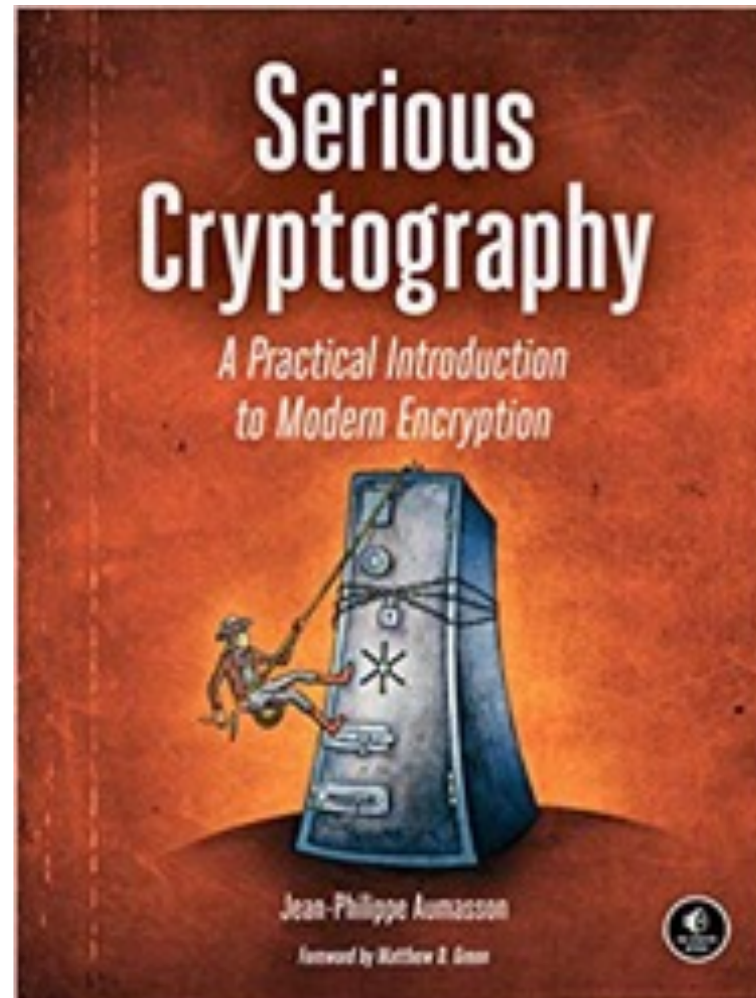
CNIT 141
Cryptography for Computer Networks



2. Randomness

Updated 8-28-2023

Topics

- Random or Non-Random?
- Randomness as a Probability Distribution
- Entropy: A Measure of Uncertainty
- Random Number Generators (RNGs) and Pseudorandom Number Generators (PRNGs)
- Real-World PRNGs
- How Things Can Go Wrong

Random or Non-Random?

What is Randomness?

- Is 11010110 more random than 00000000 ?
- Both are equally likely, as exact values
- But if the first one is described as "three zeroes and five ones" it's more likely
- So if we see something that "looks like" 11010110
- That is more likely to be truly random than 00000000

Randomness as a Probability Distribution

Probability

- A fair coin
  - 50% chance of head, 50% chance of tails
- A fair die
  - 1/6 chance of 1, 1/6 of 2, ... up to 6
- Total is always 100%
- **Uniform distribution**
  - Equal chance of every outcome

Entropy: A Measure of Uncertainty

Definition of Entropy

- Distribution has probabilities

  $p_1, p_2, \ldots p_N$

- Entropy is

  $- p_1 \log(p_1) - p_2 \log(p_2) \ldots - p_N \log(p_N)$

- log is to base 2

Examples

- One random bit: probabilities

  1/2, 1/2

- Entropy is

  - 1/2 log(1/2) - 1/2 log(1/2)

  log(1/2) = -1, so this is

  - 1/2 (-1) - 1/2 (-1) = 1 bit

- Also called ***information content***

Examples

- One random byte: probabilities

  1/256, 1/256, ... 1/256 (256 equal values)

- Entropy is

  - 1/256 log(1/256) - 1/256 log(1/256) ... (256 terms)

  log(1/256) = -8, so this is

  - 1/256 (-8) - 1/256 (-8) ... (256 terms) **= 8 bits**

Examples

- One non-random bit: probabilities

  99% of 0, **1%** of 1
- Entropy is

  - 0.99 log(0.99) - 0.01 log(0.01)

