Toward Secure Embedded Web Interfaces

Baptiste Gourdin
LSV ENS-Cachan
gourdin@lsv.ens-cachan.fr

Chinmay Soman
Stanford University
cpsoman@stanford.edu

Hristo Bojinov
Stanford University
hristo@cs.stanford.edu

Elie Bursztein
Stanford University
elie@cs.stanford.edu

Abstract

We address the challenge of building secure embedded web interfaces by proposing *WebDroid*: the first framework specifically dedicated to this purpose. Our design extends the Android Framework, and enables developers to create easily secure web interfaces for their applications. To motivate our work, we perform an in-depth study of the security of web interfaces embedded in consumer electronics devices, uncover significant vulnerabilities in all the devices examined, and categorize the vulnerabilities. We demonstrate how our framework's security mechanisms prevent embedded applications from suffering the vulnerabilities exposed by our audit. Finally we evaluate the efficiency of our framework in terms of performance and security.

1 Introduction

Virtually all network-capable devices, including simple consumer electronics such as printers and photo frames, ship with an embedded web interface for easy configuration. The ubiquity of web interfaces can be explained by two key factors. For end users, they are easy to use because the interaction takes place in a familiar environment: the web browser. For device manufacturers, providing a web-based interface is cheaper than developing and maintaining custom software and installers.

Though web interfaces are clearly an effective solution from a usability perspective, considerable expertise is required to make them secure [50]. Our first security audit of embedded web interfaces ([7]) provided the initial impetus for our work. To underscore the impact of these earlier results, we point out that compromising a networked device can be used as a stepping stone towards compromising the local network [45]. For example, compromising a photo frame in an office building can lead to an infection of a Web browser connecting to the photo frame. The infection can subsequently spread to the entire local network, and also result in privacy breaches [8]. For instance a router web interface can be exploited to steal remotely the WiFi WPA key and gain access to the entire network. Mitigating the threats posed by embedded devices, including routers, is becoming a critical task, as pointed out repeatedly in recent work [7, 45, 19, 27]. In the absence of a reference framework for building embedded web interfaces each vendor is forced to develop its own stack, which usually leads to security problems. This work takes the initial studies a step further and proposes a solution that uniformly addresses all of the known sources of vulnerabilities in embedded web applications.

We have chosen to build our reference implementation as an Android application for several reasons. First, Android has quickly become the premier open embedded operating system on the market, shipping not only on tens of millions of smart-phones every year, but also on specialized devices such as the Nook e-book reader by Barnes&Noble. Second, Android's de facto bias towards the ARM architecture makes the operating system suitable for embedding in other consumer devices such as cameras, photo frames, and media hubs. Third, the security architecture adopted by Android is particularly well-suited for embedded single-user devices as it casts the system security question into one of effectively isolating concurrent, possibly vulnerable applications.

Our main contribution in this paper, *WebDroid* [16], is the first open-source web framework specifically designed for building secure embedded web interfaces:

- *WebDroid* is designed, implemented and evaluated based on the knowledge we gained by auditing more than 30 web embedded devices' web interfaces over the two last years, and the more that 50 vulnerabilities we discovered on these devices.

- *WebDroid* is a novel composition of security design principles and techniques with a simple and intuitive configuration interface where most of the security mechanisms are enabled by default—including location and network address restrictions, as well as server-side CSP and frame-busting.

- *WebDroid* also features application-wide authentication that ensures that every embedded web application will have a secure login and logout mechanism which is resistant to attacks, including brute-forcing and session hijacking.

Similar to previous work done on building secure web servers (e.g., the OKWS server [29]), our framework separates the core web server components from the applications to protect against low level attacks. Unlike previous systems however, our framework also mitigates all of the known application-level attacks including XSS (Cross-Site Scripting) [13], CSRF (Cross Site Request Forgery) [50], SQL injection [50] and Clickjacking [44].

The remainder of the paper is organized as follows: in Section 2 we briefly go through the background necessary to understand this work. In Section 3 we present and categorize the vulnerabilities we found during our audit work. Section 4 develops the threat model that we address with our system design depicted in Section 5. In Section 6 we highlight the main defense mechanisms that are employed in our implementation. Section 7 presents the user interface for managing web applications. Section 9 discusses two application case studies and describes how *WebDroid* security mechanisms help to mitigate vulnerabilities. In Section 10 we provide a summary of relevant related work, and Section 11 concludes the paper.

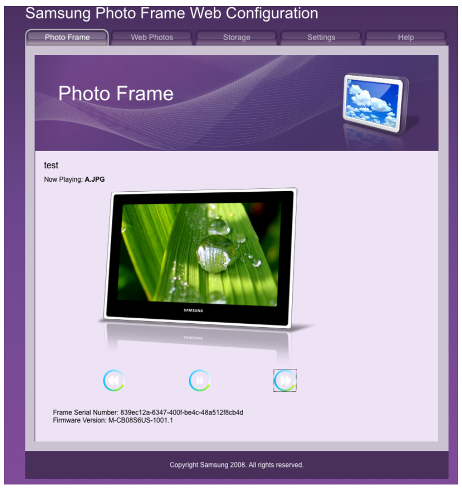
2 Background

The embedded device market is growing rapidly. For example, in the 4th quarter of 2008, 7 million digital photo frames were sold, almost 50% more than in the 4th quarter of 2007. Similarly, analysts forecast that by 2012, 12 million Network Attached Storage (NAS) devices will be sold each year. At the current pace, devices with embedded web servers will outnumber traditional web servers in less than 2 years; Netcraft reported that there are roughly 40 millions active web servers on the Internet in June 2009 [35].

In order to differentiate their products from those of their competitors, vendors are constantly adding novel features to their products, such as BitTorrent support in NAS devices.

As the number of features increases, a need for a powerful management interface on the device rapidly arises. To offer this in an intuitive, convenient, and cost effective way, vendors have started to embed web interfaces in their products. While the most well known use of these web interface is to configure network equipments such as WiFi access points and routers, many other embedded devices include web interfaces. For instance digital photo frames are an excellent example of this expansion of features and need for a rich configuration interface. Thus, it is safe to say that web interfaces have become the norm in managing embedded devices.

Our audit uncovered abundant examples of features that were hastily implemented and vulnerable to web attacks. For example the Flickr integration in digital photo frames led to XSS attacks. What is especially troublesome is the fact that we found CSRF exploits in managed network switches aimed for datacenter use. Attacks on such devices could allow remote users to reboot them and effectively DoS an entire company intranet in one step.



test

Now Playing: **A.JPG**

Frame Serial Number: 839ec12a-6347-400f-be4c-48a512f8cb4d
Firmware Version: M-CB08S6US-1001.1

Copyright Samsung 2008. All rights reserved.

