Web Camouflage

Protecting Your Clients from Browser-Sniffing Attacks

Browser cache and history are intended to be private, yet it's not difficult for malicious Web sites to "sniff" cache entries on visitors' computers and then use that information to more accurately deceive them. The authors' approach neutralizes the threat of URLs being discovered on client computers.

Markus Jakobsson
Palo Alto Research Center

Sid Stamm
Indiana University

Phishing combines the deceitful techniques of con artists with the Internet's scalability to commit identity theft by stealing credentials. Crimeware is a functionally related crime that relies on malware for identity theft; however, this malware is often installed via techniques resembling those that phishers use. The Anti-Phishing Working Group (www.apwg.com) monitors the occurrence of phishing and crimeware attacks and reports a steady increase of this threat in recent years.

It's commonly believed that phishing attacks increasingly rely on contextual information about their victims to increase their yield and lower the risk of detection. The same is likely to hold for crimeware attacks that rely on deceit. Browser caches are rife with contextual information, indicating, for example, whom users bank with or in general, the online services on which they rely. As other work has shown,[1–3] such information can easily be "sniffed" by anyone whose site the victim visits. If victims are drawn to rogue sites by receiving emails with personalized URLs pointing to these sites, then phishers can create associations between email addresses and cache contents. Attackers can also use cache information to infer which attack techniques are likely to work—for example, if an attacker learns that a given victim banks with Barclays, he would also know that the victim is likely to have antivirus protection from F-Secure (Barclays offers clients free antivirus software from F-Secure). The attacker might then spoof an email to the victim, purporting to come either from Barclays or F-Secure, warning of a purported vulnerability and suggesting that he or she immediately download a patched version of the software. The software that the victim would download, of course, wouldn't be a patch at all, but some form of crimeware. This example shows both the relation between crimeware and phishing and the importance of shielding contextual information from prying eyes; for more examples of this worrisome relationship, see a recent Symantec Press book on Crimeware (http://crimeware-book.com).

Our proposed Web camouflage solution works on the server side and lets a server "inoculate" all its clients against sniffing attacks. We camouflage accesses to internal pages by randomizing and personalizing such links; this prevents detection of internal URLs, given that the exact URL will be impossible for an attacker to guess.

Phishing

Before attackers can infer any information about a victim, they need to make their victims visit a site that they control, so that they can sniff the victim's browser state. Spam is a surprisingly successful tool in the hands of attackers wishing to attract traffic to their sites, with pornography spam the leading type with a reported 5.6 percent click-through rate, compared to 0.02 percent for pharmaceutical spam.[4] Spammers' most common goal is to make recipients click on a link and purchase something. If, instead, the spammer simply wants the message recipient to visit a site, then success rates are likely to be much higher, especially when deceit is used.[5] For example, phishers can make victims visit their sites by spoofing emails

from someone the victim knows or from within the same domain. Recent experiments by Tom Jagatic and his colleagues[6] indicate that more than 80 percent of college students will visit a site appearing to be recommended by a friend. It's worth noting that approximately 15 percent of the subjects in a control group (who received emails appearing to come from an unknown person in the same domain) entered their credentials, whereas subjects who received an email seemingly from a friend entered credentials 70 percent of the time.[6] Even though the same statistics might not apply to the general population of computer users, it's clear that spoofing is a reasonably successful technique of luring people to sites where their browsers will be silently interrogated and the contents of their caches sniffed.

Once phishers create an association between an email address and the browser cache's contents, they can use this knowledge to target the users in question with seemingly plausible phishing emails. For example, phishers can detect possible online purchases and then send notifications stating that the payment didn't go through, requesting that the recipient follow an included link to correct credit-card information and billing address. In this scenario, victims visit a site that looks just like the one at which they recently completed a transaction. A wide variety of such tricks can help increase the yield of phishing attacks, and all of them benefit from contextual information that can be extracted from the victim's browser.

