape the character using another backslash. In both these cases, using the scanner would expose the user to attacks through such URLs regardless from the browser he is using.

Intra-scanner Backslash Handling Discrepancies Certain scanners like WebInspector, gred, Google Safe Browsing, and Sucuri SiteCheck, handle the backslash differently depending on how they obtain the URL they are scanning. For example, gred and Sucuri SiteCheck drop it, when we manually submit a URL, while when the URL is in a link within the submitted page, obtained after parsing the submitted page and following the links within, they do not accept it and consider the URL invalid. We establish this by injecting various links within the submitted page and observing that only the ones containing a backslash are not accessed by the scanner. On the other hand, WebInspector canonicalizes backslashes on manual submission, while links injected in pages are URL-encoded. Finally, Google Safe Browsing treats the percent-encoded '?' differently based on how the URL is obtained, while backslashes in manually submitted URLs are processed in a more elaborate way, than when in URLs in pages, where they are trans-

Table 5. Examples of how Google Safe Browsing transforms the backslash character when manually reporting URLs.

#	Reported URL		Visited URL
1	*http://www.example.org/path\ndistorted*		*http://www.example.org/pathdistorted*
2	*http://www.example.org/path\adistorted*		*http://www.example.org/path%07distorted*
3	*http://www.example.org/path\0distorted*		*http://www.example.org/path*
4	*http://www.example.org/path\x50distorted*	$\rightarrow$	*http://www.example.org/pathQdistorted*
5	*http://www.example.org/path\x96distorted*		*http://www.example.org/path%96distorted*
6	*http://www.example.org/path\x0110distorted*		*http://www.example.org/path*
7	*http://www.example.org/path\Qdistorted*		*http://www.example.org/path*

formed to forward slashes. We further discuss Google Safe Browsing below, due to its importance.

Google Safe Browsing The backslash character is treated in many different ways by Google Safe Browsing, when a URL is manually reported, which we list below:

1. A backslash specifies a *control character*, when it is followed by one of the following characters: t, n, a, b, 0, and e. Depending on the control character, the URL is transformed in three different ways:
   '\t' '\n' The control character is deleted and the strings before and after it are joined together, as in example 1 in Table 5.
   '\a', '\b' The control character is converted to the corresponding URL-encoded character (%07 and %08 respectively), as in example 2 in Table 5.
   '\0', '\e' The control character and all trailing characters in the URL are deleted, as in example 3 in Table 5.
2. A backslash escapes a *unicode character* when it is followed by the character 'x' ($\backslash x$). In this case, the characters trailing 'x' are retrieved and interpreted as a Unicode character code in hexadecimal. The following sub-cases are possible:
   Character does not require encoding The ASCII representation of the character replaces it, as in example 4 in Table 4.
   Character requires percent-encoding The percent-encoded form of the character replaces it, as in example 5 in Table 4.
   Invalid character If the Unicode character is not allowed in the URL, for example, because it requires two percent-encoded bytes like in the case of $\backslash 0110 \rightarrow \%C4\%90$, it is dropped along with all trailing characters, as in example 6 in Table 4.
3. When a backslash is followed by any *other character*, it is treated as an invalid character and it is dropped along with all trailing characters, as in example 7 in Table 4.

5.3 Backslash in Other Applications

Applications, such as instant messengers (IMs), web email, and email clients, dynamically create links when they identify text that resembles a URL. If the information exchanged by such an application is intercepted to scan for potentially malicious URLs [24,

Table 6. How various other applications transform the backslash character.

	Program	**Transformations of backslash (\\) in URLs.**			
		URL cropped at \\	Encoded ($\%5C$)	Canonicalized (\\ $\rightarrow$ /)	Preserved (\\)
Instant Messengers	Skype		✓(Mac)	✓(Windows)	✓(Android)
	Hangouts	✓(Android)	✓(iOS)		
	QQ	✓(iOS, Android)	✓(Mac)		✓(Windows)
	WeChat	✓(iOS, Android)			
	Facebook Messenger	✓(Android)	✓(iOS)		
	Line	✓(Android)	✓(iOS)		
	iMessage		✓(Mac, iOS)		
	WhatsApp	✓(Android)	✓(iOS)		
	Viber	✓(Android, iOS)			
Web mail	GMail		✓		
	Outlook.com	✓			
	Yahoo! Mail		✓		
	Roundcube Webmail	✓			
Email clients	Opera Mail		✓(Mac, Windows)		
	Thunderbird		✓(Mac, Windows)		
	Outlook			✓(Windows,Mac)	
	Zimbra Desktop			✓(Mac, Windows)	
	eM client	✓(Windows)			
	Inky	✓[†](Windows, Mac)			
	Claws Mail				✓(Windows)
Popular Sites	Facebook	✓			
	Twitter	✓			
	LinkedIn				✓
	Tumblr		✓		

† The entire path part of the URL and part of the domain is cropped.

