[HACK DAT KIWI 2017]
[Crypto Writeup]

[shediyo @ yoburek]

Initial analysis

In all PS challenges (PS1, PS2, PS4, PSP) we get a message encrypted using a given PS cipher, but each time with different parameters.

The cipher code is included in the last section.

First we note that the message is padded with whitespaces via the str_pad function:

```
$msg=pad($message,BLOCK_SIZE);
```

```
➔ str_pad($message, max(floor( (strlen($message)-1)/$block_size)+1,1)*$block_siz
  e)
```

This is interesting, since it gives us known-plaintext values at the end.

Second, we check the key generation:

```
$key=hex2bin(md5($key));

$key=pad($key,BLOCK_SIZE);
```

The relevant (effective) secret key is actually the binary md5 of the initial secret key, and that is the one we are actually going to break.

Next we check the round key choice:

```
$roundkey=$key;

if (defined("PLUS")) $roundkey=pad(md5($key.$i),BLOCK_SIZE);
```

In the normal mode we have the same key used in each round, which can be abused if the same byte of the key is mixed several times, reducing the time for the key counting (exhaustive search) attacks.

The PLUS parameter looks like a hardening of the cipher in first sight (key repetition attacks as the one mentioned above are not possible), but there is actually a nice vulnerability here.
Notice the difference between the initial generated key and the round key – in the round key we will have a hex digest of the md5, from which only the first half will be used as the round key.
In this case each key byte has only 16 possible values (0 -9, a-f), which reduces significantly the effective key size and so the counting time needed to find the key.

Now check the core of the cipher:

```
$block=sbox($block,$i,$decrypt);
```

```
➔ for ($i=0;$i<BLOCK_SIZE;++$i) $out[$i]=chr($sbox[$round][ord($block[$i])]);
```

```
$block=pbox($block,$i,$decrypt);
```

```
➔ for ($i=0;$i<BLOCK_SIZE;++$i) $out[$i]=$block[$pbox[$round][$i]];
```

```
$block=xbox($block,$roundkey,$i,$decrypt);
```

```
➔ for ($i=0;$i<BLOCK_SIZE;++$i) $out[$i]=chr((ord($block[$i])^ord($key[$i])));
```

We can see the cipher as having a block length of 1, wuth a byte to byte path passing each round through an SBOX table and xoring with a different byte of the key, this path is defined by the permutations (PBOX tables) and can actually have key byte repetitions.

In the breaking of the cipher we can look at all bytes in relative-block-position i for $0 \leq i < 16$ and try to break their "key-path" with a smart exhaustive search, we will abuse the fact that the plaintext is printable (characters less than 0x80) to disqualify the incorrect keys - since the text is long, only the correct key will qualify.

PS1

The easiest - one round, the “key-path” consists of one sbox calculation and one key byte, easily enumerated separately for each position – $2^8 * len(msg)$ calculations overall, which is extremely fast.

Note that I was guessing here that the message will be ascii-printable, and that provides a clean and easy disqualification of incorrect keys.

```
<CODE>
msg = enc_msg.replace(' ','').decode('hex')

found_key = '\x00' * 16
for j in xrange(16):
        correct_key = found_key
        for k in xrange(256):
                failed = False
                for pos in xrange(j, len(msg), 16):
                        if SBOX_TABLE[0].index( ord(msg[pos]) ^ k ) > 0x80:
                                failed = True
                                break

                if not failed:
                        print 'Found key ', str(k), 'for position: ', str(j)
                        correct_key = found_key[:j] + chr(k) + found_key[j+1:]

        found_key = correct_key

print repr(found_key)
print ps(msg, found_key, True, True)
</CODE>
```

