Exposing Private Information by Timing Web Applications

Andrew Bortz
Stanford University
abortz@cs.stanford.edu

Dan Boneh
Stanford University
dabo@cs.stanford.edu

Palash Nandy
palashn@gmail.com

ABSTRACT

We show that the time web sites take to respond to HTTP requests can leak private information, using two different types of attacks. The first, *direct timing*, directly measures response times from a web site to expose private information such as validity of an username at a secured site or the number of private photos in a publicly viewable gallery. The second, *cross-site timing*, enables a malicious web site to obtain information from the user's perspective at another site. For example, a malicious site can learn if the user is currently logged in at a victim site and, in some cases, the number of objects in the user's shopping cart. Our experiments suggest that these timing vulnerabilities are wide-spread. We explain in detail how and why these attacks work, and discuss methods for writing web application code that resists these attacks.

Categories and Subject Descriptors

K.4.4 [**Computers and Society**]: Electronic Commerce—*Security*; K.4.1 [**Computers and Society**]: Public Policy Issues—*Privacy*

General Terms

Design, Security, Experimentation

Keywords

web application security, web browser design, privacy, web spoofing, phishing

1. INTRODUCTION

Web applications are vulnerable to a variety of well publicized attacks, such as cross-site scripting (XSS) [15], SQL injection [2], cross-site request forgery [14], and many others. In this paper we study timing vulnerabilities in web application implementations. Our results show that timing data can expose private information, suggesting that this issue is often ignored by web developers. We first discuss the type of information revealed by a timing attack and then discuss ways to prevent such attacks.

We consider two classes of timing attacks. The first, called a *direct timing attack*, measures the time the web site takes to respond to HTTP requests. We experiment with two types of direct attacks:

- Estimating hidden data size. Many sites holding user data, such as photo-sharing sites, blogging sites, and social networking sites, allow users to mark certain data as private. Photo sharing sites, for example, allow users to mark certain galleries as only viewable by certain users. We show that direct timing measurements can expose the existence of private data, and even reveal the size of private data such as the number of hidden pictures in a gallery.
- Learning hidden boolean values. Web login pages often try to hide whether a given username is valid — the same error message is returned whether the input username is valid or not. However, in many cases, the site executes a different code path depending on validity of the given username. As a result, timing information can expose username validity despite the site's attempt to conceal it.

The second class of attacks, called *cross-site timing*, is a form of cross-site request forgery [14]. The attack enables a malicious site to obtain information about the user's view of another site — a violation of the same-origin principle [11, 8]. We describe this attack in Section 4. At a high level, the attack begins when the user visits a malicious page, which proceeds to time a victim web site using one of several techniques, all of which time the exact content the user would actually see. We show that this timing data can reveal private information: for example, it can reveal whether the user is currently logged-in. In some cases, timing information can even reveal the size and contents of the user's shopping cart and other private data, as discussed in Section 4. This information enables a context-aware phishing attack [9] where the user is presented with a custom phishing page.

These attacks exploit weaknesses in server-side application software, specifically when execution time depends on sensitive information. Our results suggest that these vulnerabilities are often ignored.

1.1 Related work

Timing attacks were previously used to attack crypto implementations on smartcards [10, 12, 13] and web servers [4, 1]. Felten and Schneider [6] used a cache-based timing attack to track web users. Their idea is that once a user visits a *static* page, her local cache contains a copy of the page causing the page to load faster on subsequent visits. By measuring the time the browser takes to load a given page,

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

a malicious web site can determine whether the user visited the page before. We note that non-invasive methods exist to prevent this attack [6, 8].

Our attacks target dynamic web pages — we obtain detailed information by measuring the time a web site takes to assemble the page (i.e. the time to query the database and run application code). Since dynamic pages are not typically cacheable, and techniques exist to prevent the use of cached copies, we can ignore any caching effects.

2. WEB APPLICATION ARCHITECTURE

When an HTTP request hits a web site various components on the site are used to assemble a response. After being initially processed for required HTTP details by a web server, such as Apache or Microsoft IIS, it is routed to the appropriate application or module to generate a response. Static content, which is stored directly in a file, is the easiest to handle: the response is always just the content of the file. Dynamic content, such as data-driven HTML pages and stylesheets that are a hallmark of modern web applications, are handled by running a program.

