CAA 24-25

Randomness Generation

Alexandre Duc

Random Numbers

Question

Where do we need random numbers ?

What are the risks if they are not perfectly random ?

Bad Randomness is Destructive : IVs

- IVs are meant to be fresh but **reusing** them has different outcomes based on their usage.
- In CBC, detection of similarities in the first bloc.
- In CTR, CCM, GCM, and stream ciphers reusing an IV is catastrophic $\rightarrow$ XOR ciphertexts $=$ XOR plaintexts.

Bad Randomness is Destructive : RSA

- In RSA, p and q are prime numbers that are used to obtain the public modulus $n = pq$.
- Given p and q, one can recover the **private key**.
- If one prime number is reused between two moduli, one can **factor** them.

Bad Randomness is Destructive : DSA

- DSA is the digital signing algorithm which is based on the discrete logarithm problem.
- It is **non-deterministic** and uses randomness to sign.
- Reusing randomness in a signature implies recovering the **private key** (see further lecture).
- Allowed to recover the ECDSA private key in Playstation 3 used to sign software.

Bad Randomness is Destructive : Statistics

- *Mining your Ps and Qs : Detection of Widespread Weak Keys in Network Devices*, USENIX 2012.
- 0.75% of TLS certificates share keys
- Possible to recover **RSA private keys** of 0.5% of TLS hosts and 0.03% of SSH hosts.
- Possible to recover **DSA private keys** of 1.03% of SSH hosts (using only two signatures per host).
- Some devices for RSA were taking 2 random primes out of a hardcoded list of 9.

Random Number Generator

Random Number Generator

A **random number generator (RNG)** is a physical or a computational device that generates a sequence of numbers that appear to be random.

Statistical Tests

- NIST/Diehard/Dieharder statistical suite tests for "good" statistical properties of generated numbers.
- Check variance, χ^2, ... properties.

Question

Is it sufficient ?

Good PRNG ?

Question

What are the statistical properties of the following PRNG :

$$\text{output} = \text{AES}_0(\text{previous\_output})$$

Is it a good PRNG ?

Statistical Tests

- In cryptography : good statistical properties are **not enough !**
- We requires an additional property : **unpredictability** (also called **next-bit test**).

Next-Bit Test

Next-Bit Test

A random number generator is said to pass the **next-bit test** if an adversary knowing the first i output bits is unable to predict the $(i+1)$-th bit with a probability that is significantly different from $\frac{1}{2}$ within a feasible computational effort.

Theorem (Yao, 1982)

A random number generator passes the next-bit test if and only if it passes all efficient statistical tests.

- **Informal definitions**.
- The next-bit test is a theoretical test.

True Random Number Generators

True Random Number Generator

A **true random number generator (TRNG)** is an apparatus that generators random numbers from a physical process.

- Nuclear decay
- Coin tosses
- Atmospheric noise
- Quantum effects
- Thermal noise in a processor
- ...

Bad "TRNGs"

- Time / date
- pid
- Processor temperature

[Benadjila and Ebalard, Randomness of Random in Cisco ASA]

"In a system which expects to perform cryptographic tasks (...), no developer should start playing with time primitives to extract bits of entropy, expecting an happy end to the story. The system should be designed to include serious entropy sources (...)".

Source: https://eprint.iacr.org/2023/912.pdf

Entropy

- The **bit entropy** measures how much uncertainty there is in the bit.
- Linked to the **probability distribution of a bit**.
- Entropy of uniform bit distribution : 1 (max)
- Entropy of constant bit distribution : 0 (useless)

Bit Distribution Properties

Bias of a bit

A bit is **biased** if its distribution differs from the uniform distribution. For instance, if $\Pr[b = 0] = 0.6$.

Independent bits

Two bits are **independent** if $\Pr[b_1, b_2] = \Pr[b_1] \Pr[b_2]$.

