Beyond `/dev/urandom`: The State of Randomness in Linux

Shea Polansky

2020-08-27

Background: Random Number Generation

- Computers are deterministic machines
  - Deterministic ≠ random
- But random numbers are important
  - Obvious: key generation
  - But also: a large portion of crypto schemes are *completely broken* in the absence of *unpredictable* random numbers
- How do we resolve this quandary?

Types of Random Number Generators

Pseudorandom Number Generators (PRNGs)

- Can be very fast
- Usually "seeded" with a single value
- Can have very long periods (time until the values repeat)
- *Not* designed to be unpredictable
  - No guarantee that someone who sees N values can't predict value N+1

Cryptographically Secure PRNGs (CSPRNGs)

- Usually not that fast
- "Seeded" with as much "entropy" as possible
  - Time, I/O events, process info, hardware, network data, etc.
- May have shorter periods
- Most importantly: *designed to be unpredictable*

Random Number Generation in Practice

- Like most things, you aren’t doing this yourself
- Default RNG: `math.random()` (or equivalent)
  - Basic PRNG — fast, medium period, seeded with the time
  - Provided by LibC / language standard library
  - *Not sufficient for cryptographic purposes*
- Standard libraries do not *typically* provide their own CSPRNG
  - CSPRNGs require diverse entropy sources — standard libraries in userland can’t provide
  - Also dangerous to get wrong, so push the responsibility elsewhere

Random Number Generation in Practice — CSPRNGs

- If your standard library doesn’t give you a CSPRNG, where do you get one?
    - The OS! Specifically, the kernel.
- Linux originally provided a single source of randomness: `/dev/random`
    - Seeded by the kernel automatically by a variety of sources
    - Includes an internal entropy estimator
    - Won’t provide numbers if the entropy estimate is too low (blocks)

Aside: Sources of Entropy

- So why do we need multiple sources of entropy anyway?
- Computers are deterministic — if an attacker can provide the same inputs to the same code, they get the same outputs
  - Code is open source (plus Kerckhoff's Principle)
  - Therefore we must make the inputs difficult to predict
- Sources of entropy are usually things that only the kernel can see
  - Idea is that even a local attacker can't predict, but if they can read kernel memory then they don't need to predict RNG output
  - I/O ops, network traffic, time, etc.

Sources of Entropy — Hardware RNGs

Newer CPUs have additional source of entropy: hardware RNG

x86: RDRAND / RDSEED instructions fill memory with random bytes; ARM has equivalent

Randomness is provided via hardware

There are lots of ways to do this… but the actual implementation is only known to CPU manufacturer

Claims to be “truly random” numbers

So why not just use that and skip the effort of making a CSPRNG?

Why not skip the CSPRNG and use a hardware RNG if one is available?

1

Speed

- Hardware RNG can provide bytes only so fast
- CSPRNGs are as fast as the general CPU

2

Verifiability

- We don’t know how the hardware RNG works
- This is an obvious target to backdoor
- If we relied solely on it and it *was* backdoored, we’d be f*cked