This program, which can either be part of a specialized web application framework (e.g. PHP, Java Server Pages, or ASP.NET), or a standalone program (typically called a CGI script), outputs the content that will form the response. This program can call upon any number of resources, including databases and custom servers, which may reside either on the same machine or another machine on the network, connected either internally or across the public Internet. The time it takes to use these resources, and process the data that they return, are generally dependent on the underlying data, much of which is private. For example, an implementation of a picture gallery might respond to a user request by first retrieving from the database the list of all pictures in the gallery and then looping over all images to produce HTML only for those images marked "public". The number of loop iterations depends on the total number of images (both public and private), which is private information. Consequently, the response time leaks information about the number of private images. A popular photo sharing system called Gallery [7] is vulnerable in this way, enabling an attacker to learn the number of hidden galleries at the site. Since it doesn't matter whether the processing time is local to the web server or in another process or machine, even applications using complicated SQL queries can have computation time dependent on the data.[1]

Finally, as the response is being passed back to the web server, it is buffered and finally returned to the requesting client in one of two ways allowed by HTTP 1.1:

- **Content-length.** The server can assemble the entire response page before sending the first byte to the client. In this case, the server embeds a Content-Length HTTP header which indicates the total length of the response. When this method is used, we need only measure the time from the moment the request in sent until the first response packet is received. This time represents the total time the application took to assemble the page.
- **Chunked encoding.** Dynamic pages often take a while to assemble. With HTTP 1.1 the server can respond using *chunked encoding*, where each response chunk is sent as soon as it is available (and no Content-Length header is sent). In this case, a direct timing attack obtains more information — one can measure inter-chunk timings to determine how long each part of the page took to assemble.

Neither one of these methods is strictly more secure against timing attacks, and application server providers typically do not consider the security implications, leaving the web server to use its default settings. Apache 2.0, for example, dynamically decides whether to use the Content-Length header or to use chunked encoding. If Apache has data ready to be sent, but it has not yet seen an end-of-stream marker, Apache will use chunked encoding. Otherwise, it will use the Content-Length header.

While this architecture of application code execution is built without any regard to timing vulnerabilities, our timing attacks are effective primarily because the nature of many web applications necessarily depends on private data. The specifics of our timing attacks require no assumptions about the nature of the computation on the server-side.

3. DIRECT TIMING ATTACKS

Our first approach is to directly make requests to a target web server and carefully time the response. Using a custom program to do this, we not only get very accurate timing data (sub-millisecond) and the ability to make arbitrary requests, we also get timing data for each chunk of the response if the server uses chunked encoding, providiing a complete profile of the server's computation.

The ability to make arbitrary requests allows us to test many code paths which are normally not accessed by a properly functioning web browser. In many cases, these are in place purely to prevent accidentally broken or maliciously constructed requests from having ill effects on the application. In these cases, falling back on slow methods in unusual circumstances is normally a perfectly acceptable coding practice for even the most secure web sites. These methods, however, can easily serve as amplification for timing attacks.

In practice, however, we found that the vast majority of vulnerable web sites were vulnerable to the most simple form of attack: regular requests where only the time until the first response packet was received is measured.

3.1 Dealing with noise

In a perfect world, one could time precisely how long it takes the server to generate a response, and it would be the same every time. However, two large factors add a significant amount of noise to the process. One is varying network conditions: long delays and any packet loss can significantly affect overall timing. These conditions are additive noise, since they are not dependent on the request.[2] Another large factor is server load: when a server is handling a great number of requests concurrently, each request takes longer on average. This type of noise can be both additive (queuing

[1]The vulnerability of a SQL query depends on many factors, including the specifics of the query, the type of database server and its query optimization strategies, and the number and type of indexes for the tables involved.

[2]Only requests that are so large as to be spread out over multiple TCP packets would be affected variously by different network conditions. For the experiments in this paper, no such request was ever used.

on the server itself) and multiplicative (n threads of execution on the server take n times as long to complete).

Obviously, if these sources are not as large as the computation itself, they do not pose any difficulty in timing. As they grow more significant, multiple samples can be used to average out the large variance in timing that the noise causes. Specifically, since the noise is strictly non-negative, and in practice very skewed, the sample most likely to have small noise is the one with the smallest absolute time. Therefore, to effectively reduce the noise in these timing tests, we keep only the smallest.

