ymgve added SharifCTF7 Blobfish writeup Latest commit `eb9fe58` on Dec 20 2016

..

Blobfish.py	added SharifCTF7 Blobfish writeup	5 months ago
README.md	added SharifCTF7 Blobfish writeup	5 months ago
blobfish.tar.xz	added SharifCTF7 Blobfish writeup	5 months ago
log.txt	added SharifCTF7 Blobfish writeup	5 months ago
solver.py	added SharifCTF7 Blobfish writeup	5 months ago
test_sbox.py	added SharifCTF7 Blobfish writeup	5 months ago

README.md

Blobfish - Cryptography - 300 points

> We designed our own proprietary cipher, called Blobfish. We were very proud of it, until someone mounted a chosen-plaintext attack on the cipher, and sent the master key to us. He requested a large amount of money to show us the weakness in Blobfish. Fortunately, we have the queries (plaintexts) he made to our system, as well as the responses (ciphertexts).
>
> Our CEO decided to hire someone less greedy to help us redesign Blobfish. But he must first prove he is worthy. Look at the log, and the cipher design. If you can find the master key, use the function `make_flag()` on it to generate the flag. Send us the flag, and we'll get back to you for further negotiations.
>
> PS: `make_flag()` is defined at the end of Blobfish.py.

We are given a list of 200 plaintext -> ciphertext pairs done with an unknown encryption key, and our job is to find the key.

The Blobfish cipher is a simple Substitution-permutation network. It has a block size of 36 bits, and the substitution step uses a 6-to-6 bit S-box. It has 3 rounds, with each round using a separate 36-bit key. (As an aside, this is a totally different structure than the Blowfish cipher, which uses a Feistel network, so not sure why they picked that name)

The overall structure of the encryption of a block is:

```
sbox0
perm0
xor_key0
sbox1
perm1
xor_key1
sbox2
perm2
xor_key2
```

It is clear from the data in `log.txt` that the cipher is weak - encrypting `0x02CC0F857` gives `0xE9BFB5DFD` as a result and encrypting the same plaintext with a few bits flipped, `0x02CC0F859` results in `0xE2BFF5DFD` which is the same ciphertext with some bits flipped.

The first idea was to check the S-box function for obvious weaknesses like linearity, where one or more of the output bits are only affected by a subset of the input bits. Sadly, a quick analysis via the provided `test_sbox.py` script showed it was nonlinear.

After that, we tried to find some pattern in weak plaintext and ciphertext pairs like the ones above, using our own keys, but even though there were tons of them, no clear relation emerged.

We then considered just directly brute forcing the keys, but even with a meet-in-the-middle attack, one of the sides would have a keyspace of 2*36 bits, which is too large for a head-on attack.

But then we realized that a full brute force wasn't needed - consider the ciphertext after the `sbox2` step - we have six distinct 6-bit segments. Going forward, we can see the bits being spread out in `perm2` and xor-ed with six bits of key material, which then results in six bits of ciphertext. If we pick one of the ciphertexts in the log and pick an arbitrary 6-bit key `key2a`, we are able to partially decrypt those bits, even back through `perm2` and `sbox2`:

```
         abcdef
sbox2   |______| |______| |______| |______| |______| |______|
         ghijkl   ......   ......   ......   ......   ......
perm2   |___________________________________________________|
         .h....   ......   ....l.   .j..i.   .k....   ..g...
xor2    |.v....___......___....z.___.y..w.___.x....___..u...|         key2a = uvwxyz
ct       011001   110100   111100   100010   100111   110111
```

```
abcdef = inv_sbox(ghijkl) = inv_sbox(0^u, 1^v, 1^w, 0^x, 0^y, 0^z)
```

We can then pick an arbitrary key `key1a` and that lets us decrypt back through `perm1`:

```
sbox1   |______| |______| |______| |______| |______| |______|
         .h....   ...l..   ..g...   ...i..   ..k...   j.....
perm1   |___________________________________________________|
         ghijkl
xor1    |mnopqr_____________________________________________|         key1a = mnopqr
         abcdef
```

Given a ciphertext and two 6-bit keys `key1a` and `key2a`, we can now deduce six bits of the result after the `sbox1` substitution. While it might not seem like much, this means that for each of the sboxes, we can eliminate half the inputs, leaving 32 out of 64 possible inputs for each box.

We then look at the plaintext corresponding to that ciphertext. We see that `sbox0` and `perm0` can easily be calculated without knowing any key material. Focusing a single sbox in `sbox1` - we now know the six bits of plaintext before the xor with a six-bit `key0a`, and also 32 possible inputs for that box. Xor-ing each of those 32 inputs with the plaintext bits result in 32 possible candidate keys for `key0a`. Then the trick - **when the keys are the same, but the plaintext and ciphertext differ, the same `key0a` must be present in all candidate keys for that sbox**. Basically, given `key1a` and `key2a`, the intersection of all possible inputs for a specific sbox in `sbox1` must be nonzero, and if the intersection has exactly one element, that is a strong candidate for `key0a`.

So, the final algorithm for finding the key is - Iterate over all `key1a` and `key2a`, using the 200 plaintext-ciphertext pairs to narrow down the possible `key0a` candidates. This is a 12-bit brute force, which is easily done. We then repeat the same process six times, each time focusing on different sboxes, which eventually gives us all bits of the keys.

The final key was [83193ca54 d90e5e945 b29645f8b] and the flag we got from that key was **SharifCTF{8aae67ec50080a359fc92beca22c52e3}**