Several possible approaches can address the problem at its root—namely, at the information-collection stage. First, users should be instructed to frequently clear their browser caches and browser histories. While educational efforts (such as www.SecurityCartoon.com) can be helpful tools in the fight against online crime, experts agree that countermeasures based on education alone aren't likely to succeed (for an overview, see http://education.works-or-not.com). Moreover, attack techniques reported in other work[2,3] can still detect bookmarks on some browsers (such as Safari 1.2) that aren't affected by clearing the history or the cache and could be of equal or higher value to attackers. A second approach would be to disable all caching and not keep any history data; this approach, however, is highly wasteful in that it eliminates the significant benefits associated with caching and history files. A third avenue is a client-side solution that limits (but doesn't eliminate) the cache use via a set of rules maintained in the user's browser or browser plug-in; such an approach is taken in the concurrent work by Dan Boneh and colleagues.[7] Finally, a fourth approach—the one we propose here—is a server-side solution that prevents cache contents from being verified via personalization.

What accesses can be hidden?

External pages are camouflaged by polluting the cache with a large number of "similar" pages, making it difficult for an attacker to guess which were really accessed and which were merely part of the pollution process. This approach doesn't work for stigmatized content (such as more revealing sites like pornography) because the pollution could exasperate the problem. In the latter case, users accessing this type of content could instead use a system that works on stigmatized content as well as content that isn't or manually attach a personalized extension of the URL when entering the stigmatized site. As long as an attacker finds this extension hard to guess, it will have the desired effect.

We want to point out that neither our solution nor any other one (www.securiteam.com/securityreviews/5GP020A6LG.html) can be used to hide accesses to sites that are criminal to access. The reason is the likely difference in the adversarial model: for such content, authorities likely have alternative detection mechanisms and don't need to rely on browser sniffing for information collection. Thus, the approaches we describe in the main text simply protect users against adversaries with somewhat limited powers—peers and corporations, not governments.

Client- and server-side solutions not only address the problem from different angles, but they also address slightly different versions of the problem. Namely, a client-side solution protects those users who have the appropriate protective software installed on their machines, whereas a server-side solution protects all users of a given service (but only against intrusions relating to their use of that service). The two are complimentary, in particular, in that the server-side approach allows "blanket coverage" of large numbers of users who haven't yet obtained client-side protection, whereas the client-side approach secures users in the face of potentially negligent service providers (SPs). Moreover, if a set of users within an organization use a caching proxy, adversaries can abuse it to reveal information about the behavioral patterns of users within the group even if they employed client-side measures in their individual browsers; a server-side solution, such as our approach, can prevent this.

From a technical viewpoint, users can hide cached contents from the server side in two very different ways. One approach makes it impossible to find references in the cache to a visited site; in the other, the cache is intentionally polluted with references to all sites of some class, thereby hiding the actual references to the visited sites among these. Our solution combines these two approaches: it makes it impossible to find references to all internal URLs (as well as all bookmarked URLs), while polluting external URLs. (Here, an external URL corresponds to a URL a person would typically type to start accessing a site, whereas an internal URL is accessed from an external URL by logging in, searching, or following links.)

Background

Several different types of historical data can be stored in Web browsers which, if compromised, provide valuable information about the software's user. Let's look at them in more detail to give purpose to our tool.

Browser caches

One particular use of caches is for browsers, to avoid the repeated downloading of recently accessed material. Browser caches typically reside on individual computers, but closely related caching proxies are also common; these reside on a local network to take advantage not only of repeated individual requests for data but also of repeated requests in the group of users. The very goal of caching data is to avoid having to repeatedly fetch it; this results in significant speedups of activity—in the case of browser caches and caching proxies, these speedups result in higher download speeds.

Ed Felten and M. Schneider[1] described a timing-based attack that made it possible to determine (with some statistically quantifiable certainty) whether a given user had visited a given site or not, simply by determining the retrieval times of consecutive URL calls in a segment of HTTP code.

Browser history

In addition to having caches, most browsers also maintain a history file, to let them visually indicate previous browsing activity to users and permit users to backtrack through a sequence of previously visited sites.

