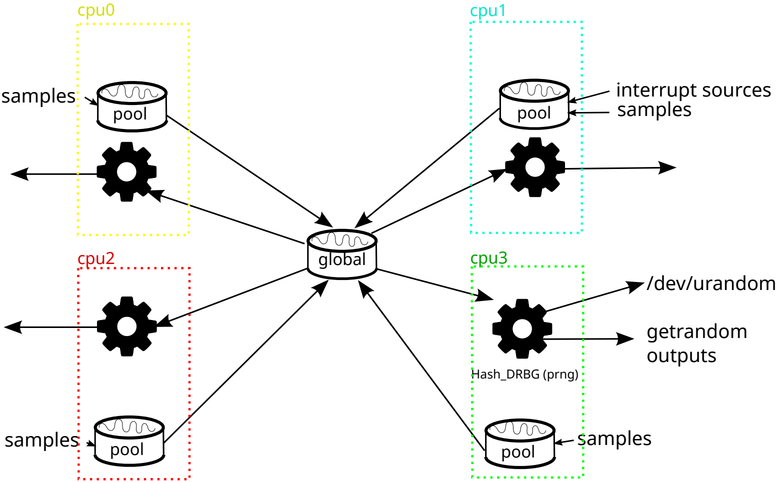
The New NetBSD Entropy Subsystem

Taylor ‘Riastradh’ Campbell
campbell@mumble.net
riastradh@NetBSD.org

EuroBSDcon 2021
nowhere and everywhere (it’s a global pandemic)
September 20, 2021

NetBSD entropy pool data flow



Computers need unpredictable secrets

- HTTPS, SSH, etc., need long-term secret keys to prevent impersonation of servers and clients.
- HTTPS, SSH, etc., need short-term secret keys to prevent forgery and eavesdropping in sessions.
- Operating systems need ephemeral secrets to swap volatile secrets onto nonvolatile media without exposing them to future theft.

What does ‘unpredictable’ mean?

- Adversary wants to impersonate, forge, eavesdrop, etc., by guessing secrets.
- Adversary has incomplete information—a state of knowledge.
- Adversary knows *process* used to choose secrets (and protocol—HTTPS, SSH, etc.), but not the secrets themselves.

Quantifying unpredictability

- Language of probability theory.
- A probability distribution represents a state of knowledge about an unknown process outcome by assigning a weight to every possible outcome.
- Example: Fair coin toss C, possible outcomes are ‘heads’ or ‘tails’.
    - $\Pr[C = \text{heads}] = 1/2$
    - $\Pr[C = \text{tails}] = 1/2$

Quantifying unpredictability

- Example: Sum S of two die rolls, possible outcomes are 2 through 12.
  - $\Pr[S = 2] = 1/36$
  - $\Pr[S = 3] = 2/36 = 1/18$
  - $\Pr[S = 4] = 3/36 = 1/12$
  - $\vdots$
  - $\Pr[S = 12] = 1/36$

Quantifying unpredictability

- Adversary wins prize if they guess the secret (and then impersonate, forge, eavesdrop, etc.).
- What's adversary's probability of success for best strategy?
- Example: Fair coin toss: $1/2$, doesn't make a difference if adversary's strategy is to guess heads or guess tails.
- Example: Sum of two die rolls: $1/6$, if they guess 7; all other outcomes have lower probability.

Quantifying unpredictability

- **Entropy** is a numeric summary of a probability distribution, or of a process whose outcomes follow a probability distribution.
  - *Not* a property of any particular value like ‘`hunter2`’ or ‘`correct horse battery staple`’!
- Many kinds of entropy (Shannon, Hartley, Renyí, min) but mainly one relevant to cryptography: min-entropy.
- **Min-entropy** of a probability distribution is the negative log of the adversary’s best chance of success at guessing the secret, i.e., the negative log of the probability of the most probable outcome:

$$H_\infty(P) = -\log \max_x P(x).$$

- (All logarithms in base 2, in units of bits.)

Quantifying unpredictability

- Min-entropy of fair coin toss: 1 bit.
- Min-entropy of die roll: $\log_2 1/6 = {\sim}2.5$ bits.
- Min-entropy of sum of two die rolls: $\log_2 1/6 = {\sim}2.5$ bits.
    - Same as one die roll even though there are almost twice as many possible outcomes!

