The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

AUTOMATING CONTENT SECURITY POLICY GENERATION

A Thesis in

Computer Science and Engineering

by

Jil Verdol

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2011

The thesis of Jil Verdol was reviewed and approved* by the following:

Patrick McDaniel
Associate Professor of Computer Science and Engineering
Thesis Adviser

Trent Jaeger
Associate Professor of Computer Science and Engineering

Raj Acharya
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

ABSTRACT

Web applications lack control over the environment in which they execute. This lack of control leaves applications open to attacks such as Cross Site Scripting, data leaks and content injection. Content Security Policy, CSP, was proposed and implemented as a way for applications to specify restrictions on how clients use content. However elaborating and maintaining a security policy can be a slow and error-prone process, especially for large websites. In this paper we propose a system to automate the policy generation process, and study its performance with an analysis of websites. The generated CSPs restrict client-server information flows of an application, using validated historical data. We reach a 0.81% miss rate, with an average of 7 directives for a given website. We consider the case where one externally maintains a security policy for a website which does not provide its own yet.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my advisor Pr. Patrick McDaniel for his constant support. I would also like to thank Thomas Moyer for his help and comments, as welI as Adam Bergstein for his reviews. I am also grateful for the feedback and comments from the SIIS Lab members.

Chapter 1

Introduction

An increasing number of applications are deployed on the web. However as the use of web applications becomes more widespread, the number of attacks targeting applications and the users' browsers also increases. The protection currently in place is provided by the Same Origin Policy (SOP). The Same-Origin Policy restricts code downloaded from a domain to only access resources from this same domain. However the user can still be tricked into downloading malicious code from a trusted website. This code can gather data from the user's session and send it to a remote server.

Different approaches have been tried to mitigate web applications vulnerabilities such as cross-site scripting (XSS) and data leaks. Kirda et al. proposed Noxes [12], a client-side web application firewall. While effective, their solution assumes the user will be able to make informed policy decisions for each of the web applications he uses. Other solutions such as WAVES by Huang et al. [8] propose crawling mechanisms and fault injection to detect vulnerabilities. However they do not offer protection from undiscovered vulnerabilities.

For large web applications, doing a full scale testing can be slow and expensive. Content Security Policy (CSP) [20] adds a protection layer to restrict which type of content is loaded in the context of an application, and from which domain it may be loaded. The approach is similar to the one taken by SOMA, proposed by Oda et al.[15]. SOMA proposes a scheme where in order to request and embed a resource in a document, the browser must receive the approval of both the origin domain of the document and the origin of the requested resource. CSP has the advantage of having an implementation by default in a popular browser. But server-side deployment remains an issue and is more likely to happen incrementally during a transition

phase. During this transition phase users might want to enjoy the benefits of CSP for sites and applications that do not implement it yet.

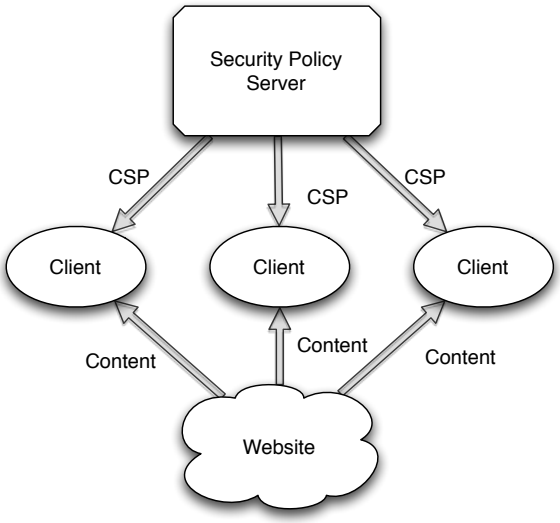

Fig. 1.1. External CSPs

In this paper we propose a scheme to generate and maintain CSPs for an external server. CSP is an evolving document. As content changes on a domain, the associated CSP must be updated. The system analyzes HTML pages of the protected domain to maintain the model. In doing so our main challenge is making sure the security policy does not affect the functionality of the website, while preventing attacks such as data leaks and classes of content injection. We take into account the temporal behavior of content. Some pieces of content will appear on the page with some periodicity and should be referenced in the policy. Other pieces of content will only be used until a certain date, and we want to remove directives for this content as soon as possible. We face three challenges. We want a solution to generate CSPs for external websites, to control

information flows between the client and other servers. In addition to generating policies, the solution must be able to maintain them over a given period of time. We also want to minimize the inaccuracies caused by this automated approach.

To solve this problem we use retention delays to postpone the expiration of directives. We study different ways to categorize content and assign retention delays to each of the categories.

Minimizing inaccuracies is important to our approach. We consider a miss occurs when a legitimate piece of content is not allowed by the security policy. In order to minimize the memory overhead, we also aim for a small number of rules maintained each day on the policy server. Therefore we want to show our approach satisfies three criteria:

- A miss rate lower than 1%
- A relatively small number of rules/day stored on the server
- The design can detect and filter potentially malicious content

To validate these points we consider a series of three experiments. The first one analyzes top Alexa websites over a period of 1 year, and tries to infer the best categories and delays for a target miss rate of 1%. The second experiment studies the evolution of number of directives over time with the model obtained in the previous part. The third experiment considers the importance of filtering results.

1.1 CSP

CSP is a content restriction enforcement scheme presented by Stamm et al. in 2010. Current web applications execute in an uncontrolled environment, which leaves them vulnerable to attacks such as XSS and content injection. CSP aims to provide to an application more control over the content displayed in its context, by restricting source domains.

