

On Randomness
Version 0.1

Ben Laurie
ben@algroup.co.uk
<http://www.apache-ssl.org/ben.html>

December 29, 2004

1 Introduction

Once upon a time, the Apache Software Foundation (ASF) wrote a portable runtime library called, imaginatively, the Apache Portable Runtime (APR). APR is used by two notable projects, the Apache HTTP Server, and Subversion, a version control system.

As APR was extended to fill the needs of the HTTP server, the subject of randomness came up. There was a big discussion about what kind of randomness should be used by default, and I, in my capacity as paranoid security guru for the ASF, argued strongly that the default should be the strongest available source of randomness.

Of course, as is well known amongst paranoid security gurus, this can be problematic. The strongest source of randomness might not be able to provide randomness quickly enough, and so things might be delayed while they wait for more randomness. This is fine, we reasoned – the casual user will have been kept safe from his own ignorance, and won't get hacked to shreds as a result of sacrificing security for convenience. The knowledgeable user can make their own decision, change the source of randomness, find more entropy, or whatever they think best. So, APR did as was thought best, and used strong randomness.

Fast forward a year or three, and a strange request was heard. The Subversion guys wanted us to use *weak* randomness. The randomness APR was providing was causing them problems by making some functions take forever. Why on Earth would you want that, I asked? If you don't care about strength, why not just use `rand()`? Or even start at one and work upwards? The answer surprised me: they wanted to be able to generate universally unique IDs (UUIDs). Now, an interesting thing about UUIDs is that you don't care about *unpredictability*, but you do care about *collisions*. Why? Because UUIDs are not secret, so unpredictability doesn't matter. But it is very bad if they collide, very bad indeed.

And that got me thinking, and talking to other paranoid security gurus. And as a result of that thinking and talking, I realised we'd got it all wrong¹.

2 Entropy

The Handbook of Applied Cryptography[1] defines entropy as follows:

The entropy of X is ... the uncertainty about the outcome before an observation of X.

In other words, the entropy is a measure of the amount of unpredictable information there is in a data source. The name of the game in producing useful randomness is to have sufficient entropy that the randomness really is random.

Once you have enough entropy, though, you can use it as a source for all your future random needs by generating pseudo-random numbers from a cryptographic pseudo-random number generator (PRNG). A cryptographic PRNG is one that

¹OK, OK, not *all* wrong, but it doesn't sound so good to say "we got it a little wrong".

cannot have its internal state recovered from its output by any method more efficient than brute force.

How much is enough? It's enough that an attacker cannot, by brute force, deduce the original entropy², by comparing output from the PRNG with candidate values of the entropy with the output as observed from the actual PRNG. I will call this number E for the rest of this paper. E is widely agreed to be 128 bits, but I use a placeholder because it may be that future advances in computation will require an increase in E .

3 Unpredictability and Collision Resistance

Why is unpredictability different from collision resistance? It's all to do with knowledge. I can have a great source of entropy, but if an attacker knows all about that source, then it gives me no unpredictability at all. However, knowledge doesn't damage collision resistance in the slightest.

That is, there is a difference between the inherent unpredictability of a source of randomness and its practical unpredictability, given an adversary's access to the source. Cryptographers tend to assume that an adversary can divine information about some sources of randomness, and hence either reduce their opinion of the value of those sources or avoid using them entirely.

Luckily, there is a formal definition of the difference between these two types of entropy. The entropy a data source has given any knowledge the adversary has (or might have) is known as conditional entropy, and the entropy ignoring any such knowledge I will call, to avoid confusion, unconditional entropy. To facilitate discussion, let's introduce a little notation. The entropy of a data source X is written $H(X)$. The conditional entropy of X given some knowledge Y is written $H(X|Y)$. It is known that $H(X|Y) \leq H(X)$, with equality occurring if and only if Y is independent of X . This ties in well with intuition: if we know something about X , then it has less entropy. If what we know has nothing to do with X , then the entropy is unchanged.