38, 42], any transformation applied by the application, introduces another point that could be exploited by an attacker (e.g., to bypass URL scanners when performing a spear phishing campaign). We tested various applications with URLs including the backslash character and report our results in Table 6. We focused our efforts on popular operating systems and platforms, such as Mac and Windows on desktops/laptops and iOS and Android on smartphones. Email clients were tested on both Windows and Mac platforms, if available (e.g., eM client and Claws Mail do not have a Mac version). IMs with the exception of Skype and QQ were tested on mobile platforms, since there is a broader variety and are more commonly used on these platforms. Skype and QQ were also tested on Windows and Mac. We do not list Skype's case on iOS, since it does not not create links for the tested URLs, essentially failing to recognize URLs with ambiguous characters. Web mail cases and popular sites were tested on both Windows and Mac using Internet Explorer, Chrome, and Firefox.

Most of the tested applications handle backslashes more strictly than browsers and stop processing when a backslash is encountered [45], essentially cropping the URL. However, since no scanner performs such a transformation, *stricter is not safer in this case*. The remainder of the tested applications either canonicalize URLs, transforming backslashes to forward slashes, preserve them, or URL- encode them. An interesting finding is that most of the applications on Android cropped the URL before the first backslash and most of the applications on iOS platform encoded the backslash. Through further investigation, we found that there is a build-in library for finding URLs in plain text, namely `android.util.Patterns.WEB_URL`, which terminates URL pattern matching when it encounters a backslash. On iOS, the build-in library `dataDetectorWithTypes:NSTextCheckingTypeLink` encodes the backslash automatically while searching for URLs. Email clients exhibit more divergence on handling URLs, which indicates that developers create their own URL parser or utilize different libraries to parse URLs. Our results indicate that the standard followed by applications may be unconsciously dictated by the platform and libraries used, some times causing the same application to handle URLs differently based on its platform version.

5.4 Backslash Handling by Different Libraries

Based on the findings presented in the previous section, we further investigate how platforms and libraries handle the backslash character in URLs. We chose some of the most commonly used languages, as reported by IEEE Spectrum [19], and widely used libraries for URL processing used when developing in these languages. The results are presented in Table 7. We observe that different libraries indeed diverge by essentially adhering to different URL RFCs. In *libcurl* 's specifications both RFC 2396 and RFC 3986 are listed, and the library preserves the character. The *cpp-netlib* library obeys RFC 3987, while libraries part of *python 2.6* do not refer to a particular RFC and, interestingly, they handle the character differently. Oracle's Java platform follows the RFC 2396 specification, but when using the *URI* class, the backslash character is not accepted. The *google-http-java-client* library follows RFC 3986. Finally, Ruby's library *net/http* uses RFC 2396. These results indicate that applications may implicitly adopt

Table 7. How various libraries handle the backslash character. We provide URLs from standard input, parse them using the corresponding libraries, and print the parsed URL to standard output.

Library	**Transformations of backslash (\) in URLs.**				
	Deleted	Encoded	Canonicalized	Preserved	Error
libcurl v7.44.0 (C)				✓	
cpp-netlib 0.11.1 (C++)	✓				
Python v2.6.8 – httplib				✓	
Python v2.6.8 – urllib		✓			
Java v1.8.0_31 – java.net (URI class)					✓
Java v1.8.0_31 – java.net (URL class)				✓	
google-http-java-client v1.20.0 (Java)		✓			
C# v4.6.00079 – System.Net			✓		
Ruby v2.2.2p95 – net/http					✓

and RFC for handling URLs based on the libraries used and the platform a developer develops for.