Question

How to transform a source of **independent**, but **biased** random bits into **unbiased** bits ?

von Neumann Randomness Extraction

- Simple solution : transform it into an unbiased bit sequence with the **von Neumann randomness extraction** procedure.
- It works as follows : consider the bit sequence by pairs of two bits. If the pair is 00 or 11, discards them. If the pair is 01, then replace it by 0, and if the pair is 10, then replace it by 1.
- Other, more efficient randomness extractor exist. Some are based on cryptographic primitives (e.g. hash functions).

Pseudorandom Number Generators

Pseudorandom Number Generator

A **pseudorandom number generator (PRNG)** is a deterministic algorithm whose aim is to generate a sequence of numbers exhibiting good statistical properties.

- The value initializing the state of a PRNG is called a **seed**.
- As a PRNG is **deterministic**, it outputs the same sequence of random numbers when seeded with the same value.

Bad PRNGs : Mersenne Twister

Wikipedia

The Mersenne Twister is a pseudorandom number generator (PRNG). It is by far the most widely used general-purpose PRNG.[. . .]
Implementations generally create random numbers faster than other methods.

- A very long period of $2^{19937} - 1$.
- The Mersenne Twister is **not secure** and should **never** be used in cryptographic applications.
- Given 624 outputs, one can recover the inner state.

Bad PRNGs : Common Libraries

Libraries

Most of the PRNGs provided by default by common libraries (Boost, GMP, ...) and languages (C/C++, PHP, Python, etc.) are **not cryptographically secure**.

Examples of **bad calls** :

- **C/C++** : `rand()`, `rand48()`, `arc4random()`
- **Java** : `java.Util.Random` / `Math.random()`
- **Python** : random

Warning

Read the documentation to see if it is cryptographically secure !

Bad PRNGs : Others

Never use the following as PRNGs :

- RC4 (stream cipher but broken)
- Lagged Fibonacci, LFSR, Lehmer, ... (mostly linear or affine)
- `random.org`, `randomnumbers.info`, ...

Cryptographically Secure PRNGs

Cryptographically Secure PRNGs

A **cryptographically secure pseudorandom number generator (CPRNG)** is a deterministic algorithm whose aim is to generate a sequence of numbers that is unpredictable for an adversary within a feasible amount of computations.

- Many implementations of CPRNGs are available.
- Stream ciphers are supposed to be CPRNGs.
- Some are based on hash functions, block ciphers, or number-theoretic hard problems.

Blum Blum Shub PRNG

- The Blum-Blum Shub PRNG is cryptographically secure if the problem of factoring large composite numbers is hard.
- It works as follows :
  - **Initialization** : Generate two sufficiently large prime numbers p and q that are congruent to 3 (mod 4), compute $n = p \cdot q$ and throw away p and q.
  - **Random Bit Sequence Generation** : Given a seed x_0 uniformly distributed on $[2, n-1]$, compute the sequence $x_i = x_{i-1}^2 \bmod n$ and output the bit $\mathrm{lsb}(x_i)$.
- The Blum Blum Shub PRNG is **extremely slow** in practice.

Blum-Micali PRNG

- The Blum-Micali PRNG is cryptographically secure if the discrete logarithm problem is hard.
- It works as follows :
  - **Initialization** : Generate a sufficiently large prime number p and find g a generator of $\mathbb{Z}_p^*$.
  - **Random Bit Sequence Generation** : Given a seed x_0 uniformly distributed on $[2, p-1]$, compute the sequence $x_{i+1} = g^{x_i} \bmod p$ and output 1 if $x_i \leq \frac{p-1}{2}$ and 0 otherwise.
- The Blum-Micali PRNG is **extremely slow** in practice.

Forward and Backward Security

Forward Security (PRNG)

The leakage of the inner state of the PRNG should not compromise the **previously** outputted random bits.

Backward Security (PRNG)

It should be possible for future outputs to be secure if current state is compromised.

Backward Security

Backward Security requires injecting fresh entropy.