For the purposes of collecting experimental data, a program timed a collection of pages at a given site many times spread uniformly over a reasonable stretch of time. This timing data generated an estimate for the actual distribution of times, and was used to calculate the estimated distribution for the various sampling methods actually used. This allowed us to estimate the reliability of the timing attacks without making millions of requests to commercial web servers.

3.2 Testing for boolean values

The most simple timing attack is a boolean test: does some condition on the server's hidden data hold or not. One such condition is used by attackers today, although with limited success: 'Is this the right password for the specified user?'. Such brute-force password attacks work only when the website is poorly designed (does not limit the rate at which a single user can attempt to log in) and the user has chosen a common or easily guessable password.

However a different attack, with a very similar idea, works surprisingly well on the majority of popular web sites: 'Does this username correspond to a valid user of this site?'. Since a great many web sites use email addresses as usernames, this data can be used to validate large lists of potential email addresses for the purposes of spam[3]. Moreover, knowledge of *which* sites the user of an email address regularly visits is useful for invasive advertising and phishing.

Because most sites do not currently consider this a significant attack, they unwittingly provide attackers with the means to get this data without timing at all, through the ubiquitous 'Forgot my password' page. This page, which is all but required on major sites, often reveals whether the specified email address is associated with a valid user or not. Some sites clearly acknowledge weaknesses in their reset page by adding layers of protection to it, such as a CAPTCHA [5], requiring the additional input of personal information of the account holder, and only sending the password or a means to reset it to the email address of the account holder. However, even well designed sites that clearly consider user account validity to be an important breach of privacy are frequently vulnerable to direct timing of their login page.

Figure 1 gives an example of two popular, high-traffic sites where timing the login page leaks user account validity. The figure shows the mean and standard deviation of the time taken to respond to a login attempt for a set of valid and invalid email addresses, taking the smallest of 10 samples. The mean and standard deviation were computed by taking

[3]Given a list of potential email addresses, an attacker can test each one against a set of popular web sites. This process will not only produce a list of valid email addresses, but also some additional personal data on each.

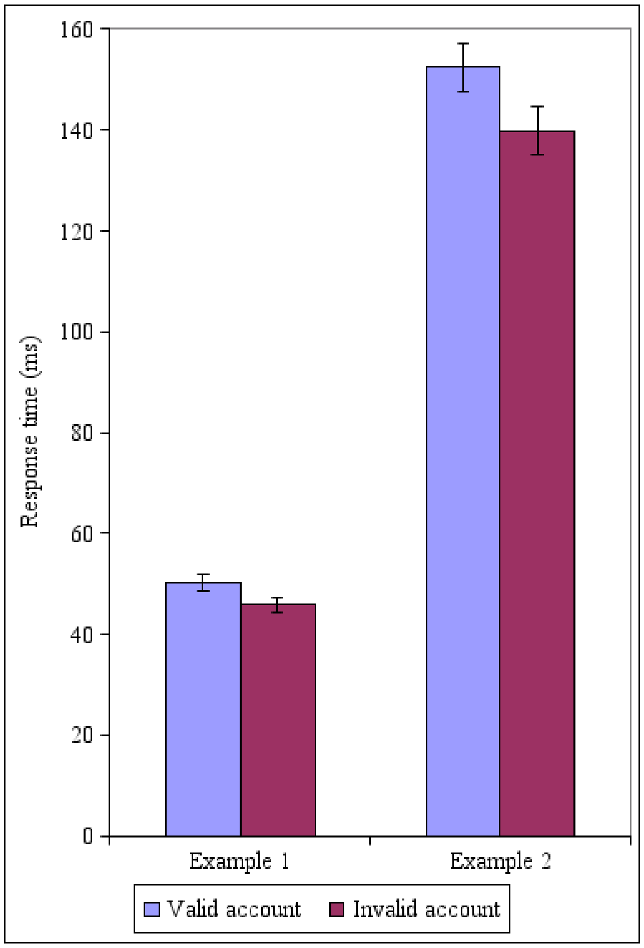

Figure 1: Distinguishing valid from invalid user accounts

many hundreds of samples of each type, and calculating the distribution that would occur when the smallest of 10 random samples is taken. The data clearly shows a separation between valid and invalid emails that is sufficient to predict accurately more than 95% of the time. Using more than 10 samples would provide an even more accurate distinguisher.