  0.014 + .066 = 0.08
- For **99.99%** of 0, **0.01%** of 1
  - Entropy is **.00147**
- If there's no randomness, entropy is zero

Python Code

```
GNU nano 2.0.6                    File: ent.py

import math

prob = [.5, .5]
print("Probabilities:", prob)

e = 0.0
for p in prob:
  e -= p * math.log(p, 2)

print("Entropy:", e)
```

```
Sam-2:141 sambowne$ python3 ent.py
Probabilities: [0.5, 0.5]
Entropy: 1.0
```

Eight Possibilities

```
  GNU nano 2.0.6                    File: ent2.py

import math

prob = [.125, .125,.125,.125,.125,.125,.125,.125]
print("Probabilities:", prob)

e = 0.0
for p in prob:
  e -= p * math.log(p, 2)

print("Entropy:", e)
```

```
Sam-2:141 sambowne$ python3 ent2.py
Probabilities: [0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125]
Entropy: 3.0
```

Weighted Coin

```
GNU nano 2.0.6                    File: ent3.py

import math

prob = [.9, .1]
print("Probabilities:", prob)

e = 0.0
for p in prob:
  e -= p * math.log(p, 2)

print("Entropy:", e)
```

```
Sam-2:141 sambowne$ python3 ent3.py
Probabilities: [0.9, 0.1]
Entropy: 0.4689955935892812
```

Kahoot!

2a

Random Number Generators (RNGs)

and

Pseudorandom Number Generators (PRNGs)

RNGs and PRNGs

- To generate randomness, computers need
  - A source of entropy
    - Provided by a **Random Number Generator** (**RNG**)
  - An algorithm to produce random bits from the entropy
    - **Pseudorandom Number Generator** (**PRNG**)

RNG

- Randomness comes from the environment
  - Analog, chaotic, unpredictable
  - Temperature, acoustic noise, random electrical fluctuations
  - Sensors: I/O devices, network or disk activity, logs, running processes, keypresses, mouse movements

QRNG

- Quantum Random Noise Generator
  - Radioactive decay, vacuum polarization, photons

PRNG

- Pseudorandom Noise Generator
  - Create many artificial random bits
  - From a few truly random bits
  - Continues working even if physical source stops (e.g., the mouse stops moving)

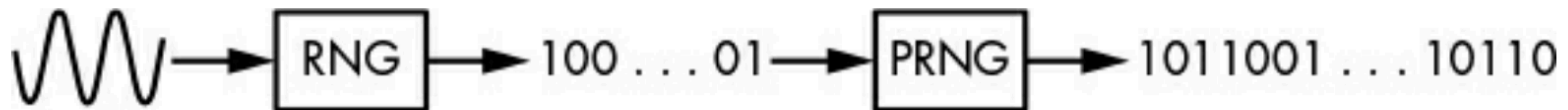


Figure 2-1: RNGs produce few unreliable bits from analog sources, whereas PRNGs expand those bits to a long stream of reliable bits.

How PRNGs Work

- PRNG receives random bits from RNG
  - at regular intervals
- Updates the ***entropy pool***
- Mixes pool's bits together when updating
  - To remove bias

