Kernel based random number generation in HP-UX 11.00

Jos Visser 19-09-2003

Table of Contents

Use the following link table to jump directly to the information you're looking for.

Introduction

Some time ago I received a call from a customer complaining that OpenSSH was terribly slow in connecting from one HP-UX node in their cluster to another. Given the fact that I had convinced them to start using OpenSSH in the first place I was somewhat emotionally involved in tracking down and solving the problem. At first it seemed sort of strange to me that this was the case since I have used OpenSSH on HP-UX before without any speed problems whatsoever. More specifically, the OpenSSH installations I had done myself on the customer's development systems were not slow at all...

Investigations showed that the slowness was caused by the process of generating random numbers within the OpenSSH client. Since HP-UX 11.00 does not have a "/dev/random" device driver, entropy has to be gathered in userspace. And since the customer did not want to run the user space Entropy Gathering Daemon (EGD) on their systems the OpenSSH binary had to gather entropy itself on demand, which is terribly slow. But, maybe I am jumping ahead of myself a bit....

Oh, and before you continue, why not get them download circuits fired up and download the HP-UX 11.00 dynamically loadable kernel module (DLKM) that this article is about from http://www.josvisser.nl/hpux11-

random" (binary and source distribution; GPL applies).

On random numbers

Lots of security protocols and algorithms depend on coughing up a random key to be used for things like encrypting network traffic (OpenSSH) or proving one's identity (Kerberos). Since the security of the protocols depend on the randomness (i.e. hard-to-guessness) of the key, the ability to generate an unpredictable key is a core feature of many security packages. Unfortunately, for a device as predictable as a computer, generating a good random number is a remarkably difficult task. The timings of instructions and the outcome of calculations are so predictable that a set of random numbers calculated by a program based on pure mathematical calculations is almost never satisfactorily random.

Speaking of which: What exactly **is** a random number?

I have always found the term "random number" somewhat strange. Numbers are not random at all. There is for instance nothing more random to the number "12" then there is to the number "7629". Probably the only non-random number in the known universe is "42" (the proof of this is left to the reader as an exercise :-). It is better to talk about a "random number generator", where the bindings should be read like "random [number generator]", and not like "[random number] generator". The generator does not generate "random numbers", it generates randomly distributed numbers (or, said differently, it generates numbers, randomly). However, the term "random number" has become so common that I will use it here as well to mean a number that has been generated by a "random [number generator]"...

Like power generators, random number generators can be "switched on" after which they will generate an endless stream of numbers (a random number sequence) that should exhibit the following characteristics:

1. The sequence should "look" random
   This means that the sequence should pass all statistical tests of randomness. People with an appetite for mathematics should now consult chapter 3 "Random Numbers" of the second volume of the timeless computer science classic "The Art of Computer Programming" by Donald Knuth. One simple test of randomness is that the mean of the random sequence is the mean of the underlying number domain. For instance in a set of random bytes, the mean of the set should be 127.5 ((0+1+2+..+255)/256=32640/256). More advanced tests are the chi-square test and the spectral test.
2. The sequence must be unpredictable
   Even when knowing the exact algorithm, the starting values and the first n numbers in the set, the $n+1$'th number should be impossible to predict with any accuracy.

Pseudo random numbers

Although it looks simple, the second characteristic is actually very difficult to ensure. From "Applied Cryptography" (by Bruce Schneier) second edition page 44: "Computers are deterministic beasts. Stuff goes in on one end, completely predictable operations occur inside, and different stuff comes out at the other end. Put the same stuff in on two separate occassions and the same stuff comes out both times. Put the same stuff into two identical computers, and the same stuff comes out of both of them..." Or, as John van Neuman aptly stated: "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin".

Lots of research has been going into finding mathematical formulae that generate number sequences that are more or less random. The most popular functions are variants of the "linear congruential" function:

```
X(n+1) = (a.X(n) + c) mod m
```

The magic integers X(0), a, c and m are the parameters of this function. Choosing good values (especially for a, c and m) is imperative if one wants to generate good random-like sequences (see section 3.2. and further of "The Art of Computing, volume 2").

The mathematical methods of generating random numbers produce sequences of numbers that pass the statistical tests for randomness, but wich are utterly predictive once you know the algorithm and the starting values of the algorithm (the seed X(0)). These sequences are usually called "pseudo random" because of their predictability.