So, I will use the term "unconditional entropy" to mean the amount of entropy a data source has ignoring any access an attacker may have to it, and "conditional entropy" to mean the amount it has given the attacker's potential knowledge. Note that you have to be a little careful with these terms, because they work in the opposite sense to the related concepts of conditional and unconditional security. However, since they all have widely accepted formal definitions, there is little sense in trying to reconcile them.

Cryptographers want conditional entropy, but for UUIDs, and other applications, what is needed is unconditional entropy.

What examples are there of things with plenty of unconditional entropy but little conditional entropy? Well, for many purposes, the system time has plenty of unconditional entropy, because it is usually different when used to generate different UUIDs. However, system time tends to have very little conditional

²Cryptographers tend to use the term "entropy" to mean the actual random data, as well as a measure of its randomness

entropy, because an attacker can often guess with a good deal of accuracy when the time was sampled by his victim.

This was used to great effect in an attack[2] on Netscape's PRNG, for example, where reliance on time as an entropy source caused a complete breach of security in the SSL implementation.

4 Forward Security

Paranoid security gurus sometimes worry about forward security. What they mean by this is that should your state be compromised at some point, it may be desirable that the state does not stay compromised forever. That is, if an attacker somehow manages to determine the current PRNG seed data at some point in time, you would probably like the attacker not to be able to reconstruct all past and future randomness, since that would likely compromise all past and future security. Preventing this type of attack is known as forward security.

It is often argued that there is little point in defending against this because if an attacker has access to your PRNG's internal state, then they clearly have seriously compromised your system, but this is not the only way state can be leaked. In particular an attacker might find a way to get access to the sources of entropy used to seed the PRNG – for example, if network traffic is used, an attacker might be able to observe the traffic (or at least most of it) from outside the network under some circumstances (for example, during a known outage of other machines on the internal network. Obviously if an attacker can *always* monitor network traffic, then it is a bad source of conditional entropy (but still a perfectly good source of unconditional entropy). Because you can't know when or what special circumstances might lead to this kind of state compromise, routinely defending against it is wise.

Forward security is easily achieved simply by reseeding the PRNG periodically, or, better yet, mixing fresh entropy into the internal state. Mixing is better because it increases the total amount of entropy, so long as the mix is done well, whereas reseeding throws away the entropy from the previous seed.

5 PRNG Seeding and Mixing

It is a really bad idea to either seed or mix less than E into your PRNG. In the case of seeding, it is simply because this exposes your PRNG to a brute force attack. For mixing, it is because if there has been a total or partial state compromise, then a brute force attack starting from the compromised state is possible, meaning the new state is then also compromised.

6 PRNG API

Discussing these conflicting requirements[3] made me realise that there isn't currently a PRNG API that provides what everyone needs. So, here's what we

came up with. Firstly, for a standard PRNG, you need E bits of conditional entropy for the seed, and from then on you're OK. The API

```
void manualforwardsecureprng(void *out,int nbytes)
```

provides `nbytes` of output from such a PRNG. Secondly, you may at various points wish to have forward security of your PRNG. The API

```
void prngbarrier(void)
```

ensures that E bits of conditional entropy are mixed into the PRNG before any further randomness is extracted. You may be too lazy to figure out when to call `prngbarrier()`, so

```
void forwardsecureprng(void *out,int nbytes)3
```

will do it periodically for you. You might also want forward security, but only when sufficient entropy happens to be available to permit it. The API

```
void lazyforwardsecureprng(void *out,int nbytes)
```

will do that. In general, this is the call that a programmer who doesn't want to think too hard about what he's doing should use.