Figure 1: The web interface embedded into a Samsung photo frame.

Figure 1 is a screenshot of the interface embedded in a high-end Samsung photo frame. This interface allows the user to control the frame's display remotely, add an Internet photo feed to be displayed on the frame, and to find out various statistics. Although at first sight this interface looks perfectly designed, we found out that in reality it is completely flawed: for example, it is possible to bypass the authentication process to view photos and it is possible to inject an exploit via a CSRF and XSS vulnerability that allows to extract photos and send them to a remote server.

3 Embedded Web Application Security: State of the Art

Over the last two years we audited the web interfaces for more than 30 embedded devices. In this section we report our audit results and discuss the insights we gained from them. These results and insights are later used to justify and guide the design of our framework security features. Note that although we discussed some of the vulnerabilities we found in a previous publication [8], this is the first time that the complete audit results are reported and discussed.

3.1 Audit coverage

The eight categories of devices we tested are: *lights-out management (LOM) interfaces* (these typically allow the administrator to power cycle a PC or control network access, bypassing the OS), *NAS* (used for shared storage accessible via Ethernet), *photo frames* (we focused on "smart" frames with network connectivity), *routers/access points* (probably the most familiar browser-managed class of consumer device), *IP cameras* (with video feeds that can be accessed over the network), *IP phones* (especially those with a web-based management interface), *switches* ("managed switches" that expose some configuration options), and *printers* (the larger ones usually have a HTTP-based interface used to configure a variety of functions, including access via e-mail). The eight device categories spanned seventeen brands: Table 1 shows which types of devices were tested for each brand. As one can see we did test devices from vendors specialized in one type of product such as *Buffalo*, and from vendors that have a wide range of products such as *D-link*.

3.2 Vulnerability classes

XSS. As a warm-up we started by testing for Type 2 (stored) cross-site scripting (XSS) vulnerabilities [13], which are common in web applications. Most devices are vulnerable, including those that perform some input checking. For example, the TrendNet switch ensures that its system location field does not contain spaces, but does not prevent attacks of the form:

```
 loc");document.write("<script/src=
'http://evil.com/a.js'></sc"+"ript>.
```

XSS attacks are particularly dangerous on embedded devices because they are the first step toward a persistent reverse XCS, as discussed below.

CSRF. Cross-site request forgery [50] enables an attacker to compromise a device by using an external web site as a stepping stone for intranet infiltration. On embedded devices it can also be used as a direct vector of attack as it allows the attacker to reboot critical network equipments such as switches, IP phones and routers. Finally we used CSRF as a way to inject Type 2 (stored) XSS and reverse XCS [9] payloads.

File security. For each device, we checked whether it was possible to read or inject arbitrary files. Some devices, such as the Samsung photo frame, allow the attacker to read protected files without being authenticated. On this device, even when the Web interface was protected by a password, it was still possible to access the photos stored in memory by using a specially crafted URL. On other devices, the Web interface could be compromised by abusing the log file.

User authentication. Most devices have a default password or no password at all. Additionally, most devices authenticate users in cleartext (i.e. without HTTPS). This was even true for several security cameras, which is surprising given that they are intended to securely monitor private spaces. We even found that some NAS and photo frames do not properly enforce the authentication mechanism and it is possible to access the user content (i.e. photos) without being traced in the logs. Similarly, nothing is done at the network level to prevent session hijacking as the traffic is in clear and the cookies are sent over HTTPS. Finally as far as we can tell not a single device implements a password policy or an anti-brute force defense.

Clickjacking attacks. Clickjacking attacks [18] are the most recent, and most overlooked attack vectors as all devices were vulnerable to them. While at first sight this does not appear to be a big issue, it turns out that being able clickjack an embedded interface gives a lot of leverage to the attacker. For example basic Clickjacking can be used to reboot devices, erase their content and in the case of routers, enable guest network access. Advanced Clickjacking [49] as demonstrated by Paul Stone at BlackHat Europe 2010 allows the attacker to steal the router WPA key or the NAS password.

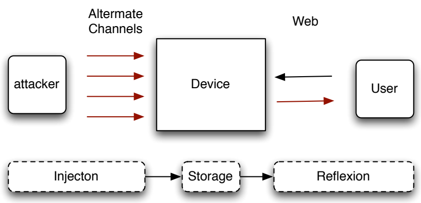


Figure 2: Overview of an XCS attack.

Brand	**Camera**	**LOM**	**NAS**	**Phone**	**Photo Frame**	**Printer**	**Router**	**Switch**
Allied								✓
Buffalo			✓				✓	
Belkin							✓	
D-Link	✓		✓				✓	
Dell		✓						
eStarling					✓			
HP						✓		
IBM		✓						
Intel		✓						
Kodak					✓			
LaCie			✓					
Linksys	✓		✓	✓			✓	
Netgear							✓	✓
SMS networks							✓	
Panasonic	✓							
QNAP			✓					
Samsung	✓							
SMC								✓
TrendNet							✓	✓
ZyXEL							✓	

Table 1: List of devices by brand.

XCS. A *Cross-Channel Scripting* attack [9] comprises two steps, as shown in Figure 2. In the first step the attacker uses a non-web communication channel such as FTP or SNMP to store malicious JavaScript code on the server. In the second step, the malicious content is sent to the *victim* via the Web interface. XCS vulnerabilities are prevalent in embedded devices since they typically expose multiple services beyond HTTP. XCS bugs often affect the interaction between two specific protocols only (such as the combination of HTTP and BitTorrent), which can make them harder to detect.

Reverse XCS. In a *Reverse XCS* attack the web interface is used to attack another service on the device. We primarily use reverse XCS attacks to exfiltrate data that is protected by an access control mechanism.

We did not look for SQL injections [21], as it was unlikely that the audited devices would contain a SQL server. However we still consider SQL injection attack to be a potential threat and therefore our framework has security mechanisms in place to mitigate them. Finally, while in some cases we found weaknesses in the networking stack (for example: predictable Initial Sequenced Numbers), we do not discuss that topic here.

3.3 Tools used

The audit of each device was done in three phases. First, we performed a general assessment using *NMap* [31] and *Nessus* [42]. Next, we tested the web management interface using *Firefox* and several of its extensions: *Firebug* [20], *Tamper Data* [26], and *Edit Cookies* [51]. We used a custom tool for CSRF analysis. In the third phase we tested for XCS using hand written scripts and command line tools such as *smbclient*.

