Managing and Understanding Entropy Usage

black hat USA 2015

Bruce Potter
bpotter@keywcorp.com || gdead@shmoo.com
Sasha Wood
swood@keywcorp.com

UBM Tech

Who are we? Why are we here?

- Bruce – CTO KEYW Corporation, Founder of The Shmoo Group
- Sasha – Senior Software Engineer, KEYW Corporation

- Our research
  - Funded by Whitewood Encryption Systems in support of ensuring the quality of entropy throughout the enterprise
  - Supported by our interns: Katherine Cleland, Jeremy Rennicks, Jeff Acquaviva
  - Whitewood is coming out of stealth at BlackHat. Stop by the booth and play our entropy generation game

WHITEWOOD
ENCRYPTION SYSTEMS, INC.
an AMFI Company

TAKE THE ENTROPY CHALLENGE!
STOP BY BOOTH 967

Our Research

- Goal: Better understand and manage entropy and randomness in the enterprise
- Determine rates of Entropy Production on various systems
- Determine rates of Entropy Consumption of common operations
- Determine correlation between entropy demand and supply of random data

If there's one theme in the work we did it's "no one really understands what's happening with respect to entropy and randomness in the enterprise"

Which of these use more entropy?

Which of these use more entropy?

```
# /bin/ls
```

```
# openssl genrsa 1024
```

Which of these use more entropy?

```
# /bin/ls
```

```
# openssl genrsa 1024
```

Which of these use more entropy?

```
# openssl genrsa 4096    # openssl genrsa 1024
```

Which of these use more entropy?

```
# openssl genrsa 4096    # openssl genrsa 1024
```

SAME!

Which of these use more entropy?

```
# openssl genrsa 4096    4096 bit TLS with PFS
```

Which of these use more entropy?

`# openssl genrsa 4096`   `4096 bit TLS with PFS`

Which of these use more entropy?

`1024 bit TLS (no PFS)`   `4096 bit TLS with PFS`

Which of these use more entropy?

`1024 bit TLS (no PFS)`   `4096 bit TLS with PFS`

SAME!

Which of these use more entropy?

`# /bin/ls`

`4096 bit TLS with PFS`

Which of these use more entropy?

`# /bin/ls`

`4096 bit TLS with PFS`

Which of these use more entropy?

`# /bin/cat`

`4096 bit TLS with PFS`

Which of these use more entropy?

`# /bin/$ANYTHING`

`4096 bit TLS with PFS`

More on this later…

Entropy vs. Random Numbers

- It’s a bit of a state of mind… there are several ways to think about it

- **Entropy** is the uncertainty of an outcome. **Randomness** is about the quality of that uncertainty from a historical perspective.
- **Full entropy** is 100% **random**.
- There are tests that measure **entropy**. **Randomness** either is or is not.
- **Entropy** has a quantity. **Randomness** has a quality.

RNG's vs. PRNG's

- Random Number Generators are non-deterministic… often based on specialized hardware and physical processes
- Pseudo Random Number Generators ARE deterministic.. But that doesn't mean they are bad
  - Tons of research in this space, and won't bore you with it here (and either you agree with the public research or you don't)
  - In a nutshell, good PRNG's with good seeding, such as those based on SHA hashes using a hardware RNG, are good for ~2^48 BLOCKS without reseeding

