Ruby master - Bug #15039

Random.urandom and SecureRandom arc4random use

08/28/2018 07:16 PM - Freaky (Thomas Hurst)

Status:	Closed		
Priority:	Normal		
Assignee:			
Target version:			
ruby -v:		**Backport:**	2.3: UNKNOWN, 2.4: UNKNOWN, 2.5: UNKNOWN

Description

Random.urandom defaults to arc4random() on a lot of platforms, including FreeBSD.

On all currently released versions of FreeBSD, arc4random() is, as the name suggests, a dubious ARC4-based userspace PRNG dating from circa 1997. Given the entire point of #9569 was that using the userspace CSPRNG in OpenSSL over /dev/urandom or equivalent is a bad idea, this seems to mean it's regressed to an *even worse* state on these platforms. Even in cases where it's using something more modern (FreeBSD 12, OpenBSD), it's still a userspace CSPRNG.

If that's fine, we might as well *pick a known-good one* and use that everywhere. Like, say, OpenSSL's.

Since the conclusion of #9569 seems to have been otherwise, I'd suggest dropping arc4random() as a potential source for Random.urandom due to it not matching the desired semantics.

Rust's OsRng seems a good template for alternative _syscall implementations:
https://docs.rs/rand/0.5.5/rand/rngs/struct.OsRng.html#platform-sources

Related issues:

Related to Ruby master - Bug #9569: SecureRandom should try /dev/urandom first	**Closed**

Associated revisions

Revision b120f5e3 - 09/04/2018 01:42 AM - shyouhei (Shyouhei Urabe)

avoid fork-unsafe arc4random implementations

Some old implementaions of arc4random_buf(3) were ARC4 based, or
unsafe when forked, or both. Resort to /dev/urandom for those
known problematic cases. Fix [Bug #15039]

Patch from Thomas Hurst tom@hur.st

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@64625 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 64625 - 09/04/2018 01:42 AM - shyouhei (Shyouhei Urabe)

avoid fork-unsafe arc4random implementations

Some old implementaions of arc4random_buf(3) were ARC4 based, or
unsafe when forked, or both. Resort to /dev/urandom for those
known problematic cases. Fix [Bug #15039]

Patch from Thomas Hurst tom@hur.st

Revision 64625 - 09/04/2018 01:42 AM - shyouhei (Shyouhei Urabe)

avoid fork-unsafe arc4random implementations

Some old implementaions of arc4random_buf(3) were ARC4 based, or
unsafe when forked, or both. Resort to /dev/urandom for those
known problematic cases. Fix [Bug #15039]

Patch from Thomas Hurst tom@hur.st

History

#1 - 08/28/2018 07:46 PM - jeremyevans0 (Jeremy Evans)

On OpenBSD, arc4random is not really a userspace PRNG in the sense that it is reseeded on a regular basis using new entropy data from the kernel

(arc4random_buf(3) -> _rs_random_buf -> _rs_stir_if_needed -> _rs_stir -> getentropy(2)). See https://github.com/openbsd/src/blob/b66614995ab119f75167daaa7755b34001836821/lib/libc/crypt/arc4random.c

I have no opinion on what the Random.urandom and SecureRandom implementations should default to on other operating systems, but on OpenBSD they should keep the current default of arc4random_buf(3).

Looking at FreeBSD 11.2 (https://github.com/freebsd/freebsd/blob/releng/11.2/lib/libc/gen/arc4random.c), it appears to also reseed on a regular basis using entropy data from the kernel (arc4random_buf(3) -> arc4_stir_if_needed -> arc4_stir -> arc4_sysctl -> __sysctl with KERN_ARND). So I don't think that should be considered a userspace PRNG, either. However, if the FreeBSD kernel is compiled without the RANDOM kernel module, it could probably fallback to being a userspace PRNG.

From the link to the Rust OSRng, it uses getentropy(2) for OpenBSD and kern.arandom for FreeBSD, which appears to be similar to what OpenBSD and FreeBSD use internally in their arc4random_buf(3) implementations.

#2 - 08/29/2018 12:31 AM - Freaky (Thomas Hurst)

jeremyevans0 (Jeremy Evans) wrote:

> On OpenBSD, arc4random is not really a userspace PRNG in the sense that it is reseeded on a regular basis using new entropy data from the kernel

It runs code in userspace to stretch a small amount of entropy into a large amount. I think that's a fair definition of "userspace PRNG", no?

In this case, that "regular basis" is every 1.6MB:

```
rs->rs_count = 1600000;
```

Why not just reseed OpenSSL's CSPRNG that often, if this was sufficient to mitigate the concerns in #9569?

> I have no opinion on what the Random.urandom and SecureRandom implementations should default to on other operating systems, but on OpenBSD they should keep the current default of arc4random_buf(3).

I think users should have a reasonable expectation of the semantics of these APIs remaining the same across systems. arc4random keeps the state predominantly in the Ruby process: you're one Spectre/Cloudbleed-style attack away from the next million bytes of SecureRandom output being guessable by an attacker and one fork-detection-mishap away from two processes generating large quantities of identical randomness.