3.4 Audit results

Table 2 summarizes which classes of vulnerabilities were found for each type of device. We use the symbol □when one device is vulnerable to this class of attacks and ■when multiples devices in the class are vulnerable. The second column from the left indicates the number of devices tested in that category. We survey the most interesting vulnerabilities in the next section.

Table 2 shows that the NAS category exhibits the most vulnerabilities, which can be expected given the complexity of these devices. We were surprised by the large number of vulnerabilities in photo frames, which are relatively simple devices.

Type	# Devices	XSS	CSRF	XCS	RXCS	File	Auth
LOM	3	■	■	■			■
NAS	5	■	■	■	■	■	■
Photo frame	3	■	■	■	□	□	■
Router	8	■	■	□		■	■
IP camera	3		■			□	■
IP phone	1	□	□	□			□
Switch	4	■	■	■			■
Printer	1	□	□		□		□

Table 2: Vulnerability classes by device type.

A possible explanation is that vendors rushed to market in order to grab market share with new features. Indeed, in the Kodak photo frame, half the Web interface is protected against XSS while the other half is completely vulnerable. IP cameras and routers are more mature, and therefore tend to have a better security. Table 2 also shows that even enterprise-grade devices such as switches, printers, and LOM are vulnerable to a variety of attacks, which is a concern as they are usually deployed into sensitive environments such as server rooms.

4 Threat Model

Our audit showed that embedded web management interfaces pose a serious security threat and are currently one of the weakest links in home and office networks. In this section we formalize our attacker model and the security objectives that our framework aims at achieving.

4.1 Attacker model

In this paper, we are concerned with securing embedded web interfaces from malicious attackers. Inspired by the threat model of [6] we are using the ”web attacker” concept with slightly more powerfully attacker as we allow the attacker to interact directly with the web framework like in the active attacker model. Accordingly our attacker model is defined as follows: we assume an honest user employs a standard web browser to view and interact with the embedded web interface content. Our malicious web attacker attempts to disrupt this interaction or steal sensitive information such as a WPA key. Typically, a web attacker can attempt to do this in two ways: by trying to exploit directly a vulnerability in the web interface, or by placing malicious content (e.g. JavaScript) in the user’s browser and modifying the state of the browser, interfering with the honest session. We allow the attacker to attempt to directly attack the web framework in any way he likes; in particular, we assume that the attacker will attempt to DDOS the web server, find buffer overflow exploits or brute force the authentication. Finally, we also assume that the attacker will be able to manipulate any non-encrypted session to his advantage.

4.2 Security objectives

Based on our audit evaluation and the attacker model described above we now formalize what security objectives our framework aims at achieving. These goals fall into four distinct umbrella objectives that cover all of the known attacks against a web interface.

Enforcing access control. The first goal of our framework is to ensure that only the right principals have access to the right data. Access control enforcement needs to be enforced at multiple levels. First, at the network level, our framework needs to ensure that the web interface is only available in the right physical or network location and to the right clients. At the application level, it means that the framework needs to ensure that every web resource is properly protected and that the attacker can not brute-force user passwords. Finally, at the user level it also means that the framework offers to the user the ability to declare whether a specific client is allowed to access a given web application.

Protecting session state. Protecting session state ensures that once a session is established with the framework, only the authenticated user is accessing the session. At the network level, protecting the session state implies preventing man in the middle attacks by enforcing the use of SSL. At the HTTP level, protecting the session means protecting the session cookies from being leaked over HTTP (as in the Sidejacking attack) or being read via JavaScript (XSS).

Deflecting direct web attacks. Deflecting direct web attacks requires that our framework is not vulnerable to buffer overflow or at least that the privileges gained in case of successful exploitation are limited. At the application level, the framework must be able to mitigate XSS [13], and SQL injection attacks [21].

Preventing web browser attacks. In order to prevent web browser attacks, the framework has to work with the browser to ensure that the attacker cannot include in a web site a piece of code (such as an iframe or JavaScript) that can abuse the trust relation between the browser and the web interface. These attacks are instances of the confused deputy problem [6]. They include CSRF and Clickjacking attacks.

5 System Overview

In this section we discuss the design principles behind our framework, provide an overview of how the framework works and describe how a web request is checked and processed.

5.1 Design principles

To address the threat model presented in the previous section, our framework is architected around the following four principles:

Secure by default. The team in charge of building an embedded web interface is usually not security savvy and is likely to make mistakes. To cope with this lack of knowledge our framework is designed to be secure by default, which means that every security feature and check is in place and it is up to the developers to make them less restrictive or turn them off. For instance, our default CSP [14] (content security policy) only allows content from *self*, which means that no external content will be allowed to load from a page in the web interface. Similarly the framework uses whitelists for input filtering: by default only a restricted set of characters is allowed in URL parameters and POST variables, and it is up to the developer to relax this whitelist if needed. As a final example, the framework injects JavaScript frame-busting code and the X-Frame-Option header in all the pages in order to prevent Clickjacking attacks. In the unlikely situation where the interface needs to be embedded in another webpage, the developer must turn the defense mechanism off.

Defense in depth. Since there is no universal fix for many types of attacks, including XSS, CSRF, and Clickjacking, our framework follows the defense in depth principle and implements all the known techniques to try and mitigate each threat as much as possible. We perform filtering and security checks at input, during processing, and during output.

Least privilege. Following the OKWS design [29], we implement the least privilege principle by leveraging the Android architecture. Each application and the framework have separate user IDs and sets of permissions; this guarantees that if the framework or one of the applications is compromised, the attacker will not take complete ownership of the data. For instance by taking over the framework one does not gain access to the phone contacts list used by one of the applications: our framework only has the network privilege. Note that the application developer must modularize his or her application to fully benefit from the least privilege design. Product features that can significantly modify device functionality, such as by executing a firmware upgrade, need to receive special consideration as well perhaps resulting in additional backend checks performed in advance.

User consent. Our last design principle is ”user consent as permission”: we let the user make the final decisions about key security policies. For example, when a new web client wants to access one of the phone web applications, it is up to the user to allow this or not because only she knows if this request is legitimate. Similarly, when the user installs a new web application, she is asked if she wants to be prompted for approval each time a client connects to that application. Finally, at install time we also provide the user with a summary of the security features that have been disabled. The user can then decide if the presented security profile is acceptable or not. While users can generally not be relied on for ensuring system security, we implement the user consent principle in order to catch potential security issues that clearly defeat common sense.

5.2 Server architecture

As shown in Figure 3, the framework is composed of four blocks and architected like the iptables firewall with a series of security checks performed at input time, and another series during output.

The Dispatcher is responsible for forwarding an HTTP request to the desired application. The forwarding decision is based on the unique port number assigned to every application. Separating applications by port number allows greater granularity for doing data encryption which is specific to every application. In addition to forwarding, the Dispatcher is also responsible for policy based enforcement of security mechanisms.