- **Seeding** operation
- **Reseeding** operation
- **Value generation**

Sources of Entropy

- To seed a PRNG, we need **entropy sources**
- It is also needed to perform reseeding operations.

Question

On a computer, what would you use as entropy source ?

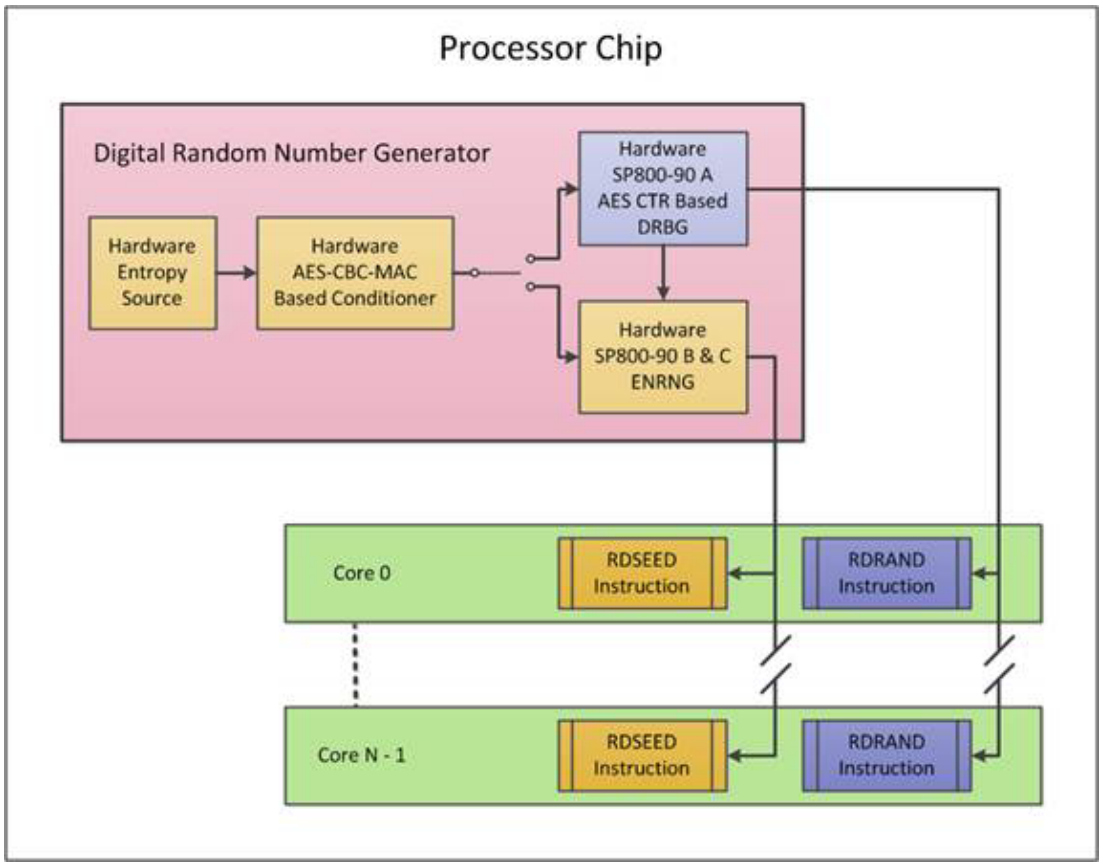
/dev/random and /dev/urandom

- On Unix-like operating systems, `/dev/random` is a special file that acts as a cryptographically secure PRNG. It collects environmental noise from device drivers (events related to disks, network, mouse, etc.)
- Since kernel 5.6, `/dev/random` is only blocking until the CPRNG is properly initialized.
- `/dev/urandom` is a **non-blocking** variant of `/dev/random`. It means that it outputs pseudorandom bytes, even if insufficient randomness is available. In extreme cases, it can become predictable.
- The presence/absence of both `/dev/random` and `/dev/urandom` and their implementation is varying among the different flavours and versions of operating systems.