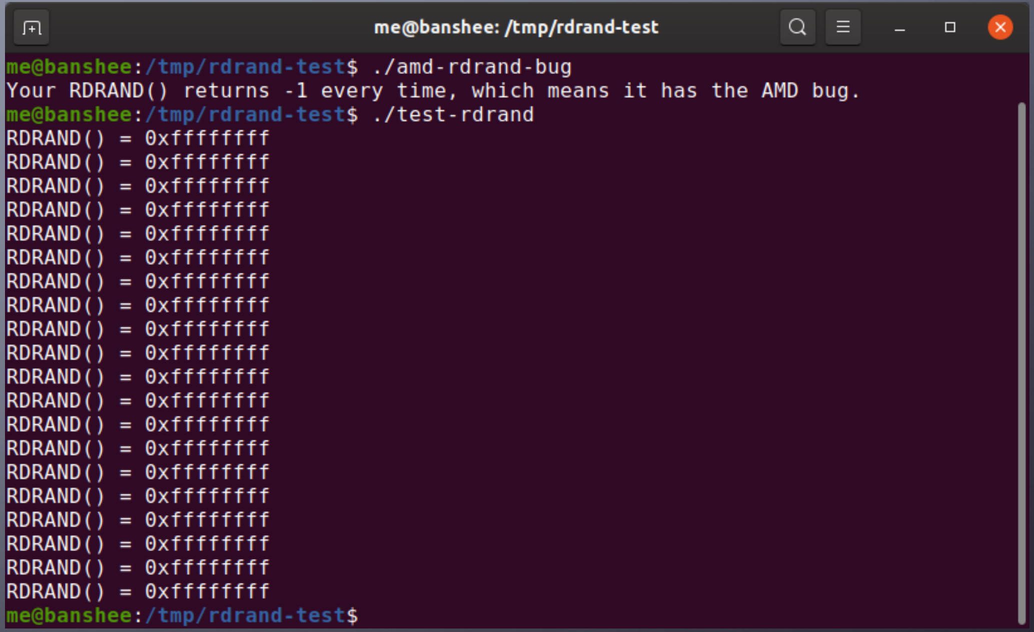
3

Single point of failure

- If there's a problem with the hardware RNG, also f*cked.

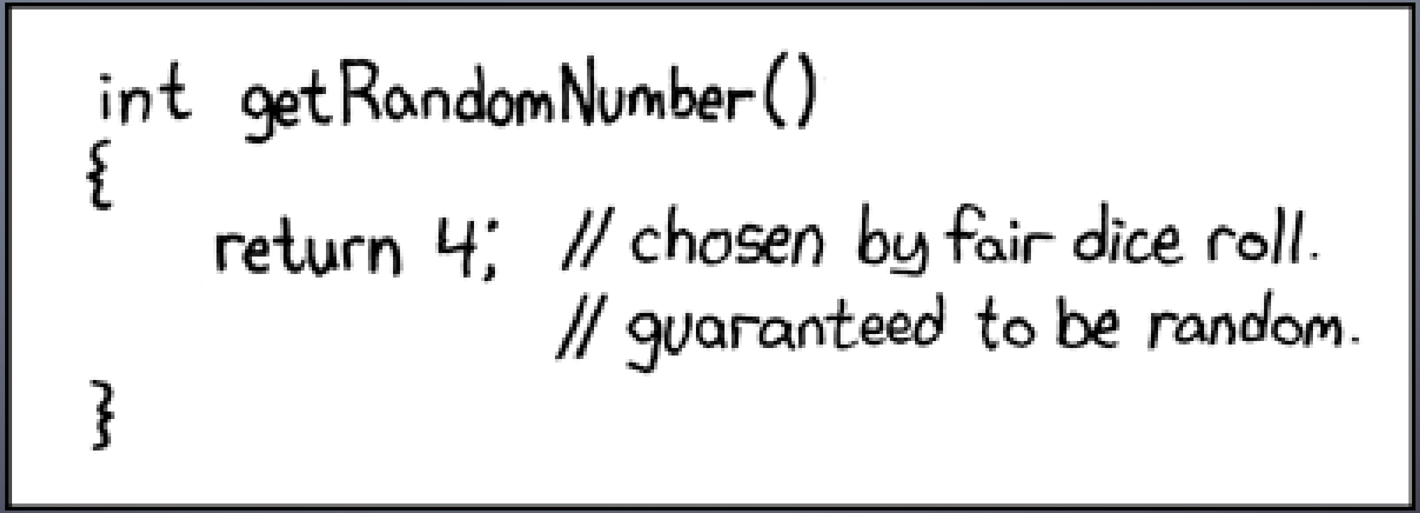
AMD RDRAND Bug

- AMD CPUs have a history of bugs in their RNG
- Most prominently, Zen 2 µArch
  - RDRAND/RDSEED *return a buffer of all 1’s*
- Pre-Ryzen architectures have other issues after suspend/resume
  - Fixed in Zen 2, but literally turned off RDRAND support bit in older CPUs



```
me@banshee:/tmp/rdrand-test$ ./amd-rdrand-bug
Your RDRAND() returns -1 every time, which means it has the AMD bug.
me@banshee:/tmp/rdrand-test$ ./test-rdrand
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
RDRAND() = 0xffffffff
me@banshee:/tmp/rdrand-test$
```