DRBG

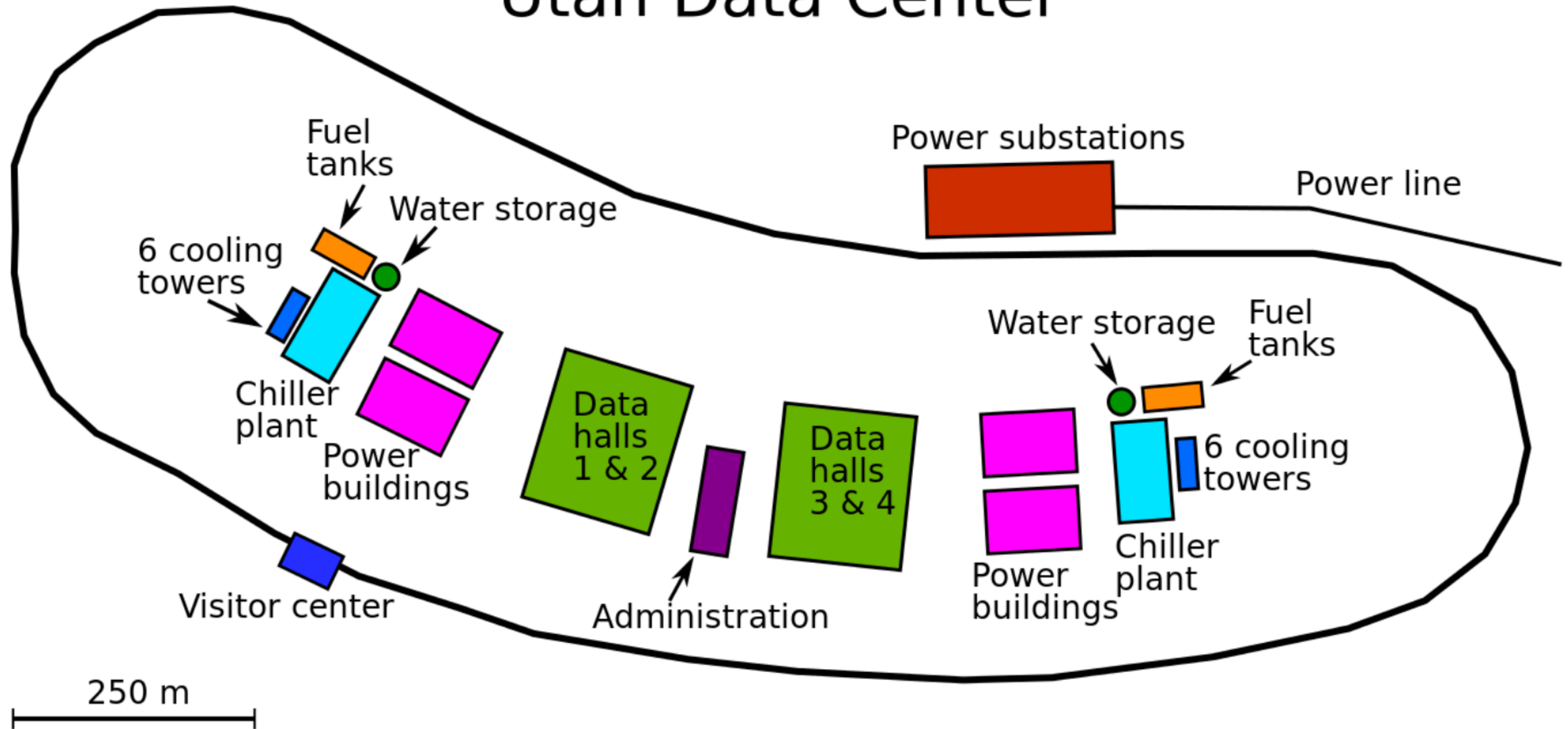
- The PRNG uses a **Deterministic Random Bit Generator (DRBG)**
- Expands some bits from the entropy pool into a much longer sequence
- Deterministic: not randomized
  - Always produces the same stream of bits from the same input

PRNG Operations

- **init()**
  - Initializes the entropy pool and the internal state of the PRNG
- **refresh()**
  - Updates the entropy pool using **R** (data from the RNG), called **reseeding**
  - **R** is called the **seed**
- **next()**
  - Returns N pseudorandom bits and updates the entropy pool

Security Concerns

- **Backtracking resistance**
  - Also called ***forward secrecy***
  - Previously generated bits are impossible to recover
- **Prediction resistance**
  - Future bits are impossible to predict



- NSA stores exabytes of captured encrypted traffic
  - 1 EB is 1 million TB
- Waiting for cryptographic keys to be found

Achieving Resistance

- **Backtracking resistance**
  - **refresh** and **next** operations must be irreversible, so
  - If attacker obtains the entropy pool, they still can't determine previously generated bits
- **Prediction resistance**
  - PRNG must **refresh** regularly with **R** values that the attacker cannot find or guess

Fortuna

- A PRNG designed in 2003 by Neils Ferguson and Bruce Schneier
    - Used in Windows
    - Uses 32 entropy pools, a 16-byte key, and a 16-byte counter
- Mac OS and iOS use ***Yarrow***
    - Designed in 1998 by Kelsey and Schneier

