How Do We Know Our PRNGs Are Working Properly?

Felix Dörre and Vladimir Klebanov

Work supported by Karlsruhe Institute of Technology and the
DFG priority program “Reliably Secure Software Systems”

- BIND9
- OpenSSH (server, user keys)
- OpenVPN, Openswan, StrongSWAN, tinc
- DNSSEC
- X.509
- Kerberos (Heimdal)
- encfs
- Tor
- postfix, exim4, sendmail
- cyrus imapd, uw-imapd, courier
- apache2 (ssl certs)
- cfengine, puppet
- xrdp
- gitosis
- pwsafe
- vsftpd, proftpd, ftpd-ssl
- telnetd-ssl
- DomainKeys, DKIM

Services Affected by the Debian OpenSSL Disaster (2006–2008)

- BIND9
- OpenSSH (server, user keys)
- OpenVPN, Openswan, StrongSWAN, tinc
- DNSSEC
- X.509
- Kerberos (Heimdal)
- encfs
- Tor
- postfix, exim4, sendmail
- cyrus imapd, uw-imapd, courier
- apache2 (ssl certs)
- cfengine, puppet
- xrdp
- gitosis
- pwsafe
- vsftpd, proftpd, ftpd-ssl
- telnetd-ssl
- DomainKeys, DKIM

https://wiki.debian.org/SSLkeys

The “Technical” Consequence

```
/* DO NOT REMOVE THE FOLLOWING CALL TO MD_Update()! */
if (!MD_Update(m, buf, j))
    goto err;
/*
 * We know that line may cause programs such as purify and valgrind
 * to complain about use of uninitialized data.  The problem is not,
 * it's with the caller.  Removing that line will make sure you get
 * really bad randomness and thereby other problems such as very
 * insecure keys.
 */
```

A Tour of Implementations

A Tour of Implementations

```
// HASHBYTES_TO_USE defines # of bytes returned by "computeHash(byte[])"
// to use to form byte array returning by the "nextBytes(byte[])" method
// Note, that this implementation uses more bytes than it is defined
// in the above specification.
```

A Tour of Implementations

/* Put the data into the entropy, add some data from the unknown state, reseed */

A Tour of Implementations

/* Put the data into the entropy, add some data from the unknown state, reseed */

/* Take some ”random” data and make more ”random-looking” data from it */

A Tour of Implementations

/* Put the data into the entropy, add some data from the unknown state, reseed */

/* Take some ”random” data and make more ”random-looking” data from it */

/* Stupid C trick */

A Tour of Implementations

/* Put the data into the entropy, add some data from the unknown state, reseed */

/* Take some ”random” data and make more ”random-looking” data from it */

/* Stupid C trick */

/* This fails silently and must be fixed. */

A Tour of Implementations

/* Put the data into the entropy, add some data from the unknown state, reseed */

/* Take some ”random” data and make more ”random-looking” data from it */

/* Stupid C trick */

/* This fails silently and must be fixed. */

Be very careful when using this function to ensure that you do not produce a poor output state. (end-user documentation)

A Tour of Implementations

/* Put the data into the entropy, add some data from the unknown state, reseed */

/* Take some ”random” data and make more ”random-looking” data from it */

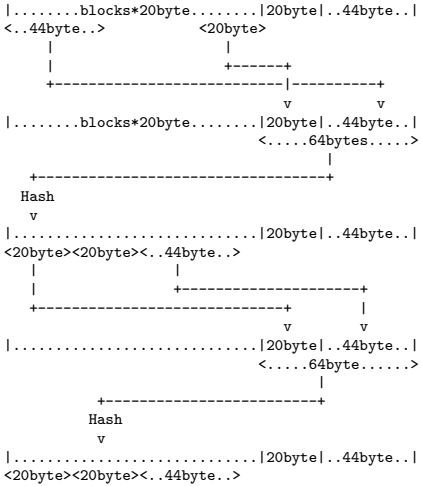
/* Stupid C trick */

/* This fail

Mind the supply chain!