3.3 Estimating the size of hidden data

Many computations that go into web applications involve taking data sets and displaying some filtered or processed version of them on a page. Frequently, the actual size of the data set itself is meant to be hidden from the user, based on some sort of access control. Simple examples that are widely used on the web include blogs where individual entries can be shown only to chosen groups of users, and photo galleries where a similar preference can be specified for albums or individual photos. Sometimes, entries can be marked 'private', visible only to the owner of the site, which can be used to edit items before making them visible to the world. The total number of items in the data set, and the relative change in items over time, represent significant hidden data that can often be discovered using timing.

Inherent in the process of dealing with a data set is having

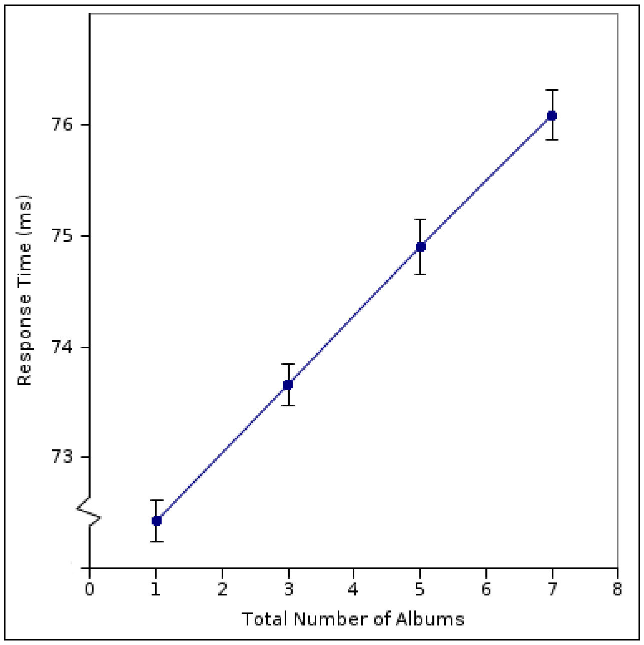

Figure 2: Counting the number of hidden albums in a gallery

to do something for each item: either check a property of it, or compute some derived data from it. These loops depend partly on the number of items and partly on the items themselves. Given that some of the properties become visible to the viewer of the page, it is possible to calculate timing data that has very strong correlation with the total number of items.

Specifically, we tested a popular photo sharing package called Gallery [7]. Taking the smallest of 20 samples was sufficient to distinguish exactly how many albums there were in a gallery, even though the gallery only had one album visible to the world.

Figure 2 shows the approximately linear relationship between the response time and the number of albums. Given the extremely small difference in response times, it was necessary to have a relatively fast and short network path to the target server. However, this attack was successful even though the server was a top-of-the-line machine under no load. This effect may or may not be present on most other web sites of this sort, including popular blogs such as LiveJournal and popular photo sites such as Yahoo's Flickr and Google's Picasa. However, as with the case of Gallery, exploiting this effect may require an unusually fast network path to the target.

4. CROSS-SITE TIMING ATTACKS

With direct attacks, it is only possible to see the 'public' side of the web. If one could make requests as another user, using that user's preferences and login credentials, it would be possible to find out information that is visible to that user alone. Since these preferences and credentials are typically sent automatically in a cookie, we merely need to time these cookie-enabled requests.

Web browsers have taken many steps to prevent one web site from learning anything about requests made by the user's browser to other sites. This broad class of attacks, known as *cross-site*, has been known and studied for some time, but remains a large source of problems on the web. Despite the presence of different preventative measures in modern web browsers, we can nevertheless still *time* cross-site content.

4.1 Browser timing techniques

Given that JavaScript is the most common form of dynamic content on the web, it will come as no surprise that it forms the basis for the most reliable method of timing cross-site content. JavaScript itself is typically prohibited from learning anything about the content of any data that is not hosted on the same domain as the page containing the script — this is a direct application of the same-origin principle. However, script *is* allowed to learn when and whether embedded content loads. This is useful in many different circumstances, including dynamic web pages using AJAX or other techniques.

One way to embed content is through the use of frames. `FRAME`s and `IFRAME`s are the official method for embedding other HTML pages into a site. Frames are very useful in creating pages that combine content from multiple sites without requiring explicit cooperation. JavaScript is provided the `onload` handler for each frame: this event allows JavaScript to be notified when the enclosed page finishes loading. While this technique is able to time a page load, it unfortunately comes with all the baggage of a typical website: all the images, applets, scripts, and stylesheets on the embedded page need to load, then the browser needs to lay out the page – even if the frame is invisible. All this adds an unacceptable amount of noise to the time measurement.