Computers and unpredictability

- Computers are usually very predictable.
  - (Software engineers in the audience furiously debugging bugs that are obviously impossible situations may dispute this.)
- But we need to maximize unpredictability for secrets!

Computers and unpredictability

- Need device drivers to make observations of unpredictable physical phenomena unknown to adversaries.
- Example: driver for device with Geiger–Müller tube pointed at an alpha emitter to count ionizing events.
- Example: driver for bored human flipping coins and entering outcomes.

Computers and unpredictability

More realistic examples: jitter between clocks.

- Common example: Ring oscillator—two circuits on a die clocked independently, one flipping bits in a loop and the other sampling the first.
    - Most devices advertised as HWRNGs on systems-on-a-chip are built out of ring oscillatrs.
- Half-example: Interrupt timings—hardware peripherals ‘sampling’ CPU cycle counter.
    - Difficult to *confidently assess* entropy of distribution.
    - Can adversary control network packet timings?
    - Can adversary guess keystroke timings?
- Non-example: Periodic timer interrupt *driven by the same clock as* the CPU cycle counter.
    - Zero entropy—deterministic relation between clocks, no jitter!

Uniformity

- Physical systems tend to have very nonuniform distributions: the possible outcomes have different probabilities.
  - Geiger counts are Poisson distributed (or, durations between are exponentially distributed).
  - Consecutive samples of ring oscillators are not independent.
  - Samples of multiple ring oscillators in parallel, with related clocks, are not independent.
  - Even honest coin tosses have small biases!
- Cryptography tends to want uniform distributions.
  - Modern cryptography can turn a short 256-bit seed with uniform distribution into an essentially arbitrarily long stream of output that appears just as uniform—adversary has no hope of telling it apart from uniform.
  - (Note: No cryptographic justification for ‘entropy depletion’—256 secret bits is enough, period. But it can be useful for testing.)

What an operating system does, roughly

So an operating system (on an otherwise essentially deterministic computer) needs to hash *enough* samples from physical systems together into uniformly distributed seeds for cryptography, to produce output from `/dev/urandom` or similar.

Iterative-guessing attacks

- Suppose physical samples come in: $s_1, s_2, s_3, \ldots$
- Each sample is from a process with *low* min-entropy, say 32 bits.
- Suppose an application immediately tries to do cryptography with what we have so far—e.g., generates a Diffie–Hellman secret for an HTTPS query, and exposes the public key on the internet.
- Software repeatedly does this for many HTTPS queries, thereby exposing some functions $f_1 = H(s_1)$, $f_2 = H(s_1, s_2)$, $f_3 = H(s_1, s_2, s_3)$, $\ldots$, of the unpredictable physical samples.
    - (Here, H produces `/dev/urandom` output, generates a DH key pair from it, and returns the public part; the details aren't important here—but are known to the adversary.)

Iterative-guessing attacks

- Recall the min-entropy of the process producing s_1 had was only 32 bits.
- So, the adversary can probably perform a *feasible* brute-force search (cost around 2^{32}) to recover s_1, using knowledge of $f_1 = H(s_1)$ to confirm a guess.
- Then, knowing what s_1 was but not s_2, the adversary can do a brute-force search to recover s_2, using knowledge of $f_2 = H(s_1, s_2)$ to confirm a guess.
- Lather, rinse, repeat, and the adversary can forge or eavesdrop on the whole session indefinitely this way—the new samples don't help if the adversary can catch up incrementally.

Iterative-guessing attacks

So an operating system should avoid exposing samples piecemeal—it needs to group them into batches with *enough* aggregate entropy from all the sources that a brute-force search is totally infeasible.

Performance issues in sampling

- Want to gather as many samples as possible to get lots of entropy.
- But incorporating samples costs computation and has some latency.
- So we gather samples into per-CPU pools—no interprocessor communication to take a sample, *except* early at boot if we've definitely not yet had 256 bits of entropy so far.
- And during interrupts we store samples in a per-CPU buffer to be processed, and just drop additional samples if the buffer is fill, to avoid high interrupt latency.