Failing that, I'd at *least* like to see arc4random only used on platforms where it's confirmed to be reasonably trustworthy.

> Looking at FreeBSD 11.2 (https://github.com/freebsd/freebsd/blob/releng/11.2/lib/libc/gen/arc4random.c), it appears to also reseed on a regular basis

In that regard it's exactly the same as on OpenBSD - it reseeds every 1.6MB. The data in between all comes from stretching that small pool of entropy using a broken stream cipher dating from the late 1980's. And yes, it has a dubious silent fail-dangerous fallback path if its seeding efforts fail, and who knows how good it is at detecting fork (also fail-dangerous judging by comments in OpenBSD compat code).

> From the link to the Rust OSRng, it uses getentropy(2) for OpenBSD and kern.arandom for FreeBSD, which appears to be similar to what OpenBSD and FreeBSD use internally in their arc4random_buf(3) implementations.

Yes, but that's *all* they use. And they'll fail if they can't get it, rather than falling back to getpid, gettimeofday and whatever junk happens to be on the stack...

#3 - 08/29/2018 02:51 AM - jeremyevans0 (Jeremy Evans)

Freaky (Thomas Hurst) wrote:

> jeremyevans0 (Jeremy Evans) wrote:
>
> > On OpenBSD, arc4random is not really a userspace PRNG in the sense that it is reseeded on a regular basis using new entropy data from the kernel
>
> It runs code in userspace to stretch a small amount of entropy into a large amount. I think that's a fair definition of "userspace PRNG", no?

srand/rand is the classic userspace PRNG, where all future output is based only on the initial input.

arc4random_buf(3) is basically like fread(3), where getentropy(2) is like read(2). Would you consider fread(3) userspace I/O simply because it buffers?

> In this case, that "regular basis" is every 1.6MB:

```
rs->rs_count = 1600000;
```

Why not just reseed OpenSSL's CSPRNG that often, if this was sufficient to mitigate the concerns in #9569?

Why do something more complex when the library function handles correctly for you (at least in OpenBSD's case)?

> I have no opinion on what the Random.urandom and SecureRandom implementations should default to on other operating systems, but on OpenBSD they should keep the current default of arc4random_buf(3).

> I think users should have a reasonable expectation of the semantics of these APIs remaining the same across systems. arc4random keeps the state predominantly in the Ruby process: you're one Spectre/Cloudbleed-style attack away from the next million bytes of SecureRandom output being guessable by an attacker and one fork-detection-mishap away from two processes generating large quantities of identical randomness.

If you are susceptible to a memory attack like Spectre/Cloudbleed, you probably have bigger issues than just arc4random_buf.

At least on OpenBSD, fork-detection-mishaps can't happen due to MAP_INHERIT_ZERO.

> Failing that, I'd at *least* like to see arc4random only used on platforms where it's confirmed to be reasonably trustworthy.

That seems fine to me. I can confirm it is "reasonably trustworthy" on OpenBSD, not sure about other operating systems.

> Looking at FreeBSD 11.2 (https://github.com/freebsd/freebsd/blob/releng/11.2/lib/libc/gen/arc4random.c), it appears to also reseed on a regular basis

> In that regard it's exactly the same as on OpenBSD - it reseeds every 1.6MB. The data in between all comes from stretching that small pool of entropy using a broken stream cipher dating from the late 1980's. And yes, it has a dubious silent fail-dangerous fallback path if its seeding efforts fail, and who knows how good it is at detecting fork (also fail-dangerous judging by comments in OpenBSD compat code).

Those may be reasons to choose something different on FreeBSD, but I make no recommendation in that area.

> From the link to the Rust OSRng, it uses getentropy(2) for OpenBSD and kern.arandom for FreeBSD, which appears to be similar to what OpenBSD and FreeBSD use internally in their arc4random_buf(3) implementations.

> Yes, but that's *all* they use. And they'll fail if they can't get it, rather than falling back to getpid, gettimeofday and whatever junk happens to be on the stack...

On OpenBSD, arc4random_buf cannot fail. It returns void, and doesn't have a fallback implementation as the kernel can always provide randomness when stirring.

Have you reviewed the OpenSSL PRNG implementation for old and recent versions of OpenSSL to confirm that they are not susceptible to any of the issues you are concerned with?

#4 - 08/29/2018 03:17 AM - shyouhei (Shyouhei Urabe)

- Related to Bug #9569: SecureRandom should try /dev/urandom first added

#5 - 08/29/2018 03:53 AM - shyouhei (Shyouhei Urabe)

Frankly, I'm not sure how less trustworthy FreeBSD libc is, compared to its kernel. They are developed by the same people in a same repository. Or is it not about the code quality in general? Maybe the problem here is that having any bits of instructions in a userland?

#6 - 08/29/2018 04:19 AM - Freaky (Thomas Hurst)

jeremyevans0 (Jeremy Evans) wrote:

> Freaky (Thomas Hurst) wrote:
>
> > jeremyevans0 (Jeremy Evans) wrote:
> >
> > > On OpenBSD, arc4random is not really a userspace PRNG in the sense that it is reseeded on a regular basis using new entropy data from the kernel
> >
> > It runs code in userspace to stretch a small amount of entropy into a large amount. I think that's a fair definition of "userspace PRNG", no?
>
> srand/rand is the classic userspace PRNG, where all future output is based only on the initial input.