Be very careful when using this function to ensure that you do not produce a poor output state. (end-user documentation)

A Tour of Implementations



A Tour of Implementations

Have a (better) design document!

Quality Assurance Measures for PRNGs

Quality Assurance Measures for PRNGs

– System-level tests and functional verification

Quality Assurance Measures for PRNGs

- System-level tests and functional verification
- Unit-level tests and functional verification

Quality Assurance Measures for PRNGs

– Statistical tests (DIEHARD, NIST SP800-22, etc.)

Quality Assurance Measures for PRNGs

- Statistical tests (DIEHARD, NIST SP800-22, etc.)
- Regression tests (in particular NIST SP 800-90)

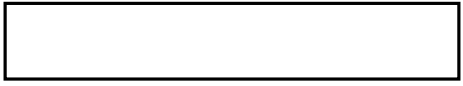
Quality Assurance Measures for PRNGs

– Manual code review

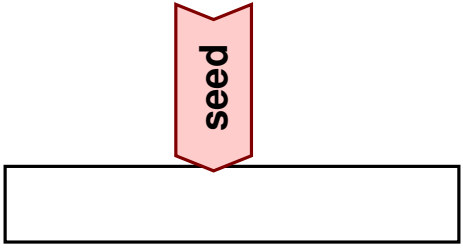
Our Contribution

- Identification of a common PRNG defect: **entropy loss**
- **Entroposcope** – a static analysis tool for detecting entropy loss in real C and Java PRNGs

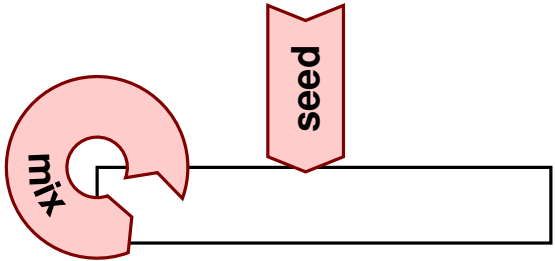
PRNG Operation Cycle (Simplified)



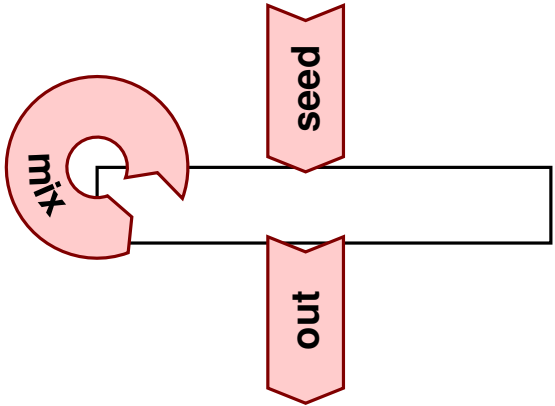
PRNG Operation Cycle (Simplified)



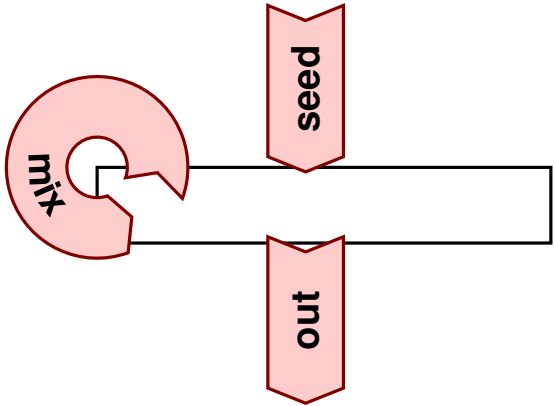
PRNG Operation Cycle (Simplified)



PRNG Operation Cycle (Simplified)

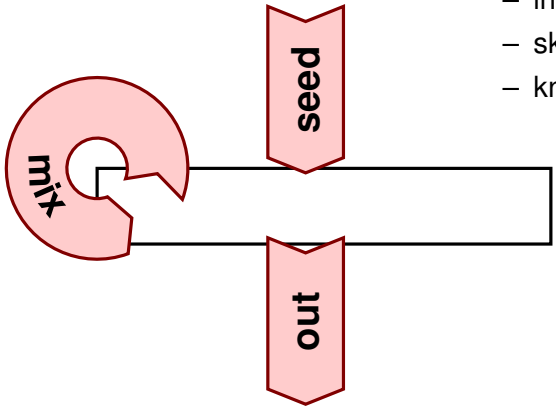


PRNG Operation Cycle (Simplified)



