Generating Global IDs

Patrick Lambert

[http://dendory.net]

September 28, 2014

Abstract

This white paper describes the process of generating Global IDs for use in computer applications, including popular methods of doing so, pros and cons, along with practical coding examples.

1 Introduction

The creation of **Identification strings (IDs)** is a requirement that many programmers and software engineers face at some point. Whether it's to identify users, database records, collections of objects or sessions in a web app, IDs are used everywhere. As a result, there have been many different methods invented for creating them, some more effective than others.

This subject was deemed so important to the construct of the Internet that several methods were devised to make a **Universally Unique ID (UUID)** described in RFC 4122 [1]. This method will be explored, along with other popular methods, in order to come up with various ways of creating these IDs. No one way is best for every case. Instead, exploration of these various methods shows that they all have their strengths and weaknesses.

1.1 Audience

The goal of this document is to offer knowledge for anyone who may have a need to create IDs. This may include junior programmers who are developing software applications, web developers making web apps having to track users, or even system administrators scripting against databases and managed user groups. It also attempts to explain some of the design decisions behind each method of creation, giving insights into why one way is best for a particular purpose.

For the purpose of this document, code samples are written in PHP. However, most if not all methods described here can be implemented in any language.

1.2 Expectations

The requirement of the methods explored here is to produce **Global IDs (GIDs)**, sometimes called **Globally Unique IDs (GUIDs)**. This means each individual ID string should be unique within a particular global scope. The scope can be an application itself, a computer system, a network or an organization. Its definition is up to the implementer and can vary based on the requirements of the application.

It however does not assume that the GIDs will be unique universally. This means that two instances of the global scope may produce IDs that collide. For example, if a web app is produced which creates GIDs for a list of users, these ID strings must be unique within the scope of the app and the list of users. However, a different app producing IDs for users in another organization unconnected to the scope of the first project, may also produce IDs which could end up being the same. This is acceptable as the two scopes are separate.

1.3 Disclaimer

As a technologist, I have long years of experience dealing with the topics described here. However, this document is provided for educational use only. I am no expert on cryptography or number theory. Everything written here is provided **as is** and with no warranty expressed or implied. Use at your own risks, and do your own research before implementing the code discussed here in a critical environment.

2 Universally Unique Identifiers (UUIDs)

The first method we will touch is the UUID function described in the Introduction. This has become a popular method for creating IDs because it offers various benefits:

- ID strings created by a UUID function are assumed unique universally, meaning that no ID will collide with any other ID created by the same function, regardless of time, scope, application or anything else.
- This in turn allows them to be not only unique but persistent, being useful for any type of application, even long term archival, as no collision is expected at any point in the future.
- They are a fixed 128 bits size with a predictable format.
- No central authority is used.

The last point is a very important benefit. For example, domain names on the Internet can be considered as a GID. They are unique among the scope of Internet domain names. However, they require a central authority, in this case ICANN [2], to make sure no two persons decide to use the same domain name. In the case of UUIDs as defined by RFC 4122, they are assumed unique thanks to programming.

A proper UUID looks like this:

```
13152fae-d25a-4d78-b318-74397eb08184
```

The RFC describes not one but five different ways to create a UUID. Each is based on the same basic principles, but vary slightly. A UUID is made up of 16 octets totaling 128 bits. The basic idea is that it includes information that is assumed to be unique universally, meaning that no other system will create the same UUID anywhere in the world, regardless of input parameters. This may include the event happening at the exact same time, executed for people with the same name, or any other number of variables coinciding to be similar.