Intel's RDSEED and RDRAND

- Embedded CPRNG on Intel CPUs.
- AMD decided to implement the same instructions.
- Two instructions are available :
  - RDRAND : get high quality random data. PRNG regularly reseeded. Faster.
  - RDSEED : get a high quality random seed for a software CPRNG. TRNG. Slower.

Intel's RDSEED and RDRAND



Source : https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide

RNRAND/RDSEED is not perfect

Documentation of RDSEED

The **Carry Flag** indicates whether a random value **is available** at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width.

- 2020 : CrossTalk vulnerability. Allows to read values of RDRAND, RDSEED across cores.
- Side-channel attack.
- Mitigations -> 3% of original speed

Pool of Entropy

- An operating system typically keeps an entropy pool which is regularly fed with fresh entropy.

Question

Linux XORs the fresh entropy into the pool. Is it a good idea ?
What if an attacker can access the RAM ?

Mixing Different Sources

Solution

Using a simple XOR works but is not the best idea. First, it is important to notice that if the attacker cannot read the value of the pool, entropy cannot decrease in the pool. However, someone generating random values and having access to the RAM can control its random value to control the whole pool. It is better to use a hash function, for instance, using $H(\text{old}\|\text{new})$ as a new pool (which is used for instance in OpenSSL).

NIST SP800-90A

- NIST standardized 4 (in fact 3) deterministic random byte generators (DRBG).
- We present here only the **general picture**.

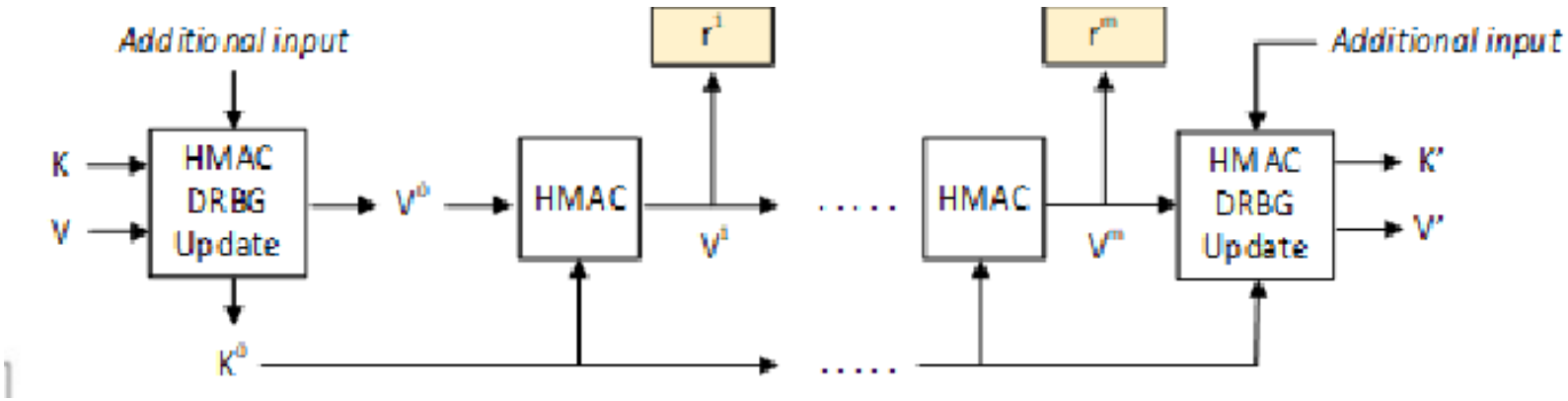
Warning

Read the standard if you want to implement it.