We treat a PRNG as a function $g\colon \{0,1\}^m \to \{0,1\}^n$

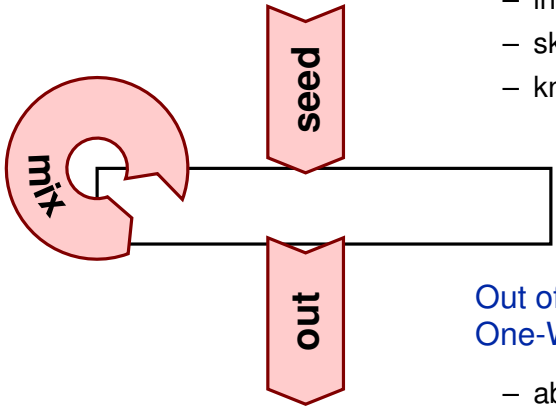
PRNG Operation Cycle (Simplified)



Out of Scope: Bad Seeds

- insufficient range (a priori)
- skewed distribution
- known to attacker

PRNG Operation Cycle (Simplified)



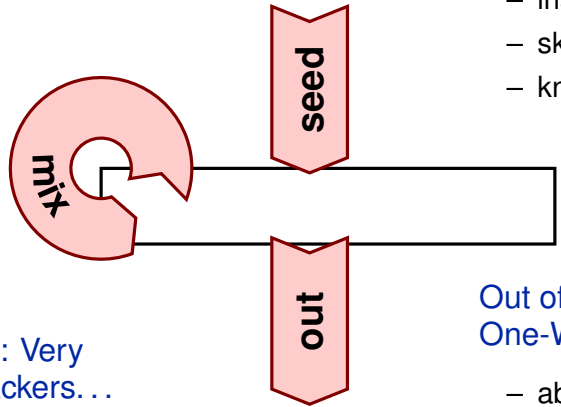
Out of Scope: Bad Seeds

- insufficient range (a priori)
- skewed distribution
- known to attacker

Out of Scope: One-Way Functions. . .

- absent or insufficient
- not one-way (e.g., Dual_EC_DRBG)

PRNG Operation Cycle (Simplified)



Out of Scope: Bad Seeds

- insufficient range (a priori)
- skewed distribution
- known to attacker

Out of Scope: Very Powerful Attackers. . .

- inspecting/corrupting PRNG state

Out of Scope: One-Way Functions. . .

- absent or insufficient
- not one-way (e.g., Dual_EC_DRBG)

Entropy Loss

The following are equivalent

- PRNG suffers from entropy loss
- Part of seed does not influence output
- Two seeds produce the same output
- Fewer possible outputs than seeds
- g is not injective

Entropy Loss

The following are equivalent

- PRNG suffers from entropy loss
- Part of seed does not influence output
- Two seeds produce the same output
- Fewer possible outputs than seeds
- g is not injective

Formally

A PRNG $g\colon \{0,1\}^m \to \{0,1\}^n$ suffers from entropy loss iff

$$\exists seed_1, seed_2.\ \left(seed_1 \neq seed_2 \wedge g(seed_1) = g(seed_2)\right)\ .$$

Entropy Loss

The following are equivalent

- PRNG suffers from entropy loss
- Part of seed does not influence output
- Two seeds produce the same output
- Fewer possible outputs than seeds
- g is not injective