Securiteam recently demonstrated a history attack analogous to the timing attack that Felten and Schneider described (www.securiteam.com/securityreviews/5GP020A6LG.html). The history attack uses Cascading StyleSheets (CSS)—specifically, the `:visited pseudo class`—to determine whether a given site has been visited, and later communicates this information by invoking calls to URLs associated with the different sites detected. An attacker-controlled computer hosts the data corresponding to these URLs, thereby helping the attacker determine whether a given site was visited. Here, the site's domain isn't detected—rather, the history attack detects whether the user has been to a specific URL or not. The same attack was recently re-crafted to show the impact of this vulnerability on phishing attacks.[2] A benevolent application based on the same browser feature was also proposed to allow postmortem detection of visits to dangerous sites and the resulting exposure to malware.[8]

The client-side solution

In work concurrent with ours, Boneh and his colleagues developed a client-side solution to the sniffing problem.[7] This solution works by making the browser follow a set of rules about when to force cache and history misses (URLs that are perceived unvisited), even if a hit could have been generated. This, in turn, hides the browser cache and history file contents from prying eyes. It doesn't, however, hide the contents of local cache proxies—unless they're also equipped with similar, more complex rule sets.

Our server-side solution

We first introduced our translator as a solution to the browser-sniffing attack at the World Wide Web conference in 2006.[9] We presented an in-depth formal design of our model and discussed initial results, configuration policies (such as when to pollute and when to redirect pages), and special considerations for more complex systems, such as those using Akamai. Let's look at our solution and its implementation in more depth.

Implementation issues

Browser sniffing attacks work in the following manner: an adversary queries the cache or history of a victim's browser, and learns whether given pages have been visited. The adversary can't read out the entire contents of the victim's cache or history file, but can only verify whether files with known names are present within. Therefore, if we can make the names of the visited Web pages impossible to guess, then we can prevent sniffing attacks.

However, to ensure compatibility with search engines, and allow Web pages to be indexed at the same time that they're inaccessible to attackers, we use the *robots exclusion standard* (www.robotstxt.org). In this unofficial standard, parts of a server's file space are deemed "off limits" to clients with specific user-agent values. However, we use the standards' associated techniques in a different manner: namely, in conjunction with a whitelist approach, to give certain privileges to pre-approved robot processes—their identities and privileges are part of a security policy for each individual site. This way, we can give special access of Web pages to search engines, but force regular users to access them in another manner.

Our implementation can rely either on browser cookies or an HTTP header called *referer* (sic). Cookies are sent from the server to the client in HTTP headers; this content isn't displayed. When a client requests a document from server *S*, it sends, along with the request, any information stored in cookies by *S*. This transfer is automatic, so using cookies has negligible overhead. The HTTP-referer header is an optional piece of information sent to a server by a client's browser—the value (if any) indicates where the client obtained the address for the requested document. In essence, it's the link location that the user clicked, so if a user types in a URL or uses a bookmark, no value for HTTP-referer is sent.

One particularly aggressive attack (depicted in Figure 1) is one in which the attacker obtains a valid pseudonym, namely, a random number, from the server and then tricks a victim into using this pseudonym (by posing as the SP in question.) Thus, attackers would potentially know the pseudonym extension of URLs for their victims and would therefore be able to query browsers about what downloads.

Informal goal specification

Our goals are to make the fullest possible use of both browser caches and histories, without allowing third parties to anticipate the names of items in these. This makes browser sniffing impossible. Specifically,

- an SP should be able to prevent sniffing of any data related to its clients (this should hold even if the data distribution is through network proxies);
- the previous requirement should hold even if a user is behind a caching proxy, and even if the adversary controls one or more user machines in a user group sharing a caching proxy; and
- search engines must retain the ability to find data served by SPs in the face of augmentations performed to avoid sniffing.

Search engines shouldn't have to be aware of whether a given SP deploys our proposed solution or not, nor should they have to be augmented to continue to function as before.

Intuition

We achieved these goals by using two techniques. First and most important, we used a customization technique for URLs, in which we "extended" each URL by using either a temporary or long-term pseudonym. This prevents a third party from interrogating the browser cache or history of a user who received customized data, given that all known techniques to do so require knowledge of the exact file name being queried. A second technique is cache pollution, which lets a site prevent meaningful information from being inferred from a cache by having spurious data entered.

Hiding vs. obfuscating

As mentioned earlier, we hide references to internal and bookmarked URLs and obfuscate references to external URLs. We hide references by using a method that customizes URLs with pseudonyms that a third party can't anticipate; obfuscation comes from polluting, or adding references to all other external URLs in a given set of URLs. This set is referred to as the *anonymity set*.