Hash_DRBG

- Based on the use of a hash function.
- Idea : $w = H(\text{state})$. state $=$ state $+ w$.
- Supports SHA-1 and SHA-2.
- Standardized in NIST SP800-90A (§10.1.1)

HMAC_DRBG



Source : http://ijns.jalaxy.com.tw/contents/ijns-v23-n1/ijns-2021-v23-n1-p33-41.pdf

HMAC_DRBG

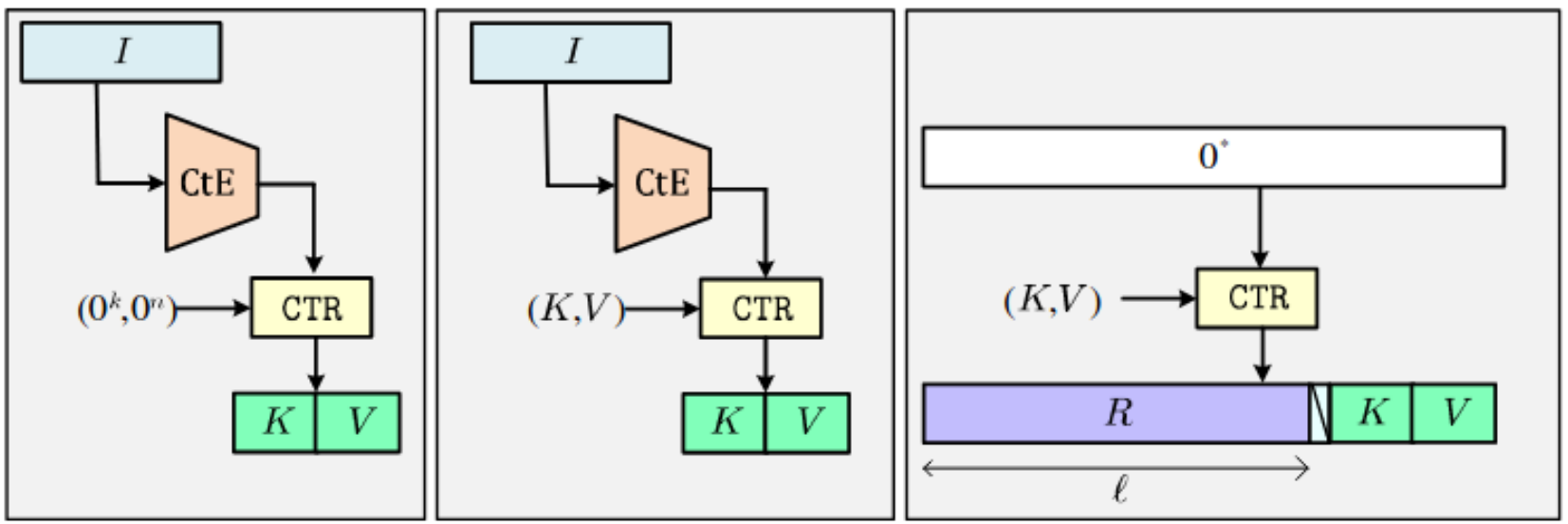
- Based on HMAC :
- Idea : Obtain key K and state V from randomness. To generate, $V \leftarrow \text{HMAC}(K, V)$. Update V and K using HMAC and additional randomness.
- Standardized in NIST SP800-90A (§10.1.2)
- Slower than Hash_DRBG but more secure.

CTR_DRBG

- Based on the use of a block cipher operated in counter mode
- Supports either 3-key Triple-DES or AES-128, AES-192 and AES-256.
- Idea : Obtain key K and state V from randomness. To generate, use CTR with key K on state V to obtain random values. Update V and K using CTR.
- Standardized in NIST SP800-90A (§10.1.3)