For very high value randomness (e.g. the key the Bank of England is going to use to sign electronic pound notes), you might want to use as many bits of conditional entropy as you are expecting for output, or at least some proportion of the number of bits in the output, rather than merely E . This is provided by

```
void trueprng(void *out,int nbytes)
```

Finally, (yes, this is the one Subversion wants), you may want to use E bits of unconditional entropy:

```
void insecureprng(void *out,int nbytes)
```

Although this doesn't need to use a cryptographic PRNG, there's really no reason not to, the only problem with the traditional approach is the seeding, not the PRNG. Also, as we will see later, it has the advantage of permitting state sharing. This function would use entropy gathered differently from the others, since it can use sources of entropy normally considered too dangerous for cryptographic use, or can allow higher estimates of the total entropy from each source.

7 Error Handling

The API above does nothing about errors. A real API should probably allow an application, at least optionally, to handle errors, though it is hard to see what can be done of value when randomness fails, other than produce better diagnostics, or apologise to the user.

There may also be an argument for producing versions of the APIs that will allow timeouts, so that programmes with user interfaces can inform users appropriately.

³There should be a version of this call with a timeout, suggested by Mark Murray - the scenario would be, for example, fresh install of FreeBSD needs to generate `sshd` key, but can't coz of no entropy - let it go anyway after a timeout, mark the key as needing replacing, but at least `sysadmin` can log in and fix the problem.

One thing that is widely provided but seems ill-advised is a facility to return less randomness than was asked for if sufficient is not available. Clearly you should ask for what you need, and getting less must surely be a security or reliability risk. Of course, it is a general feature of this API that if it can return any randomness at all, it can return a large amount of it.

8 Practical Considerations

One of the problems with PRNGs in practice is that they tend to be implemented in libraries or applications, and hence each application instance consumes at least E bits of entropy. This is bad, since many systems are chronically short of entropy. It is much better if the PRNGs are shared across all applications, which either means building them into the operating system or having them in daemons, the latter giving its own interesting security problems, of course.

Another question is how much state should be shared between the various different APIs. If one assumes the PRNG is secure, then this seems to be easily resolved: they can all share all the state, except `insecureprng()`, which requires less conditional entropy. Once there is sufficient entropy for the other APIs to start working, then even `insecureprng()` can share their state.

One thing to note when sharing state is that there is generally considered to be a “safe” amount of randomness that can be extracted from a PRNG before its state can be divined from the output, or at least some useful attack mounted on it. This amount is typically of the order of $2^{E/2}$ bits – that is, a very large number. Nevertheless, implementations should ensure that state is not shared between secure and insecure PRNGs once this limit has been exceeded, until new entropy has been mixed in. What this would mean in practice is that a point could (at least in theory) be reached where `insecureprng()` should split its state from the other PRNGs, and continue to produce output, while they block, waiting for more entropy.

Another pitfall that can occur if the PRNGs are not implemented in a daemon or device driver is that if a process forks, then the new process will produce exactly the same output as the old one. One simple answer to this is to have the parent discard a block of output from the PRNG, whilst the child takes the block and mixes it into the PRNG’s seed[4].

Finally, the data used to seed PRNGs should not include any data that should be kept private. This is because a dictionary attack on the private data coupled with the (now insufficient) other seed data could reveal the private data. It is worth noting that serial numbers or MAC addresses are often considered private because they can be used to correlate otherwise unrelated data⁴. This point particularly applies to the insecure PRNG, because other data used for entropy may be easily guessable, making the dictionary attack much easier. It is because of this privacy issue that the `insecureprng()` can’t simply be seeded with data that is guaranteed to be unique (such as the MAC address of an Ethernet card, for example).

⁴This is why Microsoft stopped using them in their UUIDs.

9 Acknowledgements

My thanks to David Wagner for pointing out major flaws in my original ideas on this subject, and providing the basis of a substantially simplified API. Robert T. Johnson and Tina Bird also provided useful comments on early drafts of this paper.

References

- [1] Menezes et al., “Handbook of Applied Cryptography”, CRC Press, 1996.
- [2] Ian Goldberg and David Wagner, “Randomness and the Netscape Browser”, Dr. Dobb’s Journal, January 1996, pp. 66–70.
<http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html>.
- [3] David Wagner, personal communication.
- [4] John Viega, personal communication.