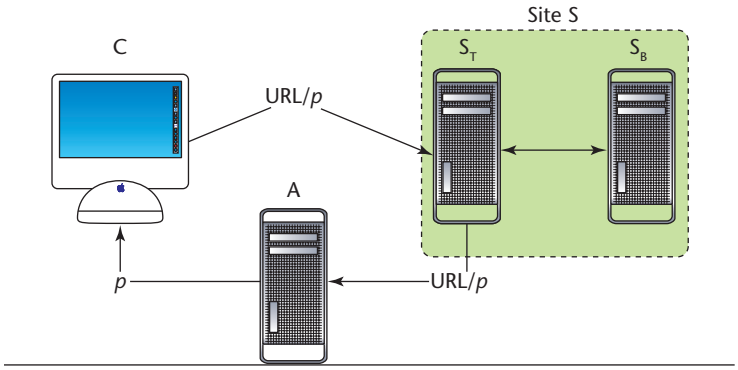

Figure 1. Aggressive attack. In this attack, *A* obtains a valid pseudonym *p* from the translator S_T of a site *S* with back end S_B and coerces a client *C* to attempt to use *p* for the next session. The goal is to query *C*'s history/cache files for the pages in the corresponding domain that *C* has visited. Our solution disables such an attack. (S_T and S_B can be different processes on the same machine.)

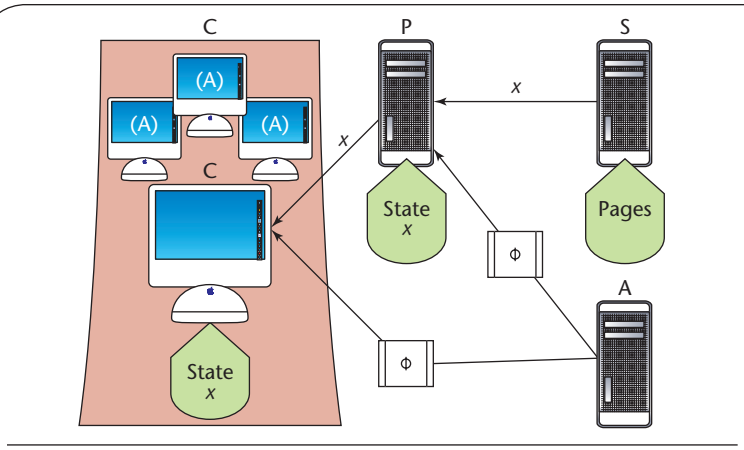

Figure 2. Formalization of a server *S*, caching proxy *p* , client *C*, attacker *A*, and attack message ϕ (sent either through the proxy or directly to *C*). *A* controls many members of *C*, allowing it—in a worst-case scenario—to generate and coordinate requests from these members. This ability lets the attacker determine what components of caching proxy *p* are likely to be associated with *C*.

What does security mean?

It's important to understand what our proposed solution achieves and against what types of attacks it protects. For example, it doesn't protect against an adversary who wants to find browsing patters and who can access the routing history, keyboard entries, or bus activity on the victim's machines (as well as an array of related types of access of information on this computer). Similarly, it provides no protection against an adversary with access to a local DNS server or local router, assuming users haven't installed a Web anonymizing proxy. Instead, we assume that the adversary will only be "reasonably powerful" and able to externally probe the victim's browser

cache and history, as described elsewhere.[2]

We also assume that the adversary might send HTTP documents to the victim, causing the victim's browser to request documents needed to display the page (see Figure 2). No revealing information has to be displayed to the user because it's still possible for the adversary to include references to arbitrary objects. Adversaries have succeeded if they can determine that any given object is present in the victim's cache or a proxy cache accessible to the victim, or if they can determine that a given object is part of the victim's browser history. In our solution, the adversary can still fully control some collection of other clients (as Figure 2 shows), but not the victim, of course. We want all pages of internal URLs to be perfectly protected—that is, make it impossible to determine whether they were accessed—and we want all pages of external URLs to be protected as well, so that an adversary can't determine whether they or related URLs (in the same anonymity set) were accessed. The solution we present has these properties. A more detailed model of adversaries and their abilities appears elsewhere.[9]

The translating filter

At the heart of our solution is a filter associated with a server whose resources and users must be protected. Similar to how middleware is used to filter calls between application- and lower-level layers, our proposed filter modifies communication between clients and servers—whether the servers were the information source or simply acting on behalf of network proxies.