These are the versions of the UUID specification:

1. The first version uses the current time, down to the nanosecond, concatenated with the machine's MAC address. This takes advantage of the fact that MAC addresses are supposed to be unique, and that the computer's processor is unlikely to run the event twice in the same nanosecond. However, in practice a MAC address can easily be changed, and there is no way to know if a future computer will be able to run this event twice in a short enough time to make the ID non-unique.
2. Version 2 was implemented on Unix systems and replaces the time stamp with the POSIX UID, which is the user ID on a local Unix system. This is good when producing IDs for individual users on a POSIX compliant system.
3. This version uses a namespace, along with a URL, domain name, object identifier or other text string to uniquely identify an event, then uses the MD5 hash of it. This is a good choice when creating IDs for items, perhaps a list of web sites stored in a database. By using the hash of the actual site, each ID is guaranteed to be unique. However, if two distinct systems choose the same namespace, then these could collide.
4. Version 4 relies solely on unique numbers. Past the 4 bits reserved for version, 122 bits are generated by a pseudo-random function. This is perhaps the easiest and thus most popular method of generating UUIDs out there.
5. Finally, this version is the same as Version 3, however it uses SHA-1 hashes instead of MD5.

Implementation of UUIDs may be the strongest strength of the specification, since there are tools and libraries that can produce IDs for you already available in most operating systems and programming languages. Linux systems by default ship with the *uuidgen* utility that can produce random or time based IDs.

PHP does not include an existing function to create a UUID following the specification, but it is fairly simple to create. The following will generate a valid Version 4 UUID:

```
function make_uuid()
{
    $a = openssl_random_pseudo_bytes(16);
    $a[6] = chr(ord($a[6]) & 0x0f | 0x40);
    $a[8] = chr(ord($a[8]) & 0x3f | 0x80);
    return vsprintf('%s%s-%s-%s-%s-%s%s%s', str_split(bin2hex($a), 4));
}

$id = make_uuid();
```

It's important to note that when creating your own ID generating functions, it's crucial to understand which functions to use. For example, the *mt_rand()* function in PHP creates random values that could be used here, however when digging deeper, one would find that *mt_rand()* generates these values solely based on the system time, which means they could be predictable and they could be non-unique. This is why entropy is needed, provided here by *openssl_random_psudo_bytes()*. A discussion of cryptography is outside the scope of this document, but RFC 4086 [3] provides a good background.

As you can see, UUID is a strong and very useful specification, and already in used by many. However, there are a couple of pitfalls. First, even random numbers could theoretically collide. Second, if the ID is generated based on the MAC address of the system, or a specific text string like a URL, those may not be unique. Even if assumed to be unique when coming out of the factory, MAC addresses can easily be changed by a user, for example. Finally, UUID strings are fairly long. This can be fine when storing those IDs in a database, but not that useful if they have to be showed as text to users, or remembered as part of URLs.

3 Session IDs

Session IDs are unique identifiers used to identify a session, typically a unique dialog between a web application and a user, in order to provide context. This allows the user to leave a web page and come back, or navigate throughout various pages, and maintain their logged in status. Session IDs are very popular, and most programming languages provide automatic ways to generate them.

In PHP, you can generate a session ID in the following way:

```
$id = session_id();
```

The benefit of this approach is that session information is kept by the programming interface itself. By using functions like *session_start()* and the *$_SESSION* variable, you can keep keys and values unique to a user. Many web apps use this method in order to generate an ID for each session and then track users as they interact with the server, until they finally log out. This ensure that connections can always be identified correctly, regardless of the amount of users making requests concurrently.

Session IDs however have some downsides. First, while the chance of collision is low, these IDs should not be considered unique. By default, PHP takes the following items to construct a session ID:

- The IP address of the connection, which ensures that every IP is guaranteed unique.
- The current timestamp, meaning that every ID produced at different times will be unique.
- Two sources of randomness, typically cryptographically secure ones such as */dev/urandom* on Unix.

These items are then hashed together (using MD5 by default) and that makes the session ID. From this knowledge, you can easily see when and where a collision may happen. It would take two users from the same IP address, creating the ID at the same exact time, with the random number generator providing the same result for both IDs. Depending on the scope of your project, this may well be satisfactory. Another possible downside is the length of the ID. An MD5 hash is lower than a UUID, at 16 bytes, but may still be too long if you wish to have users see it or try to remember it as part of a URL, for example.

