The entropic principle

/dev/u?random and NetBSD

Taylor 'Riastradh' Campbell
campbell@mumble.net
riastradh@NetBSD.org

EuroBSDcon 2014
Sofia, Bulgaria
September 28, 2014

Use `/dev/urandom`.

What is /dev/u?random?

- `/dev/random` and `/dev/urandom` are device files on all modern Unix systems.
- Can't predict what you will read from them.
- (Writes influence future reads.)
- Reading from `/dev/random` sometimes blocks.

Much cargo-cult voodoo surrounds `/dev/u?random`.

Unpredictability matters

- There are bad guys on the internet.
- They want to eavesdrop on our conversations.
- They want to intercept our conversations.

We need crypto to defend against this, and crypto needs unpredictable secrets.

Unpredictability matters

What happens if you use crypto with predictable secrets?

- Smart cards generated keys for Taiwan's national identity database with broken RNGs leading to repeated and factorable keys: `http://smartfacts.cr.yp.to/`
- Sony used ECDSA with a broken RNG to sign Playstation firmware updates, revealing the signing key.
- Millions of embedded devices on the internet have private RSA keys with factors in common generated from the same RNG states: `https://factorable.net/` (Mining Your P's and Q's).
- The NSA chose to backdoor the US's pseudorandom number generation standard, NIST SP800-90A, with Dual_EC_DRBG.

Security modelling

- APPLICATION: Generate bits with uniform distribution for user programs.
- THREAT MODEL(S):
  - Attacker may read other bits from `/dev/u?random`.
  - Attacker may influence `/dev/u?random`.
  - Attacker may compromise the kernel and get the internal state of `/dev/u?random`.
- SECURITY PROPERTY: Attacker must not predict any outputs not witnessed!

(Can't attain the security property against attacker who compromises the kernel, but some people worry about theoretically approximating it for some reason.)

Formalizing unpredictability

- Random variable X: observable physical system which can take on possible values $x_0, x_1, \ldots, x_{n-1}$.
- Probability that we observe X to have value x: $\Pr[X = x]$.
- Observation of X may not be predictable, but how do we formalize measuring how unpredictable?

Formalizing unpredictability: Shannon entropy

- Popular approach to measure unpredictability of X is *Shannon entropy*, in units of bits:

$$H[X] = -\sum_i \Pr[X = x_i] \log_2 \Pr[X = x_i],$$

  giving average bits of information per bit of observation of X.
- If X is drawn from $\{00, 01, 10, 11\}$, and

$$\Pr[X = 00] = \Pr[X = 11] = 1/2,$$
$$\Pr[X = 01] = \Pr[X = 10] = 0,$$

  then $H[X] = 1$, so since an observation of X has two bits, X has half a bit of information per bit of observation.
- Guessing this is only as hard as guessing what a single coin flip was, not two coin flips in a row.

Formalizing unpredictability: not Shannon entropy

- Shannon entropy is no good for crypto: if there are $2^{255} + 1$ possibilities for X, and

$$\Pr[X = x] = \begin{cases} 1/2, & \text{if } x = \text{`hunter2'}; \\ 1/2^{256} & \text{otherwise}, \end{cases} \tag{1}$$

  then $H[X] \approx 128$, but I don't have to try 2^{127} possibilities before I can probably guess your password — I have a pretty good guess what it is!

Formalizing unpredictability: min-entropy

- Crypto instead uses *min-entropy*:

$$H_\infty[X] = -\max_i \log_2 \Pr[X = x_i]. \tag{2}$$

- Min-entropy estimates the difficulty of the *best strategy* at guessing X.
- If X is drawn uniformly from k-bit strings, then $H_\infty[X] = k$, which is as good as you can get!
- Standard crypto practice is to use uniform distributions with $k = 128$ or $k = 256$ for key material.

/dev/u?random as random variable

- Kernel's job is to make the random variable of reading k bits from /dev/u?random have k bits of entropy.
- How does it choose the bits?
- Computer programs (single-threaded) are supposed to be deterministic!

Entropy sources

- Computers make nondeterministic observations: clock skew, network packets, keyboard input, disk seek times.

```
% gpg --gen-key
...
Please bang on the keyboard like a monkey
to ensure there's enough entropy!
```

- These *entropy sources* are random variables with highly nonuniform distribution.
- Attacker may influence them: send regular network packets, bang on the keyboard like a robot, &c.