CSP is enforced client-side, i.e. on the user's browser. It is activated by the X-Content-Security-Policy HTTP header, that specifies either the policy directly or the location of a file

describing the policy. The policy specifies which types of resources may be loaded, and from where they may be requested. The supported directives are shown in Table 1.1.

font-src	domains used for external fonts
frame-ancestors	indicates which sites may embed a protected document
frame-src	indicates which documents may be embedded in a protected document
img-src	domains used for images
media-src	domains used for audio and video elements
object-src	domains used for object, embed and applet HTML elements
script-src	domains used for external scripts
style-src	domains used for external scripts
xhr-src	indicates which sources can be used for XMLHttpRequests
report-uri	indicates where notifications of policy violations should be sent

Table 1.1. CSP Directives

CSP also disables some features by default, using the Base Restrictions. These restrictions can be disabled with the `options` directive. Base Restriction 1 prevents the execution of inline scripts. The main motivation is that inline scripts are a common attack vector for web applications. The Base Restriction 1 disables any inline script in the documents, including event handlers in HTML tags. Base Restriction 2 prevents the conversion of strings to code. I.e. it blocks calls to `eval()` and any function that converts one of its string arguments to code to be executed. The reason is that in most cases arguments to those functions come from untrusted sources, which results in vulnerabilities that could be exploited.

CSP is backward-compatible, which means that if either of the browser or server does not support it, the browser falls back on the standard same-origin policy [14].

1.2 Case Studies

In this section, we detail attacks that allow us to understand how CSP can prevent common attacks such as XSS, clickjacking and content injection. In each case, vulnerabilities and attacks are described that allow an attacker to perform these attacks.

1.2.1 Wordpress.com Cross-Site Scripting

A cross-site scripting attack (XSS) consists of an attacker inserting code in a page sent to a victim, typically JavaScript code. The attack usually involves cross-domain communication with other servers. In 2010 XSS was still the most common vulnerability on websites [6], followed by information leakage. An XSS vulnerability affected the Wordpress.com blog hosting provider in 2009 [13] . One of the themes had a feature to display messages to clients accessing a user's website. This feature could be exploited to inject JavaScript in the webpage served to visitors.

The administrator can indicate the desired message title and content of the message. The message box is then displayed in the upper section of the blog. All users accessing the website will then see the message box.

The vulnerability consisted in insufficient validation of the message contents. If JavaScript code is inserted in the message section, it will then be loaded and executed by any clients accessing the website. Since logged-in users' cookies are tied to wordpress.com, the malicious script can perform requests on behalf of the victim. It is then possible to enable the vulnerable theme for the victims too, and infect it with the malicious code. The code can then propagate to the next victim in a worm-like fashion.

1.2.1.1 CSP Stopping the XSS

The attack would be mitigated by the CSP base restriction 1. This restriction is enabled by default and states that no inline scripts will be executed. The malicious code in the message section would therefore not execute, and the victim's blog would not be infected. If a `report-uri` is provided, a violation report would also be sent via HTTP POST to the specified URI.

1.2.2 Twitter Clickjacking

Clickjacking occurs when an attacker tricks a user into clicking on a page element, while the user intended to click on a different element [16]. It is often done by making the clicked element transparent and positioning it over the visible element. A defense against clickjacking is frame-busting, where the webpage uses code to protect itself and prevent itself from being loaded in another document. In a 2010 study by Rydstedt et al. [18] studied the frame busting techniques of the top 500 Alexa sites. One of the conclusions was the fact that all the frame-busting used in those top websites could be circumvented in various ways. A clickjacking attack affected Twitter in 2009 [19]. Messages started to appear with the text "Don't click" followed by a link. When following the link and clicking on the "Don't click" button present on the landing page, users would see that a new tweet had been sent from their account with the same "Don't click" text and link.

Twitter is an online service allowing users to send short messages from their account. Users who subscribed to messages of another user are also known as followers. To send a message, users must first log into their account. They are then directed to their homepage, containing a message box. They can type in a message and click on the "update" button to send the message.

For the attack to work, users must be already logged in their account. The attack page loads Twitter.com inside an iframe using a special URL parameter `status` to preload the message box with a message. The frame is made transparent using CSS, and positioned such that the twitter "update" button is on top of the button labeled "Don't click". When users try to click

on the "Don't click" button, they actually click on the button in the invisible iframe. A message with a link to the malicious page is then posted.

1.2.2.1 CSP stopping the clickjacking attack

The first step of the attack, embedding twitter.com inside an iframe, would be stopped by the CSP directive "frame-ancestors". When enforced by the browser, the directive `frame-ancestors 'self'` prevents other websites from embedding pages served by a given server. By using this directive, twitter.com could protect itself and prevent malicious websites from embedding it inside an invisible iframe.

1.2.3 Iframe injection

An iframe injection attacks occurs when an attacker injects an iframe inside a legitimate document served by an unsuspecting server, to load a page from an external domain. A common use of iframe injection attacks is hijacking the victim website's traffic to serve malware to visitors. An advantage is that the payload of the injected page can be delivered from another server, which does not require access to the victim server's backend, and makes it harder to detect the attack [7]. There is also an increased flexibility for the attacker, who can later change the domain hosting the iframe content. A number of popular websites were compromised by malicious iframes in 2008 [3]. Using the local caching of search engines, attackers were able to inject iframe markup in search results.

Search engines often cache popular search results. The purpose is to increase the relevancy of the information presented to the user, using previous searches. When subsequent requests are made, the related popular search terms are also displayed to the user. Attackers first insert the malicious iframes in the popular results of the website's own search engines. They can do so by making large numbers of requests with malicious markup. These requests will be cached for performance reasons and as possible options to present to subsequent users. In the second step,

users make requests on the website's search engines and are often presented with these malicious search results because of insufficient server-side validation. The iframes then redirects the victims to websites controlled by the attacker.

