Obfuscating IDs

Auto-incrementing IDs are great.

When you're picking a method for generating unique IDs in a system, auto-incrementing IDs (1, 2, 3, …) are often a good choice. These have a number of advantages over more complicated techniques:

- They are very fast to generate and require no random number generation or seeding.
- Their representation is short and simple compared to e.g. GUIDs.
- There is no need to test for collisions or uniqueness.
- They are natively generated by many systems; for instance, many relational databases support them out of the box.

Of course, there are many situations in which sequential IDs aren't appropriate, but this isn't a post about choosing your identifiers.

Until they aren't.

As nice as auto-incrementing IDs are, they do have (at least) one unpleasant downside: *they're often too transparent*.

For instance, let's say that you're on a website and you notice that your own user ID is 48203.

```
http://mysite.com/userprofile?userid=48203
```

Now, using only this information, you can infer a lot more:

- If you know the ID of any other user, you now also know whether they made their profile before you or after you. If their number is close to yours, you can even know about when their profile was created.
- You know that 48202 is probably a valid user, and so is 48204. You can try to see their profiles, too.
- If the service does not return an opaque error when an invalid ID is accessed, you can quickly and easily figure out how many users are in the system.
- If you can do the above, you can also predict the IDs that new users will be assigned, which may help you mount an attack.

The problem.

Let's state it clearly, then: given that your underlying storage uses a sequential ID, how do you make it opaque to the user, while minimizing the amount of complexity the obfuscation adds to your system?

A practical method for obfuscation.

I finally found a satisfactory solution to this problem here:

A Practical Use of Multiplicative Inverses (by Eric Lippert)

Note that when Eric says "multiplicative inverse" he's referring to the modular multiplicative inverse, which caused me some confusion when looking for resources.

Basically, the principle is as follows:

Choose the biggest number you want to generate ("M").

For our system, let's say we know we'll never have more than $M = 100{,}000$ users.

Pick large number ("N") coprime to, and smaller than, the first.

"Coprime" just means "has no factors in common with". You can pick this one more or less at random. How about $N = 48029$?

Calculate the modular multiplicative inverse of N.

This is a tremendous pain to do manually since the best known algorithm is iterative. Thankfully, unless your system needs to do this on the fly, you only need to do it once. There are a number of online calculators that will do this; here's one, and here's another.

The inverse in this case is 34069.

Use your new pair of numbers to obfuscate your IDs.

The formulas we need to transform IDs are simple:

```
masked_id = (48029 * plain_id) % 100000;
plain_id = (34069 * masked_id) % 100000;
```

Let's try a few examples. How about ID 101?

```
masked_id = (48029 * 101) % 100000 = 50929
plain_id = (34069 * 50929) % 100000 = 101
```

Nice! How does it look for users?

Plain ID	Masked ID
101	50929
102	98958
103	46987
104	95016
105	43045

We've transformed a transparent and predictable sequence into an opaque and unpredictable one–and made it trivial to transform it back.

Some caveats.

Hopefully it's obvious that this adds no *actual* security to a system; it's obscurity, at best. However, for discouraging casual tampering, and creating opaque tokens out of transparent ones, it's hard to do much better without considerably more complexity.