Would rand() stop being a userspace PRNG if it automatically called srand() every 200,000 calls?

> arc4random_buf(3) is basically like fread(3), where getentropy(2) is like read(2). Would you consider fread(3) userspace I/O simply because it buffers?

Yes, and that distinction is quite important for many IO users.

The distinction can be important to CSPRNG users too, but SecureRandom/Random.urandom don't exactly help by saying "well, it depends on whether you're running Linux, Windows, or a BSD/Solaris derivative". If you wanted something fast with weaker state security guarantees you'd end up using something else anyway.

> > Why not just reseed OpenSSL's CSPRNG that often, if this was sufficient to mitigate the concerns in #9569?
>
> Why do something more complex when the library function handles correctly for you (at least in OpenBSD's case)?

Calling an OpenSSL reseed function occasionally wasn't a more complex change than reworking the Random/SecureRandom framework to favour the equivalent of /dev/urandom on popular platforms. Nor is calling getentropy() any more complex than only calling arc4random() when the implementation is sane.

Ultimately, the question is what do Random.urandom/SecureRandom *mean*.

I've been using SecureRandom expecting it to use Yarrow and Fortuna (via /dev/urandom or equivalent) as provided by my kernel, only to find it's using some terrible legacy interface lurking in libc. And even if arc4random() was just fine (as it will hopefully be in FreeBSD 12), I'd still like to be able to err on the side of caution and just directly use urandom or equivalent, as repeatedly recommended by security experts.

If SecureRandom isn't going to provide that guarantee, then maybe we need something like OsRandom that does.

> If you are susceptible to a memory attack like Spectre/Cloudbleed, you probably have bigger issues than just arc4random_buf.

Sure, and anyone with such big problems will probably appreciate not having them added to by things like "attackers can predict the next 10,000 API keys following a successful attack". If we didn't believe in defence in depth, why bother mitigating Meltdown?

> At least on OpenBSD, fork-detection-mishaps can't happen due to MAP_INHERIT_ZERO.

Neat. On FreeBSD 11.2 it's just a static pid_t compared with getpid() :(

#7 - 08/29/2018 04:50 AM - Freaky (Thomas Hurst)

shyouhei (Shyouhei Urabe) wrote:

> Frankly, I'm not sure how less trustworthy FreeBSD libc is, compared to its kernel. They are developed by the same people in a same repository.

FreeBSD /dev/(u)random's seen a *lot* more attention over the years. It would be wrong to think just because it's in the same project that it's all of uniform quality. People work on and look at the things that interest them, and old, lesser used features often languish.

> Or is it not about the code quality in general? Maybe the problem here is that having any bits of instructions in a userland?

It's a mix of the two, and it might be worth separating the issues.

One is that arc4random() is pretty much only sensibly implemented on OpenBSD, as far as I know, and its use should probably be limited to that and maybe other platforms where it's *confirmed* to be not-awful.

The other is clarifying the intent of Random.urandom and the priorities of SecureRandom. Following #9569, *are* they meant to be avoiding using userspace CSPRNGs? They do so on Linux and Windows, but don't on platforms with arc4random().

#8 - 08/29/2018 05:55 AM - jeremyevans0 (Jeremy Evans)

Freaky (Thomas Hurst) wrote:

> Would rand() stop being a userspace PRNG if it automatically called srand() every 200,000 calls?

If it received the argument to srand() from the kernel, then maybe. :)

> > arc4random_buf(3) is basically like fread(3), where getentropy(2) is like read(2). Would you consider fread(3) userspace I/O simply because it buffers?

> Yes, and that distinction is quite important for many IO users.
>
> The distinction can be important to CSPRNG users too, but SecureRandom/Random.urandom don't exactly help by saying "well, it depends on whether you're running Linux, Windows, or a BSD/Solaris derivative". If you wanted something fast with weaker state security guarantees you'd end up using something else anyway.

I think in general SecureRandom doesn't need to specify how it is implemented, just that the related bytes are cryptographically random. If that isn't true on FreeBSD <12, then I agree Ruby should use a different approach in that case, but let's limit any changes to that specific case (or other specific cases known to be problematic).

> > Why not just reseed OpenSSL's CSPRNG that often, if this was sufficient to mitigate the concerns in #9569?
>
> Why do something more complex when the library function handles correctly for you (at least in OpenBSD's case)?
>
> Calling an OpenSSL reseed function occasionally wasn't a more complex change than reworking the Random/SecureRandom framework to favour the equivalent of /dev/urandom on popular platforms. Nor is calling getentropy() any more complex than only calling arc4random() when the implementation is sane.

I think manual reseeding using the OpenSSL PRNG is significantly more complex than just calling arc4random_buf. Using getentropy instead of arc4random_buf is a little more complex, as getentropy is limited to 256 bytes (on OpenBSD).