Formally

A PRNG $g\colon \{0,1\}^m \to \{0,1\}^n$ suffers from entropy loss iff

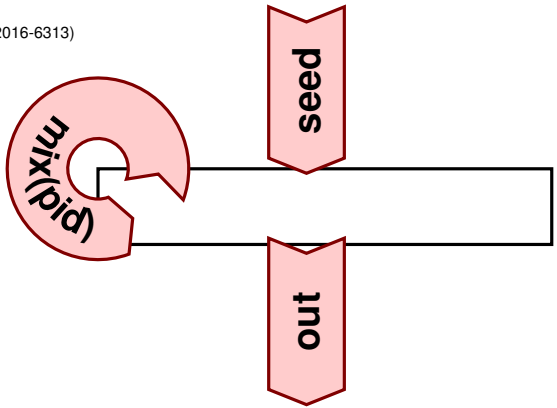
$$\exists seed_1, seed_2.\ \left(seed_1 \neq seed_2 \wedge g(seed_1) = g(seed_2)\right)\ .$$

Reasoning complicated by use of crypto functions inside g.

Instances of Entropy Loss

In well-known software

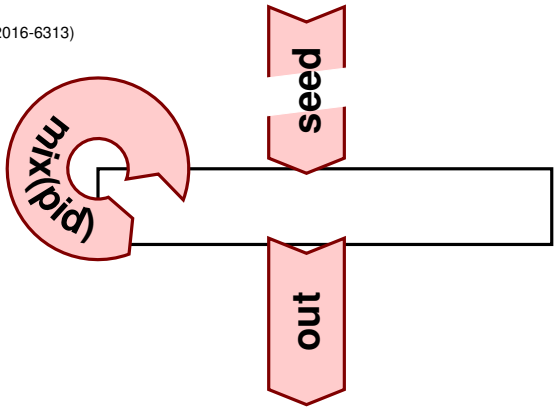
- Debian OpenSSL disaster (CVE-2008-0166)
- Android PRNG bug (CVE-2013-7372)
- Libgcrypt / GnuPG bug (CVE-2016-6313)



Instances of Entropy Loss

In well-known software

- Debian OpenSSL disaster (CVE-2008-0166)
- Android PRNG bug (CVE-2013-7372)
- Libgcrypt / GnuPG bug (CVE-2016-6313)

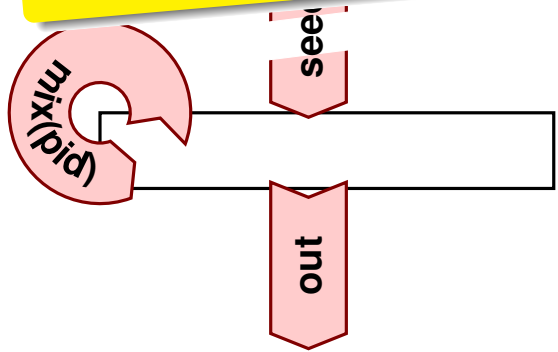


Instances of Entropy Loss

In well-known software

- Debian OpenSSL disaster (CVE-2008-0166)
- Android PRNG bug (CVE-2013-737…
- Libgcrypt / GnuPG bug (CVE-201…

"Easy" instance of entropy loss—
detectable even with gdb (read watchpoint)

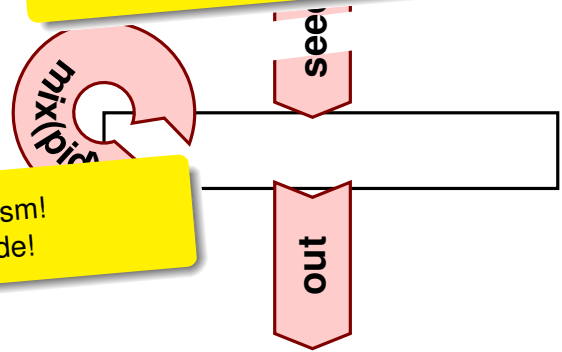


Instances of Entropy Loss

In well-known software

- Debian OpenSSL disaster (CVE-2008-0166)
- Android PRNG bug (CVE-2013-737…
- Libgcrypt / GnuPG bug (CVE-201…

"Easy" instance of entropy loss—
detectable even with gdb (read watchpoint)

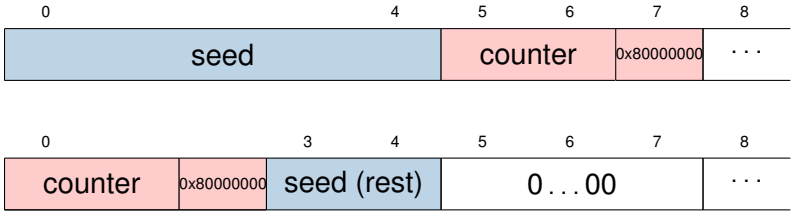


Confine non-determinism!
Test deterministic code!