CTR_DRBG

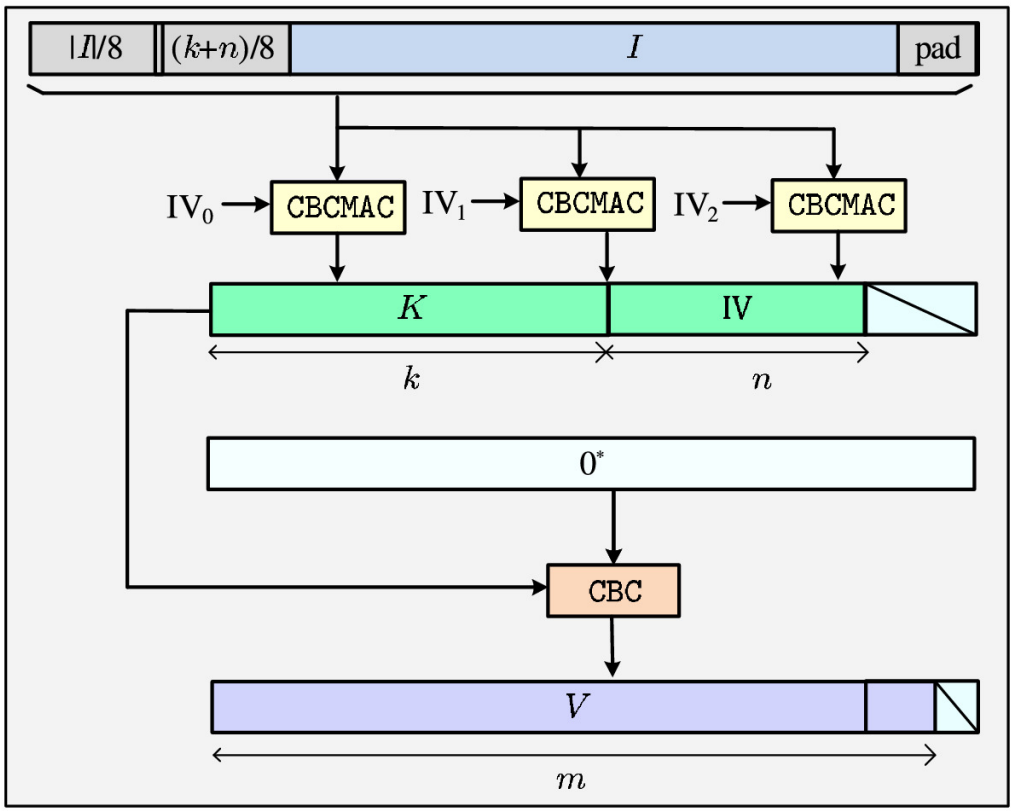
procedure setup$^E(I)$	**procedure** refresh$^E(S, I)$	**procedure** next$^E(S, \ell)$
$X \leftarrow \mathrm{CtE}[E](I)$	$X \leftarrow \mathrm{CtE}[E](I)$	$K \leftarrow S[1:k];\ V \leftarrow S[k+1:k+n]$
$K \leftarrow 0^k;\ \mathrm{IV} \leftarrow 0^n$	$K \leftarrow S[1:k]$	$r \leftarrow n \cdot \lceil \ell/n \rceil$
$S \leftarrow \mathsf{CTR}_K^{\mathrm{IV}}[E](X)$	$V \leftarrow S[k+1:k+n]$	$P \leftarrow \mathsf{CTR}_K^V[E](0^{r+k+n})$
return S	$S \leftarrow \mathsf{CTR}_K^V[E](X)$	$R \leftarrow P[1:\ell]$
	return S	$S \leftarrow P[r+1:r+k+n]$
		return (R, S)



Source : Hoang and Shen, Crypto 2020

CTR_DRBG : CtE

procedure $\mathrm{CtE}[E, m](I)$
$X \leftarrow \mathrm{pad}\big([|I|/8]_{32} \,\|\, [(k+n)/8]_{32} \,\|\, I\big)$
for $i \leftarrow 0$ **to** 2 **do**
  $\mathrm{IV}_i \leftarrow \pi([i]_{32} \,\|\, 0^{n-32})$; $T_i \leftarrow \mathsf{CBCMAC}^{\mathrm{IV}}[\pi](X)$
$Y \leftarrow T_1 \,\|\, T_2 \,\|\, T_3$; $K \leftarrow Y[1:k]$; $\mathrm{IV} \leftarrow Y[k+1:k+n]$
$C \leftarrow \mathsf{CBC}^{\mathrm{IV}}_K[E](0^{3n})$; **return** $C[1:m]$