> Ultimately, the question is what do Random.urandom/SecureRandom *mean*.
>
> I've been using SecureRandom expecting it to use Yarrow and Fortuna (via /dev/urandom or equivalent) as provided by my kernel, only to find it's using some terrible legacy interface lurking in libc. And even if arc4random() was just fine (as it will hopefully be in FreeBSD 12), I'd still like to be able to err on the side of caution and just directly use urandom or equivalent, as repeatedly recommended by security experts.

That recommendation does not necessarily apply to operating systems other than FreeBSD.

> If SecureRandom isn't going to provide that guarantee, then maybe we need something like OsRandom that does.
>
> > If you are susceptible to a memory attack like Spectre/Cloudbleed, you probably have bigger issues than just arc4random_buf.
>
> Sure, and anyone with such big problems will probably appreciate not having them added to by things like "attackers can predict the next 10,000 API keys following a successful attack". If we didn't believe in defence in depth, why bother mitigating Meltdown?

It sounds like you want a new feature: (SecureRandom|Random).sysrandom (analog of File#sysread, which does a system call for every method call).

If you think OpenSSL is a better default than arc4random_buf, you may want to look at the OpenSSL 1.0.1 implementation (oldest OpenSSL version currently supported by ruby-openssl): https://github.com/openssl/openssl/blob/OpenSSL_1_0_1-stable/crypto/rand/rand_unix.c

#9 - 08/29/2018 08:41 AM - shyouhei (Shyouhei Urabe)

Freaky (Thomas Hurst) wrote:

> > Or is it not about the code quality in general? Maybe the problem here is that having any bits of instructions in a userland?
>
> It's a mix of the two, and it might be worth separating the issues.
>
> One is that arc4random() is pretty much only sensibly implemented on OpenBSD, as far as I know, and its use should probably be limited to that and maybe other platforms where it's *confirmed* to be not-awful.

Hmm, OK. I respect your feeling that the arc4random() in FreeBSD (up to 11) is questionable. +1 to avoid using such implementation(s).

> The other is clarifying the intent of Random.urandom and the priorities of SecureRandom. Following #9569, *are* they meant to be avoiding using userspace CSPRNGs? They do so on Linux and Windows, but don't on platforms with arc4random().

As far as I understand Random.urandom intends to be cryptographically secure; nothing more. My attempt when fixing #9569 was that arc4random() supposedly adequately fulfilled this property like OpenBSD's. I did not see any reason to reject such thing. Maybe is it getting hard for a "userspace CSPRNG" in general to make sense these days? That is a possible situation and I have to change my mind then.

#10 - 08/29/2018 04:16 PM - Freaky (Thomas Hurst)

jeremyevans0 (Jeremy Evans) wrote:

> Freaky (Thomas Hurst) wrote:
>
> > Would rand() stop being a userspace PRNG if it automatically called srand() every 200,000 calls?
>
> If it received the argument to srand() from the kernel, then maybe. :)

Even if that were the case, it's storing its state in userspace, it's permuting that state in userspace. Where it comes from is beside the point, as is how often it's reseeded. Would you still think it might stop being userspace if reseeds happened every 2 million calls? 2 billion?

The only distinction for "userspace PRNG" that makes sense as far as I can see is whether or not it runs predominantly in userspace. If you're not asking the kernel *each time*, you have a userspace PRNG.

> I think in general SecureRandom doesn't need to specify how it is implemented, just that the related bytes are cryptographically random.

It currently does specify vaguely how it's implemented, albeit incorrectly:

https://ruby-doc.org/stdlib-2.6.0.preview2/libdoc/securerandom/rdoc/SecureRandom.html

It's also quite vague about what it's suitable for. "session keys in HTTP cookies, etc" :/

> If that isn't true on FreeBSD <12, then I agree Ruby should use a different approach in that case, but let's limit any changes to that specific case (or other specific cases known to be problematic).

I strongly suggest doing it the other way around. Don't blacklist known-problematic implementations, carefully whitelist known-good ones.

I think NetBSD >= 7, FreeBSD >= 12 and OpenBSD >= 5.5 should all be reasonable, DragonFlyBSD is less so.

> > Calling an OpenSSL reseed function occasionally wasn't a more complex change than reworking the Random/SecureRandom framework to favour the equivalent of /dev/urandom on popular platforms. Nor is calling getentropy() any more complex than only calling arc4random() when the implementation is sane.
>
> I think manual reseeding using the OpenSSL PRNG is significantly more complex than just calling arc4random_buf.

It's not "just calling arc4random_buf", it's calling arc4random_buf where available, calling getrandom where available, calling CryptGenRandom where available, using /dev/urandom where available. Those are what resolved #9569, a fair bit of effort, when *by your argument* it might as well have just been a tweak to ossl_rand_bytes() like:

```
static int until_reseed = 1600000;
until_reseed -= n;
if (until_reseed <= 0)
    RAND_poll();
```

Because then it wouldn't be a userspace CSPRNG, right? :P

> That recommendation does not necessarily apply to operating systems other than FreeBSD.

The arguments against userspace CSPRNG are fairly general: they add additional points of failure for the sake of performance.