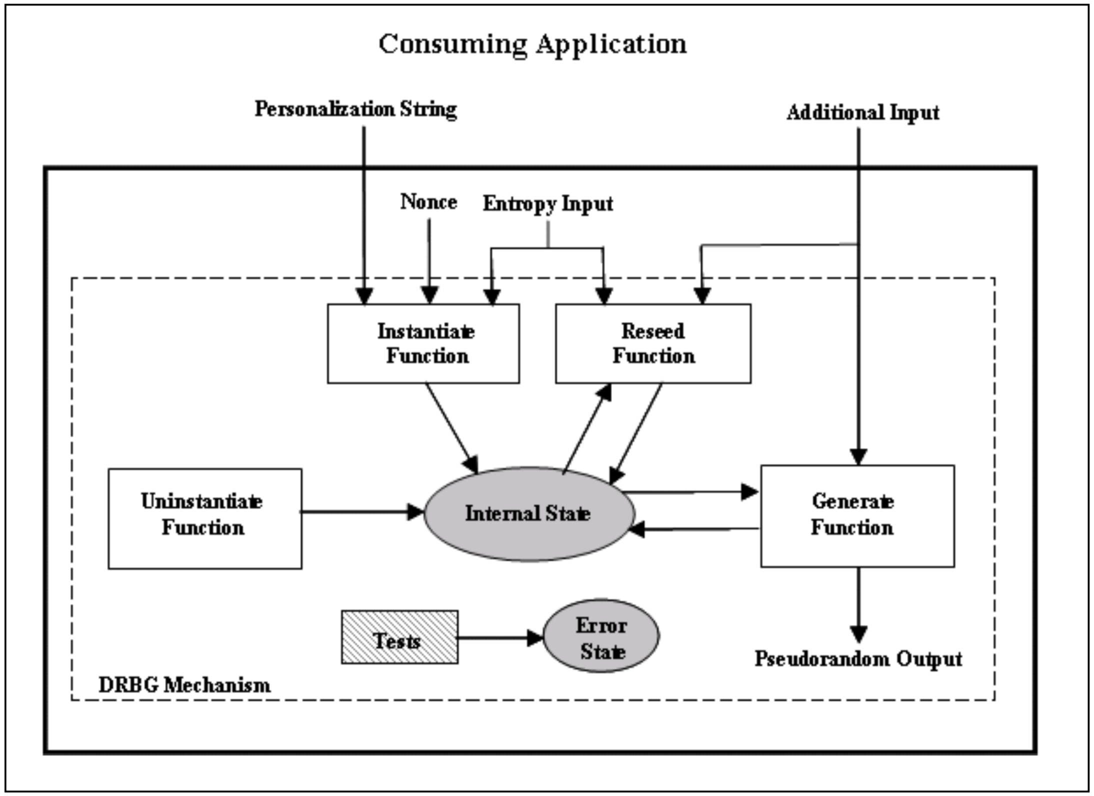


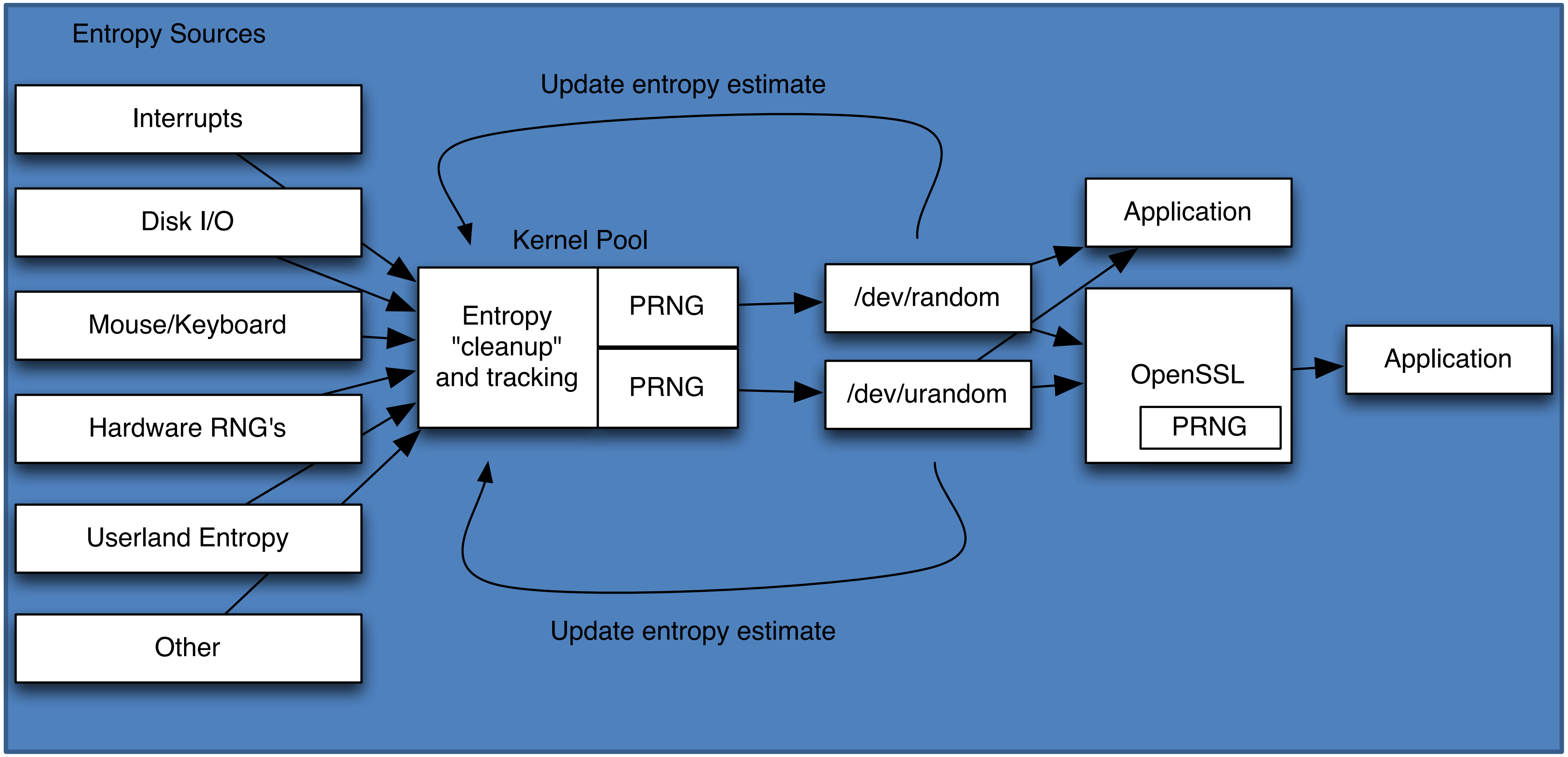
Figure 1: DRBG Functional Model

Source: NIST 800-90A

Entropy and PRNG’s

- An analogy is helpful to understand how entropy relates to PRNG’s. We’re in Vegas, so think of a deck of cards… a REALLY big deck of cards
  - Entropy generation is the act of shuffling cards
  - The PRNG is the act of dealing the deck
  - The deal is clearly deterministic, however…
  - The better the shuffling, the more random the deal will be
- In summary, the better the entropy, the better the PRNG will perform

Linux Entropy and Random Ecosystem



Entropy Production – Events and such..