Performance issues in /dev/urandom and (re)seeding

- `/dev/urandom` output is drawn from per-CPU PRNG state for scalability
- Don't want every batch of samples to trigger cross-call activity if nobody's actually using each PRNG
- Global entropy epoch counter enables lazy-reseed in chains of PRNGs (like Windows does now, according to their whitepaper!)

What to do if there’s not enough entropy and you need a key?

- For machines with on-board HWRNGs (x86 RDRAND/RDSEED, ARMv8.5-RNG RNDRRS, many newer SoCs): not a concern.
- If the operator has stored a seed on disk, NetBSD automatically updates it on boot, on shutdown, and daily.
- For other machines, well. . .

What to do if there's not enough entropy and you need a key?

- If no HWRNG and no seed, traditional answer is: block key generation!

```
$ gpg --gen-key
...
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
```

- Very annoying on servers! (Even more annoying when 'entropy depletion' is still in play.)

What to do if there's not enough entropy and you need a key?

- But does blocking at the moment of key generation *ever* make sense?
- Historically, it did, under the premise that the OS would essentially just make up an idea of the entropy of the underlying process by examining consecutive differences of samples!
- But NetBSD (and FIPS these days!) asks that any estimate be based on knowledge of how the device works, so it is necessarily driver-specific.
- Can't guarantee nonzero entropy—and thus an end to blocking—this way; e.g., timer interrupt clocked by the same clock as CPU cycle counter has zero entropy!.
- Network appliances might seem like bricks if ssh-keygen blocks first startup this way—serious usability issues invite security-destroying workarounds.

What to do if there's not enough entropy and you need a key?

- Experience with blocking `getrandom` system call in NetBSD, along with meaningful entropy estimates, has been negative—causes weird hangs in places that make no sense and gives no useful feedback.
- So we try to get the message out other ways:
  - offer option in installer to furnish seed
  - warn operator in daily security report if not enough entropy
  - one-liner in motd with reference to `https://man.NetBSD.org/entropy.7` man page

  But we need to be careful to avoid warning fatigue!
- Might remove failed `getrandom` experiment (only in HEAD so far) and instead adopt never-blocks `getentropy` from OpenBSD and like POSIX is likely to adopt soon. (Discussion ongoing.)

Cryptographic choices

- Entropy pool: Keccak-p[1600,24] sponge duplex[1]
    - 200-byte state
    - `feed(s)` enters a sample into the state
    - `fetch(L)` returns an *L*-byte string from the state affected by all inputs and erases part of the state so it can't be recovered again
    - No entropy loss from samples (unlike, e.g., naive hashing with SHA-256): any sample can be recovered from knowledge of state, all other samples, and all outputs
    - Security closely related to security of SHA-3

---

[1] Guido Bertoni, Joan daemen, Michaël Peeters, and Gilles Van Assche, 'Sponge-Based Pseudo-Random Number Generators', in Stefan Mangard and François-Xavier Standaert, eds., Cryptographic Hardware and Embedded Systems—CHES 2010, Springer LNCS 6225, pp. 33–47, `https://link.springer.com/chapter/10.1007/978-3-642-15031-9_3`, `https://keccak.team/files/SpongePRNG.pdf`

Cryptographic choices

- `/dev/urandom` pseudorandom number generator: NIST SP 800-90A Hash_DRBG with SHA-256.
  - Of the NIST SP 800-90A constructions, simplest security theorem relative to security of the hash function
  - SHA-256 naturally avoids timing side channels unlike AES
  - Used to use CTR_DRBG with AES-128 until timing attacks published[2]
  - (NetBSD kernel AES code has since been rewritten to eliminate timing side channels and a similar theorem has had a much more difficult proof exhibited for CTR_DRBG, so could go back to that now)

---

[2] Shaanan Cohney, Andrew Kwong, Shachar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom, 'Pseudorandom Black Swans: Cache Attacks on CTR_DRBG', Cryptology ePrint Archive, Report 2019/996, `https://eprint.iacr.org/2019/1996`

Cryptographic choices

Why both Keccak duplex and Hash_DRBG?

- Make it easier to approach FIPSy certificationy stuff (not actually done)—nobody ever got fired for choosing US federal government crypto.
- FIPS is (or was; things may be changing now) less picky about conditioning components than about DRBGs.

Fin

Questions?