1.2.3.1 CSP stopping the iframe injection attack

The last step of the attack can be prevented with CSP by indicating the domains that can rendered inside iframes on the webpage. Only allowing local iframes can be done with `allow 'self'` or `frame-src 'self'`. With this policy, the malicious iframes would not be rendered by a CSP-enabled browser, and users would not be automatically redirected to external websites.

Chapter 2

Thesis Statement

Web applications synthesize the code and data from various sources and users, some of which are untrustworthy. They often suffer from vulnerabilities like XSS and data leaks, and it makes them an attractive target for malicious parties. CSP (Content Security Policy) offers a solution to web applications vulnerabilities such as classes of XSS, content injection and data leaks. It adds an additional layer of protection on top of existing solutions such as the Same-Origin Policy. However deploying CSPs for large websites can be slow and error-prone. Users might also want to enjoy the benefits of CSP for websites that do not support it yet. This leads us to wonder how feasible it is to generate and maintain CSPs for external websites. We believe websites are regular enough that we can externally generate and maintain security policies. By regular we mean that the set of origin domains used on a page for any given type of content should change slowly over time and only have a few elements, in order to minimize the policy miss rate. This would allow us to use the set of observed origin domains as the basis to generate CSPs.

In the Background section we will first look at the solutions proposed to mitigate the attacks mentioned in the Introduction: XSS through inline scripts, clickjacking, and iframe injection. We will then consider various approaches for malicious content detection. Detection of malicious content is critical in order to filter potentially malicious content from the generated CSPs. In Section 4 we first detail the components of our solution and the Security Model. We consider two main aspects: policy generation and policy enforcement. For the policy generation process we study the data gathering phase and then the algorithm used. For the enforcement process we describe a simple protocol for distribution and enforcement, followed by the types

of inaccuracies that may arise. We indicate what a miss means in the context of this work. In the Section 5 we first characterize our data set, along with the content classification method used. We show how it is possible to reach a 0.82% miss rate, with only 7 directives on average per website. We show how our solution can use the Google Safe-browsing API to successfully filter malicious content. We then consider the related works and the possible future work for our design.

Chapter 3

Background

In this section we first consider how previous works protect against the attacks indicated earlier: XSS, clickjacking, iframe injection. Since our solution relies on observing content to produce policies, we must avoid the problem of false learning. In order to do so we validate content references using a third-party solution. We consider the possible solutions in the second part.

3.1 Existing Solutions

Different solutions have been proposed to mitigate the type of attacks mentioned above. The first vulnerability is XSS through inline scripts. A defense proposed by CSP is Base Restriction 1, which prevents the execution of inline scripts. BEEP [10] uses server-generated whitelists and blacklists. These list are enforced client-side to indicate which scripts should or should not run on a page. However it applies only to scripts, while CSP can be used for other elements such as images and stylesheets. BrowserShield [17] can insert runtime Javascript checks to enforce policies during code execution. But it does not focus on script origin, and the whitelists and blacklists would have to be implemented in the policy itself.

The second attack is clickjacking, and the possibility to embed a website in a page controlled by the attacker, using an iframe. CSP proposes a whitelist of domains allowed to embed the website in frames. Here the closest work would be SOMA [15]. It allows a server to indicate which domains may embed its resources on their pages, using approval files. It is more flexible than CSP in this case, as it can be queried for any resource (e.g. images) and is not restricted to the case of frames.

The third attack is content injection, more specifically iframe injection. The CSP approach is to specify a whitelist of the domains that can be embedded on the website in frames. SOMA takes a similar approach with manifest files, specifying valid content sources. However CSP offers more flexibility, as it can differentiate between content types (such as images, scripts, frames) and associate different whitelists to them.

3.2 Detection of Malicious Content

3.2.1 Server Blacklists

A solution to detect potentially malicious content is to consider the origin server. One approach to this is server blacklists. Third parties maintain lists of servers susceptible of serving malicious content. These entities offer online APIs to check whether a given server is included in one of the lists. The drawback of blacklists is that they have to be continuously updated and often become very large.

3.2.2 Server Whitelists

A similar approach is the use of whitelists. The benefit of whitelist is that they explicitly indicate a set of trusted servers. However they must handle revocation, to remove servers that send malicious content from the list .

3.2.3 Reputation Checks

Reputation checks are notably used for spam filtering. A mail server is assigned a grade, depending on the volume of email sent, received and the ratio of email sent and classified as spam. This grade can be used to estimate the trustworthiness of the mail server, and the probability that the email it sends is legit.

3.2.4 Google Safe Browsing

The Safe Browsing API [5] allows applications to check URLs against Google's blacklist of suspected fishing and malware pages. It can be embedded directly inside applications, and is in use in Firefox and Chrome. Applications can perform complex requests and download tables for client-side lookups. This is the detection system we use on our solution.

Chapter 4

Design

The actors we consider for our solution are the clients, a target website, a Security Policy Server and a third party used to validate content, the Google Safe-Browsing API. The client downloads content from the Website and external CSPs from the Security Policy Server. The website generates content but does not provide a CSP. The Security Policy Server generates and maintains CSPs for the target website. The Security Policy Server validates content using the Safe-Browsing API before integrating it into the CSPs. We consider two phases for our solution: the policy generation phase and the enforcement phase.

4.1 Security Model

Our solution uses a central server, the Security Policy Server. This central server processes content information sent by trusted clients. It generates policies from this content information, and sends them to other clients. We trust this central server, as well as the clients used for data gathering. The third-party used for malicious content detection is also trusted, as well as the domains it approves. The connections between the central server and all clients are secured for integrity using SSL. We assume an attacker can inject content onto a legitimate server, so that it is served within the same domain as legitimate content. We do not address DNS rebinding. We also do not address the possible compromise of the central server of the proposed solution, trusted clients or the third party used for malicious content detection.