Security Failures

- If RNGs fail to produce enough random bits
  - Fortuna might not notice, and produce lower-quality pseudorandom bits
  - Or stop delivering bits
- If seed files are stolen or re-used,
  - Fortuna will produce identical sequences of bits

Cryptographic vs. Non-Cryptographic PRNGs

- Most PRNGs provided for programming languages are **non-cryptographic**
  - Only concerned with statistical randomness, not predictability
  - Often use ***Mersenne Twister*** algorithm
  - Used in PHP, Python, Ruby, R, and more
- **Cryptographic** PRNGs are unpredictable

Real-World PRNGs

Unix-Based Systems

- **/dev/urandom** gets data from the crypto PRNG
  - Non-blocking: always returns data, even if entropy is low
- **/dev/random**
  - Blocking: refuses to return data if entropy is low

Blocking

- Blocking turned out to be a bad idea
  - Entropy estimates are unreliable
  - Attackers can fool them
  - **/dev/random** runs out of entropy quickly
    - Producing denial of service while waiting for more entropy
  - In practice, **/dev/urandom** is better

Fresh Tomcat takes loong time to start up

Haveged

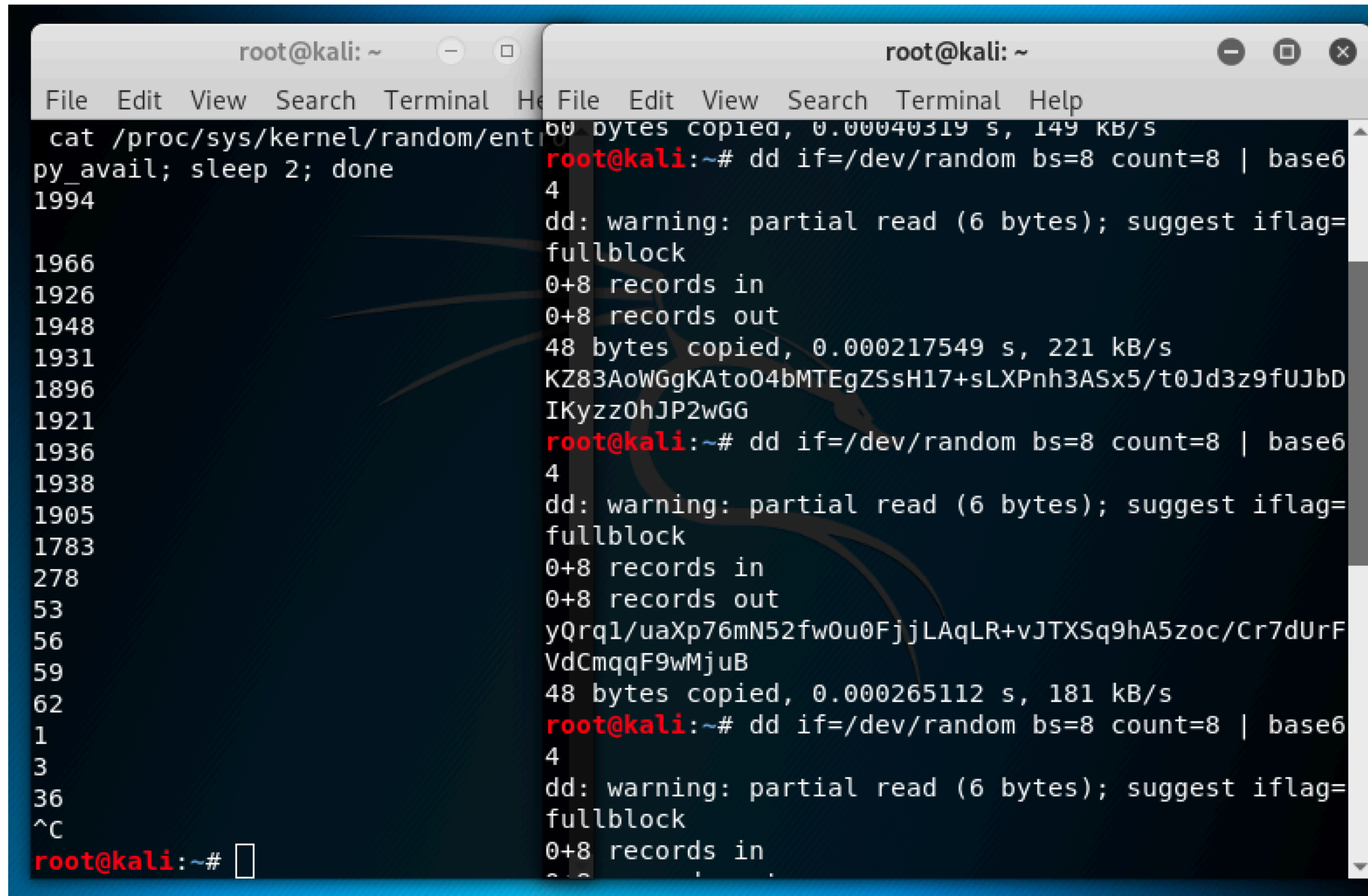
The **haveged project** is an attempt to provide an easy-to-use, unpredictable **random number generator** based upon an adaptation of the HAVEGE algorithm. Haveged was created to remedy low-entropy conditions in the Linux random device that can occur under some workloads, especially on headless servers.