When interacting with a user (through a Web browser), the filter customizes the names of all files (and the corresponding links) in a manner unique for the session and that a third party can't possibly anticipate. Thus, a third party can't verify a chosen victim's cache/history; this can only be done by someone who knows the visited pages' names. In brief, pseudonyms are used for short-lived sessions—to add randomness to all URLs served by a protected service. Pseudonyms don't affect the user's perception of a Web site, and the filter can easily identify and remove them.

Establishing and using a pseudonym

When clients first visit a site protected by our translator, they access a URL such as the index page. The translator notices the client has no personalization, so it generates a pseudonym pseudorandomly from a sufficiently large space (64 to 128 bits). The pseudonym is sent to the client in the form of a translated personalized external URL. All the URLs on translated pages are then modified by referencing the translator's domain instead of the server's and then appending the current pseudonym.

Pseudonym validity check

If an attacker, Alice, were able to convince a client, Bob, to use pseudonyms he knows, then Alice could easily guess where on the site Bob has been—effectively removing from Bob the privacy our translator creates. To avoid this, the filter must determine pseudonyms to be valid, which can be done with cookies (the cookie must equal the pseudonym), HTTP-referer (the client must come from another page in the same domain), or a combination of both. This doesn't protect against clients who spoof the HTTP-referer field or alter their cookies, but given that modifications of this nature eliminate the client's privacy, it's not rational to do so. Further investigation of pseudonym validity checking is described elsewhere.[9]

Robot policies

The same policies don't necessarily apply to robots that apply to clients representing human users. In particular, when interacting with a robot or agent, users might want to customize names of files and links differently, or by using pseudonyms that will be replaced when used (see www.robotstxt.org). The filter could use a whitelist approach to bypass customization for certain robots or employ temporary pseudonyms[9] that provide hard-to-guess URLs without requiring the private site to trust the search engine that's indexing it.

Pollution policy

Client C can arrive at a Web site via four means: by its user typing in the URL; following a bookmark; following a link from a search engine; or following a link from an external site. A bookmark might contain a pseudonym that S established, so the URL entered in C's history (and cache) is already preserving privacy. When server S obtains a request for an external URL that doesn't contain a valid pseudonym, S must pollute the cache of C in a way such that anyone analyzing C's state can't be certain which site was the intended target.

Off-site references

The translator, in effect, acts as a proxy for the actual Web server, but the Web pages could contain references to off-site (external) images, such as advertisements. In other words, an attacker could still learn that a victim has visited a Web site based on external images or other resources or even from the URLs that the Web site references. Because of this, the translator should also act as an intermediary to forward external references. This might not be necessary depending on the trust relationships between the sites in question, but for optimal privacy, it should be required. Our translator modifies off-site URLs to redirect through itself, except in cases in which two domains collaborate and

Go to google
Log in
<img src='/images/welcome.gif'>

The translator replaces any occurrences of the SB's address with its own.

Go to google
Log in
<img src='/images/welcome.gif'>

Then, based on ST 's offsite redirection policy, it changes any off-site (external) URLs to redirect through itself:

 Go to google
Log in
<img src='/images/welcome.gif'>

Next, it updates all on-site references to use the pseudonym. This makes all the URLs unique:

 Go to google
Log in
<img src='/images/welcome.gif?38fa029f234fadc3 '>

All these steps are of course performed in one round of processing, and are only separated herein for reasons of legibility. If Alice clicks the second link on the page (Log in) the following request is sent to the translator:

GET /login.jsp?38fa029f234fadc3

Figure 3. URL camouflage. Browser sniffing attacks work by querying victim Web browsers to determine if their caches or histories bear traces of Web pages of interest to the attacker. If the attacker doesn't know the complete URLs of the pages it wishes to check for, then the browser sniffing fails. Therefore, any technique that creates temporary and random extensions of Web page URLs will block browser sniffing attempts.

agree to pseudonyms set by the other. This allows the opportunity for the translator to put a pseudonym in the URLs that point to off-site data, but this leads to more work for the translator and could lead to serving unnecessary pages. Because of this, it's up to the translator's administrator (and probably the server's owner) to set a policy of what should be directed through translator S_T, referred to as an *off-site redirection policy*. Similarly, the administrator must also set a policy to determine what types of files get translated by S_T. The scanned types should be set by an administrator; this is called the *data replacement policy*.