4.2 Policy Generation

4.2.1 Content Lifetime

Our solution must take into account the fact that some objects have a short lifetime. The content on a website changes over time, and some of the directives on the central server become irrelevant. We consider objects in use as long as the trusted client reports origin information for them. The data reported by clients does not give a complete coverage, and we introduced a delay variable to take this into account. If a tuple (date, object category, origin domain) is not reported by the clients for a time period greater than the delay, then the directive for this tuple is dropped from the central server. If the tuple reappears before the expiration of the delay, then the counter is reset. In the evaluation section we show how we determine an reasonable value for this delay.

4.2.2 Data Gathering

When a trusted client browses the website, it gathers data about the origin of the content served. Then the client contacts the central server and sends the data it gathered for this specific website. The central server gathers the data sent by many clients for this same websites. It aggregates this information and produces a set of directives, which form a policy. This policy will be served when a browser requests a policy for the website. The following process outlines a trusted client sending content to the central server for analysis.

Collection of data from trusted client to central server:

1. Trusted client requests a web page
2. Trusted client receives content
3. Trusted client submits content and origin to central server
4. Central server processes web page content

4.2.3 Malicious Content Detection

Our solution aims to mitigate classes of content injection attacks. When gathering origin data from a website, we need a way to detect potentially malicious content, to avoid its immediate inclusion in the policy for this website. Various solutions such as whitelists, blacklist and reputation checks are available through on-line third-parties. For our implementation the Security Policy Server uses the Google Safe Browsing API to detect potential malicious objects.

4.2.4 Algorithm

Each content category C is assigned a delay D_c. We iterate over the days of our dataset. For each day we analyze a given website and extract a list of tuples (date, object category, origin domain). These tuples are reported by clients as indicated in Figure 4.1. Each tuple corresponds to an object observed on the website. The origin domain is the origin of the object, the object type directly refers to the directives used by CSP, e.g `img`, `frame`, `script`. The date is when the object was observed. When the server processes a tuple, it first checks whether it already saved a corresponding tuple with the same origin domain and type. If that is the case, it just updates the date entry to the new value. If there is no such tuple, it creates a new entry using the domain, type and date reported by the client. At regular intervals the freshness of the tuple is verified, using the date information. If a tuple of the category C was previously present and was not observed for D_c days, it is removed from the list of current items. If in a following snapshot this tuple reappears, the policy is updated and we consider an error occurred.Below is a description of the step function used to update directives.

S_n : Set of directives present at date $Date_n$

R_n : Set of directives reported at date $Date_n$

D_x : Date when object x was last reported

$D_{C(x)}$: Retention delay of an object x of category $C(x)$

$$S_{n+1} = \{x \in S_n \mid D_x + D_{C(x)} \geq Date_n\} \bigcup R_n$$

The right-hand side of the expression includes the objects that were observed on the date $Date_n$. The left-hand side of the expression filters the existing directives by only keeping the ones that have not expired. The resulting set of directives is the union of the two sets.

When serving a policy, the central server generates a list of allowed domains for each object type. It does so by iterating over the tuples for the protected website and returning the domains associated with each object type.

4.2.5 Example

Next, we consider an example of the output of the algorithm for the website `youtube.com`, on the day 2009-12-22 (with the yyyy-mm-dd format) with a retention delay of 18 days for scripts and 9 days for frames. For the sake of clarity we consider only a specific set of tuples. We have the following tuples on the policy server:

```
1  2009-12-03 img-src   www.google-analytics.com
2  2009-12-21 style-src s.ytimg.com
3  2009-12-21 scr-src   s.ytimg.com
4  2009-12-21 img-src   i2.ytimg.com
5  2009-12-19 frame-src ad-g.doubleclick.net
```

Only the type and domain information are used to generate the policy sent to clients. The date is used by the Security Policy Server to keep track of the expiration time of directives. On 2009-12-22 the policy server receives updates from clients, reflecting content currently visible on the website:

```
2009-12-22 style-src s.ytimg.com

2009-12-22 scr-src   s.ytimg.com

2009-12-22 img-src   i2.ytimg.com
```

The resulting policy server state on 2009-12-23 is:

```
2  2009-12-22 style-src s.ytimg.com

3  2009-12-22 scr-src   s.ytimg.com

4  2009-12-22 img-src   i2.ytimg.com

5  2009-12-19 frame-src ad-g.doubleclick.net
```

Tuple number 1 expired and was removed from the policy, since it had not been reported for 18 days. Tuples 2, 3 and 4 were updated since they were reported in the clients' updates. Tuple 5 was not updated. But since it was observed within the last 9 days (retention delay for frames), it is still kept in the policy.

4.3 Policy Enforcement

4.3.1 Distribution and Enforcement

First the browser connects to the central server over SSL, and sends a request for a given page. The web server checks to see whether it has a policy for this page. If it has one, it sends the policy. The browser then enforces the policy using the CSP module when displaying the page and executing its content. If the web server does not have a policy for the page, then it returns a negative response to the browser. In this case the browser displays and executes the page without enforcing any CSP, falling back on the Same-Origin Policy instead. The following process, illustrated in Figure 4.2 outlines a client receiving a CSP from the central server for secure browsing.

