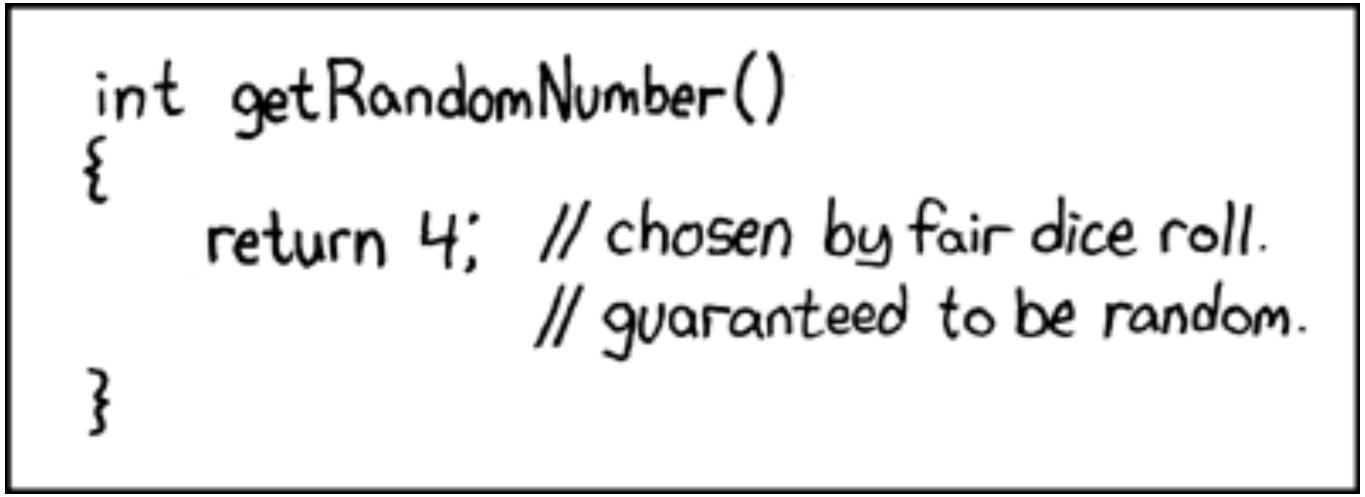
A Deep Dive into FreeBSD's Kernel RNG

W. DEAN FREEMAN AND JOHN-MARK GURNEY

Who We Are

W. Dean Freeman, CISSP CSSLP GCIH
- Sr. Test Engineer @ NSS Labs

John-Mark Gurney
- Principal Security Architect @ New Context
- Twitter: @encthenet


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

xkcd

Risks of a bad RNG

- Real world issue
  - Digital Signature Scheme's (DSS) nonce must be unique (PS3 signing key leak)
  - Debian SSH Key Issue (2006-2008): Everything using OpenSSL was broken
  - Dual_EC Deterministic Random Bit Generator (DRBG) back doored (?): RSA BSafe and Juniper
  - RSA Weak Public Keys available on the Internet paper
- Algorithm requirements
  - Nonce must be unique (AES-CTR and AES-GCM leaks message difference)
  - RSA padding must be random (RSA-PSS recommended)

Background

- Dean approached from the point of view of entropy assessment for Common Criteria and FIPS 140-2 appliances and as part of the NIAP technical community
  - How good is the entropy source seeding FreeBSD in a general purpose situation?
  - What changes would be needed to be compliant and certifiable?
- John-Mark had previously looked at FreeBSD's RNG code for improvement

RNG Overview

- TRNG - True Random Number Generator
  - Often very slow
  - Uses environmental artifacts to generate randomness
    - Reverse biased diode
    - Meta-stable state of transistors
    - Thermal Noise (ADC, etc)
    - Lava Lamps
- Pseudo Random Number Generator - PRNG
  - Uses a seed
  - Is able to generate a large amount of random data

Install-time seeding

- In the begging was the first install
- bsdinstall populates disk files with output from /dev/random, creating
  - /entropy
  - /boot/entropy
- usr.sbin/bsdinstall/script/entropy handles this function
  - Script is called for auto, jail and script installations