Relevant XKCD



```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

https://xkcd.com/221/

So anyway, kernel CSPRNGs

- Linux originally provided just `/dev/random`
  - “Blocks” if internal entropy estimate is low
  - Designed this way in case of a theoretical attack on entropy stretching algorithms
- In the meantime, also created `/dev/urandom` (“unlimited”/“unblocking”)
  - Uses entropy stretching algorithm to provide unlimited output
  - Original advice was to use `/dev/random` for extremely critical ops (e.g., master key generation), `/dev/urandom` otherwise.

The Great Random Debate

- This led to a years-long ~~internet flame war~~ argument about when if at all one should use `/dev/{u,}random`
- So when should you use them?
- The old answer: Always use `/dev/urandom`.
  - Sole exception: if you're PID 0, since the random pool may not be initialized yet and it won't tell you.

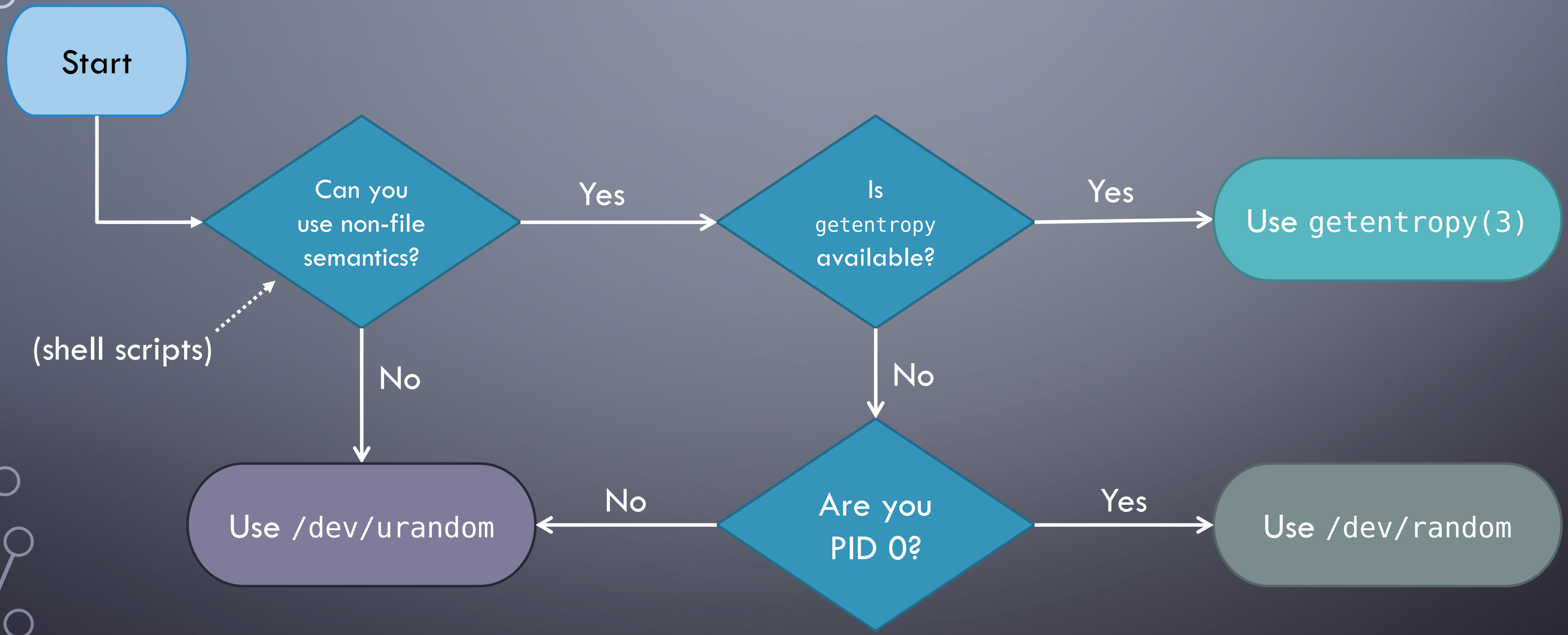
Problems With `/dev/{u,}random`

- `/dev/*` is not always available
  - Containers
- If you *are* in a situation where you might have an uninitialized entropy pool, no way to know that except by polling `/dev/random` before `/dev/urandom`
- Solution: `getrandom(2)`

CSPRNG System Calls — `getrandom(2)` / `getentropy(3)`

- Added ≈2014
- Added to GlibC in 2017 due to backward compatibility issues (what to do if the system calls aren’t available)
- `getrandom(2)` returns *up to* a programmer-requested number of random bytes
  - Uses the unblocking random source by default
  - May return less if there is insufficient entropy (e.g., random pool not initialized)
  - May also be interrupted (e.g., by signals)
- `getentropy(3)` (wrapper around syscall) returns *exactly* specified amount of bytes or none
  - Solves the problem of knowing when the pool is initialized, just check if successful

So what should you use now?



Start

Can you use non-file semantics?

(shell scripts)

Yes

Is `getentropy` available?

Yes

Use `getentropy(3)`

No

Use `/dev/urandom`

No

Are you PID 0?

No

Yes

Use `/dev/random`

(for low-level programming in C or similar languages — use a higher-level wrapper/implementation if appropriate)

“Future” Changes

- So what about `/dev/random`? What’s the point of having something you’re not supposed to ever use?