CSP Distribution from central server to client:

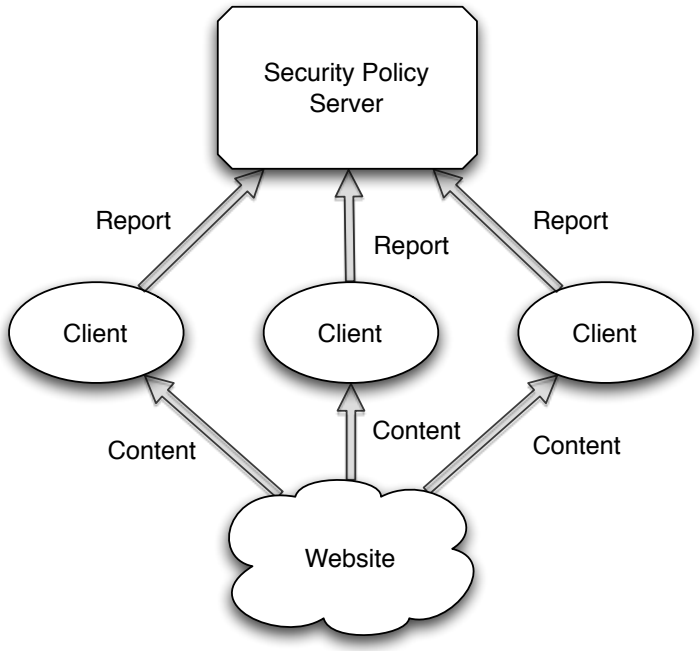

Fig. 4.1. Gathering Data for Policy Generation

1. Client browser requests a web page
2. Client browser receives web page content but does not display content
3. Client browser submits origin to central server
4. Central server sends the origin's CSP policy to the client browser
5. Client browser uses the CSP to filter out web page content before displaying the page

4.3.2 Inaccuracies

An inaccuracy occurs when there is a difference between what the policy allows, and legitimate content on the server. We consider three types of such errors:

1. When a new legitimate origin is inserted in a protected document, and this origin is not allowed by the policy yet.
2. False removes: when an origin that was previously removed from the policy appears again, but is not allowed by the policy yet.
3. When an origin does not appear in the content served, but it still allowed by the policy. This could lead to potential vulnerabilities and a large number of directives, making it hard to maintain the policy on the Security Policy server.

4.4 Compatibility with the CSP Specification

A requirement of the CSP specification is that the policy be served in the HTTP headers of the page containing the content itself. Otherwise an external policy can be indicated, using the `policy-uri` directive. An external policy must have the same origin (i.e. scheme, host and port) as the protected document. The solution we propose relaxes this part of the specification. Although we use SSL, this could leave our solution vulnerable to attacks such as DNS rebinding.

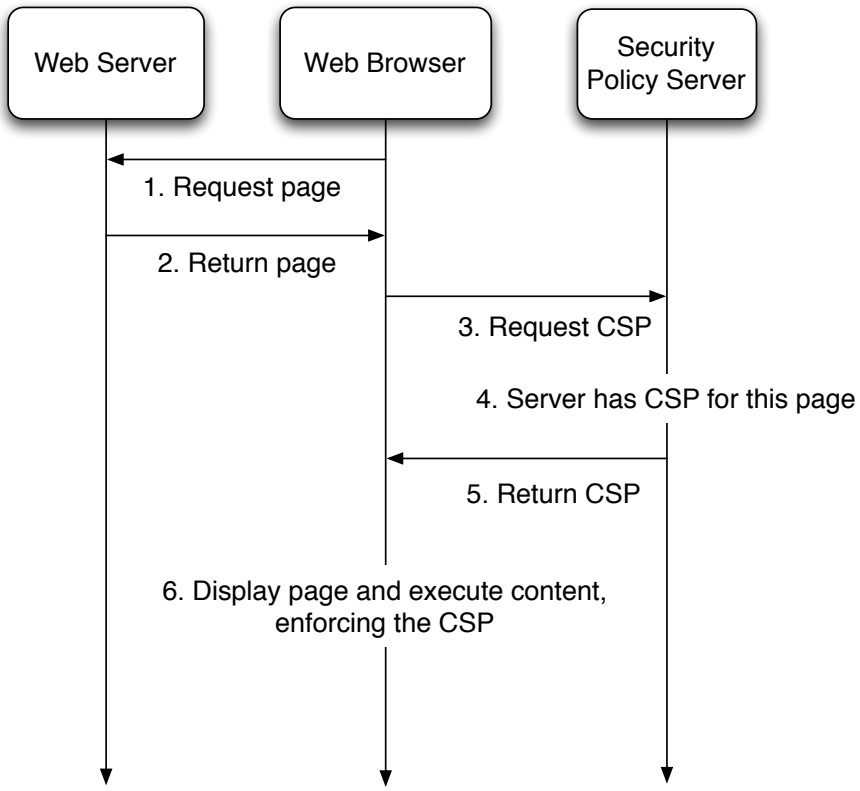

Fig. 4.2. CSP Request procedure.

4.5 Benefits and Limitations

If a page does not provide any CSP and it is not on the central server, our solution does not impose any restriction on content inclusion in the page, nor does it prevent the page from being embedded inside another page.

This solution can be used as a way to develop policies for websites, that they can maintain later. It could play a role in the incremental deployment of CSP. An advantage is that it does not require any server-side implementation for the protected domain. Content is analyzed as it appears on the website, instead of requiring a more heavy-weight source code analysis.

One limitation is the impact of false positives. In the solution we described website administrators have little control over what is allowed or blocked by the policy. Valid content could be blocked, and prevent a website from offering legitimate services to its users. Moreover, because of a single malicious object, all objects from the same type from the same domain could be blocked. Another limitation is the fact that our current solution uses static analysis. So it does not detect XmlHttpRequests, or elements dynamically included through JavaScript.

Chapter 5

Evaluation

In this section we evaluate the performance of our solution. We use the front page of 89 websites saved on the wayback machine, over the year 2009. The main metric we use is the number of misses caused by the security policy. A miss occurs when the security policy blocks legitimate content. In order to minimize the number of directives in the policy, we decide to categorize content and to assign appropriate delays to each of the categories.

In the first part of this section we describe the data set used, and how we determine parameters for our model. In the second part we use these parameters to study the evolution of the number of directives stored over time. In the third part we outline the importance of filtering, and give an example of unfiltered and filtered policies.