A good example of an implementation of pseudo random number generation is the "rand" functions in your system's C library and/or Perl interpreter. The "srand" function sets the starting point of the sequence and the "rand" function then delivers random numbers (usually in the range 0..1). When run twice in a row with the same starting seed, the same sequence is generated. For instance type in the following Perl program and run it a couple of times:

```
#!/usr/bin/perl

srand(42);
```

```
for ($i=0; $i<10; $i++) {
        $n=rand;
        print "$n\n";
}
```

If you run this program repeatedly you will see that it generates the same sequence of numbers every time round! On my system the generated list is:

```
0.744525000061007
0.342701478718908
0.111085282444161
0.422338957988309
0.0811111711783106
0.856440708026625
0.498799422194079
0.478814290644628
0.690812444305639
0.834593765962154
```

The mathematical average of this list is 0.506122252152382, so it at first sight it probably satisfies one property of randomness. But then again, because the same list appears every time, the sequence can not be truly random. In order to alleviate this problem, we should seed the random generator with a random number... Uhhh... wait a minute, we've just bit ourselves in the tail here....

To make the sequence slightly less predictable most programs use something like the current time in seconds since 1970 as the seed for the random generator. This ensures that a different sequence pops up if we run the program twice at different times. However, if we run it twice in a second, the same sequence appears again. The Perl Cookbook therefore advices to throw in the current process id as well, seeding the pseudo random generator with the current time exclusively or'ed with the process id of the current process:

```
srand(time^$$);
```

Although this does indeed "guarantee" a different pseudo random sequence every time the program starts, the sequence is still way too predictable for applications that need a high level of randomness (like creating session keys for protecting information). The problem is that it is too easy for an attacker to guess which random numbers will appear since he knows what time it is and there are a limited number of process id's "for sale" when a process starts... With this information the hacker can generate a number of sequences that are likely to occur within the application under attack.

Another problem with pseudo random number generators is that the sequences they produce eventually repeat themselves. The so called "period" of the pseudo random number generator describes how long the sequence is before it starts repeating itself. Good pseudo random number generators have periods like 2^32 (approximately 4 billion) numbers.

Why is predictability bad?

Predictability of random number generation is bad in cases where the random numbers are used in an algorithm that strives to keep something secret. The prime example is the creation of a encryption keys by programs like PGP and SSH. The secrecy of the protected data is directly dependent on the fact that the keys used are impossible to guess with any accuracy so that the attacker must search the entire key space which is not computationally feasible as long as the keys are long enough (currently anything above 64 bits will do; 128 bits is preferred).

However, if the random numbers used in the key generation are predictable, the attacker can redo the key generation algorithm with the predicted random numbers and suddenly the number of keys to try in the attack drops dramatically. In the early days of Netscape Navigator, the pseudo random number generator used by the navigator to create SSL session keys was broken, leading to an SSL session being decrypted and the data exchange to become visible (more information on this on the following web page: http://pauillac.inria.fr/~doligez/ssl/).

Are pseudo random number generators completely useless then? No, they are not. There are a lot of applications in the fields of gaming and simulation that can use pseudo random numbers. However for security applications pseudo random numbers are usually not random enough.

Does randomness exist?

It is at this point in the discussion that someone usually postulates that true randomness can not exist. Not in computers, nor in the universe at large. Many people see the universe as a big state machine where the entire future can be predicted as long as one knows the exact position, energetic charge, speed and direction of every atom. It might take a computer the size of Deep Tought (or its successor) to work out the future (or so the advocates of this

world view postulate), but in theory It Can Be Done. From this starting point randomness can not exist; everything can be calculated and all the fun would be taken out of your next spring trip to Las Vegas.

Fortunately for us these statements are not true. Quantum mechanics teaches us that truly indecisable (random) processes exist and Heisenberg's Uncertainty Principle states that it is impossible to know both the exact location and speed of a particle. The implication of this last statement is that you can not know the state of the entire universe, and that it is therefore impossible to predict the next state (however whether the slot machines in Las Vegas are quantum mechanical by nature is something else all together :-).

Scientists have discovered lots of processes that are relatively easy to measure but are quantum mechanical by nature. Examples of these processes are cosmic radiation, the movements of lava lamps and certain behaviours of heat sensitive resistors. True random data can be obtained by measuring values of these processes and using or tabulating them...