- Time/Date (VERY low entropy)
- Disk IO
- Interrupts
- LSB from network activity / inter arrival time
- Keyboard input
- Mouse movements

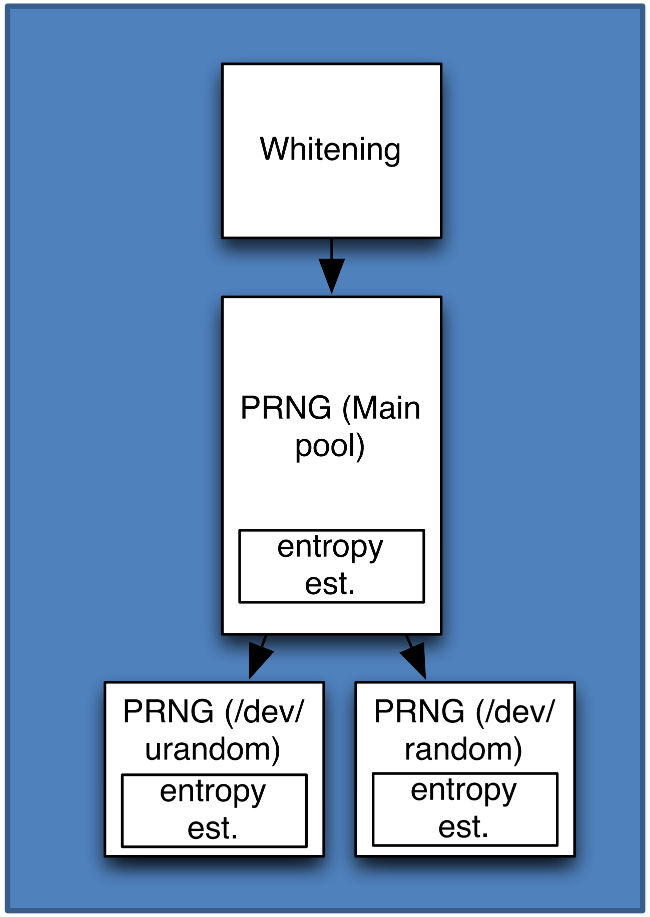
Entropy Production – Hardware RNG's

Sources of Entropy
Thermal Noise
Atmospheric Noise
A/D conversion Noise
Beam Splitting
Quantum
Shot Noise
Lava Lamps

Implementations
Ivy Bridge (Thermal Noise) – Intel integrated hardware into Ivy Bridge Chipsets (access via RDRAND)
Entropy Key (Shot Noise) – Simtec's USB connected entropy generator
BitBabbler et al (Shot Noise)
Whitewood et al (Quantum)
Lavarand (Lava Lamp) – Developed by SGI. Uses images of Lava Lamp to find entropy