Instead, images are a much more effective method for timing. `IMG` tags are commonly used to embed images into a page, and surprisingly can be used to time *any* web-accessible url. When a source is loaded via an image tag, the browser cannot know in advance that the source will actually be an image. It sends a normal request[4], and when the response header indicates that it is not an image, the browser stops and notifies JavaScript via the `onerror` handler. This allows enterprising JavaScript to accurately time responses for *arbitrary content*.

Our primary technique is to use an invisible image and JavaScript to take several timing samples of the same or different pages sequentially. Figure 3 shows that this code is not at all complicated.

Restricting access to these handlers does not solve this problem, since one can execute a cross-site timing attack without JavaScript at all. Specific tags, such as `LINK` and `SCRIPT`, force the browser to finish downloading and processing one before moving on to the next. Using a tag of this type pointing at a target page in between two tags pointing to the attacker's site, the attacker can remotely time the target page.

The use of these methods to obtain multiple, precise tim-

[4]Using the Accept header in HTTP, a browser actually makes a slightly different request for a source found in an image tag than found in a frame tag. However, in an effort to be maximally compatible, most browsers do not specify that *only* images are acceptable – merely a preference for an image.

```
<html><body><img id="test" style="display: none">
<script>
  var test = document.getElementById('test');
  var start = new Date();
  test.onerror = function() {
    var end = new Date();
    alert("Total time: " + (end - start));
  }
  test.src = "http://www.example.com/page.html";
</script>
</body></html>
```

Figure 3: Example JavaScript timing code

ing samples from an unwilling user's browser are incredibly realistic, since all of these attacks can be done invisibly in the background. A malicious web site need only distract the user for a few seconds for the attacks to complete.

4.2 Why cross-site timing is harder than direct

Unlike a direct attack, a cross-site timing attack does not have a stable, known network configuration. A particular user could have virtually any type of Internet connection at almost any geographical location. Therefore, an absolute comparison of timed responses is not very useful. Instead, a robust cross-site attack must time at least two sources, in order to correct for differences in individual network conditions. One source will be the page whose computation time is dependent on the hidden data the attacker wants to discover. The other source must have as little dependency as possible on the hidden data, to serve as a timing baseline. Almost any page on the web satisfies this second criteria, even pages on the attacker's site; however, the ideal second source is a static page on the target web server that does not depend on hidden data. For example, one could measure the response time for a non-existent page on the same site — most sites' 404 error pages do not depend on user data.

None of the available techniques for cross-site timing can measure arrival times for individual chunks (as can be done in direct attacks, see Section 3). Fortunately, the most valuable timing data – the time from the initial request to the first chunk of the response – is accessible from JavaScript using image tags and the `onerror` handler. [5]

For efficiency of data collection, the timing data for all the cross-site attacks was generated using the same program for collecting the direct timing data. It simulated legitimate browser requests and only gathered the data that would be available to the browser through the use of JavaScript and image tags. The models generated from this timing data were then experimentally verified using an actual in-browser attack page. The timing noise generated by several browsers, including Firefox and Safari, was not found to considerably impact the accuracy of the generated models.

[5]The first chunk is the least likely to be dropped due to the network, least likely to be delayed by TCP window artifacts, and most likely to have been generated after the important queries of the page have been executed and after the hidden data is otherwise processed and output to HTML.

4.3 Testing for boolean values

One obvious attack is to determine what, if any, relationship the user has with a given site. With cross-site timing, it is often possible to distinguish between four types of users: those who have never been to a site, users who have been to a site but never logged in, users who are currently logged in, and users who are not logged in but have logged in sometime in the past. At least one distinguishing attack is not only present but easily exploitable on every major web site tested. Here we give two examples which serve to illustrate two common vulnerabilities present in many other web sites.

In these specific attacks, the goal is to distinguish between a logged in user and all other types of users. Since we require two timing sources, for the first example, the chosen 'test page' was the front page of the website, and the 'reference page' was the 'Contact Us' page. Looking at the difference between the time to load the test and reference pages in Figure 4, we can clearly distinguish a logged in user with only 2 samples per page. This is primarily because the web site in question externally redirects a logged in user from the front page to the primary member page, which adds a full network round-trip to the time it takes the browser to complete the request. The data in the figure also suggests that it may be possible to distinguish whether a user has a cookie. In fact, although not included in the figure, using the difference between the 'Contact Us' page and an arbitrary page that is not present (which returns a 404 error) we can distinguish whether the user has ever been to the web site in question with the same 2 samples per page.[6]

Even though the second example web site does not do any external redirecting, it's reference page takes longer to load for a logged in user than for a user who is not logged in because a logged in user has a more complicated and data-rich page than an anonymous user. This is easily distinguished by the difference between the time to load the reference page and the time to load an arbitrary page that is not present, again with only 2 samples per page.