Instances of Entropy Loss

In well-known software

- Debian OpenSSL disaster (CVE-2008-0166)
- Android PRNG bug (CVE-2013-7372)
- Libgcrypt / GnuPG bug (CVE-2016-6313)



Instances of Entropy Loss

In well-known software

- Debian OpenSSL disaster (CVE-2008-0166)
- Android PRNG bug (CVE-2013-7372)
- Libgcrypt / GnuPG bug (CVE-2016-6313)

later

Analysis Procedure

1. User isolates the deterministic part of PRNG

Analysis Procedure

1. User isolates the deterministic part of PRNG
2. User chooses analysis scope (m,n)

Analysis Procedure

1. User isolates the deterministic part of PRNG
2. User chooses analysis scope (m,n)
3. User replace crypto functions with idealizations

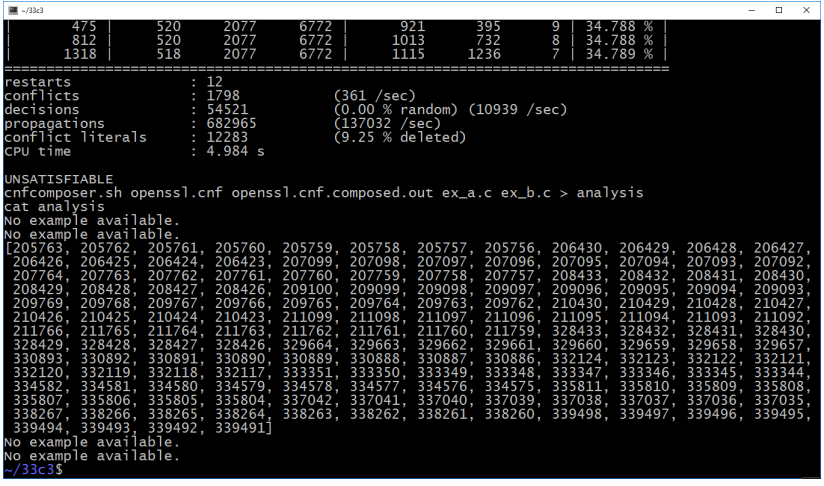
Analysis Procedure

1. User isolates the deterministic part of PRNG
2. User chooses analysis scope (m,n)
3. User replace crypto functions with idealizations
4. Entroposcope generates & checks verification condition

Analysis Procedure

1. User isolates the deterministic part of PRNG
2. User chooses analysis scope (m,n)
3. User replace crypto functions with idealizations
4. Entroposcope generates & checks verification condition
5. If potential entropy loss found, visualization