Source : Hoang and Shen, Crypto 2020

DUAL_EC_DRBG

- Published by NIST in 2006 in NIST SP-800-90A in collaboration with NSA. Removed in 2012.
- Only proposal in NIST SP-800-90A based on mathematics $\rightarrow$ thousand time slower.
- Uses elliptic curves with parameters P, Q coming out of nowhere.
- Too many bits are output : 0.1% bias guessing the next bit.
- Potential **backdoor** : knowing e such that $Q = eP$ would allow to predict all the future outputs.

Randomness Generation in Practice

- **C/Linux :system** : `getrandom()` system call (`getentropy()` on BSD). Reads from `urandom` if has a high enough entropy level. Non blocking afterwards.
- **C++/Windows** : `BCryptGenRandom`. Possibility to choose algorithm via **providers**. Default : `CTR_DRBG`.
- **C/C++ libraries** : OpenSSL : `RAND_bytes` (hard to use correctly). Libsodium : `https://libsodium.gitbook.io/doc/generating_random_data`
- **Java** : `java.security.SecureRandom`. Possibility to choose algorithm via providers. Default : `SHA1PRNG`.
- **Python** : `secrets` module. Uses underlying OS RNG.
- **Rust** : `rand::rngs::OSRng`. Takes randomness from OS. `rand::rngs::StdRng` is a CPRNG. Uses currently Chacha20.

Warning

Warning

In all these cases, **read** the documentation and **check** the return values !

Generating Random Numbers

- Typically, a CPRNG will deliver various amounts of random bytes. This is fine when generating parameters for symmetric cryptography.
- However, when dealing with public-key algorithms, one is often asked to generate uniformly distributed random numbers that are strictly smaller than an arbitrary number n.
- Going from random bytes to a random number is a delicate operation.

Generating Random Numbers

Question

How would you generate a random integer modulo 18 (i.e., a random number in $\mathbb{Z}_{18}$)?

Wrong Methods to Generate Random Numbers

We want to generate a uniform number smaller than n, where n is an ℓ-bit number.

- one generates an ℓ-bit number and reduces it modulo n ;
- one generates an ℓ-bit number r and if $r > n$, then one clears the most significant bit ;
- one generates an $\ell - 1$-bit number (i.e., computing $r \bmod 2^{\ell-1}$).

The Right Way : the Rejection Method

- The right way is to employ a **rejection method**. To generate a number $r < n$ uniformly at random, where n is an ℓ-bit number, one proceeds as follows :
  1. Generate an ℓ-bit string r uniformly at random.
  2. If $r < n$, then output it. Otherwise, jump to 1.
- The loop will be taken a variable number of times, but it will quickly finish with high probability.

Question

What is the probability of failure in the worst case for one iteration ?

The Debian Fiasco

- From 2006 to 2008 on Debian : completely broken OpenSSL PRNG.
- Only 32768 possible SSH keys.
- Combination of **bad software design** and **too clever undocumented tricks**.

OpenSSL PRNG

- Gathers entropy from different sources (e.g. `/dev/urandom`) and combines them using a hash function.
- Hash function is using Merkle-Damgård construction. Buffer is compressed into state : `MD_Update(&m,buf,j);`
- `RAND_add(buf, n, e)` function : adds a buffer of size n into the state with e bits of entropy.
- Allows to update the **entropy estimate**.