4.4 Estimating the size of hidden data

Even more so than with direct attacks, there is a tremendous amount of 'countable' data that should only be visible to the user, and not to any arbitrary web site that user connects to. It is impossible to list here all the possible places

[6]Since a user can manually clear all browser state, including cookies and history, at any time, "ever" really means "since the user last cleared his cookies".

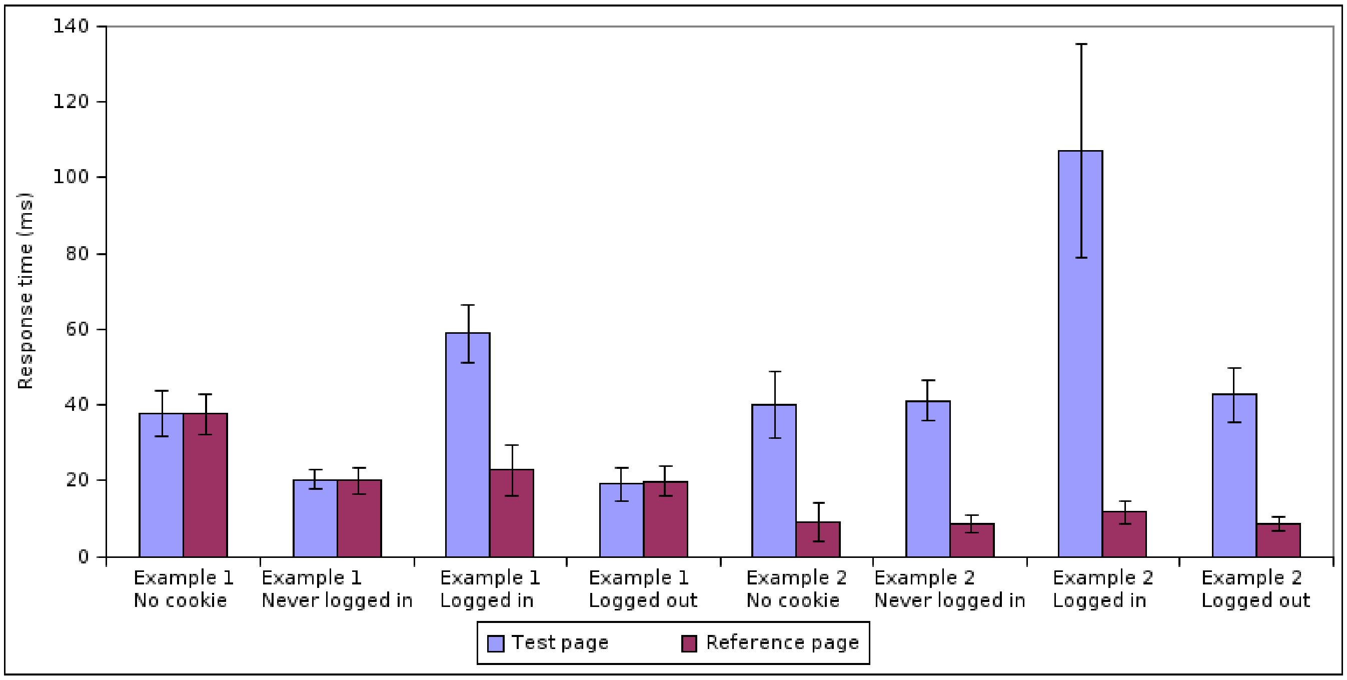

Figure 4: Distinguishing if a user is logged in

in which this scenario is true on the web today, but the most obvious would include counting the number of transactions at a bank or brokerage site, auctions at an auction site, or emails at any of the popular webmail sites. These counts could even be conducted on search results, a common feature of many web sites, giving an attacker the power to see only items meeting some chosen criteria.

