The Debian OpenSSL disaster

CRASH INVESTIGATION

Find out what we can learn from the Debian OpenSSL disaster. **BY KURT SEIFRIED**

After a plane crash, a crash investigation begins. Investigations reveal that most airplane crashes are due either to human error or to some new confluence of circumstances that was never anticipated. Consequently, airline travel is one of the safest forms of travel per passenger mile.

Software is another matter. Unlike a plane crash, when a software crash occurs, a typical response is simply to address the immediate problem with a source code patch, which does nothing to address the underlying problems. Thus, we are in a constant state of only treating the symptoms but never the underlying problems.

Because these underlying problems are never corrected, we keep seeing the same software flaws over and over (temporary file creation, buffer overflows, stack overflows, etc.).

This month, I'll investigate the Debian OpenSSL disaster in an effort to find any root problems.

In a nutshell, the problem occurred when a Debian package maintainer for OpenSSL ran Valgrind, a code analysis tool, against OpenSSL and found several uses of uninitialized memory. The use of uninitialized memory is potentially dangerous because you have no idea what could be in it – all 0's, all 1's, or an attacker's injected code, just to name a few possibilities.

The maintainer then went online to the openssl-dev mailing list and asked about the following code:

```
MD_Update(&m,buf,j);
```

Several replies later, the developer decided it was safe to remove the offending code from Debian's OpenSSL package. These changes were committed to Debian, and life went on as usual.

The code in question occurs twice: once in *ssleay_rand_bytes()* and once in *ssleay_rand_add()*. Although the code is identical, it serves two very different functions. In *ssleay_rand_bytes()*, the code simply returns random data from memory into a buffer. However, in the function *ssleay_rand_add()*, it tries to be clever by adding some uninitialized memory to the entropy pool. In the best case, it adds to entropy, and in the worst case, it doesn't hurt anything.

This buffer is used as the primary source of entropy for any applications using OpenSSL (unless they use a custom PRNG source). By commenting it out entirely, the developer removed virtually all the randomness used during key creation by most applications. The only remaining "random" data used during key creation was the process ID, reducing the key space from 2^(large number, such as 128 or 1024) to 2^16 (or less, in some cases). Oops.

What Went Wrong?

Several things went wrong, and like most disasters, everything would have been fine if the chain of events had been broken at any point. Instead, every Debian administrator had to patch every

single box and re-generate every encryption key that was created in the past two years.

One of the immediate issues was code that did something unnecessary in a potentially dangerous manner: Adding uninitialized memory to true randomness doesn't gain much and makes the code look like it might be doing something else entirely.

Nor was the code well documented. For example, the following comment might be appropriate:

```
 /* This is critical code
* that reads from a random
* pool of data,
* pretty much all the
* critical randomness used
* in OpenSSL
* comes from here, if you
* monkey with it you may
* break OpenSSL
* and any applications that
* rely upon it entirely
*/
```

This is why military equipment has nice big warnings like "this side to enemy."

Chances are, if this code were easier to understand, the developer wouldn't have needed to go to the openssl-dev mailing list for help. This leads nicely into the second issue: unclear communications.

When dealing with security-related code or security-related issues, it is critical to communicate with the right people or forum. Otherwise, you might receive what looks like an authoritative answer but is in fact incorrect – perhaps because the question was misunderstood or because the answer is simply wrong. In this case, it looks like pretty much everything that could go wrong did go wrong.

The OpenSSL team claims that the question was asked on the wrong mailing list, whereas other people claim that the supposedly correct mailing list isn't advertised.

Additionally, the question was somewhat unclear: The example code snippet doesn't give full context, and because of the nature of the code, this might have led to a misunderstanding of exactly what was going on.

Prefacing your email with something like, "If you know a better person or forum to ask, please let me know," is probably more effective than, "Is this the right place to ask," and certainly better then blindly firing off an email and hoping for an authoritative answer.

Additionally, a reporter can check the CVS (or subversion, or git) check-in messages to find out who checks in lots of changes in the affected code to learn who is probably responsible for the code in question.

As you can see, tracking down the right person can be quite a chore, so clearly documenting how to contact – and who to contact – for various issues will go a long way toward preventing problems with your software.

Code Changes

Because Debian packagers maintain their own code repositories, the source code change wasn't noticed by anyone outside of the Debian project, but it still got widespread distribution. If the source code patch had officially been sent upstream to OpenSSL by the Debian Project, it would have probably raised red flags and been removed. This leads to a situation in which even if the upstream project continues to update and maintain software, a simple patch maintained by Debian could introduce a subtle – or in this case, a significant – flaw. The solution to this problem, sending all patches upstream for inclusion, is not simple. Another possibility is officially to run all patches upstream for review.

Lack of Testing

Finally, I will address the most difficult issue. In the airplane industry, materials, designs, and often entire components (such as wing assemblies) are tested to the point of destruction to see just how much abuse they can take before failing. To my knowledge, a testing framework for OpenSSL that generates a statistically significant number of keys – for example, several hundred thousand or million – and then analyzes them to check for randomness doesn't officially exist.

Additionally, even if such a framework existed, it would need to be applied regularly to new versions – and not just the official upstream version, but to any versions that have modifications applied to them by vendors.

Of course, this type of testing framework should be applied to all products, for example, a firewall-testing protocol that applies a variety of rulesets – ranging from simplistic to complex – and then sends traffic to it that tries to evade the rulesets.

Until such frameworks exist, it is almost certain that serious flaws will continue to crop up.

Conclusion

Unfortunately, it is much cheaper in the short term simply to treat the most damaging symptoms of bad software engineering than it is to address the underlying problems and causes. However, in the long run, this leads to huge amounts of time spent by end users applying patches and updates and developers needing to address the same problems repeatedly.

The good news is that many of the solutions to these problems are not that expensive, and most require little if any technology to implement.

Simply commenting code, documenting communications channels, and asking questions clearly – with as much context as possible – will go a long way. Also, it's important to remember that open source isn't just about access to source code, but access to the very culture that writes the source code, which means everyone has the chance to help make it that much better. ■

INFO

DSA-1571-1 openssl: *http://www.debian.org/security/2008/dsa-1571*

Key rollover: *http://www.debian.org/security/key-rollover/*

SSLkeys: *http://wiki.debian.org/SSLkeys*

OpenSSL bug report: *http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=363516*

THE AUTHOR

Kurt Seifried is an Information Security Consultant specializing in Linux and networks since 1996. He is married and has four cats but no fish (because the cats are more hungry than afraid of water). He often wonders how it is that technology works on a large scale but often fails on a small scale.