> It sounds like you want a new feature: (SecureRandom|Random).sysrandom (analog of File#sysread, which does a system call for every method call).

I'd rather have SystemRandom that was guaranteed to do a syscall, then it can have all the same utility methods and pretty much be a drop-in replacement for SecureRandom where desired.

A CryptoRandom counterpart always guaranteed to be a decent fast userspace CSPRNG might be nice, too.

Which SecureRandom should behave like is bit of a bikeshed. I'd look at Secure and wiggle my eyebrows. Keeping CSPRNG state in the kernel is more secure, right? And this is how it already behaves on Linux and Windows.

#11 - 08/29/2018 04:34 PM - Freaky (Thomas Hurst)

shyouhei (Shyouhei Urabe) wrote:

> Freaky (Thomas Hurst) wrote:

> The other is clarifying the intent of Random.urandom and the priorities of SecureRandom. Following #9569, *are* they meant to be avoiding using userspace CSPRNGs? They do so on Linux and Windows, but don't on platforms with arc4random().

> As far as I understand Random.urandom intends to be cryptographically secure; nothing more.

Doesn't the name rather waggle its eyebrows at you while glancing pointedly at /dev/urandom? Why else call it that?

> My attempt when fixing #9569 was that arc4random() supposedly adequately fulfilled this property like OpenBSD's. I did not see any reason to reject such thing. Maybe is it getting hard for a "userspace CSPRNG" in general to make sense these days? That is a possible situation and I have to change my mind then.

Userspace is always going to be a bit riskier - the state is stored in the process itself rather than isolated in the kernel, fork() can trigger difficult to detect edge-cases (hopefully nobody breaks minherit()!), and their implementations tend to have a more, er, "varied" history.

The payoff, of course, is some orders of magnitude better performance. I'd like that option, but I'm not sure SecureRandom should be taking it by default.

#12 - 08/29/2018 08:42 PM - naruse (Yui NARUSE)

I agree with your fundamental concept: if there's more code, there's more bugs.

I don't fully agree with "userspace CSPRNG is harmful" because it's required.

For example, ruby can't read /dev/urandom if it runs in jail.
And as Jeremy already pointend, OpenBSD doesn't have urandom alternative syscall because max output size of getentropy(2) is 256 bytes.
(It means getentropy(2) is not a urandom alternative; just call getentropy(2) enough times is not acceptable)
Therefore the use of arc4random(3) is not avoidable on OpenBSD.

The use of kern.arandom on FreeBSD can be acceptable.
On NetBSD, we should use cprng functions.
http://netbsd.gw.com/cgi-bin/man-cgi?cprng+9.i386+NetBSD-7.0.2
On Darwin, there seems similar issue as OpenBSD.

#13 - 08/29/2018 11:37 PM - Freaky (Thomas Hurst)

naruse (Yui NARUSE) wrote:

> I agree with your fundamental concept: if there's more code, there's more bugs.
>
> I don't fully agree with "userspace CSPRNG is harmful" because it's required.
>
> For example, ruby can't read /dev/urandom if it runs in jail.

If the OS can't provide entropy to a process through urandom or an equivalent syscall, how is it going to safely seed a fallback CSPRNG?

(Rust's rand can use JitterRng, for what it's worth: https://docs.rs/rand/0.5.5/rand/rngs/struct.JitterRng.html - seems a little esoteric).

> And as Jeremy already pointend, OpenBSD doesn't have urandom alternative syscall because max output size of getentropy(2) is 256 bytes.
> (It means getentropy(2) is not a urandom alternative; just call getentropy(2) enough times is not acceptable)
> Therefore the use of arc4random(3) is not avoidable on OpenBSD.

Both the existing /dev/urandom *and* getrandom() paths have to support calling their respective syscalls enough times to fill the requested bytes. Why is it suddenly unacceptable when it's a guaranteed cap of 256 bytes instead of an arbitrary dynamic cap?

getentropy() on OpenBSD is certainly a lot simpler to support than getrandom() on Linux.

> On Darwin, there seems similar issue as OpenBSD.

SecRandomCopyBytes looks fairly sane:
https://developer.apple.com/documentation/security/1399291-secrandomcopybytes?preferredLanguage=occ

#14 - 08/30/2018 07:49 AM - naruse (Yui NARUSE)

Freaky (Thomas Hurst) wrote:

> naruse (Yui NARUSE) wrote:
>
> > I agree with your fundamental concept: if there's more code, there's more bugs.

> > I don't fully agree with "userspace CSPRNG is harmful" because it's required.
> >
> > For example, ruby can't read /dev/urandom if it runs in jail.
>
> If the OS can't provide entropy to a process through urandom or an equivalent syscall, how is it going to safely seed a fallback CSPRNG?
>
> (Rust's rand can use JitterRng, for what it's worth: https://docs.rs/rand/0.5.5/rand/rngs/struct.JitterRng.html - seems a little esoteric).

There's /dev/random and getentropy.
But the size of those entropy is limited.