Linux Entropy Pool

- Linux maintains an entropy pool where entropy is stored for use by `/dev/random` and `/dev/urandom`
- Data from entropy sources may have bias that's removed via whitening
- Note on entropy estimation… there is no absolutely correct measure of entropy in the pool so the kernel makes an estimation
  - A polynomial analysis is done on incoming events to see how "surprising" the event is. If it's REALLY surprising, you get a good bump in entropy. If it's not, you don't get much.
- The `/dev/random` and `/dev/urandom` pools are generally close to zero. Entropy is fed from the main pool when necessary.
- Pool can be queried for an approximation of how much entropy is available (`/proc/sys/kernel/random/entropy_avail`). Value returned is from the main pool.
- Pool is normally 4096 bits. Can be increased (but unlikely to be needed)



Entropy Production - Stats

- Entropy production rates for various use cases in virtual machine and bare metal systems
  - 5 runs @ 300 seconds each

Use Case	Entropy Production Rate (VM)	Entropy Production Rate (Bare Metal)
Unloaded	1.94 bits/s	2.4 bits/s
Apache with 10 reqs/s	3.85 bits/s	4.84 bits/s
Apache with 50 reqs/s	5.56 bits/s	7.66 bits/s
Pinged every .1s	2.2 bits/s	2.95 bits/s
Pinged every .001s	3.77 bits/s	13.92 bits/s
Heavy Disk IO	3.26 bits/s	4.75 bits/s
Unloaded with HAVEGED	9.08 bits/s	1.2 bits/s
Pinged every .1s with HAVEGED	9.17 bits/s	1.54 bits/s
Pinged every .0001s with HAVEGED	12.6 bits/s	2.91 bits/s
Heavy Disk IO with HAVEGED	9.09 bits/s	2.33 bits/s
RDRAND	N/A	0.58 bits/s

Entropy Production - Stats

Use Case	Entropy Production Rate (VM)	Entropy Production Rate (Bare Metal)
Unloaded	1.94 bits/s	2.4 bits/s
Pinged every .001s	3.77 bits/s	13.92 bits/s
Pinged every .0001s with HAVEGED	12.6 bits/s	2.91 bits/s
RDRAND	N/A	0.58 bits/s

- Unloaded
  - Very little entropy available any way you cut it
- Virtual Machines
  - Not as much entropy available as with bare metal… run HAVEGED if you can
- Bare Metal
  - ??? With HAVEGED was far worse than without in our testing
  - Tried several different setups with same results. Interested in other's findings
- RDRAND
  - Unavailable (normally) in the VM. However, even on bare metal, kernel entropy estimation was not helped by RDRAND
  - Turns out, due to recent concerns regarding RDRAND, even though RDRAND can be used to reseed the entropy pool, the entropy estimation is NOT increased by the kernel… on purpose

Entropy Production - Stats

- Entropy production rates for various Android devices

Device, Use Case	Entropy Production Rate
S3, IO Load	20.7 bits/s
S3, IO Load, web browsing	24.4 bits/s
HTC One m8, IO Load	38.4 bits/s
HTC One m8, unloaded	32.9 bits/s
Apache (atom based tablet), unloaded	23.7 bits/s
Proprietary phone, unloaded	38.5 bits/s
Proprietary phone, shaken	34.7 bits/s

- Across the board, more entropy production than on our servers and VM's (even without load)
- We tried to "shake" a phone to see if the accelerometer would add entropy. No joy.

random vs urandom

- `/dev/random`
  - Provides output that is roughly 1:1 bits of entropy to bits of random number
  - Access depletes kernel entropy estimation
  - Blocks if entropy pool is depleted and waits
- `/dev/urandom`
  - Never blocks
  - Will reduce entropy estimation IF sizeof($request) < (Entropy estimation – 128 bits)… and only will reduce estimation down to 128 bits
  - Will not reduce entropy estimation from the pool if the pool is less than 192 bits (64bits is minimum amount of entropy to be provided)
  - Each read produces a hash which is immediately fed back in to the pool
  - For most (all) purposes `/dev/urandom` is fine (if used correctly… ie: you check to see if you actually got entropy)
- `get_random_bytes()`
  - Call for the kernel to access the entropy pool without going through the character devices
  - Just a wrapper to access the non-blocking pool just like /dev/urandom

Things that aren't random

- C `rand()` – A Linear Congruential Generator. The same seed results in the same output. And if you know two consecutive outputs, you know all outputs
  - https://jazzy.id.au/2010/09/20/cracking_random_number_generators_part_1.html
- Python `random.py` – Implements a Mersenne Twister. Better than `rand()` but still not suitable for crypto (624 consecutive outputs gives up everything)
  - https://jazzy.id.au/2010/09/22/cracking_random_number_generators_part_3.html

An interesting thing happened on the way home from the entropy pool…

```
# cat /proc/sys/kernel/random/entropy_avail
882
# cat /proc/sys/kernel/random/entropy_avail
777
# cat /proc/sys/kernel/random/entropy_avail
672
# cat /proc/sys/kernel/random/entropy_avail
567
```

This is an unloaded server… almost no activity.
What's going on here?