Cryptographically secure pseudo random generators

Unfortunately, generating truly random numbers (based on random physical processes) is beyond the scope of most hard- and software. For applications that want high quality random numbers but without the overhead of a satellite dish to capture random cosmic background noise we have to devise a really good random number generator, one that is *cryptographically secure*.

A cryptographically secure random generator is one that generates random sequences that pass the statistical tests for randomness and with a level of unpredictability that make it "computationally infeasible to predict what the next random bit will be, given complete knowledge of the algorithm or hardware generating the sequence and all of the previous bits in the stream" (quote from Bruce Schneier; Applied Cryptography; page 45). Although a computer can only be in a finite number of states, that number is very large and if we include enough bits of the state in a smart way we can accomplish the unpredictability that we are looking for.

Interactive programs commonly use things like mouse movements or keyboard interaction to generate the randomness that we are looking for. Clearly this method of random number generation is not suitable for batch or background applications, so we need to come up with something else...

Information entropy

Before we continue our discussion on random numbers and good ways of generating them with a computer I want to spent some time explaining the concept of information entropy (see also: http://www.wikipedia.org/wiki/Information_entropy).

The entropy of a data set (e.g. a file, or a buffer) is a measure for the amount of information in that data set. Most ways of storing information conveniently have internal redudancy. For instance this HTML file uses one byte for each letter whereas you can be absolute sure that the first bit of every byte is never used (since I am not in the habit of typing non-ASCII (char code > 127) characters in an HTML file. Of the seven bits left I also seldomly use the lower 32 characters of the ASCII character set, only a stray newline character (10) here and there and maybe a tab (9) if I'm really going wild. So that leaves approximately 90 values (all letters (52), digits (10), and some special characters like <, > and so forth) out of the possible 256 (for each byte) that I am actively using. I need about 6.5 bit to encode the characters that I use. There are then other predictions that I can make that could decrease the number of bits necessary to store this file even more. For instance I use the word "the" a lot, so it would be more efficient if I use a special one byte code (say character code 0x01) to denote "the". And so forth... Actually at the moment that I am writing this (with the article three quarters finished) the information entropy of this file is about 4.796493 bits per byte (as calculated by John Walker's "ent" program, see http://www.fourmilab.ch/random/; an HP-UX 11.00 version of "ent" (compiled) can be downloaded at http://www.josvisser.nl/hpux11-random/ent.gz).

Actually compression programs like "gzip" and "bzip2" have got everything to do with information entropy. The programs work because they can devise a way to store the same information in less bits, thereby increasing the entropy of the file to as close to 100% as possible. For instance the gzip'ed version of this HTML file has an entropy of 7.971679 bits per byte. You can now easily deduce why compressing a compressed file has no effect any more. The entropy of a compressed file is already close to 100% and to increase it even more is usually not computationally feasible.

Random data has by definition an entropy of 100%. Every bit in the data set is used efficiently (since all values are equally likely to occur, so no bit-gain there) and there is no predictability that would allow us to chop the bit count further. Therefore the entropy test is also a good test for the randomness of a data set.

Entropy as measure for chaos

The concept of information entropy has been adapted from the thermodynamical understanding of entropy (see also http://www.wikipedia.org/wiki/Ent ropy) which can amongst other things be thought of as a measure of the disorder (chaos) in a thermodynamical system (such as the universe).

I will therefore now start using the term "entropy" to loosely mean "chaos" in the remainder of this article...

Generating random numbers in the OS kernel

Let us move back to the topic of generating random numbers of a suitably high quality so that they can be used for security purposes such as OpenSSH key generation.

It has for some time now been fashionable to add a mechanism for generating good random numbers to the kernel of the operating system. Usually these random numbers are continuously gathered and then made available through a device file such as "/dev/random" and/or "/dev/urandom". The random device driver in the kernel "ensures" that a good size pool of random data is always available so that obtaining a set of random numbers is usually an I/O-less system call (read(2)) away.

The random drivers in de kernel do usually not use a mathematical function to determine random data, but instead gather entropy from kernel-level processes such as I/O patterns, interrupt timings and other hard-to-predict things. These data items are then stirred into a pool of entropy data using hashing and CRC functions to obscure the source data and maintain the statistical equilibrium.

The theory behind this mode of operation is that the things going on in the kernel are almost quantum-mechanical by nature (ultimately being driven by things like user requests in the network) and as good as impossible to predict (if done correctly).