5.1 Content Classification and Retention

5.1.1 Data Set

The Wayback Machine [2] is a project of the Internet Archive [1]. It stores snapshots of websites and allows people to browse these sites as they were on a given date. It was extremely useful to us as it allowed us to study the content embedded by websites over 2009.

Our first approach was instrumenting a web browser and crawling a list of sites on the Wayback Machine over 2009. However this approach was problematic. Content such as images, applets or flash objects took many seconds to load and often resulted in timeouts. In such cases, content sources were not logged and affected the statistics. Also the time needed made it difficult to scale the method to hundreds of websites.

We decided to analyze html pages of the target websites. The results are not as complete as for the approach with the browser - for instance we do not observe XmlHttpRequests - but it is much faster and more robust. We gathered content information from 89 websites from the archive, using the Alexa top 500 websites. The content of each of those sites was analyzed for the year 2009. For each website, each day is associated with a HTML snapshot of the site. The data from the archive does not cover every day, so we used a simple interpolation algorithm, where we assume content does not change between two successive snapshots. We simulate the execution of the policy server using the algorithm described in Section 4.

5.1.2 Classification by Type

The classification by type was the closest to the semantics used by CSP, which made it easier to analyze the results. We use the following types: images, external script files, stylesheets, objects such as java applets and flash elements. For each object of our data set we keep track of the distance in days between successive appearances. The results are shown in Figure 5.1. The figure shows the cumulative distribution (CCDF) of these distances. We use it to evaluate the percentage of misses we can cover for each type, when picking a specific retention delay. We see that for a same retention delay, each type of content presents a different coverage of misses. Images exhibit the highest coverage. As shown by Table 5.1 they represent 79% of the content, which explains their significant impact on the average miss rate.

We were interested in seeing how we could use this data to reach a given miss rate. We first targeted a coverage of 99% of misses. We obtained the appropriate delays by simply using a threshold of 1% over the CCDF of misses described above. These delays are indicated in Table 5.2. Then we ran our algorithm over the data set using these delays, and reached an average miss rate of 0.42%. As expected the average miss rate is close to the miss rate for images (0.41%), because of the high percentage of images in our data set.

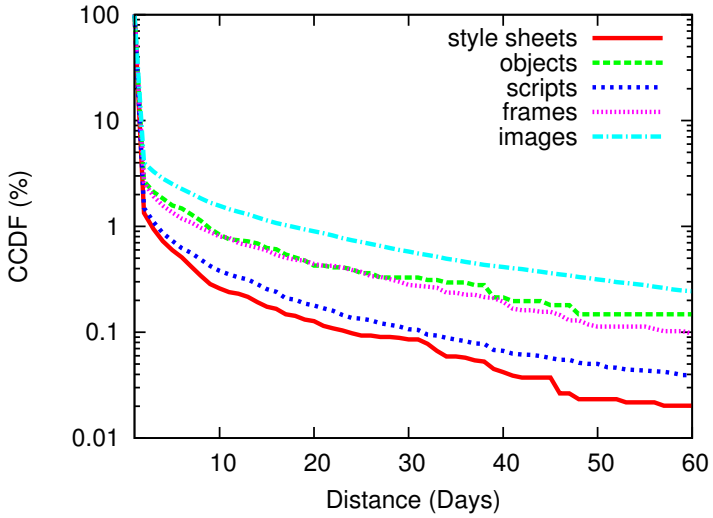

Fig. 5.1. Counter Cumulative Distribution Function of Misses per Type

Type	Count	%	avg. per site
images	1077454	78.95	33.16
scripts	189331	13.87	5.82
css	68873	5.04	2.12
frames	21586	1.58	0.66
objects	16316	0.70	0.23
total	1364603	100	42.00

Table 5.1. Content statistics for 89 archived sites during 2009

Type	λ (days)	Miss Rate(%)
images	18	0.41
scripts	4	0.69
css	3	0.97
document	8	0.09
objects	9	1.31
average	-	0.42

Table 5.2. Values for a 0.42% total miss rate

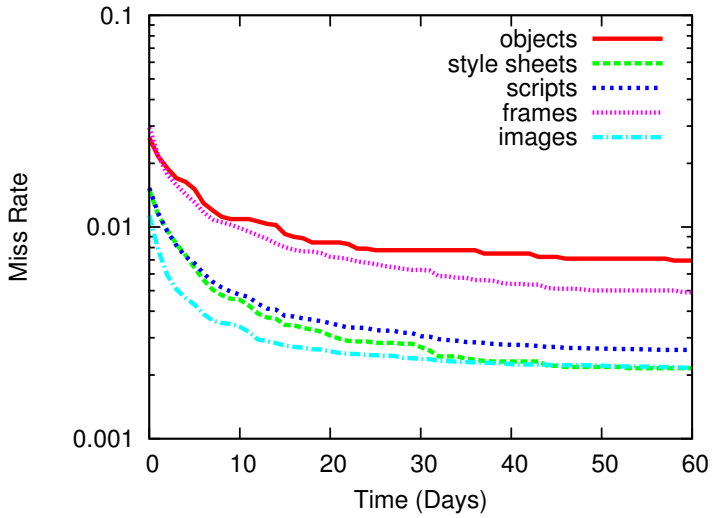

Fig. 5.2. Misses per Type

Although this approach is simple, we notice a difference between the coverage of misses in the CCDF (1%) and the actual miss rate resulting from the delays picked (0.42%). We decided to use a more direct approach: computing the misses per type for each delay, then selecting the delays for a 1% miss rate or lower. Figure 5.2 shows the corresponding plots. Similarly to the CCDF, different types exhibit different miss rates for the same delays. It confirms the choice of selecting a different miss rate for each type in order to reach a total miss rate lower than 1%. The resulting miss rate is 0.81%, which is closer to the expected 1% than the previous miss rate of 0.42%. Table 5.3 show the corresponding delays. We use those delays in the following sections.