6 Discussion

6.1 The Problematic Backslash Character

Web developers have noticed the differences in how different browsers handle the backslash character before us. In a *stackoverflow* post a developer reports that the handling of backslashes from Chrome prevents him from using it legitimately [2]. The response from another user is enlightening: '*The unified solution to deal with backslash in a URL is to use %5C. RFC 2396 did not allow that character in URLs at all (so any behavior regarding that character was just error-recovery behavior). RFC 3986 does allow it, but is not widely implemented, not least because it's not exactly compatible with existing URL processors.*'. More recently, a Google+ user and web developer also identified the discrepancy and pointed that it could lead to another type of vulnerability [28]. In particular, changing the URL can affect the verification of the message origin when using $postMessage()$. They had to update their web application to account for backslash transformations. It is clear that it is unclear which standard each browser and URL scanner adheres to. Moreover, attempts to auto-correct user typos, such as typing a backslash instead of a slash, have been widely adopted by graphical programs, such as browsers. On the other hand, only a few URL scanners seem to be aware of such schemes.

6.2 Impact and Responsible Disclosure

Our results show that there is a clear gap on the use of Google Safe Browsing from Firefox. That is, because an attacker can create URLs including backslashes, which can be followed by Firefox but transformed by Google before checking them for malware. We disclosed the problem to both Google's Safe Browsing team and Mozilla. They have acknowledged it and are working towards a solution. At the moment of writing, the solution is not clear cut due to multiparty involvement. Firefox could adopt canonicalization as the rest of the main stream browsers. Until that happens Google may be looking out for backslashes in encountered URLs. A member of the Google Safe Browsing team has confirmed that such URLs (not the ones submitted by us) are present in their malicious-URLs database, despite our inability to get such URLs scanned. This confirms that even within Google backslash handling is not uniform. Based on our results with various scanners and applications, we suspect that other solutions based on different URL scanners to filter or block malicious URLs are suffering from the same issue.

6.3 Remediation

Adhering Strictly to a Single Standard The obvious solution to the problem would be that every URL parser implementation adheres to the same standard and be bug-free. Unfortunately, experience has showed that this is probably not a realistic solution. Just recently a bug in how Skype for Windows parses URLs caused it to crash when it parsed the string "http://:" [39]. Browser developers have been devising ways for years to auto-correct common errors made by web developers and display pages that would not be parsed by a strict HTML parser. HTTP, the protocol running the web, is also frequently incorrectly implemented, as a quick search for " incorrect HTTP handling" reveals.

Using Multiple URL Scanners Our results show that for all tested scanners and browsers, there is no single scanner that could be adopted by any browser and have no discrepancies that leave room for attacks. However, combining multiple scanners could solve the problem, as they would cover different links. As these scanners may already be exchanging data, we designed a test to evaluate whether they already do. More specifically, we checked whether Google Safe Browsing utilizes other scanners' databases. For this test, we created unique URLs that point to a malicious executable file and submit them to each scanner through the appropriate interface. After five days, we check whether the link is stored inside the corresponding scanner's database. We could do this for VirusTotal, Scumware, WebInspector, and Zscaler Zulu that offer a database query interface. Then we access these links through Chrome to check whether they are blocked. Unfortunately, none of them was, indicating that the scanners do not share data.

Broader Scanning and URL Collection The most viable solution seems to be that when URLs are found to contain characters or patterns, which may be interpreted differently by a client that the scanner checks all possible variations. If such patterns are not broadly used by benign websites, then the additional overhead imposed on the scanner will be relatively small. Our results show that the gred URL scanner already does something like this for $\%3F$ (?) and $\%23$ (#). Another option is that scanners take the URLs actually sent by the browser to web servers as-is and use them for scanning. However, this option may violate a user's privacy, as the URL may contain private information and exposes the sites the user visits.

7 Related Work

Identifying malicious web sites before the user visits them to block them, take them down, etc. has been a popular area of research. A score of techniques are used to identify malicious content, using both dynamic and static analysis techniques. While not being exhaustive, we attempt to discuss some of the most prominent works here. Note that the security problem highlighted in this paper does not relate to the techniques and methods used to detect malicious content, such as malware, exploits, and phishing sites. Instead, it has to do with they way users and security systems obtain and parse URLs. That is, security issues arise because an attacker can use a URL to hide his malicious content from a security system, while the client, usually a browser, reaches malicious content through the same URL. Some of the works described below do involve the URL in the classification of web pages used to detect malicious content. It is possible that these approaches could be extended to include heuristics that identify problematic URL patterns as potentially malicious, however, the effectiveness of such measures also depends on how frequently such patterns are encountered on benign sites.