Linux is consuming entropy all the time

- …including at process start time. For ASLR, KCMP, and other aspects of `fork/copy_process()`, the kernel can consume up to 256 bits of entropy each time you start a process
- Not consistent, still doing research.
- In our observation, the pool was never maxed out, and in general was never above 25% full
- Calls to `/dev/urandom` were often close enough to the 192 bit cutoff that even though data was returned, no entropy was removed from the pool
- Even without doing “crypto” there is constant pressure on the entropy pool due to OS activities
  - Makes it even harder to get good seeds for cryptographic operations.

Entropy in OpenSSL

- OpenSSL maintains its OWN PRNG that is seeded by data from the kernel
- This PRNG is pulled from for all cryptographic operations including
  - Generating long term keys
  - Generating ephemeral/session keys
  - Generating nonces
- Draws against this PRNG are not known about by kernel and are not tracked outside of OpenSSL

Entropy in OpenSSL – Issue #1

- OpenSSL only seeds its internal PRNG once per runtime
- No problem for things like openssl genrsa 1024
- Different situation for long running daemons that link to openssl… like webservers
- Our data shows that an Apache PFS connection requires ~300-800 bytes of random numbers
  - Nonce for SSL handshake (28 bytes)
  - EDH key generation (the rest)
- If your application doesn't put limits on # of connections per child, these nonces and keys will be derived from the same PRNG that is never reseeded

```
md_rand.c (openssl):
 if (!initialized) {
        RAND_poll();
        initialized = 1;
    }

[[initialized is set back
to 0 at exit]]


ssl_engine_init.c (mod_SSL):
ssl_rand_seed(base_server, ptemp,
      SSL_RSCTX_STARTUP, "Init: ");
```

Entropy in OpenSSL – Issue #2

- OpenSSL pulls seed from `/dev/urandom` by default (and stirs in other data that is basically knowable)
- OpenSSL does NOT check to see the quality of the entropy when it polls `/dev/urandom`
  - It asks for 32bytes of random data at init. `/dev/urandom` will happily provide it
  - If entropy estimate is <384, only a limited amount of entropy is removed from the pool
  - If entropy estimate is <192, no entropy is removed from the pool
- This affects both command line use and library use of OpenSSL. It does not discriminate.

Entropy in mod_SSL

- mod_SSL is the de facto mechanism to implement SSL/TLS in Apache
- Given that SSL does not reseed in long running processes, mod_SSL *should* periodically reseed
  - Best practice according to https://wiki.openssl.org/index.php/Random_Numbers
  - Programs can defensively call `RAND_poll` (somewhat dangerous on some platforms), `RAND_add` or `RAND_seed` (requires you to get a new seed yourself)
- Does NOT call `RAND_poll()` or `RAND_add()`
  - Rather, mod_SSL attempts to add entropy itself to the OpenSSL PRNG

Entropy in Mod_SSL - Issue

- mod_SSL's attempt to generate entropy is not very sophisticated. On every request, it stirs in
  - Date/time (4 bytes)
  - PID
  - 256 bytes off the stack
- Date/Time is low resolution and guessable
- PID is a limited search space
- 256 bytes off the stack does not vary from call to call

mod_SSL Entropy Analysis

- Analysis of the 256 bytes from the stack
  - mod_SSL is declaring an unsigned char[256], not zeroizing it, and then reading the data in that array as part of the attempt to gather entropy
  - Of our ~600k observations of this process, the array yielded 4568 unique values (ie: <8% of the time the value changed)
  - Combined with PID (which for server processes will always be the same) and date (with only 1 second resolution), there is very little entropy added through mod_SSL
- In summary, mod_SSL tries really hard but doesn’t do much better than doing nothing at all

nginx Entropy Analysis

- Let’s not let nginx off the hook…
- nginx does does nothing to attempt to stir in more entropy
  - Relies solely on OpenSSL’s built in entropy decisions

Entropy vs. Random Numbers

- How much entropy goes in to each random byte you need?
  - It depends on what you're doing
- Tested various common actions in OpenSSL
  - These are averages. Public key generation can vary wildly based on how "lucky" you are when generating pseudo primes.
  - Assume OpenSSL was seeded with 32 bytes of entropy. In practice, the amount of entropy is likely lower