Problematic Code ?

```
char buf[100];
fd = open("/dev/random", O_RDONLY);
n = read(fd, buf, sizeof buf);
close(fd);
RAND_add(buf, sizeof buf, n);
```

Other Occurrence

```
i=fread(buf,1,n,in);
if (i <= 0) break;
/* even if n != i, use the full array */
RAND_add(buf,n,i);
```

Obtaining an Output

- Function `RAND_bytes(buf, sizeof buf)`.
- Adds the values of the buffer to the entropy pool before obtaining output.
-

```
char buf[100];
n = RAND_bytes(buf, sizeof buf);
```

- Valgrind got crazy : conditional jump based on an uninitialized value.

```
When debbuging applications that make use of openssl using
valgrind, it can show alot of warnings about doing a conditional
jump based on an unitialised value.  Those unitialised values are
generated in the random number generator.  It's adding an
unintialiased buffer to the pool.

The code in question that has the problem are the following 2
pieces of code in crypto/rand/md_rand.c:

247:
                MD_Update(&m,buf,j);

467:
#ifndef PURIFY
                MD_Update(&m,buf,j); /* purify complains */
#endif

...

What I currently see as best option is to actually comment out
those 2 lines of code.  But I have no idea what effect this
really has on the RNG.  The only effect I see is that the pool
might receive less entropy.  But on the other hand, I'm not even
sure how much entropy some unitialised data has.

What do you people think about removing those 2 lines of code?
```

Answers

OpenSSL dev :

```
> But on the other hand, I'm not even
> sure how much entropy some unitialised data has.
>
Not much. If it helps with debugging, I'm in favor of removing them.
(However the last time I checked, valgrind reported thousands of bogus
error messages. Has that situation gotten better?)
```

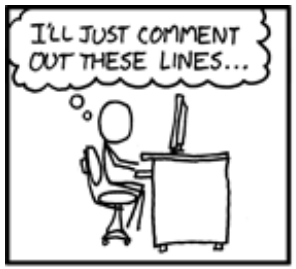
Not an OpenSSL dev :

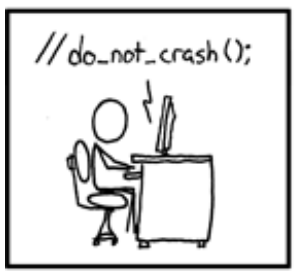
```
I recently compiled vanilla OpenSSL 0.9.8a with -DPURIFY=1 and on Debian
GNU/Linux 'sid' with valgrind version 3.1.1 was able to debug some
application using both TLS/SSL as S/MIME without any warning or error
about the OpenSSL code. Without -DPURIFY you're indeed flooded with
warnings.

So yes I think not using the uninitialized memory (it's only a single
line, the other occurrence is already commented out) helps valgrind
```

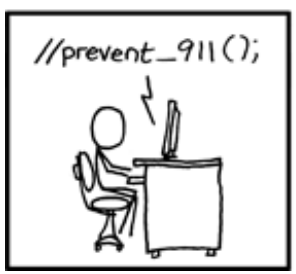
Result

- The only randomness introduced in the PRNG is the PID of the process.
- The problem comes from partially documented clever code.
- We also had duplicated code with different comments
- More infos on `https://research.swtch.com/openssl`

Conclusion







IN THE RUSH TO CLEAN UP THE DEBIAN-OPENSSL FIASCO, A NUMBER OF OTHER MAJOR SECURITY HOLES HAVE BEEN UNCOVERED:

AFFECTED SYSTEM	SECURITY PROBLEM
FEDORA CORE	VULNERABLE TO CERTAIN DECODER RINGS
XANDROS (EEE PC)	GIVES ROOT ACCESS IF ASKED IN STERN VOICE
GENTOO	VULNERABLE TO FLATTERY
OLPC OS	VULNERABLE TO JEFF GOLDBLUM'S POWERBOOK
SLACKWARE	GIVES ROOT ACCESS IF USER SAYS ELVISH WORD FOR "FRIEND"
UBUNTU	TURNS OUT DISTRO IS ACTUALLY JUST WINDOWS VISTA WITH A FEW CUSTOM THEMES

https://www.xkcd.com/424