- Links Ch 2b, Ch 2c

Linux Commands (Before 2022)

- To see entropy pool
  - **`for i in {1..100}; do cat /proc/sys/kernel/random/entropy_avail; sleep 2; done`**
- To consume entropy
  - **`dd if=/dev/random bs=8 count=8 | base64`**

Demo: Without Haveged



Demo: With Haveged

```
root@kali: ~
File  Edit  View  Search  Terminal  Help
1783
278
53
56
59
62
1
3
36
^C
root@kali:~# for i in {1..100}; do
 cat /proc/sys/kernel/random/entro
py_avail; sleep 2; done
2449
2401
2408
1966
1337
1951
1451
2464
1448
^C
root@kali:~#
```

```
root@kali: ~
e  Edit  View  Search  Terminal  Help
 records in
 records out
WpEQ09avqaeN6lQIiuwD+hDfUM4n18jkek/mCQaIqEHqIGzk
xEytGJ1B+T
bytes copied, 0.000158845 s, 302 kB/s
t@kali:~# dd if=/dev/random bs=8 count=8 | base6

 warning: partial read (7 bytes); suggest iflag=
lblock
 records in
 records out
bytes copied, 0.000177802 s, 298 kB/s
S6paYw7VI1CMgjN8sztp0HrTiIxtm61artAr2BQVk+mw5b60
97ReVb3H9JY0sL6uY=
t@kali:~# dd if=/dev/random bs=8 count=8 | base6

 warning: partial read (6 bytes); suggest iflag=
lblock
 records in
 records out
bytes copied, 0.000168294 s, 285 kB/s
GmfJRtZ4MumTtQUBqh8EQTqeYmbacLOyR5ldKWEeo8Piz9fb
zdyF2XA925
t@kali:~#
```

256 Bits

- The **/dev/random** RNG was overhauled
- Now it always has 256 bits available entropy
- You can't drain it
  - https://www.reddit.com/r/archlinux/comments/v75t5w/entropy_of_devrandom_stuck_at_256/

Windows

- CryptGenRandom() function
  - Now replaced by BcryptGenRandom()
- Takes entropy from the kernel mode driver **cng.sys** (formerly **ksedd.sys**)
- Loosely based on Fortuna

Intel RDRAND

- Hardware RNG introduced in 2012 with Ivy Bridge
- Uses **RDRAND** assembly language instruction
- Only partially documented
- Some people fear that it has an NSA backdoor

On the Possibility of a Back Door in the NIST SP800-90 Dual Ec Prng

Dan Shumow
Niels Ferguson
Microsoft

- Talk given in 2007
- Link Ch 2d

Conclusion

- WHAT WE ARE NOT SAYING:
  NIST intentionally put a back door in this PRNG
- WHAT WE ARE SAYING:
  The prediction resistance of this PRNG (as presented in NIST SP800-90) is dependent on solving one instance of the elliptic curve discrete log problem.
  (And we do not know if the algorithm designer knew this before hand.)