4 Custom ID creation

Many more methods of generating Global IDs exist out there, and you may be tempted to create your own. There are good reasons to do this, but also some pitfalls that need to be avoided. Certainly, the simplest way to create a unique ID would be the use of a random number. This can be trickier than at first glance, however, as no computer can generate truly random numbers. They all rely on pseudo-random generators, which need to have enough entropy. Plus, random numbers can collide. The chance of that happening depends on how many bits the number has, meaning that for a good implementation, you may require quite a bit of length.

Another popular way to ensure uniqueness is to concatenate various items, which is used by the previous two methods. This may be your best bet since you can then select which items to pick based on your requirements. For example, if you create a web app, you may want to take items that are unique to network connections such as an IP address. If you generate database entries for real world products, then perhaps these items have QR codes that can be used.

Also, you may wish to add other requirements that impact uniqueness. Perhaps you are making an online form and you actually don't want the same user to be able to generate a new ID every nanosecond. Here, creating a Global ID that takes its uniqueness only based on a time stamp down to the second will ensure that another ID can't be created within this 1-second time frame, thus rejecting potential spam or denial of service attacks. Security on the Internet is a complex topic, and you should read RFC 3631 [4] for a good overview of other mitigation methods.

In our case study, we will implement a function that creates a GID based on the time and the user IP, and must be small enough in length to display in URLs and for users to optionally remember them. This will create enough uniqueness for our scope, which is allowing any one IP to create an entry in our web app at the most once per second. Note that this scope is specific to this one example. You may wish to change the IP address to existing user IDs, or to a combination of items such as IP, browser agent, and a random source. The more items you add, the longer the length of the resulting ID.

Here is the code in PHP:

```php
function make_id()
{
   $a = dechex(str_replace(".", "", $_SERVER['REMOTE_ADDR']));
   $b = dechex(time());
   return base_convert($a . $b, 16, 36);
}

$id = make_id();
```

This function generates an hexadecimal value for the IP address, along with the current timestamp, then converts from base 16 to base 36. One nice thing about base conversion is that no information is lost, and the full breath of the ASCII alphanumerical values can be used to reduce the length of the ID. Here are some example values created:

```
1sr2jco9laiu
1vnfi6ffb
6fd9rl9ahr8k4g
1qpfx5r29m7qm
```

As you can see, the values produced are between 9 and 14 bytes in length. This is far shorter than the 20 bytes of SHA-1 and even 16 bytes of MD5. It is of course possible to produce smaller IDs, for example simply using the time, but as a result you would lose uniqueness to a point that may no longer be acceptable for your particular scope.

5 Additional constraints

So far we've presented the basics of generating GIDs, but depending on the scope of your work, you may have many more constraints to take into account. For example, most of the built-in functions generate alphanumerical strings, but perhaps you require numerical IDs. For such a situation, you can still use the functions we discussed, and then simply convert them to base 10 using code like this:

```
$id = base_convert(make_id(), 36, 10);
```

Another constraint is where those GIDs will be stored. In most cases, they are kept in memory, such as a session ID for a web app, or stored in a database, such as user IDs that have access to specific resources on a system. But sometimes they need to be sent over shared networks. In these cases, IDs may need to not only be unique, but also hashed or encrypted. A hash is a one-way conversion from clear text to fixed-sized. Note that not all hash functions are the same. MD5 for example is known to have collision vulnerabilities, while SHA-1 has known attacks. As such, while they are fine to generate IDs, they should not be relied upon for secure communications. Instead, SHA-256 should be used. Wikipedia has a good article [5] on the subject of hashes.

IDs stored in a database can often be gibberish, but when it's something that has to be remembered by a user, your requirements may not only be about length, but also the actual text used. This is why most user IDs are taken from another source, such as part of the name of a person or an email address. Back in the early days of the Internet, user IDs were typically numbers. People who used the instant messaging system called ICQ had to memorize their numerical IDs, while early mail systems used gibberish as well. Generating unique GIDs that can be easily remembered may bring an additional level of difficulty. For example, a name is typically a poor ID, because many people share the same name. An email address is far better, since thanks to the uniqueness of domain names, each email address is unique.