Demo


~/33c3
475	520 2077 6772	921 395 9	34.788 %
812	520 2077 6772	1013 732 8	34.788 %
1318	518 2077 6772	1115 1236 7	34.789 %
restarts : 12
conflicts : 1798 (361 /sec)
decisions : 54521 (0.00 % random) (10939 /sec)
propagations : 682965 (137032 /sec)
conflict literals : 12283 (9.25 % deleted)
CPU time : 4.984 s
UNSATISFIABLE
cnfcomposer.sh openssl.cnf openssl.cnf.composed.out ex_a.c ex_b.c > analysis
cat analysis
No example available.
No example available.
[205763, 205762, 205761, 205760, 205759, 205758, 205757, 205756, 206430, 206429, 206428, 206427,
206426, 206425, 206424, 206423, 207099, 207098, 207097, 207096, 207095, 207094, 207093, 207092,
207764, 207763, 207762, 207761, 207760, 207759, 207758, 207757, 208433, 208432, 208431, 208430,
208429, 208428, 208427, 208426, 209100, 209099, 209098, 209097, 209096, 209095, 209094, 209093,
209769, 209768, 209767, 209766, 209765, 209764, 209763, 209762, 210430, 210429, 210428, 210427,
210426, 210425, 210424, 210423, 211099, 211098, 211097, 211096, 211095, 211094, 211093, 211092,
211766, 211765, 211764, 211763, 211762, 211761, 211760, 211759, 328433, 328432, 328431, 328430,
328429, 328428, 328427, 328426, 329664, 329663, 329662, 329661, 329660, 329659, 329658, 329657,
330893, 330892, 330891, 330890, 330889, 330888, 330887, 330886, 332124, 332123, 332122, 332121,
332120, 332119, 332118, 332117, 333351, 333350, 333349, 333348, 333347, 333346, 333345, 333344,
334582, 334581, 334580, 334579, 334578, 334577, 334576, 334575, 335811, 335810, 335809, 335808,
335807, 335806, 335805, 335804, 337042, 337041, 337040, 337039, 337038, 337037, 337036, 337035,
338267, 338266, 338265, 338264, 338263, 338262, 338261, 338260, 339498, 339497, 339496, 339495,
339494, 339493, 339492, 339491]
No example available.
No example available.
~/33c3$

Crypto Function Idealization (Example)

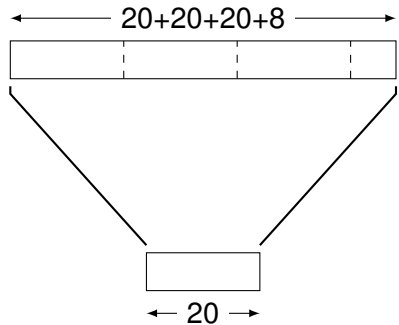
From OpenSSL's `RAND_add()`:

```
sha1(local_md[0..19] | state[0..19] | buf[0..19] | md_count)
```

Crypto Function Idealization (Example)

From OpenSSL's `RAND_add()`:

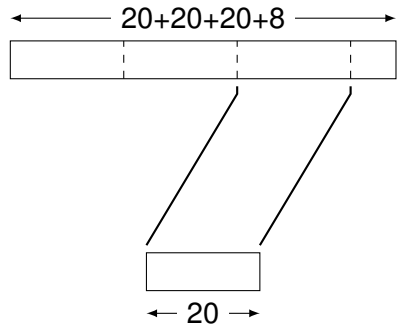
```
sha1(local_md[0..19] | state[0..19] | buf[0..19] | md_count)
```



Crypto Function Idealization (Example)

From OpenSSL's `RAND_add()`:

```
sha1(local_md[0..19] | state[0..19] | buf[0..19] | md_count)
```



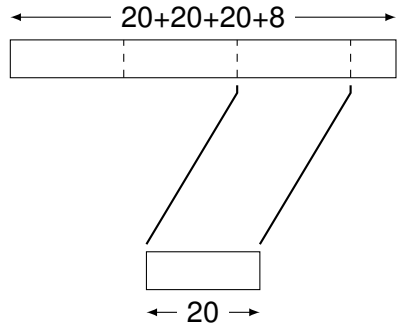
Crypto Function Idealization (Example)

From OpenSSL's `RAND_add()`:

```
sha1(local_md[0..19] | state[0..19] | buf[0..19] | md_count)
```

Unsound idealization (useful!)

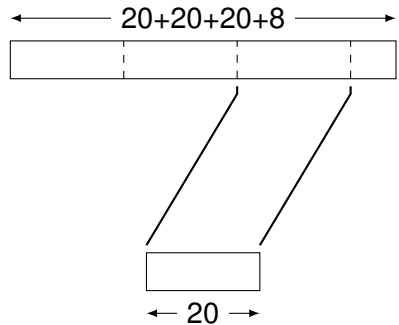
`memcpy()`



Crypto Function Idealization (Example)

From OpenSSL's `RAND_add()`:

```
sha1(local_md[0..19] | state[0..19] | buf[0..19] | md_count)
```



Unsound idealization (useful!)