PS2

This time we have 2 rounds, each “key-path” consists of two key bytes, again can be enumerated separately for each position – $2^{16} * len(msg)$ calculations overall, which is also extremely fast.

```
<CODE>
msg = enc_msg.replace(' ','').decode('hex')

found_key = '\x00' * 16
key_index_pairs = [(i, PBOX_TABLE[1][i]) for i in range(16)]
for p in key_index_pairs:
        correct_key = found_key
        for k1 in xrange(256):
                for k2 in xrange(256):
                        failed = False
                        for pos in xrange(p[0], len(msg), 16):
                                if SBOX_TABLE[0].index( SBOX_TABLE[1].index(
                                    ord(msg[pos]) ^ k1 )^ k2 ) > 0x80:
                                          failed = True
                                          break

                        if not failed:
                                print 'Found key', str((k1, k2))
                                correct_key = found_key[:p[0]] + chr(k1) +
                                                    found_key[p[0] + 1:]
                                correct_key = correct_key[:p[1]] + chr(k2) +
                                                    correct_key[p[1] + 1:]

        found_key = correct_key
# We actually get several values for keys (4,7) - the last one is the correct
one :)
print repr(found_key)
print ps(msg, found_key, True, True)
</CODE>
```

PS4

This time we have 4 rounds, which is 4 key bytes per "path", so enumerating like before will cost us too much time - $2^{32} * len(msg)$.

So we check if we have any "key-paths" which reuse several key bytes, this is probable since the permutations are random.

The decryption key-position paths we get from the current permutations are as follows:

15 -> 1 -> 5 -> 6 [-> 0]
14 -> 13 -> 9 -> 0 [-> 5]
13 -> 8 -> 7 -> 1 [-> 4]
12 -> 4 -> 12 -> 14 [-> 15]
11 -> 5 -> 2 -> 13 [-> 12]
10 -> 12 -> 3 -> 7 [-> 1]
9 -> 0 -> 8 -> 11 [-> 14]
8 -> 7 -> 4 -> 8 [-> 9]
7 -> 9 -> 13 -> 4 [-> 6]
6 -> 14 -> 10 -> 3 [-> 13]
5 -> 6 -> 15 -> 10 [-> 2]
4 -> 2 -> 1 -> 9 [-> 3]
3 -> 10 -> 11 -> 15 [-> 11]
2 -> 15 -> 0 -> 2 [-> 10]
1 -> 11 -> 6 -> 5 [-> 7]
0 -> 3 -> 14 -> 12 [-> 8]

The green paths are the ones with key byte repetitions, following the break order below we won't have more than 3 key bytes to count each time:
12 -> 4 -> 12 -> 14 [-> 15]
6 -> 14 -> 10 -> 3 [-> 13]
3 -> 10 -> 11 -> 15 [-> 11]
1 -> 11 -> 6 -> 5 [-> 7]
11 -> 5 -> 2 -> 13 [-> 12]
8 -> 7 -> 4 -> 8 [-> 9]
9 -> 0 -> 8 -> 11 [-> 14]

Now that gives us a run-time bounded by $2^{24} * len(msg)$ which is feasible, but we can do better.

Notice that we need to break 3 key bytes only for the two first paths, and their relative positions in the cipher blocks are 15, 13.
We remember that we have whitespace padding at the end, we can reduce the checks using MITM (meet in the middle):

1. We map in a dictionary each 2-round encryption of a whitespace for all (k1, k2) to the key pair
2. We calculate the decryption of the corresponding cipher byte for each key pair (k3, k4) and take only the key pairs in the dictionary matching the calculated middle value.

We'll be left with roughly 2^{24} pairs which satisfy the 8 bit 'meet' condition.

Now from those pairs we will only take the ones satisfying the key conditions (k1 = k3, k2 = [14^{th} key byte]) and only those tuples will be passed through the ascii checks.

Using this method our runtime is bounded by $2^{24} + 2^{16} * len(msg)$ which is much better.

The generic code for key breaking:

