Wed, 23 Apr 2014

plaidCTF 2014 - parlor (crypto250)

By aDR4eA

Filed under plaidCTF2014 blog Crypto length extension

For PlaidCTF2014, Eindbazen and fail0verflow joined forces as 0xffa, the Final Fail Alliance. Don't miss out on other write-ups at Eindbazen's site!

```
The Plague is running a betting service to build up funds for his massive
empire. Can you figure out a way to beat the house? The service is running
at 54.197.195.247:4321.
```

Shall we play a game?

```
/-----------------------------------------------------------------------------\
| Welcome to the betting parlor!                                              |
|                                                                             |
| We implement State of the Art cryptography to give you the fairest and most |
| exciting betting experience!                                                |
|                                                                             |
| Here's how it works: we both pick a nonce, you tell us odds, and you give us |
| some money.                                                                 |
| If md5(our number + your number) % odds == 0, you win bet amount*odds.     |
| Otherwise, we get your money! We're even so nice, we gave you $1000 to start.|
|                                                                             |
| If you don't trust us, we will generate a new nonce, and reveal the old nonce|
| to you, so you can verify all of our results!                               |
|                                                                             |
| (Oh, and if you win a billion dollars, we'll give you a flag.)             |
\_____________________________________________________________________________/

====================
  1) set your odds
  2) set your bet
  3) play a round
  4) get balance
  5) reveal nonce
  6) quit
====================
```

How about Global Thermonuclear War?

```
1
Please pick odds (as a power of 2 between 1 and 100): 16
Odds set to 2^16, good luck!
[... banner ...]
3
Okay, send us a nonce for this round!
fail
Betting $1 at odds of 2^16
Too bad, we generated 25514, not 0... better luck next time!
[... banner ...]
5
What? You think we're cheaters? Fine, the nonce has been 41716a6f6093e9f60529dfa9cff590e3
Who is the cheater now, huh?
```

Let's see if they cheat:

```
>>> import hashlib
>>> int(hashlib.md5("41716a6f6093e9f60529dfa9cff590e3".decode('hex') \
... + "fail").hexdigest(), 0x10) % (1 << 16)
56779L
```

Hm. That didn't work.

```
>>> int(hashlib.md5("41716a6f6093e9f60529dfa9cff590e3" + \
... "fail").hexdigest(), 0x10) % (1 << 16)
4795L
>>> int(hashlib.md5("41716a6f6093e9f60529dfa9cff590e3" + \
... "fail".encode('hex')).hexdigest(), 0x10) % (1 << 16)
34010L
```

That neither. Maybe the server *does* cheat in the end? One more attempt:

```
>>> int(hashlib.md5("41716a6f6093e9f60529dfa9cff590e3".decode('hex') \
... + "fail\n").hexdigest(), 0x10) % (1 << 16)
25514L
```

Ah! This now makes more sense. So let's recap: We send a string (actually: binary blob) to the server, the server prepends a secret, calculates the MD5 digest of the result, takes as many bits as we requested as odds, and if the result is zero, we win. The value we win is the odds multiplied with the amount we bet.

Now, assuming that MD5 is randomly distributed, the expected value is exactly - zero (-$bet + $bet * $odds * (1 / $odds)). On average, we're losing as much as we win. So, is this a fair game?

One attempt would be to bet $0, try random values until we win (which happens 50% for $odds=2^1), and keep on supplying the same (known-good) value, with a maximum bet then of course.

Let's try this:

```
[... banner ...]
1
Please pick odds (as a power of 2 between 1 and 100): 1
Odds set to 2^1, good luck!
[... banner ...]
3
Okay, send us a nonce for this round!
fail0
Betting $1 at odds of 2^1
Too bad, we generated 1, not 0... better luck next time!
[... banner ...]
3
Okay, send us a nonce for this round!
fail1
Betting $1 at odds of 2^1
Wow! You won, congratulations!
[... banner ...]
3
Okay, send us a nonce for this round!
fail1
What part of NONCE don't you understand, asshole?
[... disconnect ...]
```

Ok, that didn't work.