Let's look at an example. A client, Alice, navigates to a requested domain http://test-run.com (previously described as S) that the translator S_T protects. In this case, the translator is located at that address, and the server is hidden from the public at an internal address (10.0.0.1 or S_B) that only the translator can see. The S_T recognizes Alice's user-agent (provided in an HTTP header) as not being a robot, and so proceeds to preserve her privacy. It calculates a pseudonym for her (say, 38fa029f234fadc3) and queries the actual server for the page. The translator then receives a page, as described in Figure 3.

The translator notices the pseudonym at the end of the request, removes it, verifies that it's valid, and then forwards the request to the server. When the server responds, the translator re-translates the page using the same pseudonym that it obtained from the request.

Security argument

Our proposed solution satisfies the previously stated security requirements. The most important thing to recognize is that if the attacker doesn't know the full URL of the pages for which he wants to query a victim's browser, then the attack fails. However, to conclude that our proposal protects users against browser sniffing, one must consider two separate and important cases: *internal* pages and *external* pages.

Perfect privacy of internal pages

Our solution doesn't expose pseudonyms associated with a given user/browser to third parties, except when temporary pseudonyms are used (this only exposes the fact that the user visited the page) and where shared pseudonyms are used (in which case, the referring site is trusted.) A site replaces any pseudonyms generated by either itself or its trusted collaborators, so assuming the user didn't intentionally disclose URLs, and given the pseudorandom selection of pseudonyms, a third party can't infer the pseudonyms associated with a given user or browser. Similarly, a third party can't force a victim to use a pseudonym given to it by the server to the attacker: this would cause the pseudonym to become invalid (and detectable). Thus, our solution offers perfect privacy for internal pages.

n-privacy of external pages and searchability

Assuming pollution of n external points from a set X by any member of a set of domains corresponding to X, a third party can't distinguish access of one of these external points from the access of another from cache/history data alone.

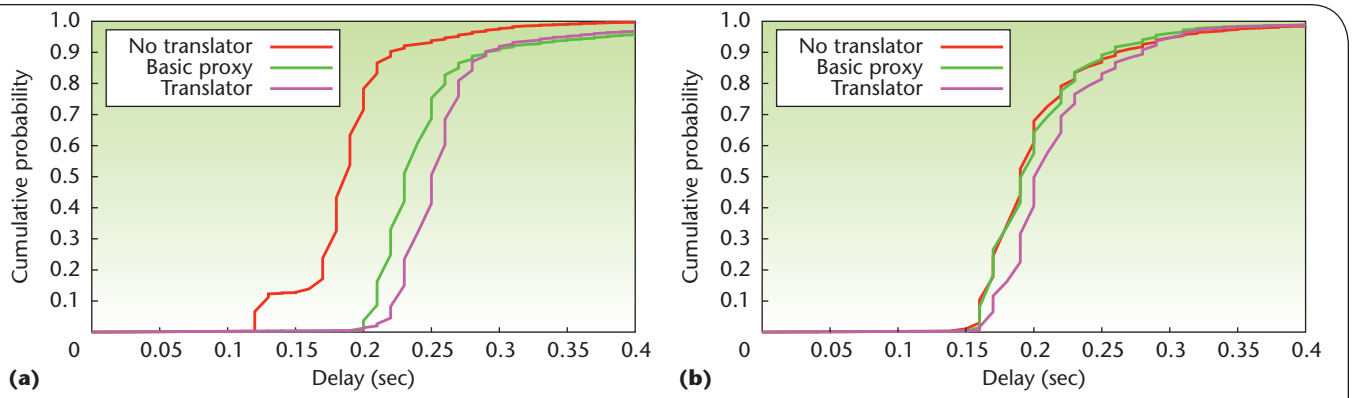

Figure 4. Cumulative distribution of our data. The vast majority of the results from each of the three test cases appears in a very short range of times, indicating cohesive results for both (a) the stand-alone configuration and (b) the loop-back configuration. Additionally, the loop-back configuration makes the performance hit negligible—even for full translation of the Web pages enabled.