/dev/random on Linux

A good open source example of generating random numbers in the kernel is the "/dev/random" implementation of Linux (by Theodore T'so). The "random" character device driver operates (not by coincidence :-) as was described in the previous section. It gets its basic data from the time intervals between kernel events such as interrupts, mouse movements and key presses. These timings are then manipulated using hashing and CRC functions and stirred into a pool of entropy data along with other data like the scan code of the key pressed or the location of the mouse pointer.

When random numbers are requested by the application, data is retrieved from the pool and returned to the requestor. Two interfaces are provided: the "random" and the "urandom" device files. When reading from the "random" interface ("/dev/random") the highest quality random numbers are returned. If there is not enough bits of entropy in the pool the driver blocks the requesting process until enough data is available (woken up by the lower half of the driver that fills the entropy pool). The "urandom" interface ("/dev/urandom") never blocks but just reuses the current state of the pool until the application request has been satisfied. If the pool is currently smaller than the number of bytes requested this leads to lower quality random numbers (but usually of enough quality for most programs).

This behaviour (under Linux) can easily be tested with the following simple experiment:

First execute the following command:

```
dd if=/dev/urandom of=aap bs=1 count=4096
```

This reads 4 KB of random numbers from the random pool. This command returns immediately and on my laptop yields a file with an entropy of approximately 7.94 bits per byte. If we repeat this command but now reading from the "random" interface the command blocks:

```
dd if=/dev/random of=aap bs=1 count=4096
```

The read blocks because the pool is not large enough to return 4096 bytes and the "random" interface then blocks the reading process until the pool has been refilled by the driver's lower half that runs on interrupt basis. To unblock the read just move your mouse a bit to generate entropy and whoppa, there it goes...!

User space alternatives

For those that are not blessed with good kernel based random number generators there exist a number of userspace alternatives that generate high quality random numbers by obtaining entropy from the system using user level commands like "netstat", "ps" and "ipcs". The output of these commands is typically hashed with algorithms like MD5 or SHA and then combined into a pool of random numbers that is returned to the application. In fact, OpenSSH uses this method as a last resort if the internal random number generator (provided by OpenSSL) has not been seeded from

a better (i.e. kernel based) random number generator.

The Entropy Gathering Daemon (EGD, see http://egd.sourceforge.net) is a deamon that continuously gathers entropy in this way and makes it available to other programs (like GPG or OpenSSH) via a pool file or a named pipe. The Pseudo Random Number Generator Daemon (see http://freshmeat.net/projects/prngd) is another example of this type of software.

Obviously gathering entropy in userspace by running programs that display system status costs lots of CPU cycles and takes a considerable amount of time. Another drawback to using userspace entropy gathering is that in a highly loaded system running the system commands necessary to gather the entropy might not be possible for lack of swap space or process table slots.

So although random number generation using only userspace tools is feasible, a kernel based alternative is much preferred.

/dev/random on HP-UX

HP-UX is one of these Unix implementations that is always somewhat slow in implementing new features. Consequently, until recently, it did not have a "/dev/random" implementation and the one that has come available is only for HP-UX 11i, leaving 11.00 users "in the cold"... (The 11i-only kernel based random number generator can be downloaded at http://www.software.hp.com/cgi-bin/swdepot_parser.cgi/cgi/displayProductInfo.pl?productNumber=KRNG11I.)

At one of my customers they use a great number of HP-UX 11.00 systems (over 20) to develop and run their core business software (in the field of electronic payments and financial transactions). A couple of years ago I convinced them to move to (Open)SSH for logging onto these systems for development and system administration. However, since HP-UX 11.00 lacks a kernel based random generator they were forced to rely on userspace entropy gathering. For reasons best left undiscussed they also decided not to run the Entropy Gathering Daemon. This meant that they had to rely on OpenSSH's (slow) internal entropy gathering mechanisms to generate the random numbers that are used in determining a unique random session key.

To give you and idea of how slow OpenSSH's internal entropy gatherer is I executed the "ssh-rand-helper" program a couple of times under control of the "time" command. The following is typical:

```
# time ./ssh-rand-helper
ff792a74705576ae4dc6791814d3ae4d26d674daf89efd97389564139fbf44afb6f2abb42d3e407b3b476d6f3c2e4ecd

real    0m2.21s
user    0m1.31s
sys     0m0.76s
```

As you can see the program uses somewhere around 2 whole seconds of CPU time (on a D380) to gather its entropy. It uses this time running the entropy gathering commands, as the following excerpt from "ps -ef" shows:

```
    root  4305  3849  4 10:23:19 pts/ta    0:00 ./ssh-rand-helper
    root  4307  4305  1 10:23:19 pts/ta    0:00 ls -alni /var/adm
```

These two seconds of CPU time add directly to the time needed to establish an SSH connection to another node which makes it a slow and irritating experience...