As an example, we look at counting the number of items in a user's shopping cart at a popular Internet retailer. Measured at a single moment, it reveals information about that user's overall shopping habits. If a user could be convinced or forced to visit an attacker's web site more than once, the relative change in the shopping cart could be used to infer purchase quantities and dates.

Experimentally, a reference page was chosen whose timing did not substantial depend on the number of items in the shopping cart. This task was not trivial on this site, which includes a feature-filled header on every page. Unexpectedly, the time to compute the header itself was also correlated with the number of items in a shopping cart.

As Figure 5 clearly shows, the difference between the shopping cart and this reference page is linearly related with the number of items in a user's shopping cart very precisely up to the count of 10, which is the number of items the shopping cart will display on a single page. After that, there is still a noticeable dependency, but it is smaller and less precise. Overall, the number of items can be determined with overwhelming probability to within a factor of 10% with only 10 timing samples per page. More samples would allow an attacking site, under realistic conditions, to count exactly. This data is drawn for a user that is not logged in (anonymously browsing and shopping) — for unknown reasons, this attack is more effective for a logged in user.

4.5 Combining with cross-site request forgery

In theory, even more powerful attacks can be created by combining the cross-site timing attack with existing cross-site request forgery. Cross-site request forgery (CSRF) [14] is an attack where one site directs the browser to make a request that actually changes state on another site, even though the browser prevents the attacking site from viewing any data that did not originate from the same domain. A simple and effective solution to this problem is well-known: add a hidden field to every form containing a random string, and check that a valid random string is present in every form request. Despite this ready solution, cross-site request forgery remains a pervasive problem on the web.

Most CSRF attacks that are at the moment annoyances – such as adding specific items to a user's shopping cart – can become a serious privacy breach when combined with timing. For example, an attacker able to add arbitrary items to a user's cart can test if the cart contains a particular item. To see how, recall that shopping carts have a per-item quantity field. Hence counting items in a shopping cart (using cross-site timing) actually counts the number of *distinct* items in the cart. To test if an item is presently in a shopping cart the attacker first counts the current number of items in the cart, it then adds an item, then counts again. If the number of items did not change, then the added item must have already been in the shopping cart. Since a second CSRF can be used to remove the 'test' item, this attack could be executed invisibly.

5. DEFENSES

Generally speaking, any control flow statement that depends on sensitive data could lead to timing vulnerabilities. For example, an application that retrieves a list of records

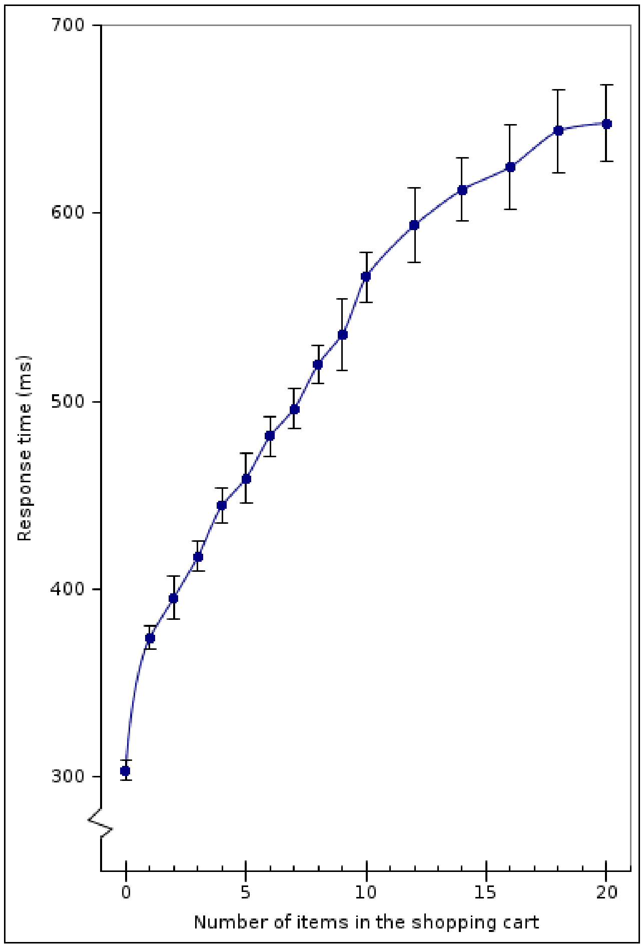

Figure 5: Counting the number of items in a user's shopping cart

from the database and then selectively decides which ones to display will be vulnerable to leaking the total number of records. One could look for such coding patterns to detect basic timing vulnerabilities and correct them, but this is likely to be very error-prone.