Any search engine that's not included in the customization of indexed pages will be oblivious to the translation that's otherwise imposed on accesses, unless the search engine's software applies temporary pseudonyms. Similarly, a search engine that receives already customized data from the translator can remain oblivious to the included pseudonyms, given that users will get the same URLs, which will then be re-customized.

It's worth noting that although clients can easily manipulate pseudonyms, there's no benefit associated with doing so, and what's more, it could have detrimental effects on overall client security. Thus, we won't worry about such modifications here because they're irrational.

Implementation details

To test the security of our sniffing countermeasure, and to measure its impact on the time it takes to serve Web content, we implemented and tested our technique. We now describe these tests' results.

Stand-alone configuration

In the summer of 2005, we implemented a translator to estimate our solution's ease of use as well as determine its approximate efficiency and accuracy. We wrote our translator as a Java application that sat between a client *C* and protected site *S*. The translator performed user-agent detection (for identifying robots), pseudonym generation and assignment, translation, and redirection of external URLs. We initially placed the translator on a separate machine from *S* to get an idea of the worst-case timing and interaction requirements, although they were on the same local network. We set up the remote client on the Internet outside the local network.

Loop-back configuration

As a refinement of our initial prototype,[6] we also tested the ideal situation in which an existing Web server is changed in the following manner to use the loop-back interface:

- The translation software is installed on the Web server.
- The HTTP daemon is reconfigured to listen only on the loop-back interface.
- The translation software is configured to listen on the interfaces formerly monitored by the HTTP daemon.
- The translation software "privatizes" and acts as proxy for the Web server that's now hidden from the outside world.

To the outside world, the Web server doesn't seem to have changed, and the communication channel between the translator and HTTP daemon has a very high bandwidth—it isn't restricted by hardware interfaces, only the Web server's operating system.

Pseudonyms and translation

Our prototype calculated pseudonyms that use the java.security.SecureRandom pseudorandom number generator to create a 64-bit random hexadecimal string.

A client sent requests to our prototype, and our filter scanned the URL for an instance of the pseudonym. If the pseudonym wasn't present, it was generated for the client and then stored only until the response from the HTTP daemon was translated and sent back to the client.

Most of the parsing was done in the header of the HTTP requests and responses. We implemented a simple data replacement policy for our prototype: any value for the user-agent that wasn't "robot" or "wget" was assumed to be a human client. This let us easily write a script using the command-line `wget` tool in order to pretend to be a robot to simulate a search en-

Table 1. Seconds of delay in our prototype translator.*

SET UP	AVERAGE	STANDARD DEVIATION	MAXIMUM	MINIMUM
No translator	0.2078	0.0946	0.2057	0.2098
Separate proxy	0.2529	0.0971	0.1609	1.9910
Loop-back proxy	0.2078	0.0972	0.2056	0.2099
Separate translator	0.2669	0.0885	0.1833	1.9750
Loopback translator	0.2211	0.1002	0.2189	0.2233

*The second delays are for both the loop-back configuration (in which the HTTP daemon and translator shared a machine) and multiple-computer configuration (in which the translator was on its own machine). When the translator operates in proxy mode, it simply forwards all traffic between the server and clients without modifying it. When in translation mode, the software customizes pages with pseudonyms.

gine crawler's perspective. Any content would simply be served in basic proxy mode if the user-agent was identified as one of these two.

Additionally, if the content type wasn't text or HTML, then the associated data was simply forwarded from the HTTP daemon to the client. HTML data was intercepted and parsed to replace URLs in common context locations:

- Links (...)
- Media (<{*tag*} src='{URL}'>)
- Forms (<form action='{URL}'>)

More contexts could easily be added; the prototype used Java's regular expressions for search and replace. Our prototype didn't contain any optimizations because it was a simple proof-of-concept model and we wanted to calculate worst-case timings.

Redirection policy

The prototype also implemented a very conservative redirection policy: for all pages p served by the Web site hosted by HTTP daemon S_B, any external URLs on p were replaced with a redirection for p through S_T. Any pages q that weren't served by S_B weren't translated; instead, they were simply forwarded, and the URLs on q were left alone. This redirection doesn't cause the translator to behave like a proxy server—it simply serves the first off-site page. Any further links followed by the client went directly to its host and weren't translated.