The Configuration Manager handles per-application tuning of the security policies. When an application is first registered with the web server, all the security mechanisms are turned on by default. The administrator can then enable or disable individual mechanisms using the configuration interface. The resulting configuration is captured in a database and made available to the Dispatcher for policy enforcement.

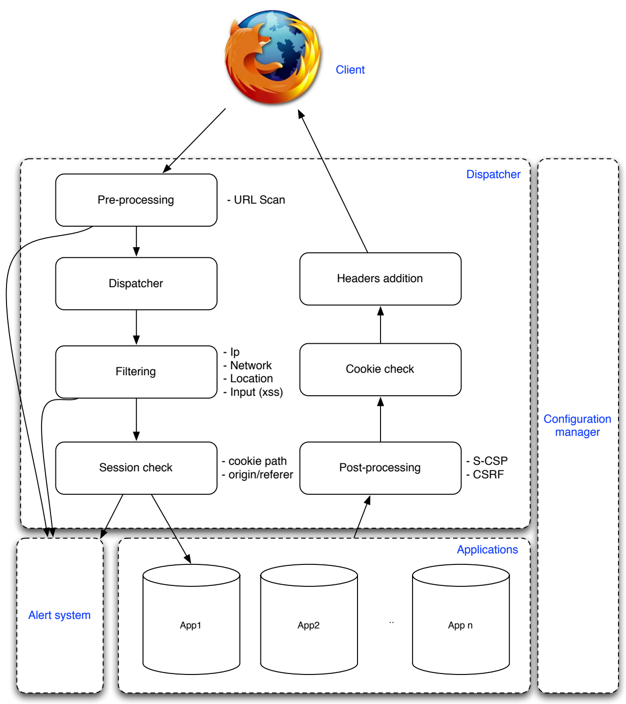


Figure 3: Overview of the framework design showing the interaction of the different web server components (dispatcher, applications, and alert system) involved in the processing a client request.

The Alert System is used to control how the administrator is to be notified for different events. For instance, the administrator may want to be explicitly alerted for every new client connection. The Alert System also handles notifications caused by malicious web requests as detected by the Dispatcher. Notifications can either be passive or active depending on whether they need approval from the administrator.

Finally, the framework also provides an API for efficiently implementing web applications. The core functionality includes methods to handle HTTP requests and generate the response. It also provides handlers with build in security mechanisms for content generation such as HTML components, CSS, JavaScript, JSON etc. For instance, the HTML, XML and JSON handlers provide parameterized functions required to escape dynamic content before being added to the rendered page. In addition, the framework provides methods for allowing applications to construct HTTPOnly or secure cookies.

5.3 Request processing

As depicted in Figure 3 a new web request goes through a series of input security checks and processing, and is subsequently forwarded to the actual application. The response generated is subjected to another iteration of checks and processing before being sent to the client. If any check fails then the processing is aborted and a notification is sent via the Alert System.

The pre-processing step performs two rounds of security checks. First, the origin of the request is compared to the client restriction policy in order to block queries coming from unwanted sources. Second, the HTTP query is validated through regular expression whitelists. The corresponding web application is then identified (based on the port number) and the session and CSRF tokens validation checks can be done.

After validation, the request is sent to the web application which generates a page using our framework and sends it back to the web server. Before reaching the network, the response is passed through post-processing security mechanisms like S-CSP and CSRF token generation. This usually results in the inclusion of additional headers and modification of certain HTML elements. The result is then returned to the client.

6 Security Mechanisms

A broad range of mechanisms and best practices have been developed over the last few years to counter the most severe web security problems. It is clear that no single technique or framework will make a web application secure. In addition, expecting developers to understand and deploy all of these mechanisms on their own is unrealistic. Table 3 maps the mechanisms that we embed in our secure web server implementation against the threats they are designed to mitigate. We now describe each security mechanism and provide further references. Note that in many scenarios we depend on a correct browser implementation for security capabilities. Wherever possible, we use additional mechanisms that can add security even if the browser is not up-to-date or compliant.

HTTPOnly cookies. Many XSS vulnerabilities can be mitigated by reducing the amount of damage an injected script can inflict. HTTPOnly cookies [33] achieve this by restricting cookie values to be accessible by the server only, and not by any scripts running within a page. In practice, most cookies used in web application logic are inherently friendly to this concept, and this is why we have chosen to build it in. (HTTPOnly cookies are not implemented by Android HttpCookie.)

Category	Access control		Session		Direct attack				Browser attack	
Defense/Threat	Bypass	Pass guess	MITM	Hijack	XSS	SQLi	XCS	RXCS	CSRF	Clickjack
HTTP only cookie				✓	✓			✓		
Server side input filtering					✓	✓		✓		
CSP					✓		✓			
S-CSP					✓		✓			
CSRF random token								✓	✓	
Origin header verification								✓	✓	
X-FRAME-OPTION										✓
JS frame-busting code										✓
SSL			✓	✓						
HSTS			✓	✓						
Secure cookie				✓						
Parametrized queries						✓				
URL scanning										
Application-wide auth	✓									
Password policy		✓								
Anti brute-force		✓								
Restrict network/location	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DOS protection										

Table 3: Threats and corresponding security mechanisms

Server-side input filtering. Even though filtering or whitelisting of user input can fail if implemented incorrectly [3, 2, 1], it is still very important to sanitize user data before web pages are rendered with it. Input filtering can prevent scripting exploits as well as SQL injections. When applied to data coming from other embedded services, input filtering can also prevent many XCS attacks.

CSP (Content Security Policy). Pages rendered by the typical embedded web application have little need to contact external web sites. Correspondingly our server is configured to offer restrictive CSP [14] directives to browsers, limiting the impact of any injected code in the page.

S-CSP (Server-side Content Security Policy). For browsers that do not support CSP, we introduce Server-side CSP. While rendering a particular site, the server looks at the CSP directives present in the header (or the policy-uri) and modifies the HTML code accordingly. Instead of standard input filtering, the changes are based on the custom policies defined by the administrator: such as valid hosts for the different HTML elements, use of inline-scripts, eval functionality usage and so on. Its novelty lies in the fact that the resulting HTML page as received by the browser automatically becomes CSP compliant. In addition to filtering, S-CSP can also support reporting of CSP violations via 'report-uri' directive which ordinarily is not possible for incompatible browsers.

X-Frame-Options. Clickjacking is a serious emerging threat which is best handled by preventing web site framing. Since embedded web applications are usually not designed with mash-up scenarios in mind, setting the option to DENY is a good default configuration.