Boot-time loading

- Early loading of entropy added starting in 10.0-R
- Provides seeding of DRBG before file systems are loaded
  - Loaded from file on boot device (Default /boot/entropy)
  - After mixing, original seed is zeroed out in memory
- File on disk is overwritten with output from /dev/random after read
  - On UFS file systems, the blocks are overwritten but artifacts may remain, depending on the properties of the underlying disk device
  - ZFS is copy-on-write (COW), so the file is never really destroyed
  - Clones and snapshots may cause copies to persist indefinitely

RC Time Loading

- The random rc script handles seeding as well as setting the entropy source mask
- Mask values control which entropy sources are leveraged at runtime
- Writes out new entropy file when shutting down

Runtime Entropy Collection - Sources

- FreeBSD has a pluggable framework for both PRNG implementations and entropy sources
  - Environmental and platform-provided sources supported in GENERIC
  - Full list of entropy sources can be found in *usr/src/sys/random.h* and include:

CACHED
ATTACH
KEYBOARD
MOUSE
NET_TUN
NET_ETHER
NET_NG
INTERRUPT
SWI
FS_ATIME
UMA

PURE_OCTEON
PURE_SAFE
PURE_GLXSB
PURE_UBSEC
PURE_HIFN
PURE_RDRND
PURE_NEHMEIAH
PURE_RNDTEST
PURE_BROADCOM

Runtime Entropy Collection - Methods

- Three main methods for seeding the DRBG
  - random_harvest_direct()
    - Used by RANDOM_ATTACH when new hardware is attached
    - Used to collect from registered, “pure,” entropy sources, such as RDRND
  - random_harvest_fast()
    - Only used if the kernel is built with RANDOM_ENABLE_UMA
  - random_harvest_queue()
    - Everything else goes in via random_harvest_queue()
      - Currently, this includes RANDOM_PURE_BROADCOM and RANDOM_PURE_OCTEON, which should probably go through random_harvest_direct()

Mixing and Feeding

YARROW, FORTUNA AND ARC4RANDOM

Yarrow

- FreeBSD’s PRNG before 11.0-R was based on Yarrow
- Designed by Bruce Schneier, John Kelsey and Niels Ferguson
  - Fast and slow accumulator pools
  - Entropy is collected and then initially whitened with SHA-256
  - When a request for random bytes is made, CTR-mode AES further whitens as the pools are drained

Fortuna

- Default DRBG implementation in FreeBSD since 11.0-R
- Designed by Bruce Schneier, Niels Ferguson and Tadayoshi Kohno
  - Designed to withstand concerted cryptanalytic attack
  - Successor to the Yarrow algorithm
- Features 32 entropy accumulator pools
  - Raw entropy is collected and distributed over the pools
  - Uses SHA-256 to effectively create an infinitely long string of entropy
- When random bytes are requested, selected pools are drained, such that later pools are used less frequently
  - On drain, the bytes in the pool are fed through a CTR-MODE AES implementation

arc4random

- Developed by OpenBSD and import in 1999
- Originally contained an rc4 implementation (hence the name), but HEAD now uses ChaCha20
  - ChaCha20 leverages 256-bit keys and provides AES-like strength with the benefit of greater speed on hardware which lacks acceleration for AES
  - Even on hardware w/ AES-NI, FPU restrictions would likely prevent it's use
- The arc4random DRBG is seeded with the output of the mixer

Device Nodes

- Two device nodes provided: /dev/random and /dev/urandom
- On FreeBSD, the latter is a symlink to the former, unlike other implementations (e.g., Linux)
  - Both will block until seeded
  - Combined when Yarrow was added in 2000
- Device can be read from to provide whitened output from the in-use DRBG (i.e., Fortuna)
- Device can be written to
  - Anything written from userland is whitened the same way as any system entropy collected by the kernel
  - This is how the random script updates the seed with the stored entropy files

Entropy Analysis

Evaluating Entropy - Overview

- An Entropy Assessment Review (EAR) is required as a first step for Common Criteria evaluations
  - Reviews done by the Information Assurance Directorate (IAD) at the National Security Agency
  - Sufficient initial seed values for the entropy device are required to be accepted for evaluation and approval for government use
- NIST SP800-90B, “Recommendation for the Entropy Sources Used for Random Bit Generation (Second Draft)”
  - Published December 2016
  - Provides guidance for assessing strength of the entropy used to seed a DRBG
- Two general tracks for assessing entropy
  - Independent, identically-distributed (IID)
  - Non-IID
- FreeBSD’s entropy sources were evaluated as non-IID
  - An appliance vendor with a custom hardware entropy source may qualify for the IID track, but in a GP OS on commodity hardware this is not the case