- In BSD-land, `/dev/random` is actually a symbolic link to `/dev/urandom` now
- And as of Kernel 5.6 (March 2020, but not filtered down to all distros yet), `/dev/random` behaves as `/dev/urandom` after the pool is initialized!
  - The debate is over! And you can stop asking that question as a gotcha in job interviews :)
- However, since you can’t assume that people are running newer kernels, the previous flowchart is unchanged.

Quick Note: CSPRNGs on Other OSs

- Windows: use `BCryptGenRandom` (part of CryptoAPI: Next Generation) or `rand_s` for native code and `RNGCryptoServiceProvider` in .NET
  - Both call into the same system managed CSPRNG
- iOS: Use `SecRandom`
- MacOS: Use `SecRandom` (preferred) or `/dev/urandom`
- Android (and any JVM): use `java.security.SecureRandom`
- JavaScript: `crypto.randomBytes` (NodeJS) / `Crypto.getRandomValues` (Browser)
- PHP: `random_bytes` / `random_int`
- Python: `secrets` module

Quick Takeaways

- Generating cryptographic-quality random numbers is hard
- If you're not writing an init system, don't use `/dev/random`
- Use `getentropy(3)` if you can, `/dev/urandom` if you can't
- The difference between `/dev/random` and `/dev/urandom` is going to slowly disappear, but stick with the old advice for backwards compatibility

Questions?

Thank you.

Further Reading

- https://man7.org/linux/man-pages/man2/getrandom.2.html
- https://man7.org/linux/man-pages/man3/getentropy.3.html
- https://en.wikipedia.org/wiki/Entropy-supplying_system_calls
- Other OS's:
  - Windows: https://docs.microsoft.com/en-us/windows/win32/api/bcrypt/nf-bcrypt-bcryptgenrandom and https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/rand-s?view=vs-2019 and https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rngcryptoserviceprovider?redirectedfrom=MSDN&view=netcore-3.1
  - MacOS/iOS: https://developer.apple.com/documentation/security/randomization_services
  - Android: https://developer.android.com/reference/java/security/SecureRandom
  - JVM: https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html
  - JavaScript: https://nodejs.org/api/crypto.html#crypto_crypto_randombytes_size_callback (Node) / https://developer.mozilla.org/en-US/docs/Web/API/Crypto/getRandomValues (Browser)
  - PHP: https://www.php.net/manual/en/ref.csprng.php
  - Python: https://docs.python.org/3/library/secrets.html