<CODE>

```
def find_good_key_tuples(dec_val, msg_val, bad_cond):
        msg_dict = {}
        for i in xrange(256):
                msg_dict[i] = []

        tups = []
        for i in xrange(256):
                for j in xrange(256):
                        msg_dict[ SBOX_TABLE[2].index( SBOX_TABLE[3].index(
                                ord(msg_val) ^ i ) ^ j )  ].append((i, j))

        for i in xrange(256):
                for j in xrange(256):
                        for k in msg_dict[
                        SBOX_TABLE[1][SBOX_TABLE[0][
                        ord(dec_val)] ^ i] ^ j ]:
                                p = (k[0], k[1], j, i)
                                if not bad_cond(p):
                                        tups.append (p)

        return tups

def break_key_seq(msg, pos, dec_val, bad_cond):
        tups = find_good_key_tuples(dec_val, msg[len(msg) - 16 + pos],
                                        bad_cond)
        good_tups = []
        for p in tups:
                failed = False
                for s in xrange(pos, len(msg), 16):
                        z = SBOX_TABLE[2].index( SBOX_TABLE[3].index(
                                ord(msg[s]) ^ p[0] ) ^ p[1] )
                        if SBOX_TABLE[0].index( SBOX_TABLE[1].index(
                                                        z ^ p[2] ) ^ p[3] ) > 0x80:
                                failed = True
                                break

                if not failed:
                        good_tups.append(p)

        return good_tups
```

</CODE>

We use it as follows:

```
<CODE>
msg = enc_msg.replace(' ','').decode('hex')
found_key = [-1] * 16

# 12 -> 4 -> 12 -> 14 [-> 15]
found = break_key_seq(msg, 12, ' ',lambda p: p[0] != p[2])
print 'First key seqs:', found, 'for pos:', str((12, 4, 12, 14))
assert len(found) == 1
found = found[0]
found_key[12] = found[0]
found_key[4] = found[1]
found_key[14] = found[3]

# 6 -> 14 -> 10 -> 3 [-> 13]
found = break_key_seq(msg, 6, ' ',lambda p: p[1] != found_key[14])
print 'Second key seqs:', found, 'for pos:', str((6, 14, 10, 3))
assert len(found) == 1
found = found[0]
found_key[6] = found[0]
found_key[10] = found[2]
found_key[3] = found[3]

# The next ones are easier and not written here :)
</CODE>
```

PSP

In this challenge we have 8 rounds, but we also have the PLUS property which reduces our key size significantly because of the hex-digits vulnerability.
We have 8 keys with 16 options for each so counting over all keys naively will take us $16^8 * len(msg) = 2^{32} * len(msg)$ break time.

This time, we do not have paths containing repeating keys, but we can use the MITM method to reduce the calculation time for the paths at the padding offsets, reducing to $2^{24} * len(msg)$ break time.

We can reduce even more via guessing - we will guess an additional pair for the MITM check, this time "meeting" on 16 bits and promising that roughly 2^{16} will pass the initial tuple choice and so we get $2^{16} * len(msg)$ calculation time for each guess.

First we guess naively options "a"-"z" for each position before the last block, the chance of hitting a correct position is high since we have an ascii text as the plaintext, this way we find the last 4 plain bytes out of each 16 bytes, having the whitespace padding at the end.

After that, we do not have whitespace padding, but we can do educated guesses of the following bytes based on the known bytes.

For example - after getting the last 4 bytes we can print the 5 last bytes from each cipher block:

```
'TThat'
'$e th'
'pally'
'\x02age '
'Z Cha'
'\xb7 lit'
'T Hit'
'\xe80 or'
'\x95d he'
'Tfor '
'THis '
'$ Cha'
'\xe2 mag'
'\xb7ours'
'\xbb    '
```

We can see that 'T' it is probably a whitespace (offset 5 from the end), and 'p' is probably 'e' (offset 13 from the end), we can check those guesses and if we get a good key tuple – we proceed to the next byte.