Activity	Average Random Bytes required
RSA 1024 bit key gen	5721 bytes
RSA 2048 bit key gen	14694 bytes
RSA 4096 bit key gen	139315 bytes
Apache TLS 1024 (no PFS)	498 bytes
Apache TLS 1024 (PFS)	635 bytes

Impact

- How much entropy do you _really_ need in order to be secure
  - Depends on your risk threshold and types of attacks you care about
- Attacks against PRNG’s come in three major forms
  - Control/Knowledge of “enough” entropy sources
  - Knowledge of the internal state of the PRNG
  - Analysis of the PRNG through observed traffic

Impact

- Controlling inputs
  - By default, kernel pulls from a variety of sources to create entropy pool
  - Difficult to control all of them, and ~256bits of entropy from any one source CAN result in cryptographically strong numbers
- Knowledge of state
  - Internal state is very complex, but not impossible to understand
  - Amount of entropy in pool and amount of entropy in the seed pulled from pool is a good surrogate to know the "guessability" of the pool
  - If you're pulling data without appreciable entropy, you're putting yourself at risk
- Observing output
  - According to NIST, you need at least 2^48 blocks of output
  - That's "blocks of output".. Not bytes. Some PRNG blocks are 256bits or more
  - Apache needs 1,000,000 million bits of randomness per minute on a fully loaded core
  - That's 10,000 years (give or take) before NIST thinks you need to reseed
- BUT….
  - NIST's guidance assumes correctness. What we've described isn't correct. Successful crypto requires attention to detail. While we don't know if the issues we've talked about today are exploitable, they're certainly not good
  - All things being equal, we'd be skeptical of the current system

libWES Release

- We created the WES Entropy Client as a solution to the wild west of entropy generation and distribution.
  - Initial release is for OpenSSL
- The client allows users to:
  - Select which entropy sources are used,
  - How to use each source; seed or stream,
  - Which PRNG to use and at what security strength,
  - Provide our own PRNG for use,
  - How often to re-seed the PRNG,
  - Whether or not to use prediction resistance, and
  - View generation and consumption statistics.

libWES vs egd vs status quo

- Status quo
  - Does not allow insight into the production or consumption of entropy.
  - Cannot block OpenSSL from reading from /dev/urandom if no entropy has been produced (startup on VM clone, etc)
- EGD
  - Simply puts entropy into the kernel pool, therefore all the issues above apply
- WES Entropy Client
  - Provides it’s own entropy pool and distribution within OpenSSL.
  - Allows users to pick which entropy sources are used and how they are used
  - Will not distribute entropy until properly seeded if using PRNG.