JavaScript frame-busting. Not all browsers support the X-Frame-Options header, and therefore our framework automatically includes frame-busting code in JavaScript. The particular piece of code we use is as simple as possible and has been vetted for vulnerabilities typically found in such implementations [44].

Random anti-CSRF token. Cross-site request forgery is another web application attack which is easy to prevent, but often not addressed in embedded settings. Our framework automatically injects random challenge tokens in links and forms pointing back at the web application, and checks the tokens on page access [39].

Origin header verification. Along with checking CSRF tokens, we make sure that for requests that supply any parameters (either POST or GET) and include the Origin [5] or Referer header, the origin/referer values are as expected. We do this as a basic measure to prevent cross-site attacks. When the Referer header is available, we also check for cross-application attacks, making sure that each application is only accessed through its entry pages.

SSL. Securing network communications often ends up being a low-priority item for application developers, and this is why our web server uses HTTPS exclusively by default, with a persistent self-signed certificate created during device initialization.

HSTS (HTTP Strict Transport Security) and Secure cookies. In addition to supporting SSL out of the box, our server implements the HSTS standard [22] and requests that all incoming connections be over SSL, which prevents several passive and active network attacks [23]. Moreover, browser cookies are created with the Secure attribute, preventing the browser from leaking them to the network in plaintext.

Parametrized rendering and queries. Android already supports parametrized SQLlite queries [52] and we encourage developers to make use of this facility. We have also added the ability to parametrize dynamic HTML rendering, in which case escaping of the output is performed automatically.

URL scanning. Incoming HTTP requests are sanitized by applying filtering similar to that offered by the URLScan tool in Microsoft IIS [34]. Our filter is configured to restrict both the URL and query parts of a request, while changes by the web application developer are allowed if necessary. URLScan is most useful in preventing web application vulnerabilities due to incorrect or incomplete parsing of request data.

Application-wide authentication, password policy, and password anti-bruteforcing. Recognizing that user authentication is often a weak spot for web applications, we have implemented user authentication as part of the web server, freeing the developers from the need to implement secure user session tracking. In addition, the password strength policy can be changed according to requirements, and a mechanism to prevent (or severely slow down) brute-force attacks is always enabled.

Network restrictions. Most embedded web servers have a relatively constrained network access profile: either the device should serve requests only when connected to a specific network or WiFi SSID, or the hosts requesting service might match a profile, such as a specific IP or MAC address. This feature, while easily accessible, can not be configured by default due to the differences in individual application environments.

Location restrictions. Similar to network restrictions, the server can be configured to operate only when the device is at specific physical locations, minimizing the opportunities for an attacker to access and potentially compromise the system.

DDoS. While distributed denial-of-service (DDoS) protection is difficult, we believe that much can be done to mitigate such threats. For most applications, maintaining local service is of top priority, and so we throttle HTTP requests such that those coming from the local network always have a guaranteed level of service. Of course, this can not prevent lower-level network DDoS attacks: these have to be taken care of separately, outside of the web server.

7 User interface

This section briefly describes the user interface required for basic administration of the web server and security policy management. In the following description, we refer to the owner of the smart phone or embedded device as the Admin user.

7.1 Configuration management

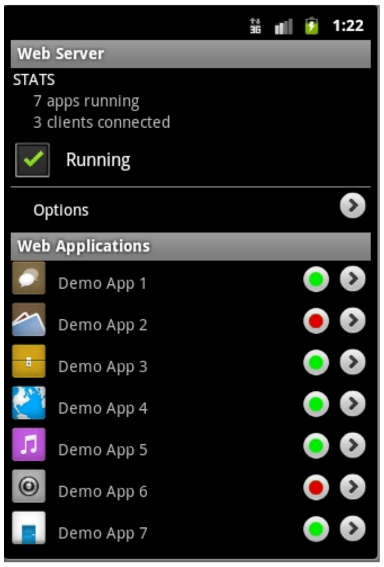
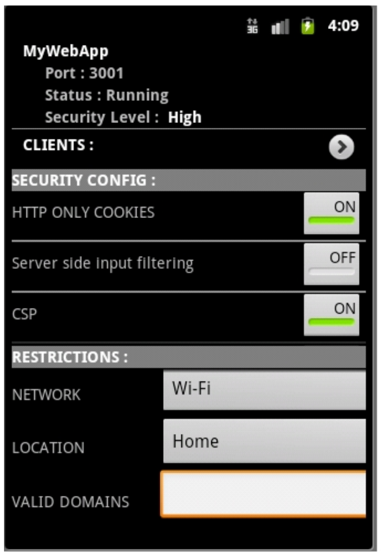


Figure 4: Main web server configuration interface.

This interface is used to control the server settings across all the applications. As shown in Figure 4, it provides the ability to disable each web application. It also displays the web server overall statistics such as the number of active application and the number of active connections session.

Web server logs. Accessible from the menu options, the logged events such as failures, new connections and configuration changes can be visualized.

Settings. From this interface, the Admin overrides some security features in order to enforce certain mechanisms for all applications, irrespective of their individual configuration.


4:09

MyWebApp

Port : 3001

Status : Running

Security Level : High

CLIENTS :

SECURITY CONFIG :

HTTP ONLY COOKIES ON

Server side input filtering OFF

CSP ON

RESTRICTIONS :

NETWORK Wi-Fi

LOCATION Home

VALID DOMAINS

Figure 5: Web application configuration interface, allowing per-application customizations (secure settings highlighted in green).

7.2 Configuration per web application

This interface enables the Admin user to control some web application parameters such as the port number, the application name, and its password or tune the security policy for every application. As shown in Figure 5, it displays the name, path, security level and status information along with the currently enabled security mechanisms. Since all the mechanisms are turned on by default, policy administration is not strictly necessary. However, this allows flexibility in the framework that can be useful in special circumstances. For instance, the Admin user may wish to disable the heavy S-CSP mechanism in the case of a restricted set of trusted users. The different functionalities provided by the interface are described below.

Alarm system configuration. Each new client connection request can be monitored by setting the alarm notification level to one of the three possibilities: Disabled, Passive, or Approval. Both Passive and Approval notifications alert the administrator about the new connection. Approval mode has the additional feature of requiring the Admin user to grant access before proceeding.

Network and location restriction. The web server can restrict clients connecting based on the network properties (serving WiFi or 3G only for example) or based on the current location such as home or office.