The generic code for key breaking:

```
<CODE>
def find_good_key_tuples(dec_val1, msg_val1, dec_val2, msg_val2, ms):
        search_range = range(0x30, 0x3a) + range(0x61, 0x67)
        msg_dict = {}
        for i in xrange(256):
                for j in xrange(256):
                        msg_dict[(i, j)] = []

        tups = []
        for i1 in search_range[:]:
                for i2 in search_range[:]:
                        for j1 in search_range[:]:
                                for j2 in search_range[:]:
                                  z7 = SBOX_TABLE[7].index(ord(msg_val1) ^ i1)
                                  z6 = SBOX_TABLE[6].index( z7 ^ i2 )
                                  z5 = SBOX_TABLE[5].index( z6 ^ j1 )
                                  z4 = SBOX_TABLE[4].index( z5 ^ j2 )
                                  z7 = SBOX_TABLE[7].index(ord(msg_val2) ^ i1)
                                  z6 = SBOX_TABLE[6].index( z7 ^ i2 )
                                  z5 = SBOX_TABLE[5].index( z6 ^ j1 )
                                  y4 = SBOX_TABLE[4].index( z5 ^ j2 )
                                  msg_dict[(z4, y4)].append((i1, i2, j1, j2))

        for i1 in search_range[:]:
                for i2 in search_range[:]:
                        for j1 in search_range[:]:
                                for j2 in search_range[:]:
                                  z0 = SBOX_TABLE[0][ord(dec_val1)] ^ i1
                                  z1 = SBOX_TABLE[1][z0] ^ i2
                                  z2 = SBOX_TABLE[2][z1] ^ j1
                                  z3 = SBOX_TABLE[3][z2] ^ j2
                                  z0 = SBOX_TABLE[0][ord(dec_val2)] ^ i1
                                  z1 = SBOX_TABLE[1][z0] ^ i2
                                  z2 = SBOX_TABLE[2][z1] ^ j1
                                  y3 = SBOX_TABLE[3][z2] ^ j2
                                  for k in msg_dict[(z3, y3)]:
                                    failed = False
                                    for m in ms:
                                      z7 = SBOX_TABLE[7].index( m ^ k[0] )
                                      z6 = SBOX_TABLE[6].index( z7 ^ k[1] )
                                      z5 = SBOX_TABLE[5].index( z6 ^ k[2] )
                                      z4 = SBOX_TABLE[4].index( z5 ^ k[3] )
                                      z3 = SBOX_TABLE[3].index( z4 ^ j2 )
                                      z2 = SBOX_TABLE[2].index( z3 ^ j1 )
                                      z1 = SBOX_TABLE[1].index( z2 ^ i2 )
                                      z0 = SBOX_TABLE[0].index( z1 ^ i1 )
                                        if z0 > 0x80:
                                          failed = True
                                          break
                                        if not failed:
                                          p = (k[0], k[1], k[2], k[3],
                                                j2, j1, i2, i1)
                                          tups.append(p)
        return tups
```

```
def change_in_offset(found_key, offss, off_val):
	return found_key[:offss] + chr(off_val) + found_key[offss + 1:]

def break_and_update_key(msg, change_bytes, found_keys, dec_val1, offset1,
				dec_val2 = None, offset2 = None):
	pos = change_bytes[0]
	ms = [ord(msg[s]) for s in xrange(pos, len(msg), 16)]
	if dec_val2 is None:
		for offset2 in xrange(3, len(msg)):
			print 'Guess offset: ' + str(offset2)
			if offset2 == offset1:
				continue
			for guess in xrange(ord('a'), ord('z')):
				tups = find_good_key_tuples(dec_val1,
					msg[len(msg) - 16 * offset1 + pos],
					chr(guess),
					msg[len(msg) - 16 * offset2 + pos],
					ms)
				if len(tups) != 0:
					print 'Found!: ' + chr(guess)
					break
			if len(tups) != 0:
				break
	else:
		tups = find_good_key_tuples(dec_val1,
			msg[len(msg) - 16 * offset1 + pos], dec_val2,
			msg[len(msg) - 16 * offset2 + pos], ms)

	assert len(tups) == 1
	print tups[0]
	new_found_keys = found_keys[:]
	change_vals = list(tups[0])[::-1]
	change_bytes = change_bytes[::-1]
	for i in xrange(len(change_bytes)):
		new_found_keys[i] = change_in_offset(new_found_keys[i],
					change_bytes[i] , change_vals[i])
	return new_found_keys
```
</CODE>