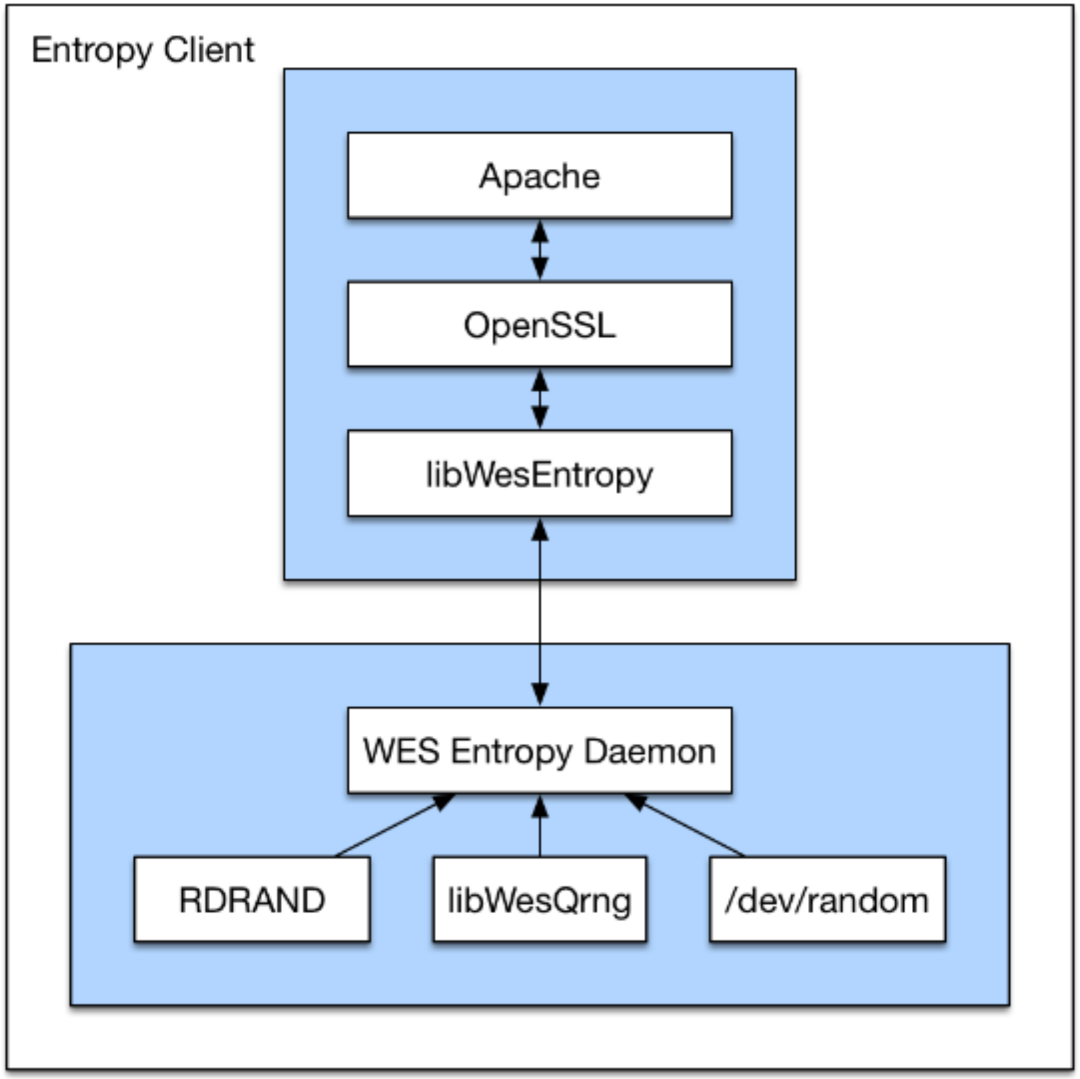
RDRAND, /dev/random, and LKML

- 2011- Linux kernel contained code that said “if you’ve got RDRAND, use that as sole source of entropy”
- 2012 – Theodore Ts’o (re)took control of that part of the kernel and added other sources of entropy back in
- Spring 2013 – Snowden
- Fall 2013 – A call to remove ALL support for RDRAND from Linux random subsystem (including online petitions and the like)
- Fall 2013 – Linus says “Where do I start a petition to raise the IQ and kernel knowledge of people? Guys, go read drivers/char/random.c. Then, learn about cryptography. Finally, come back here and admit to the world that you were wrong.
- “Short answer: we actually know what we are doing. You don't.”

Given our findings on process forking vs. OpenSSL use, we’re not sure anyone knows what they’re doing. So, we did it ourselves.