Non-IID Track Estimation - Tests

- SP800-90B provides for a battery of statistical tests for estimating min-entropy value for non-IID sources
  - Most Common Value Estimate
  - Collision Estimate
  - Markov Estimate
  - Compression Estimate
  - T-Tuple Estimate
  - Longest Repeated Substring (LRS)
  - Multi Most Common in Window Prediction Estimate
  - Lag Prediction Estimate
  - Multi-MMC Prediction
  - LZ78Y Prediction Estimate

Collection of Entropy

- Need to collect non-whitened entropy for evaluation
- Last place all seed data is available prior to any whitening is randomdev_hash_iterate()
- How to collect?
  - Could patch the kernel and provide a way to dump the data
  - Could use DTrace
- For expedience, DTrace was used to collect the data
  - tracemem() used to dump raw bytes
- Patch developed to print entropy so early boot collection could be evaluated
- DTrace output collected to a file then parsed with a Perl script to create a binary file

Evaluation of Entropy

- NIST provides a Python script to evaluate an input file against either IID or the non-IID track
- We are looking at the non-IID track, so noniid_main.py is used
- The worst-case value provided by all analysis formulas is taken as “min-entropy”
  - Min-entropy value is the key number for EARs specifically, as assuming things are as bad as they possibly could be is the most prudent course
- Typically, we want to see a min-entropy between 4-6
  - Less than 4 would require additional scrutiny

Evaluating a Control Sample

```
PS D:\Projects\SP800-90B_EntropyAssessment> python .\noniid_main.py -v .\truerand_8bit.bin 8
reading 1000000 bytes of data
Read in file .\truerand_8bit.bin, 1000000 bytes long.
Dataset: 1000000 8-bit symbols, 256 symbols in alphabet.
Output symbol values: min = 0, max = 255

Running entropic statistic estimates:
- Most Common Value Estimate: p(max) = 0.00428909, min-entropy = 7.86511
- Collision Estimate: p(max) = 0.0127256, min-entropy = 6.29613
- Markov Estimate (map 6 bits): p(max) = 1.13787e-223, min-entropy = 5.78597
- Compression Estimate: p(max) = 0.00870919, min-entropy = 6.84325
- t-Tuple Estimate: p(max) = 0.004124, min-entropy = 7.92174
- LRS Estimate: p(max) = 0.00391357, min-entropy = 7.9973

Running predictor estimates:
Computing MultiMCW Prediction Estimate: 100 percent complete
        Pglobal: 0.003937
        Plocal: 0.002136
MultiMCW Prediction Estimate: p(max) = 0.0039373, min-entropy = 7.98858

Computing Lag Prediction Estimate: 100 percent complete
        Pglobal: 0.004073
        Plocal: 0.002136
Lag Prediction Estimate: p(max) = 0.00407281, min-entropy = 7.93976

Computing MultiMMC Prediction Estimate: 100 percent complete
        Pglobal: 0.004110
        Plocal: 0.002136
MultiMMC Prediction Estimate: p(max) = 0.00410955, min-entropy = 7.92681

Computing LZ78Y Prediction Estimate: 100 percent complete
        Pglobal: 0.004110
        Plocal: 0.002136
LZ78Y Prediction Estimate: p(max) = 0.00410961, min-entropy = 7.92678
-----------------------
min-entropy = 5.78597

Don't forget to run the sanity check on a restart dataset using H_I = 5.78597
```