> > And as Jeremy already pointend, OpenBSD doesn't have urandom alternative syscall because max output size of getentropy(2) is 256 bytes.
> > (It means getentropy(2) is not a urandom alternative; just call getentropy(2) enough times is not acceptable)
> > Therefore the use of arc4random(3) is not avoidable on OpenBSD.
>
> Both the existing /dev/urandom *and* getrandom() paths have to support calling their respective syscalls enough times to fill the requested bytes. Why is it suddenly unacceptable when it's a guaranteed cap of 256 bytes instead of an arbitrary dynamic cap?
>
> getentropy() on OpenBSD is certainly a lot simpler to support than getrandom() on Linux.

Linux's getrandom has flags argument.
By default it behaves like getentropy and /dev/random, but if GRND_NONBLOCK is given it behaves like /dev/urandom.

> > On Darwin, there seems similar issue as OpenBSD.
>
> SecRandomCopyBytes looks fairly sane:
> https://developer.apple.com/documentation/security/1399291-secrandomcopybytes?preferredLanguage=occ

Thanks, I look it further.

#15 - 08/30/2018 11:06 AM - naruse (Yui NARUSE)

naruse (Yui NARUSE) wrote:

> > On Darwin, there seems similar issue as OpenBSD.
>
> SecRandomCopyBytes looks fairly sane:
> https://developer.apple.com/documentation/security/1399291-secrandomcopybytes?preferredLanguage=occ
>
> Thanks, I look it further.

As far as I investigated, it's implemented with CCRandomCopyBytes() and fork safe. We should use this.
https://stackoverflow.com/a/21736905/515572

#16 - 08/30/2018 01:29 PM - Freaky (Thomas Hurst)

naruse (Yui NARUSE) wrote:

> Freaky (Thomas Hurst) wrote:
>
> > naruse (Yui NARUSE) wrote:
> >
> > > For example, ruby can't read /dev/urandom if it runs in jail.
> >
> > If the OS can't provide entropy to a process through urandom or an equivalent syscall, how is it going to safely seed a fallback CSPRNG?
>
> There's /dev/random and getentropy.

... but you've just described concerns over situations where such APIs are unavailable. If you have no /dev/urandom why would you have a /dev/random? If you have no non-blocking syscall for entropy why would you have a blocking one?

> But the size of those entropy is limited.

Entropy's not really limited, the entire point of CSPRNGs is that you can stretch 256 bits of entropy out practically forever because the effort required to recover those bits and predict anything *exceeds the physical limitations of the universe*.

> > Both the existing /dev/urandom *and* getrandom() paths have to support calling their respective syscalls enough times to fill the requested bytes.
> > Why is it suddenly unacceptable when it's a guaranteed cap of 256 bytes instead of an arbitrary dynamic cap?
> >
> > getentropy() on OpenBSD is certainly a lot simpler to support than getrandom() on Linux.
>
> Linux's getrandom has flags argument.
> By default it behaves like getentropy and /dev/random, but if GRND_NONBLOCK is given it behaves like /dev/urandom.

http://man7.org/linux/man-pages/man2/getrandom.2.html

No it doesn't. "*By default, getrandom() draws entropy from the urandom source*".

GRND_NONBLOCK's only affect is to return an error instead of blocking if the urandom source hasn't been seeded - i.e. very early in the boot sequence, which I'm not sure Ruby should be caring about. /dev/urandom also blocks like this on most platforms.

#17 - 08/31/2018 08:19 AM - naruse (Yui NARUSE)

Freaky (Thomas Hurst) wrote:

> naruse (Yui NARUSE) wrote:
>
> > Freaky (Thomas Hurst) wrote:
> >
> > > naruse (Yui NARUSE) wrote:
> > >
> > > > For example, ruby can't read /dev/urandom if it runs in jail.
> > >
> > > If the OS can't provide entropy to a process through urandom or an equivalent syscall, how is it going to safely seed a fallback CSPRNG?
> >
> > There's /dev/random and getentropy.
>
> ... but you've just described concerns over situations where such APIs are unavailable. If you have no /dev/urandom why would you have a /dev/random? If you have no non-blocking syscall for entropy why would you have a blocking one?

Even if /dev/random is not accessible, it can use getentropy(2).

> But the size of those entropy is limited.

> Entropy's not really limited, the entire point of CSPRNGs is that you can stretch 256 bits of entropy out practically forever because the effort required to recover those bits and predict anything *exceeds the physical limitations of the universe*.

Yes.
But Random.urandom shouldn't be limited.

> > Both the existing /dev/urandom *and* getrandom() paths have to support calling their respective syscalls enough times to fill the requested bytes.
> > Why is it suddenly unacceptable when it's a guaranteed cap of 256 bytes instead of an arbitrary dynamic cap?
> >
> > getentropy() on OpenBSD is certainly a lot simpler to support than getrandom() on Linux.
>
> Linux's getrandom has flags argument.
> By default it behaves like getentropy and /dev/random, but if GRND_NONBLOCK is given it behaves like /dev/urandom.

> http://man7.org/linux/man-pages/man2/getrandom.2.html
>
> No it doesn't. "*By default, getrandom() draws entropy from the urandom source*".
>
> GRND_NONBLOCK's only affect is to return an error instead of blocking if the urandom source hasn't been seeded - i.e. very early in the boot sequence, which I'm not sure Ruby should be caring about. /dev/urandom also blocks like this on most platforms.