One defense is to ensure that the web server always takes a constant amount of time to process a request. Blaze [3] proposed an operating system level mechanism for doing so. A similar system could be built for web servers. However, simply ensuring that total request time is constant is insufficient. If the server is using chunked encoding, inter-chunk timings could reveal sensitive information, even though the total response time is constant. For chunked encoding it is critical that all inter-chunk times are constant.

We implemented this specific defense as an Apache module called **mod_timepad**. The module ensures that each chunk is sent at a time since the request was made which is a multiple of n milliseconds where, say $n = 100$ms. n is a user-adjustable parameter that can be specified for each page, directory, site, and server. If n is set greater than the maximum time to prepare a chunk for a given page, then responding to a request for that page will leak no timing information to an attacker. If n is insufficiently large (for example, if the page could take any amount of time to compute), then the module dramatically reduces the resolution of timing data available to an attacker. While certainly not a perfect solution, this module can be used effectively to thwart the attacks demonstrated in this paper with very little modification to existing web applications.

While the correct way to fix timing vulnerabilities is at the web site, the cross-site timing attack may also be defeated using browser modifications. For example, one could block our JavaScript timing method by applying the same-origin policy to `onerror` and `onload` events. As a result, the attacking site would have no information on how or when the target page was loaded. This approach, however, is very brittle and unlikely to provide security — there are many different methods for measuring page load time and they would all have to be blocked.

Finally, we note that simply adding random delays at the web server will not defeat this timing attack. It will only slow down the attack by forcing the attacker to sample multiple times to average out the noise. The ineffectiveness of random delays was already discussed in [10].

6. CONCLUSION

This paper discusses a pervasive bug in web application software. The fact that timing data at many web sites leaks private information suggests that this side channel is often ignored by web developers. We presented a number of direct and indirect measurement techniques that can effective exploit real-world leaks of private information, including a new *cross-site timing* method that can reveal private user state. While a difficult problem to solve, one approach to fixing these vulnerabilities is carefully controlling the time taken to respond to any request, either through careful server-side coding or a web server module that automatically regulates the time at which responses are sent.

Acknowledgments

We thank John Mitchell, Collin Jackson, Adam Barth, Jeremiah Grossman, and Daniel Walling for their suggestions.

7. REFERENCES

[1] Onur Aciicmez, Werner Schindler, and Cetin Koc. Improving Brumley and Boneh timing attack on unprotected SSL implementations. In *Proceedings of the 12th ACM conference on Computer and communications security*, 2005.

[2] C. Anley. Advanced SQL injection in SQL server applications, 2002. `http://www.nextgenss.com/papers/advanced_sql_injection.pdf`.

[3] Matt Blaze. Simple UNIX time quantization package. Previously available on the web.

[4] D. Boneh and D. Brumley. Remote timing attacks are practical. *Journal of Computer Networks*, 48(5):701–716, 2005. Extended abstract in Usenix Security 2003.

[5] The CAPTCHA project. `http://www.captcha.net`.

[6] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *ACM Conference on Computer and Communications Security*, pages 25–32, 2000.

[7] Gallery. `http://gallery.menalto.com/`.

[8] Collin Jackson, Andrew Bortz, Dan Boneh, and John Mitchell. Protecting browser state from web privacy attacks. In *Proceedings of the 15th ACM World Wide Web Conference (WWW 2006)*, 2006.

[9] Markus Jakobsson. Modeling and preventing phishing attacks, 2005. `http://www.informatics.indiana.edu/markus/papers/phishing_jakobsson.pdf`.

[10] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. *Advances in Cryptology*, pages 104–113, 1996.

[11] Jesse Ruderman. The same origin policy, 2001. `http://www.mozilla.org/projects/security/components/same-origin.html`.

[12] Werner Schindler. A timing attack against RSA with the chinese remainder theorem. In *CHES 2000*, pages 109–124, 2000.

[13] Werner Schindler. Optimized timing attacks against public key cryptosystems. *Statistics and Decisions*, 20:191–210, 2002.

[14] Chris Shiflett. Cross-site request forgeries, 2004. `http://shiflett.org/articles/security-corner-dec2004`.

[15] The cross-site scripting FAQ. `http://www.cgisecurity.net/articles/xss-faq.shtml`.