Evaluating a Sample From FreeBSD

```
PS D:\Projects\SP800-90B_EntropyAssessment> python .\noniid_main.py -v .\xaa 8
reading 1048576 bytes of data
Read in file .\xaa, 1048576 bytes long.
Dataset: 1048576 8-bit symbols, 256 symbols in alphabet.
Output symbol values: min = 0, max = 255

Running entropic statistic estimates:
- Most Common Value Estimate: p(max) = 0.252273, min-entropy = 1.98694
- Collision Estimate: p(max) = 0.266191, min-entropy = 1.90947
- Markov Estimate (map 6 bits): p(max) = 2.09263e-18, min-entropy = 0.458823
- Compression Estimate: p(max) = 0.355327, min-entropy = 1.49278
- t-Tuple Estimate: p(max) = 0.883155, min-entropy = 0.179262
- LRS Estimate: p(max) = 0.755781, min-entropy = 0.403961

Running predictor estimates:
Computing MultiMCW Prediction Estimate: 99 percent complete
        Pglobal: 0.252040
        Plocal: 0.785120
MultiMCW Prediction Estimate: p(max) = 0.78512, min-entropy = 0.349015

Computing Lag Prediction Estimate: 99 percent complete
        Pglobal: 0.418056
        Plocal: 0.732048
Lag Prediction Estimate: p(max) = 0.732048, min-entropy = 0.44999

Computing MultiMMC Prediction Estimate: 99 percent complete
        Pglobal: 0.403212
        Plocal: 0.787947
MultiMMC Prediction Estimate: p(max) = 0.787947, min-entropy = 0.34383

Computing LZ78Y Prediction Estimate: 99 percent complete
        Pglobal: 0.296099
        Plocal: 0.787947
LZ78Y Prediction Estimate: p(max) = 0.787947, min-entropy = 0.34383
-----------------------
min-entropy = 0.179262

Don't forget to run the sanity check on a restart dataset using H_I = 0.179262
```

FreeBSD’s Min-Entropy is a Little Grim

- Several measurement samples taken
  - Both virtual and bare metal
    - Xeon and i7 processors with RDRND, AES-NI etc.
  - Attempted to make boxes as busy as possible
    - Generate network traffic, build ports, etc.
- None got a min-entropy of even 1 bit per byte
- Why?
  - Raw data contains lots of repeat values, null bytes, and predictable values
  - Best source of high-value entropy is RDRND*, but wasn’t mixed in
  - Mixers use SHA-256 hash compression to make this less of an issue
- Linux isn’t really any better
  - Vanilla entropy sources in Linux are rather weak
  - Typically, jitter rng patches, havaged, or rng-tools (or some combination of all three) with additional hardware are needed to get suitable entropy values

Sample Captured Entropy

```
   randomdev_hash_iterate:entry 16
 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
6e 5f b9 6f 7a 6a e1 63 00 00 00 00 04 01 ba 08
4c 8f 8f 70 bf 0f 89 ed 00 00 00 00 04 01 bb 08
36 42 e5 70 5e 7d f4 39 00 00 00 00 04 01 bc 08
ec 51 12 73 d6 32 cf f7 00 00 00 00 04 01 bd 08
9b 79 2b 74 1b 26 6e 65 00 00 00 00 04 01 be 08
6c f0 93 74 f9 0f a9 37 00 00 00 00 04 01 bf 08
55 9e fd 75 8a 5f 2d 93 00 00 00 00 04 01 c0 08
b3 8f 15 76 91 84 dd 0e 00 00 00 00 04 01 c1 08
13 36 97 77 26 16 25 6b 00 00 00 00 04 01 c2 08
69 9c 42 78 d2 cd f8 6f 00 00 00 00 04 01 c3 08
8f 00 ee 78 ad 23 7b 5d 00 00 00 00 04 01 c4 08
c3 e7 18 79 9b f0 66 aa 00 00 00 00 04 01 c5 08
58 85 9a 7a 8f 60 91 af 00 00 00 00 04 01 c6 08
08 26 1c 7c 7b e2 65 45 00 00 00 00 04 01 c7 08
```

Conclusion

Min-entropy of the data itself is lacking, but the volume makes up for this.

The DRBG is of a strong design, and can deal w/ large amounts of low min-entropy data

To help prevent attacks, add a quota system that limits the rate at which a user can request data (such that other users are not impacted)

Code needs some clean up with some questionable practices

Improvements can be made to seeding

Q&A

Scripts used for evaluation:

https://github.com/badfilemagic/fbsd-entropy

NIST SP800-90B tools:

https://github.com/usnistgov/SP800-90B_EntropyAssessment

bsd-rngd in progress:

https://github.com/badfilemagic/bsd-rngd