Also, the game.

How else can we cheat? We can't ask for the nonce, because the server generates a new nonce afterwards. However the server returns us up to 100 bits of the result as part of the "Too bad, we generated <...>, not 0..." message.

The answer is a length-extension attack - Merkle-Damgård FTW! Basically, MD5 can be described as runnning a function `h'=F(h, data)` over each 64-byte block of `data`. The initial `h` value is fixed for MD5, and there is padding at the end to assure that the last block of `data` is 64-byte in size. For this, a specific string (which includes the message length) is appended to the message. If there is not enough space in the current 64-byte block for this padding, a new block is appended.

So let's say we upload a 116-byte `client_nonce`. MD5 would operate on three blocks:

1. `server_nonce[0..15] + client_nonce[0..47]`
2. `client_nonce[48..111]`
3. `pad(client_nonce[112..115], 116)`.

(`pad(data, size)` is described here).

During the bet, the following steps are executed:

1. `h := h_initial`
2. `h := F(h, server_nonce[0..15] + client_nonce[0..47])`
3. `h := F(h, client_nonce[48..111])`
4. `h := F(h, pad(client_nonce[112..115], 116))`
5. return `h % $odds`

If we can figure out any intermediate result, for example the value of `h` after step 2 or 3, then we can predict the final result, since we know all remaining information. Even better, we could then go and modify the last part of the `client_nonce`, which still allows us to predict the final result, but allows us to play multiple times.

But how can we figure out intermediate results? Easy - the server tells us up to 100 bits at the end, so by sending a short client_nonce, we can get an intermediate result. However, it always includes a padding block at the end. So even if we send a `client_nonce` of just 48 bytes (completing an MD5 block together with the server nonce), we get

1. `h := h_initial`
2. `h := F(h, server_nonce[0..15] + client_nonce[0..47])`
3. `h := F(h, pad("", 64))`
4. return `h % $odds`

Winning the game

If we set `client_nonce[48..111] = pad("", 64)`, in both cases the first three steps are identical. In the second version, we get 100 bits of the final result - which is the same as the intermediate result for the first version!

Knowing the intermediate result, we can easily modify `client_nonce[112..115]` to whatever we want - and we can predict the MD5 digest.

But - we only know 100 bits of the intermediate result, and without knowing the complete intermediate result, we can't predict *anything* of the final result. But we can obtain a few final results, and just guess the remaining 28 bits (MD5, after all, only produces 128-bit of digest) of the first result, until we can reproduce the few final results that we collected.

There is one more complication - the padding block is not valid ASCII. In fact, what do we do with the `\n`? Doesn't `client_nonce[47]` has to be `\n`, which means that for the full `client_nonce` we'll have a `\n` in the middle?

In short - no. This also reveals why `\n` is part of the calculation. There is no special handling for `\n`. The server is not waiting for a single line, it's waiting for a single TCP packet, probably via a single `read()`-call. This allows us to send anything we want, including as many newline-characters and zero-bytes as we want.

Once the intermediate result is known, you still cannot directly generate nonces that win, but you can test a nonce before submitting to the server. By just randomizing `client_nonce[112..115]` you can eventually find a nonce that will win. There is a trade-off how much local processing vs. how much network transactions to do. You can either set the odds high (the goal was to get $1B, so with the initial balance of $1000 this would require winning with at least 1:2^20), or low (say, 1:1), and play multiple times.

(This is a of course just me not wanting to admit that playing multiple times with low odds is a much clever way than brute-forcing the lower 20 bits to become zero. But flag is flag. `i_dunno_i_ran_out_of_clever_keys`, to be specific.)

© fail0verflow, all rights reserved. Not a member of the Cheezburger© Group.