Type	λ (days)	Miss Rate(%)
images	1	0.77
scripts	2	0.95
css	2	0.96
document	10	0.98
objects	15	0.92
average	-	0.81

Table 5.3. Values for a 0.81% total miss rate

5.1.3 Lifetime

In this section we consider the lifetime of content. We used 7 categories: from 0 to 3 days old, from 4 days to 8 days old, from 9 days to 2 weeks, from 2 weeks to 1 month, from 1 month to 2 months, from 2 months to 6 months, and 6 months old and older. Figure 5.3 shows the distribution on November 22nd 2009. The values and percentages are indicated in Table 5.4. The `Total` column represents the total number of objects of a given category in our dataset from

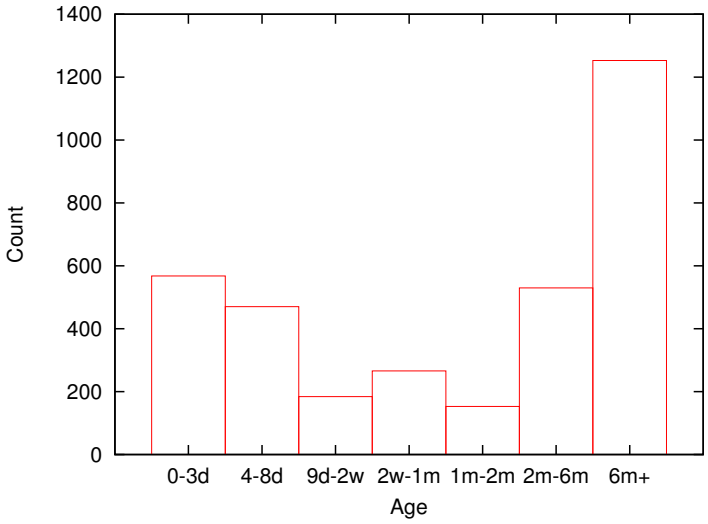

Fig. 5.3. Age Distribution Chart on November 22nd 2009

Category	Total	(%)
0 to 3 days	568	16.58
4 to 8 days	470	13.72
9 days to 2 weeks	184	5.37
2 weeks to 1 month	266	7.76
1 month to 2 months	153	4.46
2 months to 6 months	530	15.47
6 months and older	1253	36.59
total	3424	100.0

Table 5.4. Age Distribution

the Wayback Machine. The percentage column represents the proportion of each category to the total number of objects. We see that 52% of the objects are at least 2 months old.

5.2 Directives

In this section we study the directives produced by the model we obtained with the previous retention delays. First we consider the number of directives. Table 5.5 shows the average number of directives per website per day, ordered by type. The percentage shows the contribution of a given type to the total number of directives. On average there were 7.37 directives at any given day, resulting in 5.70 pieces of content per directive on average. These results validates the first and second requirements of our system: relatively low miss rates and number of rules per day per website. This means we can expect this design to be usable in a real environment.

Type	Average per site	Total	%
images	3.23	104914	43.94
scripts	2.37	76826	32.17
css	0.95	30955	12.96
document	0.54	17778	7.44
objects	0.25	8301	3.47
aggregate	7.37	238774	100.0

Table 5.5. Number of directives

Then we consider how directives evolve over time. Table 5.6 show the average number of new directives added per day, organized by type. The percentage column shows how each type contributes to the total number of new directives. Similarly Table 5.7 shows the number of directives removed per day, organized by type. When combining these results we obtain the data

presented in Table 5.8, which shows the net average number of directives added per day. We see that there is a very slow increase in the number of directives for a website over time. It is a net increase of 0.003 policy directives per day on average.This increase could be caused by a growing number of content sources for websites.

Type	Average	%
images	0.061	59.00
scripts	0.024	23.81
css	0.010	10.30
document	0.005	5.01
objects	0.0019	1.85
average	0.1043	100.0

Table 5.6. Average number of directives added per day per website

5.3 Filtering of malicious content

In Section 4 we outlined the need for filtering in our solution. In this section we use the Safe Browsing API and test it in our solution. We used a list of 8000 domains from the DNS-BH project [4], an aggregate of domains flagged by various sources. 3444 domains tuned out to be reachable. We analyzed the HTML files of those sites and extracted their content sources.

1864 sites had three sources of content or more. Out of those sites, 647 (34.7%) had content that was flagged by the API. Table 5.9 shows the total number of objects detected, organized by type. A total of 69618 references to embedded content was found, and 17647 (25.3%) were flagged by the safe browsing API. So there were 37.34 embedded objects per site on average, and 9.46 potentially malicious objects per site on average.

Type	Average	%
images	0.060	59.76
scripts	0.023	23.36
css	0.010	10.40
document	0.004	4.64
objects	0.0018	1.82
average	0.1046	100.0

Table 5.7. Average number of directives removed per day per website

Type	Average	%
images	0.001	31.52
scripts	0.0011	40.21
css	0.0001	6.52
document	0.0005	18.47
objects	0.0001	3.26
average	0.003	100.0

Table 5.8. Total average number of directives added per day per website

Type	Total	%	avg per site
images	57778	82.99	30.99
scripts	6815	9.78	3.65
css	3112	4.47	1.66
objects	1256	1.80	0.67
frames	657	0.94	0.35
total	69618	100	37.34

Table 5.9. Content statistics for infected sites

Type	Total	%	avg per site
images	3836	43.13	2.05
scripts	2736	30.76	1.46
css	1422	15.99	0.76
objects	476	5.35	0.25
frames	423	4.75	0.22
total	8893	100	4.77

Table 5.10. Distinct domain statistics for infected sites

We considered the case of one of the domains. Without filtering the resulting policy is:

```
X-Content-Security-Policy: allow 'self';
object-src 'self' widget-b1.slide.com www.youtube.com;
script-src 'self' sepahan-e.com addonrock.ru;
img-src 'self' sstatic1.histats.com;
frame-ancestors 'self';
```

With filtering we obtain:

```
X-Content-Security-Policy: allow 'self';
object-src 'self' widget-b1.slide.com www.youtube.com;
script-src 'self' sepahan-e.com;
img-src 'self' sstatic1.histats.com;
frame-ancestors 'self';
```

The domain `addonrock.ru` referenced as a scripts source was flagged by the Safe Browsing API and excluded from the policy. We see that using the API our design successfully filtered a potentially malicious object.