Entropy pooling and distribution

- Kernel combines entropy sources with crypto magic called entropy extractors.
- Kernel uses output as seed for deterministic pseudorandom number generator when you read from `/dev/u?random`.

What if there's not much entropy?

- Your system autogenerated `sshd` keys from `/dev/u?random` before you've had a chance to bang on the keyboard like a monkey!
- Keys are predictable! Bad guys can guess them and log in! Even Debian is laughing at you! How do we prevent this?

What if there's not much entropy?

- Naïve answer: wait until system is unpredictable enough.
- `/dev/random` traditionally estimates how much entropy the system has observed, and blocks until it reaches a threshold.
- Use it as an 'unpredictability barrier': read once from `/dev/random` and once it unblocks, use `/dev/urandom` to generate keys.
- `/dev/urandom` never blocks: whatever has been fed into the entropy pool, the kernel uses it to seed a pseudorandom number generator and generate output.

What if there's not much entropy?

Problem: Can't say whether a *state* is unpredictable. Can only say whether a *process* is unpredictable.
(Obligatory Dilbert reference.)

`http://dilbert.com/strips/comic/2001-10-25/`

(Dilbert meets the accounting department's random number generator, a troll who only says 'nine' over and over again. Scott Adams won't let me include his strip verbatim for this noncommercial use without paying him money.)

Estimating this is a hard problem! No good solution. Typical approaches are *ad hoc*.

Running out of entropy?

- `/dev/random` *also* traditionally blocks sometimes long after boot.
- Original theory was if you read too much from it you run out of entropy, so you'd better wait for more.
- But entropy is *not a scarce resource* like oil.
- Entropy is a *property of a physical process*.
- So why block after boot?

Topping off the entropy tank

- `/dev/random` blocking after boot is still useful!
- If you use it as an unpredictability barrier, you keep your blocking code paths exercised.
- If you use it to generate keys always, you will notice when your application blocks — instead of being told two years from now that it doesn't work on an embedded system you never even heard of because it doesn't have enough entropy at boot!
- But most applications just need `/dev/urandom`.

What if there is *no* entropy?

- What if there are no entropy sources?
- No disk, no mouse, no keyboard, no monkey.
- Kernel is totally deterministic: can't be unpredictable.
- Can't usefully serve `/dev/u?random`.
- Example: embedded appliances (Mining P's & Q's).
- Solution: save entropy from the factory installer onto small (32-byte) nonvolatile storage on install and shutdown, restore on boot.
- This system engineering avoids need for `/dev/random` as unpredictability barrier.

Exotic threat models

- Attacker can influence network packet timings? (Easy.)
- Attacker can influence keytrokes and timings?
- Attacker can compromise your CPU? (Paranoid view of Intel RDRAND!)

Good entropy extractors thwart manipulation of one entropy source or another.

Hardware random number generators

- PCI devices: HIFN 7751, Broadcom BCM58xx.
- SoC on-board devices: Broadcom BCM2385 (Raspberry Pi).
- CPU instructions: Intel RDRAND, VIA PadLock.

Hardware random number generators

- . . . The coin in your trouser pocket:

```
% echo hhhttttthhththttthhht... >> /dev/random
```

NetBSD

Current code written mainly by Thor Lancelot Simon and me.

- `/dev/u?random` uses per-open or per-CPU pseudorandom number generator state, so it scales.
- Kernel uses slow NIST CTR_DRBG with AES-128 for key material and `/dev/u?random`: attacker must never predict unseen outputs.
- Kernel uses fast ChaCha8 without backtracking resistance for non-key material, e.g. NFS transaction ids: attacker must not predict new outputs ahead of time, but may predict old ones.
- Userland `arc4random(3)` API soon to be reimplemented with per-thread state and ChaCha8 instead of global RC4.
- (Let me know if you've heard of `arc4random(3)` being used for key material! I wouldn't recommend it!)

Questions?

(Use /dev/urandom!)

Appendix: Entropy game!

`http://www.loper-os.org/bad-at-entropy/manmach.html`

My password manager:

```
{ tr -cd '[:graph:]' < /dev/urandom | head -c 20; } | \
scrypt enc /dev/stdin password.scrypt
```