Domain whitelist. The Admin can define a list of domains that are allowed in the CSP policy by writing a comma separated list of domains/IP addresses. If this field is empty, the web server will enforce the restrictive 'allow self' policy and block all other sources.

```
<WebServerConf>
<WebApp>
<path>com.android.websms</path>
<Enabled>1</Enabled>
<CSRF>1</CSRF>
<HttpOnlyCookie>1</HttpOnlyCookie>
<XFrame>1</XFrame>
</WebApp>
</WebServerConf>
```

Figure 6: Web server configuration sample

IP whitelist. The Admin user can explicitly allow access for a specific set of trusted hosts by adding a comma-separated list of IP addresses. For a new connection request, if the source IP is in this list then access is permitted regardless of the restrictions described above.

7.3 Configuration without the UI

For embedded devices without a display to access the configuration interface, the web server can be configured through an XML file present in the application package as a raw resource. With this file, the web server administrator can enforce security mechanisms for specific web applications or disable all web application that do not respect some requirements. The web server configuration can also be done after installation by modifying the SQLite database on the device.

8 Implementation

In this section we describe how our system is implemented and how Android applications interact with it. Our system consists of two main components: the *Dispatcher* (a web server that processes and routes requests to applications) and our *framework API* that Android applications can access.

The Dispatcher works as an Android background service. As a starting block we used the Tornado open-source web server that we hardened and modified to work with our framework. The web server follows the least privilege principle, and runs with the minimal permissions set needed to handle HTTP communications: *android.permission.INTERNET*. To be allowed to expose a web interface, an application requests a new permission that we created called *com.android.webserver.WEB_APPLICATION*. This novel permission is more restrictive than *android.permission.INTERNET* and only allows the

```
mountWebContent("websms",
                Home.class);
mountWebContent("websms/send",
                SendSMS.class);
mountWebContent("websms/view",
                SMSHistory.class);
mountWebContent("websms/theme.css",
                RawRessource.class,
                RawRessource.CSS,
                R.raw.hello);
```

Figure 7: WebSMS code used to declare the exposed web interface.

application to serve web requests via the dispatcher.

At launch time the Dispatcher browses the list of installed applications for new ones requesting the web application permission. By retrieving the ContentProvider associated to the framework, it queries the security configuration. Following the consent as permission principle we prompt the user every time a new web application wants to register. When an application set the same URL path than another one, the registration is discarded and a possible malicious application warning is displayed to the user.

The framework API is a Java library that handles communications between the web server and the web application (which run as separate processes). It also provides a set of classes that help generating web content. Similarly to many modern web framework (i.e. Rails), every web page need to registered it web path through a function call, in our case this function is *mountWebContent*. This function bind a path to a java class entry point. For example our WebSMS web application register 4 web pages: 3 HTML pages and 1 CSS stylesheet (Figure 7). Note the use of the *RawRessource.class* which allows developer to expose directly raw data to the web such as CCS files. Our framework provides a set of classes to help building HTML pages, or handling other resources request such as pictures, CSS stylesheets or JavaScript libraries. The java classes Home, SendSMS and SMSHistory extends the framework class HTMLPage which provides various methods to add dynamic content to the pages. In particular the HTMLPage class has the method `appendHTMLContent(content, String[] vars)` that allows to programmatically append content to the page. Text variables are represented by $ which are substituted by the corresponding var string after it is filtered to prevent XSS. While the authors can bypass the filtering process if they want by default it is in place. Similarly, the HTMLPage class ensures that the data passed to the application is properly sanitized and that parametrized SQL queries are used in order to prevent SQL injection.

When an HTTP request is received, it goes through all pre-processing security mechanisms and is dispatched to the corresponding web application. The framework API embeds an Android *ContentProvider* used by the web server to query pages. HTTP headers, body and security tokens are added to the query and then transmitted to the web application. Using the framework API, the web page is build and send back as answer to the query. This one is finally checked by all post-process security mechanisms and send back to the web client.

9 Case Studies

In this section we present two case studies that demonstrate how our framework effectively mitigates web vulnerabilities. We describe the applications we built, their attack surface, how the framework protects them, and finally show that when using off-the-shelf security scanners the framework is indeed able to mitigate the vulnerabilities found in the apps.

To study the effectiveness of our the system we built two sample applications that take advantages of the phone's capabilities to provide useful services: the first one, *WebSMS*, is used for reading and sending SMS from the browser; the second one, WebMedia, provides a convenient web interface to browse and display the photos and videos stored on the smartphone. We argue that these two applications—while limited—are good case studies of what developers might want to built in order to leverage a device's capabilities in the form of web applications.

9.1 Applications

WebSMS. When loaded in a client browser, the user can choose to view the current SMS inbox or send a new one. For the second choice, the application displays a list of contacts fetched from the phone's directory along with a search box. Clicking on a particular contact allows to send a SMS directly from the browser. The SMS content is sent by the browser to the application via a POST request that contains the contact ID.

WebMedia. This application displays a gallery of photos and videos stored on the Android device (Figure 8). When a thumbnail is clicked, a full size view of the media file is displayed. The application provides a convenient way to display photos and videos to friends and family on a big screen. In addition, this application enables seamless sharing of content with trusted users (friends or family).

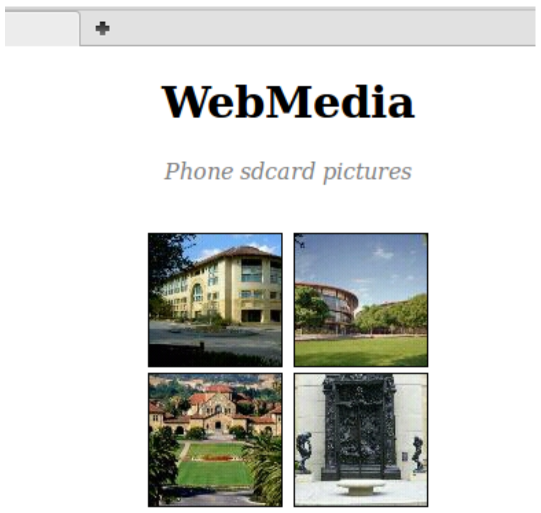


Figure 8: The WebMedia embedded web application.

9.2 Attack surfaces

Without framework support, the web applications suffer from multiple vulnerabilities. In the WebSMS application, the contact search can be a vector for reflected XSS or SQL injection. Also, the capacity to send message and view their contents afterward can lead to a stored XSS in the sending and in the receiving phone. The WebMedia application is vulnerable to CSRF attacks as well. The XSS attack allows the attacker to steal private information as the contact list of the sent and received SMS contents. A CSRF can be conducted to send SMS on behalf of the user, which can lead to embarrassing situations or financial loss. In extreme cases, if the phone is used as a trusted device to authorize sensitive operations such as bank transfers, then the combination of XSS and CSRF attacks will allow a malicious user to bypass this security mechanism and conduct fraudulent operations.