And thus the time had come for a "/dev/random" kernel based random number generator for HP-UX 11.00 (or so the customer thought :-).

The random DLKM for HP-UX 11.00

Operating under the motto "beter goed gejat dan slecht zelf bedacht" (Dutch for: "I'd rather steal something good than inventing something bad myself") I hacked away for some time throwing the code of the "dlclass" DLKM (Dynamically Loadable Kernel Module; example from HP's device driver development guide) and the "random" driver of Linux into a big bowl and after stirring for a couple of days and letting it rise out came the "random" DLKM for 32-bit and 64-bit HP-UX 11.00 kernels.

Since I could copy most of the difficult code from the two great source bases described above (for which many thanks to Theodore T'so and an unknown HP programmer) the only question left to break my head over was where to get my entropy from. Since the HP-UX kernel is not exactly open source, most interesting and good ways for gathering entropy are closed off for me. HP-UX 11i's KRNG (by HP themselves) comes with a set of kernel patches that modify other parts of the kernel to maintain entropy data for use by the KRNG module. This was clearly not possible for me.

After some deliberation it was clear to me that the easiest way to hook into ongoing kernel activity was by revectoring the I/O routines of other devices in the system. In my thoughts the easiest and most consistent sources of activity in an average HP-UX server are those of the SCSI disk driver (especially on the servers my customer has). So after some deliberation I decided to revector the strategy routine of the "sdisk" driver (which has fixed major number 31) and to get my entropy from there...

My initial idea was to take the interval time (in microseconds) between a fixed number of SCSI disk I/O's. Because the disk I/O's are physical I/O's (mostly between the buffer cache and the disk) they are very hard to predict since their exact timings depend on all sorts of system activity like memory usage, user requests and the operations of the syncer daemon. In order not to stress the system too much with this extra activity, I introduced a tunable parameter "random_io_threshold" which configures the amount of physical disk I/O's to measure. On a lightly loaded system 16 might be a good value whereas on a busy system 256 or maybe even higher might be appropriate (after installation of the software check the "/etc/rc.config.d/random" file for the current value).

Although the timings are (as far as my thinking goes right now) a good entropy source my concern was that the timings alone might not fill the entropy pool fast enough. I therefore decided to mix in some more entropy from the actual data block read/written by the SCSI Disk. So what the "random" DLKM does is that after stirring the timing interval into the entropy pool it hashes (SHA) the first 16 bytes of the data block read/written by this I/O and adds bytes from the hash into the pool as well.. Together I hoped these sources would provide entropy from a high enough quality to fill the pool fast enough and to provide high quality random numbers...

Downloading and installing

The DLKM (source and binary) can be downloadad from the following web site: http://www.josvisser.nl/hpux11-random". After unpacking you get a directory structure with the source code, necessary configuration files and two binary modules for 32 and 64 bit kernels. Building the module from the source code requires the HP ANSI C compiler (as far as I know gcc can not be used to compile kernel modules). The provided "Makefile" also takes care of the installation in the system, it copies everything into place and then loads the module in the system:

```
# make install
Determined that the 32 bit module is needed on this system...
        cp mod$(getconf KERNEL_BITS).o mod.o
        kminstall -u random
        test -f /etc/rc.config.d/random || cp random.conf
/etc/rc.config.d/random
        cp init-script.sh /sbin/init.d/random
        ln -f -s /sbin/init.d/random /sbin/rc2.d/S990random
        ln -f -s /sbin/init.d/random /sbin/rc1.d/K110random
        /sbin/init.d/random start
Generating module: random...
Requesting loadable module update...

  Specified module(s) below is(are) activated successfully.
      random

kmadmin: Module random loaded, ID = 1
        dmesg | tail -1
random> version=0.0.2; wait channel=092d4ac0; spinlock=08e93e60; io_treshold=16
```

>From this moment on, entropy is being gathered and made available through the "/dev/random" and "/dev/urandom" interfaces. The system is also set up to automatically load the module at system boot and unload it at system shutdown. If you want to manually (un)load the module you can call the "/sbin/init.d/random" script with the "start", "stop" and "restart" arguments (e.g after changing the values of the module's parameters in "/etc/rc.config.d/random").