Cova et al. [20] present JSand, a dynamic analysis system that visits web sites using an instrumented browser, collecting run-time events as the browser executes the website. Anomaly detection methods are applied on features extracted from the events to classify websites and identify malicious ones. JSand is part of the Wepawet scanner, which we tested in this work, and utilizes Mozilla's Rhino interpreter. This is probably the reason it processes backslashes in a Firefox-like manner. Prophiler [18] later improves JSand by accelerating the process of scanning web pages by allowing for benign pages to be quickly identified and filtered out. Features extracted from page content, the URL, and information about the host of the page are used to quickly identify benign pages. EvilSeed [29] follows the reverse direction and begins from known malicious websites, which it uses as seeds to search the web more efficiently for malicious content. This is accomplished by extracting terms that characterize the known-to-be malicious sites and using them to query search engines, hence, obtaining results more likely to be malicious or compromised.

In 2007, Google researchers introduced a system for identifying all malicious pages on the web that attempt to exploit the browser and lead to drive-by downloads [41]. Based on the fact that Google already crawls a big part of the web, the researchers begun an effort to extract a subset of suspicious pages that can be more thoroughly scanned. Simple heuristics are used to greatly reduce the number of pages that need to be checked. In a later paper, Provos et al. [40] present results showing the prevalence of drive-by download attacks, using features such as out-of-place inline frames, obfuscated JavaScript, and links to known malware distribution sites to detect them. Their findings estimate that 1.3% of search queries made to Google returned at least one URL labeled as malicious.

Dynamic analysis techniques that scan the web to identify malicious pages, frequently employ client honeypots. That is, a modified collection of programs that act as a user operating a browser to access a web site. Moshchuk et al. [44] developed Strider HoneyMonkeys, a set of programs that launch browsers with different patch levels, concurrently accessing the same URL, to detect exploits. The approach is based on detecting the effects of a compromise, like the creation of new files, alteration of configuration files, etc.

Some recent works that aim to improve the detection of malicious websites include JStill [47], which performs function invocation-based analysis to extract features from JavaScript code to statically identify malicious, obfuscated code. Kapravelos et al. [30] also focused on detecting JavaScript that incorporates techniques to evade analysis. Another approach, Delta [16], relies on static analysis of the changes between two versions of the same web site to detect malicious content.

Some other works have focused on aspects of the URL itself to detect malicious sites. ARROW [48] looks at the redirection chains formed by malware distribution networks during a drive-by download attack. Garera et al. [25] classify phishing URLs using features that include red-flag keywords in the URL, as well as feature based on Google's page rank algorithm. Statistical features and lexical and host-based features of URLs have been also used in the past to identify malicious URLs with the help of machine learning [33, 34, 46]. Malicious URLs are frequently hidden by using JavaScript to dynamically generate them on-the-fly. Wang et al. [43] employ dynamic analysis to be extracted such hidden URLs.

Besides the URL scanners mentioned in this paper, there exist another type of scanner called Web Application Scanners. The Web Application Scanner is a kind of scanner that is fed with a URL or a set of URLs, retrieves the pages that URLs pointed to, follows the links inside until identifying all the reachable pages in the application (under a specific domain), analyze the pages with crafted inputs if necessary, and figure out whether this site is vulnerable to some web-specific vulnerabilities (e.g., Cross-Site Scripting, SQL injection, Code Injection, Broken Access Controls). Doupé et al. [21] presents an thorough evaluation of eleven this kind of web application scanners by constructing a vulnerable web site and feeding this website to scanners. Khoury et al. [31] evaluate three scanners against stored SQL injection. Bau et al. [15] analyze eight web application scanners and evaluate their effectiveness against vulnerabilities. For this kind of scanners, they are out of the scope of this paper. In our paper, we assume that the web site is controlled by the attacker and the attacker can planted any malicious content into any link belongs to this site while the web application scanners are targeting the benign sites that may potentially be exploited. The web application scanners usually are not capable of detecting malicious content and phishing pages as well.

8 Conclusions

The procedure of developing a common URL parser framework or enforcing a standardization model can be a hard and challenging task for both application and service vendors, due to expeditious changes in the technology field, and variations and gaps among multiple web services.

In this work, we experimentally test all major browsers and URL scanners, as well as various applications that parse URLs. We expose multiple discrepancies on how they actually parse URLs. These differences leave users vulnerable to malicious web content because the same URL leads the browser to one page, while the scanner follows the same URL to scan another page.