9.3 Security evaluation

In order to evaluate whether our framework is able to mitigate the attacks against our vulnerable applications we have run the web scanners Skipfish and Nexpose against our applications with the framework defense mechanisms off and then on. When the framework defenses are turned off, both Skipfish and Nexpose detected reflected XSS and stored XSS vulnerabilities in the WebSMS application. When the framework defenses are turned on, no vulnerabilities are reported. Note that neither scanner reported the CSRF vulnerabilities.

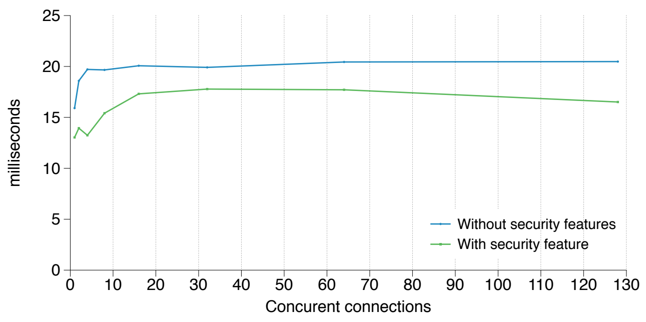


Figure 9: Average number of request per second with and without security features enabled.

This limited experiment shows that our framework can help effectively and transparently mitigate vulnerabilities that may exist in embedded web interfaces even though it can not completely replace good coding practices and careful code review.

9.4 Performance evaluation

While as stated earlier performance should not be the focus of a mobile web framework, we still ran a basic performance evaluation using the Apache benchmark tool to evaluate the impact of enabling security features on *WebDroid* performance. To reflect as accurately as possible real world usage, we ran these benchmarks over WiFi with *WebDroid* on a standard HTC Desire phone with Android 2.3. We were not able to test over 3G as IP are not routable.

WebDroid performance in term of requests per second for the WebSMS application when the number of simultaneous connections increase is reported in figure 9. The figure 10 depicts how fast *WebDroid* is able to process each request as the number of simultaneous connections increase. As visible in the diagrams, *WebDroid* take between a 10% to 30% performance hits when the security features are turned one depending on the number of simultaneous connections. On average *WebDroid* performance take a 20% hit when the security features are enabled. While this performance hit might not be acceptable for a regular website, for an embedded interface we argue that it is acceptable as even when there are 128 simultaneous connections, *WebDroid* is able to serve every request in less than 80 ms which is below what is the optimal user tolerance time: 100ms [37].

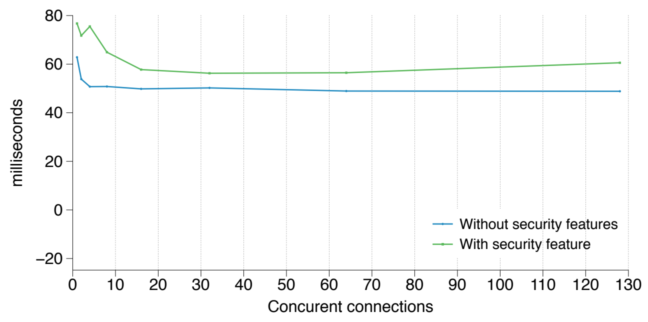


Figure 10: Average time to process a request with and without security features enabled.

10 Related Work

Browser defenses. Mozilla Foundation's *Content Security Policy* (CSP) [14] proposal allows a site to specify restrictions on content served from the site, including which external resources the content can load. The CSP policy is specified as an HTTP header in each HTTP response. For example, the CSP header

```
X-Content-Security-Policy: allow self
```

prevents the content from loading any external resources or executing inline scripts. Replacing "allow self" with "allow whitelist" allows external resources from the given whitelist. Another system, SiteFirewall [9], takes a similar approach but also allows persistent browser-side policy storage (via cookies or other, more secure objects). SiteFirewall is capable of blocking some types of XCS attacks from being completed. The system uses a browser extension that acts as a firewall between vulnerable, internal web sites, and those accessed by the user on the open Internet. A third proposal called SOMA [38] implements a mutual consent policy on cross-origin links. That is, both the embedding and the embedded content must agree to the action being initiated. As with CSP, SOMA is implemented as a content-specific policy rather than a global site policy. Finally Content Restrictions [32] is another approach to defining content control policies on web sites.

Frameworks. Generic web frameworks, such as *Ruby on rails* [41] and *Django*, implement numerous features such as built-in CSRF defenses that help developers to build secure web interfaces more easily. However this kind of generic framework is very heavy and therefore not suitable for being used in embedded devices. We are not currently aware of any framework specially designed for embedded devices. Additionally, while designed with security in mind, these frameworks do not make secure web application design intuitive for the developer.

In contrast, we strive for a secure by default system where a developer has to do little if anything in order to build a secure web application.

Web servers. At the process level, flow control enforcement such as the one presented in *Histar* [54], *Asbestos* [11] and *Flume* [30] can be used to achieve some of our goals such as document sanitization. The Android OS [15] capability model can also be extended to enforce network restrictions. As far as we know, none of the lightweight web servers like *Tornado* [12] were built with the objective of enforcing security principles. Previous work on security centric web servers such as [29] were only designed to mitigate low level attacks by enforcing privilege separation. None of them offered a framework to mitigate web vulnerabilities.

Other related work. The log injection attack, a simple form of XCS, has been known for several years [47], most notably in the context of web servers resolving client hostnames. Recently, CSRF and XSS attacks have attracted much attention, including work on various defense techniques [6]. NAS security has been a topic for discussion since the early days of networked storage [10].

IP telephony security has also been scrutinized. However this has only been done for specific protocols, not for complete systems [48]. Most other work in web security[13, 24, 4, 17, 25, 28, 43, 32, 36, 40, 53, 46] has focused on web servers on the open Internet, as opposed to devices on private intranets, which are the topic of this work.

11 Conclusion

We present *WebDroid* the first web application framework that is explicitly designed for embedded applications, with a particular emphasis on secure web application design. We motivate our work with extensive results from audits carried out over the last two years on a broad range of embedded web servers. We evaluate *WebDroid* performance and show that despite the fact that that performance take a 20% hit when we all the security features are activated, *WebDroid* remains sufficiently fast for its purpose. Finally as a case study we build two sample web applications.

Acknowledgment

We thank Samuel King and anonymous reviewers for their comments and suggestions. This work was partially supported by the National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research.