5.4 Net effect

As indicated in the previous sections, the pages of our data set have approximately 50 embedded elements on average. This would lead to an average of a miss every 2 page loads. However in most cases there will be more than one user. The rate perceived by a single user will depend on how often the policies are updated, and the number of users. Our 1% relates only to the first time an object is encountered on a given website. All subsequent accesses will not experience a miss. Thus the average user will experience an inconsequent number of misses.

Chapter 6

Related Work

In 2003 Huang et al. propose a web application vulnerability assessment framework WAVES [8]. Their solution identifies data entry points with a black-box approach, and then performs fault injection to detect vulnerabilities. While our solution also relies on crawling the application, it aims to provide protection for the unknown vulnerabilities of web applications through CSP. In 2004 Huang et al. proposed WEbSSARI, which performs a verification on source code [9] using flow analysis. As indicated above, our solution aims to provide an additional protection layer. It also has the advantage of being more lightweight and easier to operate than a complete flow analysis. In 2006 Jovanovic et al. proposed Pixy [11], a tool also based on data flow analysis. It allows them to detect taint-style vulnerabilities automatically. However they had a false positive rate close to 50%, meaning a manual inspection of results would still be needed. Our solution would require minimal intervention from an administrator. The same year Kirda et al. proposed Noxes, a client-side solution to mitigate XSS attacks and data leak attacks. It is a web firewall applications that runs as a service on the client. In our opinion, the disadvantage of this approach is that it assumes the user is savvy enough to make informed choices about the policy that must be enforced by the firewall. BEEP was proposed by Jim et al. in 2007. It uses policies supplied by the server and enforced client-side. The main limitation is that these policies only apply to which scripts may execute. We believe CSP offers more coverage and flexibility. Oda et al. proposed SOMA [15] in 2008, a policy for constraining communications and inclusions in web pages. It is close to CSP but has the advantage of requesting a mutual approval of both servers (the one hosting the document and the one hosting embedded content)

before including content. The solution uses manifest files and approval files to specify policies. However the question of generating these files for large applications was not addressed.

Chapter 7

Conclusion

Implementation and deployment remain a major issue for security solutions. While previous works proposed solutions and policies to restrict how content may be used by web applications, no work considered how those policies would be generated and maintained in a scalable way. In this paper we proposed an approach to automatically generate security policies for websites that do not provide their own yet. The generated CSPs restrict client-server information flows of an application, using validated historical data. We decided to stay close to the semantics of CSP by categorizing content according to its type, and then showed how different delays can be assigned to achieve a target error rate, with a relatively small number of directives. On our data set we had a 0.81% miss rate, with 7 directives per website on average over 1 year. It suggests that content domains indeed change slowly enough that generating and maintaining security policies from observation is possible. Our design also addressed the need to filter potentially malicious content from the policies. In the ideal case the website should be able to analyze its content and update its policy accordingly, but the solution we propose could be useful during a transition phase. Future work will consider which delays are best suited for different classes of web applications, and how to include hooks in the Content Management Systems of web applications to automatically update policies.

References

[1] Internet archive. http://www.archive.org/.

[2] Internet Archive. The wayback machine. http://www.archive.org/web/web.php.

[3] Dancho Danchev. Wired.com and history.com getting rbn-ed. http://ddanchev.blogspot.com/2008/03/wiredcom-and-historycom-getting-rbn-ed.html, 03 2008.

[4] DNS-BH. Malware domain blocklist. http://www.malwaredomains.com/.

[5] Google. Google safe browsing api. http://code.google.com/apis/safebrowsing/.

[6] Jeremiah Grossman. Whitehat website security statisics report. http://www.whitehatsec.com/home/assets/WPstats_fall10_10th.pdf, 2010.

[7] Joel Hruska. Ongoing iframe attack proving difficult to kill. http://arstechnica.com/security/news/2008/03/ongoing-iframe-attack-proving-difficult-to-kill.ars, March 2008.

[8] 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 conference on World Wide Web*, WWW '03, pages 148–159, New York, NY, USA, 2003. ACM.

[9] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[10] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[11] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6 pp. –263, May 2006.

[12] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 330–337, New York, NY, USA, 2006. ACM.

[13] Pedro Laguna. Wordpress.com permanent xss vulnerability. `http://equilibrioinestable.wordpress.com/2009/04/08/wordpresscom-permanent-xss-vulnerability/`, 04 2009.

[14] Mozilla. Introducing content security policy. `https://developer.mozilla.org/en/Security/CSP/Introducing_Content_Security_Policy`, 2010.

[15] Terri Oda, Glenn Wurster, P. C. van Oorschot, and Anil Somayaji. Soma: mutual approval for included content in web pages. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 89–98, New York, NY, USA, 2008. ACM.

[16] OWASP. Clickjacking. `http://www.owasp.org/index.php/Clickjacking`.

[17] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: vulnerability-driven filtering of dynamic html. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association.

[18] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: a study of clickjacking vulnerabilities on popular sites. In *IEEE Oakland Web 2.0 Security and Privacy Workshop*, 2010.

[19] Daniel Sandler. Quick explanation of the "don't click" attack. `http://dsandler.org/outgoing/dontclick_orig.html`, 02 2009.

[20] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 921–930, New York, NY, USA, 2010. ACM.