As far as we are aware of, this is the first time browsers and URL scanners have been cross-evaluated in this way. The current work can be used as a reference to anyone interested in better understanding the facets of this fast evolving area. It is also expected to foster research efforts to the development of fully-fledged solutions that put emphasis mostly to the technological, but also to the standardization aspect.

Acknowledgements

We want to express our thanks to the anonymous reviewers for their valuable comments. We would also like to acknowledge Paul Spicer's contribution, who initially investigated the problem.

References

1. Uniform resource identifier (URI): Generic syntax (January 2005), `https://www.ietf.org/rfc/rfc3986.txt`
2. Different behaviours of treating (backslash) in the url by FireFox and Chrome. stackoverflow (May 2012), `http://stackoverflow.com/questions/10438008/different-behaviours-of-treating-backslash-in-the-url-by-firefox-and-chrome`
3. gred (Mar 2015), `http://check.gred.jp/`
4. Online link scan – scan links for harmful threats! (2015), `http://onlinelinkscan.com/`
5. PhishTank — join the fight against phishing (2015), `http://www.phishtank.com/`
6. scumware.org - just another free alternative for security and malware researchers (2015), `http://www.scumware.org/`
7. Stopbadware — a nonprofit organization that makes the web safer through the prevention, mitigation, and remediation of badware websites. (May 2015), `https://www.stopbadware.org/`
8. Sucuri sitecheck - free website malware scanner (Mar 2015), `https://sitecheck.sucuri.net/`
9. urlquery.net - free url scanner (Mar 2015), `http://urlquery.net/`
10. VirusTotal – free online virus, malware and URL scanner (2015), `https://www.virustotal.com/en/`
11. web inspector – inspect, detect, protect (2015), `http://app.webinspector.com/`
12. Website/url/link scanner safety check for phishing, malware, viruses - scanurl.net (Mar 2015), `http://scanurl.net/`
13. Zscaler zulu url risk analyzer - zulu (Mar 2015), `http://zulu.zscaler.com/`
14. Akhawe, D., Felt, A.P.: Alice in warningland: A large-scale field study of browser security warning effectiveness. In: Proceedings of the 22th USENIX Security Symposium. pp. 257–272 (2013)
15. Bau, J., Bursztein, E., Gupta, D., Mitchell, J.: State of the art: Automated black-box web application vulnerability testing. In: Security and Privacy (SP), 2010 IEEE Symposium on. pp. 332–345 (May 2010)
16. Borgolte, K., Kruegel, C., Vigna, G.: Delta: Automatic identification of unknown web-based infection campaigns. In: Proceedings of the ACM SIGSAC Conference on Computer & Communications Security (CCS). pp. 109–120 (2013)