Using /dev/random on HP-UX 11

Once the module is active, random data can immediately be read from the module's device files (that are automatically created/removed when the module is loaded/unloaded through the script in "/sbin/init.d"). OpenSSL (and by inference, OpenSSH) will do so automatically when they detect the existence of the device files. These commands will therefore be faster straight away!

```
# /sbin/init.d/random stop
kmadmin: Module 1 unloaded
# time ssh josv@gatekeeper date
Fri Sep 12 22:15:26 CEST 2003

real    0m3.58s
user    0m2.19s
sys     0m0.70s
```

```
# /sbin/init.d/random start
Generating module: random...
Requesting loadable module update...

  Specified module(s) below is(are) activated successfully.
      random

kmadmin: Module random loaded, ID = 1
# time ssh josv@gatekeeper date
Fri Sep 12 22:15:43 CEST 2003

real    0m1.19s
user    0m0.88s
sys     0m0.06s
```

As you can see from this example using the "/dev/random" interface (or actually, "/dev/urandom" because that is what OpenSSL uses) cuts a whopping 1.95 CPU seconds from OpenSSH's processing!

Remember that you read about processes blocking on reading "/dev/random" if there was not enough entropy data available? In the following example you can see just that:

```
# dd if=/dev/random of=/tmp/aap bs=32 count=32 &
[1]     4785
# ps -l -p 4785
  F S        UID   PID  PPID  C PRI NI             ADDR   SZ WCHAN   TTY        TIME COMD
  1 S          0  4785  3849  0 148 24          91aaf00   15 92d4ac0 pts/ta     0:00 dd
```

Here I try to read one kilobyte of random data from the pool. There probably is not enough data in the pool and thus the program blocks. The "WCHAN" field in the "ps -l" output gives the "wait channel address"; this is an internal kernel address of a data structure that the process is blocking on. In this case (on this machine, with this kernel), the address "0x92d4ac0" is the wait channel used by the random driver. You can verify that in the "dmesg" output shown earlier where the driver reports its own wait channel address...

If I now start generating disk I/O's, the random drivers will start retrieving entropy and the command will finish:

```
# find / >/dev/null
32+0 records in
32+0 records out
[1] +  Done                    dd if=/dev/random of=/tmp/aap bs=32 count=32 &
# ls -l /tmp/aap
-rw-r--r--   1 root       sys            1024 Sep 12 14:33 /tmp/aap
```

How good is "my" random data...

An interesting question obviously is how good the random data is that is provided by the DLKM. As a test I ran the "ent" program described earlier over a four kilobyte random data obtained in a manner like was described above (but now with 64 blocks of 64 bytes). The "ent" program performs a number of statistical tests for randomness and thereby gives a clue about the quality of a random number sequence. In my test run, "ent" produced the following output:

```
# ./ent /tmp/aap
Entropy = 7.954200 bits per byte.

Optimum compression would reduce the size
of this 4096 byte file by 0 percent.

Chi square distribution for 4096 samples is 262.25, and randomly
would exceed this value 50.00 percent of the times.

Arithmetic mean value of data bytes is 126.6506 (127.5 = random).
Monte Carlo value for Pi is 3.114369501 (error 0.87 percent).
Serial correlation coefficient is -0.005663 (totally uncorrelated = 0.0).
```

The entropy of this set is rather good and the other statistical measurements look good too. However I must confess that I still need to do most of the basic math on the current implementation of the DLKM (any help would be appreciated :-). When compared to the kernel based random number generator of Linux the figures are more or less comparable.

Concluding

Although HP-UX 11.00 is not new, there are still a fair number of users out there that could benefit from a kernel based random number generator. I wholeheartedly invite them to start using this piece of software. It is 32/64 bit safe and has been stress tested on SMP systems without giving any problems. Because of the way entropy is generated I

add a number of instructions to every SCSI disk I/O, but given the time involved in doing I/O in the first place I feel the trade off is well worth it.

And, as is the custom with Open Source Software, I invite everyone to file their complaints, comments and suggestions with me, josv at osp.nl.

Other articles by Jos Visser:

- Comal has been set free
- Unix failover clusters - If you've seen one, you've seen 'm all
- Interessante ontwikkelingen in Linux
- Rock Linux: Not for woozies!

E-mail: Jos Visser

[Index]