References

[1] Minded security research labs: Http parameter pollution a new web attack category (not just a new buzzword :p). Web http://blog.mindedsecurity.com/2009/05/http-parameter-pollution-new-web-attack.html, 2009. 8

[2] Minded security research labs: A twitter domxss, a wrong fix and something more. Web http://blog.mindedsecurity.com/2010/09/twitter-domxss-wrong-fix-and-something.html, 2010. 8

[3] Minded security research labs: Bypassing csrf protections with clickjacking and http parameter pollution. Web http://blog.andlabs.org/2010/03/bypassing-csrf-protections-with.html, 2010. 8

[4] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, 2008. 13

[5] A. Barth, C. Jackson, and I. Hickson. The http origin header. web http://tools.ietf.org/id/draft-abarth-origin-03.html, 2009. 8

[6] A. Barth, C. Jackson, and J. Mitchell. Robust defenses for cross-site request forgery. In *proceedings of ACM CCS '08*, 2008. 5, 6, 13

[7] H. Bojinov, E. Bursztein, and D. Boneh. Embedded Management Interfaces: Emerging Massive Insecurity. In *Blackhat USA*, July 2009. Invited talk. 1

[8] H. Bojinov, E. Bursztein, and D. Boneh. XCS: cross channel scripting and its impact on web applications. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 420–431. ACM, 2009. 1, 3

[9] H. Bojinov, E. Bursztein, and D. Boneh. Xcs: Cross channel scripting and its impact on web applications. In *CCS 2009: 16th ACM Conference on Computer and Communications Security*, Nov 2009. 3, 4, 13

[10] Cifs security consideration update, 1997. http://www.jalix.org/ressources/reseaux/nfs-samba/~cifs/CIFS-Security-Considerations.txt. 13

[11] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 17–30. ACM, 2005. 13

[12] Facebook. Tornado web server. Web http://developers.facebook.com/news.php?blog=1&story=301, 2009. 13

[13] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. Petkov. *XSS Exploits: Cross Site Scripting Attacks and Defense*. Syngress, 2007. 2, 3, 5, 13

[14] M. Foundation. Content security policy, 2009. wiki.mozilla.org/Security/CSP/Spec. 6, 8, 13

[15] Google. Android os. Web http://www.android.com/, 2008. 13

[16] B. Gourdin. Webdroid: Google code project. http://code.google.com/p/android-secure-web-server/. 1

[17] O. Hallaraker and G. Vigna. Detecting malicious javascript code in mozilla. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2005. 13

[18] R. Hansen. Clickjacking. ha.ckers.org/blog/20080915/clickjacking. 3

[19] C. Heffner. How to hack millions of routers. In *Blackhat USA*, 2010. 1

[20] J. Hewitt and R. Campbell. Firebug 1.3.3, 2009. http://getfirebug.com/. 4

[21] T. Holz, S. Marechal, and F. Raynal. New threats and attacks on the world wide web. *Security & Privacy, IEEE*, 4(2):72–75, March-April 2006. 4, 5

[22] Http strict transport security (HSTS), 2011. http://http://bit.ly/lwqdlu. 9

[23] C. Jackson and A. Barth. Forcehttps: Protecting high-security web sites from network attacks. In *Proceedings of the 17th International World Wide Web Conference (WWW2008)*, 2008. 9

[24] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *in proc. of 16th International World Wide Web Conference*, 2007. 13

[25] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS)*, 2006. 13

[26] A. Judson. Tamper data 10.1.0, 2008. http://tamperdata.mozdev.org/. 4

[27] S. Kamkar. mapxss: Accurate geolocation via router exploitation. http://samy.pl/mapxss/, January 2010. 1

[28] E. Kirda, C. Kruegel, G. Vigna, , and N. Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In *In Proceedings of the 21st ACM Symposium on Applied Computing (SAC), Security Track*, 2006. 13

[29] M. Krohn. Building secure high-performance web services with OKWS. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, page 15. USENIX Association, 2004. 2, 6, 13

[30] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 321–334. ACM, 2007. 13

[31] G. F. Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*, volume 978-0470170779. Nmap Project, 2007. 4

[32] G. Markham. Content restrictions, 2007. www.gerv.net/security/content-restrictions/. 13

[33] Microsoft. Mitigating cross-site scripting with http-only cookies. Web http://msdn.microsoft.com/en-us/library/ms533046.aspx, 2009. 7

[34] Microsoft. Urlscan 3.1. Web http://www.iis.net/download/urlscan, 2011. 9

[35] Netcraft. Totals for active servers across all domains. Website http://news.netcraft.com/archives/2009/06/17/june_2009_web_server_survey.html, Jun 2009. 2

[36] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *In Proceedings of the 20th IFIP International Information Security Conference*, 2005. 13

[37] J. Nielsen's. Response times: The 3 important limits. http://www.useit.com/papers/responsetime.html. 12

[38] T. Oda, G. Wurster, P. van Oorschot, and A. Somayaji. Soma: mutual approval for included content in web pages. In *ACM CCS'08*, pages 89–98, 2008. 13

[39] P. Petefish, E. Sheridan, and D. Wichers. Cross-site request forgery (csrf) prevention cheat sheet. web http://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet, 2010. 8

[40] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID)*, 2005. 13

[41] Ruby on rails. http://rubyonrails.org/. 13

[42] R. Rogers. *Nessus Network Auditing, Second Edition*, volume 978-1597492089. Syngress, 2008. 4

[43] RSnake. Xss (cross site scripting) cheat sheet for filter evasion. http://ha.ckers.org/xss.html. 13

[44] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*, 2010. 2, 8

[45] G. Rydstedt, B. Gourdin, E. Bursztein, and D. Boneh. Framing attacks on smartphones, dumb routers and social sites: Tap-jacking, geo-localization and framing leak attacks. In *Woot*, 2001. 1

[46] P. Saxena and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *proceedings of NDSS'08*, 2008. 13

[47] Log injection attack and defense, 2007. http://bit.ly/kbMebK. 13

[48] Basic vulnerability issues for sip security, 2005. http://download.securelogix.com/library/SIP_Security030105.pdf. 13

[49] P. Stone. Next generation clickjacking. media.blackhat.com/bh-eu-10/presentations/Stone/BlackHat-EU-2010-Stone-Next-\Generation-Clickjacking-slides.pdf, 2010. 3

[50] D. Stuttard and M. Pinto. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*, volume 978-0470170779. Wiley, 2007. 1, 2, 3

[51] B. Walther. Edit cookies 0.2.2.1, 2007. `https://addons.mozilla.org/en-US/firefox/addon/4510`. 4

[52] D. Wichers. Sql injection prevention cheat sheet. web `http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet`, 2011. 9

[53] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *In Proceedings of the USENIX Security Symposium*, 2006. 13

[54] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *7th Symposium on Operating Systems Design and Implementation*, 2006. 13