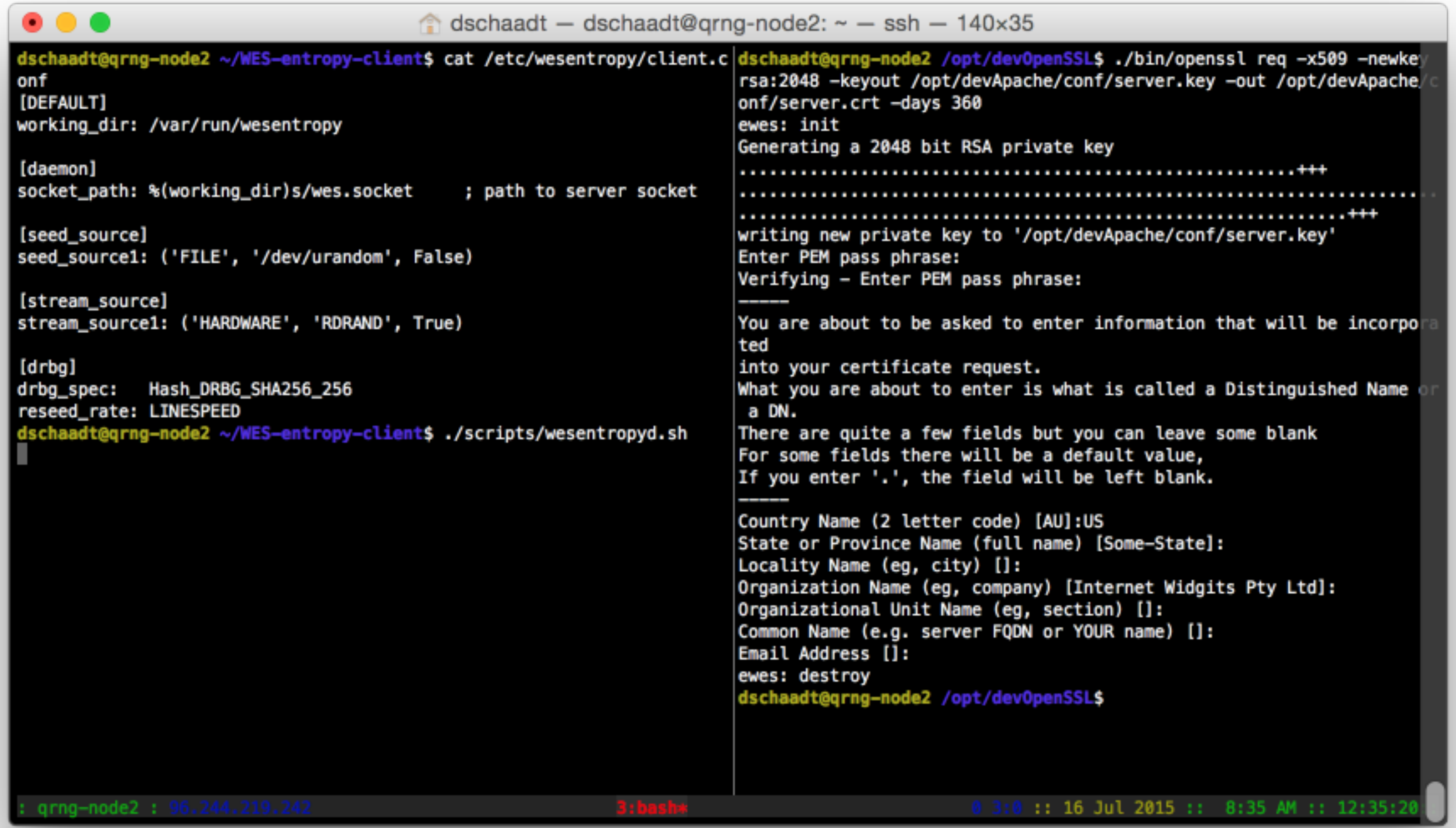
libWES Architecture

- libWesEntropy – Replaces OpenSSL random engine
- WES Entropy Daemon – Interfaces with external entropy sources and feeds entropy in to OpenSSL directly
  - Entropy sources can even be network based. Entropy can be provided as a service to libWES clients
- Must configure OpenSSL to use libWES
- Most software doesn't look at OpenSSL configuration and just uses "stock"
  - To get mod_SSL integration with libWES, you have to recompile mod_SSL to call OpenSSL so it reads configuration



libWES Client Example

Generating a server key and certificate:



dschaadt — dschaadt@qrng-node2: ~ — ssh — 140×35

```
dschaadt@qrng-node2 ~/WES-entropy-client$ cat /etc/wesentropy/client.c
onf
[DEFAULT]
working_dir: /var/run/wesentropy

[daemon]
socket_path: %(working_dir)s/wes.socket     ; path to server socket

[seed_source]
seed_source1: ('FILE', '/dev/urandom', False)

[stream_source]
stream_source1: ('HARDWARE', 'RDRAND', True)

[drbg]
drbg_spec:   Hash_DRBG_SHA256_256
reseed_rate: LINESPEED
dschaadt@qrng-node2 ~/WES-entropy-client$ ./scripts/wesentropyd.sh
```

```
dschaadt@qrng-node2 /opt/devOpenSSL$ ./bin/openssl req -x509 -newkey
rsa:2048 -keyout /opt/devApache/conf/server.key -out /opt/devApache/c
onf/server.crt -days 360
ewes: init
Generating a 2048 bit RSA private key
.......................................................+++
..................................................................
.............................................................+++
writing new private key to '/opt/devApache/conf/server.key'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorpora
ted
into your certificate request.
What you are about to enter is what is called a Distinguished Name or
 a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:
ewes: destroy
dschaadt@qrng-node2 /opt/devOpenSSL$
```

```
: qrng-node2 : 96.244.219.242             3:bash*             0 3:0 :: 16 Jul 2015 ::  8:35 AM :: 12:35:20
```

WES Client Release

- Currently available through
  - http://whitewoodencryption.com/
- Client is under active development, so please provide feedback and bug reports
- Future plans
  - Integration with nginx, OpenSSH, and other large-scale OpenSSL consumers
  - Integration with Whitewood’s QRNG and Entropy as a Service offering
  - Focus on stability and scalability

Blackhat Sound Bytes

- Entropy use is not well understood and managed; Developers and administrators don't have complete ownership so issues persist
- OpenSSL has several weaknesses in how it handles entropy, particularly when linked in to long running processes like servers
- Need to think about entropy lifecycle, just like we think about other crypto lifecycles

Questions?