The usage is as follows:

```
<CODE>
msg = enc_msg.replace(' ','').decode('hex')

found_keys = [0] * 8
for i in range(8):
        found_keys[i] = '\x00' * 16

# 15 -> 6 -> 15 -> 1 -> 9 -> 3 -> 7 -> 14 [-> 15]
found_keys = break_and_update_key(msg, [15, 6, 15, 1, 9, 3, 7, 14],
                                      found_keys, ' ', 1)

# 5 -> 8 -> 1 -> 2 -> 6 -> 13 -> 5 -> 4 [-> 14]
found_keys = break_and_update_key(msg, [5, 8, 1, 2, 6, 13, 5, 4], found_keys,
                                      ' ', 1)

# 3 -> 5 -> 2 -> 7 -> 0 -> 1 -> 4 -> 0 [-> 13]
found_keys = break_and_update_key(msg, [3, 5, 2, 7, 0, 1, 4, 0], found_keys,
                                      ' ', 1)

# 1 -> 3 -> 14 -> 0 -> 4 -> 4 -> 6 -> 3 [-> 12]
found_keys = break_and_update_key(msg, [1, 3, 14, 0, 4, 4, 6, 3], found_keys,
                                      ' ', 1)

'''
# find next byte guesses
dec = ps_multi(msg, found_keys, True)
for i in range(len(dec) - 16 * 33, len(dec), 16):
        print repr(dec[i + 11: i + 16])
'''

# Now we have educated guesses based on the decrypted values

# 7 -> 15 -> 3 -> 14 -> 3 -> 2 -> 10 -> 2 [-> 11]
found_keys = break_and_update_key(msg, [7, 15, 3, 14, 3, 2, 10, 2],
                                      found_keys, ' ', 5, 'e', 13)

# 6 -> 1 -> 13 -> 10 -> 14 -> 12 -> 1 -> 11 [-> 10]
found_keys = break_and_update_key(msg, [6, 1, 13, 10, 14, 12, 1, 11],
                                      found_keys, ' ', 2, 'r', 13)

# 9 -> 4 -> 5 -> 4 -> 8 -> 0 -> 9 -> 7 [-> 9]
found_keys = break_and_update_key(msg, [9, 4, 5, 4, 8, 0, 9, 7], found_keys,
                                      ' ', 7, 'a', 11)

# 0 -> 7 -> 12 -> 8 -> 5 -> 11 -> 14 -> 13 [-> 8]
found_keys = break_and_update_key(msg, [0, 7, 12, 8, 5, 11, 14, 13],
                                      found_keys, ' ', 4, 'c', 14)

# 11 -> 13 -> 0 -> 11 -> 10 -> 9 -> 13 -> 9 [-> 7]
found_keys = break_and_update_key(msg, [11, 13, 0, 11, 10, 9, 13, 9],
                                      found_keys, ' ', 10, 'e', 14)
```

```
# 4 -> 10 -> 7 -> 15 -> 1 -> 15 -> 0 -> 10 [-> 6]
found_keys = break_and_update_key(msg, [4, 10, 7, 15, 1, 15, 0, 10],
                                          found_keys, ' ', 6, 'b', 14)

# 14 -> 9 -> 11 -> 6 -> 7 -> 6 -> 2 -> 1 [-> 5]
found_keys = break_and_update_key(msg, [14, 9, 11, 6, 7, 6, 2, 1],
                                          found_keys, 'o', 1, ' ', 14)

# 2 -> 2 -> 10 -> 13 -> 13 -> 14 -> 11 -> 6 [-> 4]
found_keys = break_and_update_key(msg, [2, 2, 10, 13, 13, 14, 11, 6],
                                          found_keys, 'h', 9, ' ', 4)

# 12 -> 14 -> 9 -> 9 -> 15 -> 10 -> 3 -> 8 [-> 3]
found_keys = break_and_update_key(msg, [12, 14, 9, 9, 15, 10, 3, 8],
                                          found_keys, 'd', 13, ' ', 1)

# 10 -> 11 -> 4 -> 12 -> 2 -> 8 -> 8 -> 5 [-> 2]
found_keys = break_and_update_key(msg, [10, 11, 4, 12, 2, 8, 8, 5],
                                          found_keys, 'y', 11, ' ', 2)

# 13 -> 12 -> 8 -> 5 -> 11 -> 5 -> 15 -> 12 [-> 1]
found_keys = break_and_update_key(msg, [13, 12, 8, 5, 11, 5, 15, 12],
                                          found_keys, ' ', 16, 'a', 18)

# 8 -> 0 -> 6 -> 3 -> 12 -> 7 -> 12 -> 15 [-> 0]
found_keys = break_and_update_key(msg, [8, 0, 6, 3, 12, 7, 12, 15],
                                          found_keys, 't', 9, ' ', 1)

dec = ps_multi(msg, found_keys, True)
print dec
</CODE>
```