Ah, you are correct.
But I mean GRND_RANDOM.
getentropy(2) always behaves as GRND_RANDOM and Random.urandom with getentropy(2) will require userspace CSPRNG.
But getrandom without GRND_RANDOM behaes as /dev/urandom, and Random.urandom with getrandom works without userspace CSPRNG.

#18 - 08/31/2018 04:49 PM - Freaky (Thomas Hurst)

naruse (Yui NARUSE) wrote:

> Freaky (Thomas Hurst) wrote:
>
> > ... but you've just described concerns over situations where such APIs are unavailable. If you have no /dev/urandom why would you have a /dev/random? If you have no non-blocking syscall for entropy why would you have a blocking one?
>
> Even if /dev/random is not accessible, it can use getentropy(2).

As far as I know getentropy(2) behaves as urandom, just with a size limit.

> > But the size of those entropy is limited.
>
> Entropy's not really limited, the entire point of CSPRNGs is that you can stretch 256 bits of entropy out practically forever because the effort required to recover those bits and predict anything *exceeds the physical limitations of the universe*.
>
> Yes.
> But Random.urandom shouldn't be limited.

Yes, with the caveat that it should block if the urandom source has yet to be seeded (because then it can't give a reasonable guarantee of cryptographic randomness, which is what it's for, right?)

> GRND_NONBLOCK's only affect is to return an error instead of blocking if the urandom source hasn't been seeded - i.e. very early in the boot sequence, which I'm not sure Ruby should be caring about. /dev/urandom also blocks like this on most platforms.
>
> Ah, you are correct.
> But I mean GRND_RANDOM.

Well, that's not being used by Ruby. And I don't think it ever should, it's basically just asking for the brain-dead Linux-style /dev/random behaviour where it tries to guess how much entropy is "used" and blocks arbitrarily based on guessing how much more it's gathered.

GRND_NONBLOCK *is* used if need_secure is false, which I suppose is fair enough - it can fall back to /dev/urandom and provide some maybe-predictable randomness instead. This would be quite Linux-specific, though.

> getentropy(2) always behaves as GRND_RANDOM and Random.urandom with getentropy(2) will require userspace CSPRNG.

What makes you think OpenBSD's getentropy blocks?

https://man.openbsd.org/getentropy.2: no mention of blocking, and the implementation:

https://github.com/openbsd/src/blob/b66614995ab119f75167daaa7755b34001836821/sys/dev/rnd.c#L916-L933

Literally just a kernel-space arc4random_buf call.

#19 - 08/31/2018 06:18 PM - naruse (Yui NARUSE)

Freaky (Thomas Hurst) wrote:

> naruse (Yui NARUSE) wrote:
>
> > Freaky (Thomas Hurst) wrote:
> >
> > > ... but you've just described concerns over situations where such APIs are unavailable. If you have no /dev/urandom why would you have a /dev/random? If you have no non-blocking syscall for entropy why would you have a blocking one?
> >
> > Even if /dev/random is not accessible, it can use getentropy(2).

> As far as I know getentropy(2) behaves as urandom, just with a size limit.

As you mentioned, it's actually urandom.
I misunderstood.

> > But the size of those entropy is limited.
>
> Entropy's not really limited, the entire point of CSPRNGs is that you can stretch 256 bits of entropy out practically forever because the effort required to recover those bits and predict anything *exceeds the physical limitations of the universe*.

> Yes.
> But Random.urandom shouldn't be limited.

> Yes, with the caveat that it should block if the urandom source has yet to be seeded (because then it can't give a reasonable guarantee of cryptographic randomness, which is what it's for, right?)

Yeah, it should block if there's no entropy.

> > GRND_NONBLOCK's only affect is to return an error instead of blocking if the urandom source hasn't been seeded - i.e. very early in the boot sequence, which I'm not sure Ruby should be caring about. /dev/urandom also blocks like this on most platforms.
>
> Ah, you are correct.
> But I mean GRND_RANDOM.

> Well, that's not being used by Ruby. And I don't think it ever should, it's basically just asking for the brain-dead Linux-style /dev/random behaviour where it tries to guess how much entropy is "used" and blocks arbitrarily based on guessing how much more it's gathered.
>
> GRND_NONBLOCK *is* used if need_secure is false, which I suppose is fair enough - it can fall back to /dev/urandom and provide some maybe-predictable randomness instead. This would be quite Linux-specific, though.

What I said in this is come from misunderstand that "getentropy(2) is similar to random".
It is wrong.

> > getentropy(2) always behaves as GRND_RANDOM and Random.urandom with getentropy(2) will require userspace CSPRNG.
>
> What makes you think OpenBSD's getentropy blocks?
>
> https://man.openbsd.org/getentropy.2: no mention of blocking, and the implementation:
>
> https://github.com/openbsd/src/blob/b66614995ab119f75167daaa7755b34001836821/sys/dev/rnd.c#L916-L933
>
> Literally just a kernel-space arc4random_buf call.