17. Burns, J.: Cross site request forgery: An introduction to a common web application weakness. White paper, Information Security Partners, LLC. (2007)
18. Canali, D., Cova, M., Vigna, G., Kruegel, C.: Prophiler: A fast filter for the large-scale detection of malicious web pages. In: Proceedings of the International Conference on World Wide Web (WWW). pp. 197–206 (2011)
19. Cass, S.: The 2015 top ten programming languages, `http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages`
20. Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious JavaScript code. In: Proceedings of the International Conference on World Wide Web (WWW). pp. 281–290 (2010)
21. Doupé, A., Cova, M., Vigna, G.: Why johnny cant pentest: An analysis of black-box web vulnerability scanners. In: Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 111–131. Springer (2010)
22. Egele, M., Wurzinger, P., Kruegel, C., Kirda, E.: Defending browsers against drive-by downloads : mitigating heap-spraying code injection attacks. In: Proceedings of the International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) (2009)
23. Egelman, S., Cranor, L.F., Hong, J.: You've been warned: An empirical study of the effectiveness of Web browser phishing warnings. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI). pp. 1065–1074 (2008)
24. FireEye: Email security – detect and block spear phishing and other email-based attacks (May 2015), `https://www.fireeye.com/products/ex-email-security-products.html`
25. Garera, S., Provos, N., Chew, M., Rubin, A.D.: A framework for detection and measurement of phishing attacks. In: Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM). pp. 1–8 (2007)
26. Google: Safe browsing API – Google developers (2015), `https://developers.google.com/safe-browsing/`
27. Ikinci, A., Holz, T., Freiling, F.: Monkey-Spider: Detecting malicious websites with low-interaction honeyclients. In: Proceedings of Sicherheit, Schutz und Zuverlässigkeit (2008)
28. Imperial-Legrand, A.: Vulnerability writeups. Google+ (March 2014), `https://plus.google.com/+AlexisImperialLegrandGoogle/posts/EQXTzsBVS7L`
29. Invernizzi, L., Benvenuti, S., Cova, M., Comparetti, P.M., Kruegel, C., Vigna, G.: EvilSeed: A guided approach to finding malicious web pages. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy. pp. 428–442 (2012)
30. Kapravelos, A., Shoshitaishvili, Y., Cova, M., Kruegel, C., Vigna, G.: Revolver: An automated approach to the detection of evasive web-based malware. In: Proceedings of the USENIX Security Symposium. pp. 637–652 (2013)
31. Khoury, N., Zavarsky, P., Lindskog, D., Ruhl, R.: An analysis of black-box web application security scanners against stored sql injection. In: Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on. pp. 1095–1101 (Oct 2011)
32. Kirda, E.: Cross site scripting attacks. In: van Tilborg, H., Jajodia, S. (eds.) Encyclopedia of Cryptography and Security, pp. 275–277. Springer US (2011), `http://dx.doi.org/10.1007/978-1-4419-5906-5_651`
33. Ma, J., Saul, L.K., Savage, S., Voelker, G.M.: Beyond blacklists: Learning to detect malicious web sites from suspicious URLs. In: Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD). pp. 1245–1254 (2009)
34. Ma, J., Saul, L.K., Savage, S., Voelker, G.M.: Identifying suspicious URLs: An application of large-scale online learning. In: Proceedings of the International Conference on Machine Learning (ICML). pp. 681–688 (2009)

35. Microsoft: Microsoft security intelligence report, volume 13. Technical report, Microsoft Corporation (2012)
36. Microsoft: Smartscreen filter (2015), http://windows.microsoft.com/en-us/internet-explorer/products/ie-9/features/smartscreen-filter
37. Moshchuk, A., Bragin, T., Deville, D., Gribble, S.D., Levy, H.M.: Spyproxy: Execution-based detection of malicious web content. In: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium. pp. 3:1–3:16. SS'07, USENIX Association, Berkeley, CA, USA (2007), http://dl.acm.org/citation.cfm?id=1362903.1362906
38. proofpoint: Targeted attack protection (May 2015), https://www.proofpoint.com/us/solutions/products/targeted-attack-protection
39. Protalinski, E.: These 8 characters crash Skype, and once they're in your chat history, the app can't start (update: fixed). VentureBeat (May 2012), http://venturebeat.com/2015/06/02/these-8-characters-crash-skype-and-once-theyre-in-your-chat-history-the-app-cant-start/
40. Provos, N., Mavrommatis, P., Rajab, M.A., Monrose, F.: All your iFRAMEs point to us. In: Proceedings of the USENIX Security Symposium. pp. 1–15 (2008)
41. Provos, N., McNamee, D., Mavrommatis, P., Wang, K., Modadugu, N.: The ghost in the browser analysis of web-based malware. In: Proceedings of the Workshop on Hot Topics in Understanding Botnets (HOTBOTS) (2007)
42. Symantec: Symantec Web Security.cloud (2015), http://www.symantec.com/web-security-cloud/
43. Wang, Q., Zhou, J., Chen, Y., Zhang, Y., Zhao, J.: Extracting URLs from JavaScript via program analysis. In: Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE). pp. 627–630 (2013)
44. Wang, Y.M., Beck, D., Jiang, X., Verbowski, C., Chen, S., King, S.: Automated web patrol with Strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. In: Proc. of NDSS (February 2006)
45. WHATWG: URL living standard (May 2015), https://url.spec.whatwg.org/
46. Whittaker, C., Ryner, B., Nazif, M.: Large-scale automatic classification of phishing pages. In: Proc. of NDSS (February 2010)
47. Xu, W., Zhang, F., Zhu, S.: JStill: Mostly static detection of obfuscated malicious javascript code. In: Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY). pp. 117–128 (2013)
48. Zhang, J., Seifert, C., Stokes, J.W., Lee, W.: ARROW: GenerAting signatuRes to detect dRive-by dOWnloads. In: Proceedings of the International Conference on World Wide Web (WWW). pp. 187–196 (2011)