But what if you are trying to create user IDs that will be used to generate email addresses, meaning that you don't already have an email address you can use? A typical way to do this is by using a person's last name, the first letter of their first name, then adding a single digit at the end should the ID collide. This ensures uniqueness and presents an ID that is still human readable. While no method will be perfect, often requirements can be attained with creative thinking.

Here is a code sample that may accomplish this. It assumes an array *$userids* containing a list of existing users, but can be modified to check against a database:

```
function make_id($first_name, $last_name)
{
    $k = 0;
    $b = $a = $last_name . $first_name[0];
    for ($i = 0; $i < count($userids); ++$i)
    {
        if($a == $userids[$i])
        {
            $a = $b . strval($k);
            $i = 0;
            $k++;
        }
    }
    return $a;
}

$id = make_id("John", "Doe");
```

Finally, GIDs may need to be created in a distributed manner. What if you scale your application to several instances, each accepting connections from users who may need to create IDs in a central database and thus need to be unique?

Typical GIDs should be made using non-deterministic algorithms, meaning that someone cannot guess what their ID will be based on the input values. This is a basic security requirement, and also helps with uniqueness. For example, if your ID generating function only takes known values such as the time stamp and IP address, it will create a predictable ID. This means it should never be used for sensitive applications, like as a password. However, this determinism can be useful in the case of distributed systems, where you need to be able to predict IDs made by other instances. Wikipedia has an article [6] on the topic.

6 Conclusion

We've seen various different methods for creating Global IDs. Each method has its strengths and pitfalls, and as a result no one method is best. What you use depends on your requirements and the scope you attempt to address. The more bits needed for uniqueness, the longer your ID will be. We've seen the difference between using pseudo-numbers and concatenating various items, along with potential pitfalls of both methods. We've also seen some practical examples of creating IDs, and built-in methods that can be leveraged.

One source of uniqueness that is used in almost every method is time, because it's a great way to ensure persistence. Another source is a pseudo-random generator. The generation of *true* randomness is a difficult topic, and Wikipedia has a good article [7] about the subject. In many cases, just these two sources may be enough, but others can be used based on requirements, such as the MAC address, IP address, or some other value.

Global ID generation is a topic that touches both software development as well as cryptography and maths. As such, it's important to be careful when reinventing the wheel. If security is a key requirement, proper care should be taken when determining whether your numbers are unique enough or not. It's possible to have truly unique IDs for a specific scope, but not for all scopes. If all inputs can be modified, then there is a chance of collision, as slim as it is. Your job is often determining how slim the chance is, and whether that is acceptable.

Often, the creators of large and complex applications have no idea that their products would grow to the sizes that they did when they first started. Even if you think your project is small, such important tasks as creating unique IDs should be done correctly, since it's much harder to fix down the road.

7 References

References

[1] RFC 4122: *A Universally Unique IDentifier (UUID) URN Namespace*
http://www.rfc-editor.org/rfc/rfc4122.txt

[2] ICANN: *Internet Corporation for Assigned Names and Numbers*
https://www.icann.org/

[3] RFC 4086: *Randomness Requirements for Security*
http://www.rfc-editor.org/rfc/rfc4086.txt

[4] RFC 3631: *Security Mechanisms for the Internet*
http://www.rfc-editor.org/rfc/rfc3631.txt

[5] Wikipedia: *Hash function*
http://en.wikipedia.org/wiki/Hash_function

[6] Wikipedia: *Deterministic algorithm*
http://en.wikipedia.org/wiki/Deterministic_algorithm

[7] Wikipedia: *Pseudorandom Number Generator*
http://en.wikipedia.org/wiki/Pseudorandom_number_generator