PS Cipher Code

<CODE>

```
<?php
function pad($message,$block_size)
{
        return str_pad($message, max(floor( (strlen($message)-
                        1)/$block_size)+1,1)*$block_size);
}
function sbox($block,$round=0,$reverse=false)
{
        static $sbox=null;
        if ($sbox===null) //generate sbox
        {
                srand(SEED);
                $sbox=array_fill(0, ROUNDS, array_fill(0,256,0));
                for ($k=0;$k<ROUNDS;++$k)
                {

                        $base=range(0,255);
                        for ($i=0;$i<256;++$i)
                        {
                                $r=rand(0,count($base)-1);
                                $index=array_keys($base)[$r];
                                $sbox[$k][$i]=$base[$index];
                                unset($base[$index]);
                        }
                }
        }

        $out=str_repeat(" ", BLOCK_SIZE);
        if ($reverse)
                for ($i=0;$i<BLOCK_SIZE;++$i)
                        $out[$i]=chr(array_search(ord($block[$i]),
$sbox[$round]));
        else
                for ($i=0;$i<BLOCK_SIZE;++$i)
                        $out[$i]=chr($sbox[$round][ord($block[$i])]);
        return $out;
}
function pbox($block,$round=0,$reverse=false)
{
        srand(SEED);
        static $pbox=null;
        if ($pbox===null) //generate pbox
        {
                srand(SEED);
                $pbox=array_fill(0, ROUNDS, array_fill(0,BLOCK_SIZE,0));
                for ($k=0;$k<ROUNDS;++$k)
                {

                        $base=range(0,BLOCK_SIZE-1);
                        for ($i=0;$i<BLOCK_SIZE;++$i)
                        {
                                $r=rand(0,count($base)-1);
                                $index=array_keys($base)[$r];
```

```
                                $pbox[$k][$i]=$base[$index];
                                unset($base[$index]);
                        }
                }
        }
        $out=str_repeat(" ", BLOCK_SIZE);
        if ($reverse)
                for ($i=0;$i<BLOCK_SIZE;++$i)
                        $out[$pbox[$round][$i]]=$block[$i];
        else
                for ($i=0;$i<BLOCK_SIZE;++$i)
                        $out[$i]=$block[$pbox[$round][$i]];
        return $out;
}
function xbox($block,$key)
{
        $out=str_repeat(" ", BLOCK_SIZE);
        for ($i=0;$i<BLOCK_SIZE;++$i)
                $out[$i]=chr( (ord($block[$i])^ord($key[$i]))%256 );
        return $out;
}
function ps2_block($block,$key,$decrypt=false)
{
        $key=hex2bin(md5($key));
        $key=pad($key,BLOCK_SIZE);
        if ($decrypt)
                for ($i=ROUNDS-1;$i>=0;--$i)
                {
                        $roundkey=$key;
                        if (defined("PLUS"))
                                $roundkey=pad(md5($key.$i),BLOCK_SIZE);
                        $block=xbox($block,$roundkey,$i,$decrypt);
                        $block=pbox($block,$i,$decrypt);
                        $block=sbox($block,$i,$decrypt);
                }
        else //encrypt
                for ($i=0;$i<ROUNDS;++$i)
                {
                        $roundkey=$key;
                        if (defined("PLUS"))
                                $roundkey=pad(md5($key.$i),BLOCK_SIZE);
                        $block=sbox($block,$i,$decrypt);
                        $block=pbox($block,$i,$decrypt);
                        $block=xbox($block,$roundkey,$i,$decrypt);
                }
        return $block;
}
function ps2($message,$key,$decrypt=false)
{
        $msg=pad($message,BLOCK_SIZE);
        $blocks=str_split($msg,BLOCK_SIZE);
        $res=[];
        foreach ($blocks as $block) //ECB mode
                $res[]=ps2_block($block,$key,$decrypt);
        return implode($res);
}
</CODE>
```