Schneier on Security

Blog | Newsletter | Books | Essays | News | Talks

Blog >

The Strange Story of Dual_EC_DRBG

- Dual_EC_DRBG is 1000x slower than other options
- Championed by the NSA
- Schneier said to avoid it in 2007
  - Link Ch 2f

Secret Documents Reveal N.S.A. Campaign Against Encryption

Documents show that the N.S.A. has been waging a war against encryption using a battery of methods that include working with industry to weaken encryption standards, making design changes to cryptographic software, and pushing international encryption standards it knows it can break. Related Article »

TOP SECRET//SI/TK//NOFORN

(U) COMPUTER NETWORK OPERATIONS
(U) SIGINT ENABLING

This Exhibit is SECRET//NOFORN

	FY 2011[1] Actual	FY 2012 Enacted			FY 2013 Request			FY 2012 — FY 2013	
		Base	OCO	Total	Base	OCO	Total	Change	% Change
Funding ($M)	298.6	275.4	—	275.4	254.9	—	254.9	-20.4	-7
Civilian FTE	144	143	—	143	141	—	141	-2	-1
Civilian Positions	144	143	—	143	141	—	141	-2	-1
Military Positions	—	—	—	—	—	—	—	—	—

[1]Includes enacted OCO funding. Totals may not add due to rounding.

- TOP SECRET leaks from Snowden
- New York Times, 2013 (Link Ch 2h)

(U) Project Description

(TS//SI//NF) The SIGINT Enabling Project actively engages the US and foreign IT industries to covertly influence and/or overtly leverage their commercial products' designs. These design changes make the systems in question exploitable through SIGINT collection (e.g., Endpoint, MidPoint, etc.) with foreknowledge of the modification. To the consumer and other adversaries, however, the systems' security remains intact. In this way, the SIGINT Enabling approach uses commercial technology and insight to manage the increasing cost and

(U) Base resources in this project are used to:

- (TS//SI//REL TO USA, FVEY) Insert vulnerabilities into commercial encryption systems, IT systems, networks, and endpoint communications devices used by targets.
- (TS//SI//REL TO USA, FVEY) Collect target network data and metadata via cooperative network carriers and/or increased control over core networks.
- (TS//SI//REL TO USA, FVEY) Leverage commercial capabilities to remotely deliver or receive information to and from target endpoints.
- (TS//SI//REL TO USA, FVEY) Exploit foreign trusted computing platforms and technologies.
- (TS//SI//REL TO USA, FVEY) Influence policies, standards and specification for commercial public key technologies.

NIST formally chops NSA-tainted random number generator

By Juha Saarinen
Jun 29 2015
6:47AM

Dual_EC_DRBG algorithm no longer part of standard.

- Link Ch 2g

How Things Can Go Wrong

Poor Entropy Sources

- Netscape's SSL in 1996
  - Seeded from process ID and system time in microseconds
  - Predictable values
  - Total entropy only 47 bits, but should have had 128

Crypto shocker: four of every 1,000 public keys provide no security (updated)

Almost 27,000 certificates used to protect webmail, e-commerce, and other ...

DAN GOODIN - 2/15/2012, 4:00 AM

- In 2012, researchers tested 7.1 million 1024-bit RSA public keys
- 27,000 of them had a shared prime factor
  - (p or q)
  - Link Ch 2i

Insufficient Entropy at Boot Time

- Cause: devices generated public keys early after bootup, before collecting enough entropy

```
prng.seed(seed)
p = prng.generate_random_prime()
q = prng.generate_random_prime()
n = p*q
```

Non-Cryptographic PRNG

- Old version of MediaWiki, used for Wikipedia
- mt_rand is a Mersenne Twister

```
        /**
           * Generate a hex-y looking random token for various uses.
           * Could be made more cryptographically sure if someone cares.
           * @return string
           */
function generateToken( $salt = '' ) {
    $token = dechex(mt_rand()).dechex(mt_rand());
    return md5( $token . $salt );
}
```

Sampling Bug with Strong Randomness

- Cryptocat had an off-by-one error
- Values had 45 bits of entropy instead of 53

```
Cryptocat.random = function() {
    var x, o = '';
    while (o.length < 16) {
        x = state.getBytes(1);
        if (x[0] <= 250) {
            o += x[0] % 10;
        }
    }
    return parseFloat('0.' + o)
}
```

2b