Indeed.
I misunderstood about that.
I finally agree that

- Use getentropy with loop on OpenBSD
- Use sysctl KERN_ARND on FreeBSD 11 or prior

I also note that FreeBSD 12 will provide getrandom(3) like Linux:
https://www.freebsd.org/cgi/man.cgi?query=getrandom&sektion=2&apropos=0&manpath=FreeBSD+12-current

#20 - 08/31/2018 06:56 PM - jeremyevans0 (Jeremy Evans)

naruse (Yui NARUSE) wrote:

> I finally agree that
>
> - Use getentropy with loop on OpenBSD

As the maintainer of the OpenBSD ruby port, I'm requesting that ruby continue to use arc4random_buf(3) on OpenBSD. The OpenBSD getentropy(2) man page states:

```
    getentropy() is not intended for regular code; please use the
    arc4random(3) family of functions instead.
```

#21 - 08/31/2018 10:03 PM - Freaky (Thomas Hurst)

jeremyevans0 (Jeremy Evans) wrote:

> As the maintainer of the OpenBSD ruby port, I'm requesting that ruby continue to use arc4random_buf(3) on OpenBSD. The OpenBSD getentropy(2) man page states:
>
> ```
>     getentropy() is not intended for regular code; please use the
>     arc4random(3) family of functions instead.
> ```

I think there's a reasonable argument for providing access to both. What I'd like is:

1. Weak, fast random with an in-process Mersenne Twister or similar.
2. Strong, fast, crypto random. Generated using an in-process CSPRNG.
3. Strong, slow, seed random. Generated by either a system service or the kernel, with state managed *outside* the Ruby process.

Rust has these with SmallRng for 1, StdRng for 2, and OsRng/EntropyRng for 3.

Python has random for 1, SystemRandom for 3 (backed by os.urandom, which indeed uses OpenBSD's getentropy()).

Ruby has Random for 1, but 2 *or* 3 are provided by SecureRandom/Random.urandom depending on which platform you're on: 2 with arc4random_buf(), 3 with getrandom()/CryptGenRandom//dev/urandom.

#22 - 09/01/2018 11:16 AM - shyouhei (Shyouhei Urabe)

Freaky (Thomas Hurst) wrote:

> jeremyevans0 (Jeremy Evans) wrote:
>
> > As the maintainer of the OpenBSD ruby port, I'm requesting that ruby continue to use arc4random_buf(3) on OpenBSD. The OpenBSD getentropy(2) man page states:
> >
> > ```
> >     getentropy() is not intended for regular code; please use the
> >     arc4random(3) family of functions instead.
> > ```
>
> I think there's a reasonable argument for providing access to both. What I'd like is:

While I'm not against these points, I feel it is not a good idea to wait for such new API. Can we focus on fixing FreeBSD first? Maybe the OpenBSD issue should be handled separately in another thread.

#23 - 09/01/2018 04:19 PM - Freaky (Thomas Hurst)

shyouhei (Shyouhei Urabe) wrote:

> While I'm not against these points, I feel it is not a good idea to wait for such new API. Can we focus on fixing FreeBSD first? Maybe the OpenBSD issue should be handled separately in another thread.

Yeah, I guess for now let's focus on making sure we're only using arc4random_buf() where it's a modern CSPRNG with robust fork detection.

OpenBSD has since 5.6: https://www.openbsd.org/plus56.html "Use MAP_INHERIT_ZERO in arc4random(3), to zero out the RNG state if the process forks."

NetBSD has since 7.0 (ChaCha20, uses MAP_INHERIT_ZERO): https://github.com/NetBSD/src/blob/netbsd-7-0/lib/libc/gen/arc4random.c

FreeBSD has since this commit to 12-CURRENT: https://v4.freshbsd.org/commit/freebsd/src/338061

So something like:

```
#if (defined(__OpenBSD__) && OpenBSD >= 201411) || \
    (defined(__NetBSD__)  && __NetBSD_Version__ >= 700000000) || \
    (defined(__FreeBSD__) && __FreeBSD_version >= 1200079)
```

Earlier versions are likely to be ARC4 and/or have weaker fork detection vulnerable to pid wraparounds.

#24 - 09/03/2018 01:04 AM - shyouhei (Shyouhei Urabe)

naruse (Yui NARUSE) Are you willing to provide a fix for this? Or can I commit the proposed #if guard?

#25 - 09/03/2018 03:29 PM - naruse (Yui NARUSE)

shyouhei (Shyouhei Urabe) wrote:

> naruse (Yui NARUSE) Are you willing to provide a fix for this? Or can I commit the proposed #if guard?

Could you commit it?

#26 - 09/04/2018 01:42 AM - shyouhei (Shyouhei Urabe)

- Status changed from Open to Closed

Applied in changeset trunk|r64625.

---

avoid fork-unsafe arc4random implementations

Some old implementaions of arc4random_buf(3) were ARC4 based, or
unsafe when forked, or both. Resort to /dev/urandom for those
known problematic cases. Fix [Bug #15039]

Patch from Thomas Hurst tom@hur.st