Timing

Our prototype translator caused only negligible overhead while translating documents. Given that we only translated HTML documents, the bulk of the content (images) were, and should be, simply forwarded. Because of this, we didn't include in our results the time taken to transfer any files other than HTML. Our test Web site served only HTML pages and no other content, so all content passing through the translator had to be customized. This situation represents the absolute worst-case scenario for the translator. Our data might be a conservative representation of translator speed, though, because most content normally served on the Web—such as images—would simply be served by the translator without processing.

We measured the amount of time it took to send the client's request and receive the entire response. We noted the times for eight differently sized HTML documents 1,000 times each. We set up the client to load only single HTML pages as a conservative estimate—in reality, less of the traffic would be translated because many requests for images go through the translator. Because of this, we can conclude that the translator's actual impact on a robust Web site will be less significant than our findings.

Our data (see Figure 4) shows that the page translation doesn't create noticeable overhead on top of what it takes for the translator to act as a basic proxy. Moreover, acting as a basic proxy creates so little overhead that delays in transmission via the Internet completely overshadow any performance hit from our translator (see Table 1). We conclude that using a translator in the fashion we describe won't cause a major performance hit on a Web site.

Translation optimization

Given that the server's administrator is most likely in control of the translator, too, he or she has the opportunity to speed up the static content's translation. When a static HTML page is served, the pseudonym will always be placed in the same locations, no matter what it is. This means that the locations in which pseudonyms should be inserted can be stored along with the content so that the translator can easily place pseudonyms without having to search through and parse data files.

Through the use of random strings and history pollution techniques, we've shown how a translator

mechanism can be used (without noticeable overhead for either client or server) to neutralize browser sniffing attacks. Furthermore, the effort to employ this translator is minimal, and a protected server will appear no different on the Web than if it weren't translated. The results of protecting a server aren't trivial: it provides security against sniffing attacks for all of a protected site's clients. We aren't aware of any company using our techniques, but hope that they will start to be used before attackers start relying on browser sniffing as a tool to improve the success of spear phishing attacks, or as a way to evade user privacy. Our defense techniques are not patented, and can be used free of charge by anybody. □

References

1. E.W. Felten and M.A. Schneider, "Timing Attacks on Web Privacy," *Proc. 7th ACM Conf. Computer and Communication Security*, S. Jajodia and P. Samarati, eds., ACM Press, 2000, pp. 25–32.
2. M. Jakobsson, T. Jagatic, and S. Stamm, "Phishing for Clues," 5 July 2005; www.browser-recon.info.
3. SecuriTeam, "Timing Attacks on Web Privacy," 20 Feb. 2002; www.securiteam.com/securityreviews/5GP020A6LG.html.
4. A. Mindlin, "Seems Somebody Is Clicking on that Spam," *The New York Times*, 3 July 2006; www.nytimes.com/2006/07/03/technology/03drill.html?_r=1&oref=slogin.
5. M. Jakobsson, "The Human Factor in Phishing," Privacy & Security of Consumer Information '07; www.informatics.indiana.edu/markus/papers/aci.pdf.
6. T. Jagatic et al., "Social Phishing," *Comm. ACM*, vol. 50, no. 10, October 2007, pp. 94–100.
7. C. Jackson et al., "Web Privacy Attacks on a Unified Same-Origin Browser," *Proc. 15th Ann. World Wide Web Conf.* (WWW 06), 2006; http://crypto.stanford.edu/sameorigin/sameorigin.pdf.
8. M. Jakobsson, A. Juels, and J. Ratkiewicz, "Remote Harm-Diagnostics," www.ravenwhite.com/files/rhdpdf.
9. M. Jakobsson and S. Stamm, "Invasive Browser Sniffing and Countermeasures," *Proc. 15th Ann. World Wide Web Conf.* (WWW 06). 2006; www.stop-phishing.com.

Markus Jakobsson *is principal scientist at Palo Alto Research Center, cofounder of RavenWhite, and an adjunct associate professor at Indiana University. His research interests include cybersecurity and cryptographic protocols, fraud, deceit, crimeware, and phishing. Jakobsson has a PhD in computer science from the University of California, San Diego. Contact him at markus@markus-jakobsson.com.*

Sid Stamm *is a PhD candidate in computer science at Indiana University. His research includes abuses of Web technologies for purposes of fraud and how to protect people from such abuses. Contact him at sstamm@cs.indiana.edu.*