`memcpy()`

Sound idealization

Underspecified injective function

Implementation

based on

CBMC
bounded model checker
for C and Java
cprover.org

MINISAT
boolean satisfiability checker

minisat.se

Implementation

– CBMC computes $g(\cdot)$ as propositional formula

Implementation

- CBMC computes $g(\cdot)$ as propositional formula
- Entroposcope generates

$$seed_1 \neq seed_2 \wedge g(seed_1) = g(seed_2) \qquad (*)$$

Implementation

- CBMC computes $g(\cdot)$ as propositional formula
- Entroposcope generates

$$seed_1 \neq seed_2 \wedge g(seed_1) = g(seed_2) \qquad (*)$$

- MINISAT checks $(*)$ for satisfiability

Implementation

- CBMC computes $g(\cdot)$ as propositional formula
- Entroposcope generates

$$seed_1 \neq seed_2 \wedge g(seed_1) = g(seed_2) \qquad (*)$$

- MINISAT checks $(*)$ for satisfiability
- If $(*)$ has a model, Entroposcope visualizes the entropy loss

Implementation

- CBMC computes $g(\cdot)$ as propositional formula
- Entroposcope generates

$$seed_1 \neq seed_2 \wedge g(seed_1) = g(seed_2) \qquad (*)$$

- MINISAT checks $(*)$ for satisfiability
- If $(*)$ has a model, Entroposcope visualizes the entropy loss

Analysis duration ~30s

Results of Applying the Tool

- BoringSSL
- Yarrow (Apple XNU port)
- s2n
- Android PRNG (Apache Harmony)
- OpenSSL
- Libgcrypt / GnuPG

Results of Applying the Tool

- BoringSSL
- Yarrow (Apple XNU port)
- s2n
- Android PRNG (Apache Harmony)
- OpenSSL
- Libgcrypt / GnuPG

The good cases

No entropy loss detected in analysis scope

Results of Applying the Tool

- BoringSSL
- Yarrow (Apple XNU port)
- s2n
- Android PRNG (Apache Harmony)
- OpenSSL
- Libgcrypt / GnuPG

Android PRNG (Apache Harmony)

Known entropy loss detected

Results of Applying the Tool

- BoringSSL
- Yarrow (Apple XNU port)
- s2n
- Android PRNG (Apache Harmony)
- OpenSSL
- Libgcrypt / GnuPG

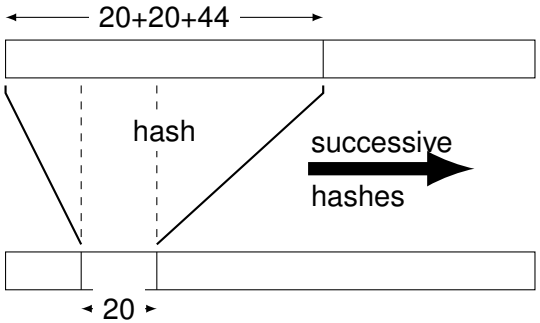
OpenSSL

- Debian disaster detected
- Entropy loss by design detected
- Previously unknown entropy loss detected

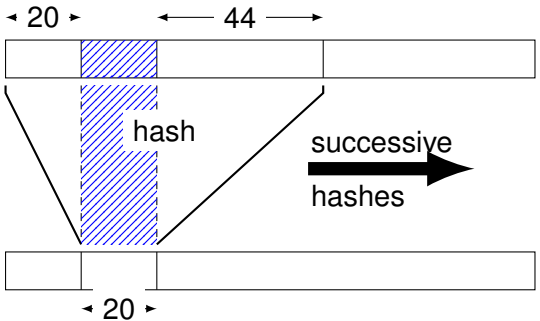
Results of Applying the Tool

- BoringSSL
- Yarrow (Apple XNU port)
- s2n
- Android PRNG (Apache Harmony)
- OpenSSL
- Libgcrypt / GnuPG

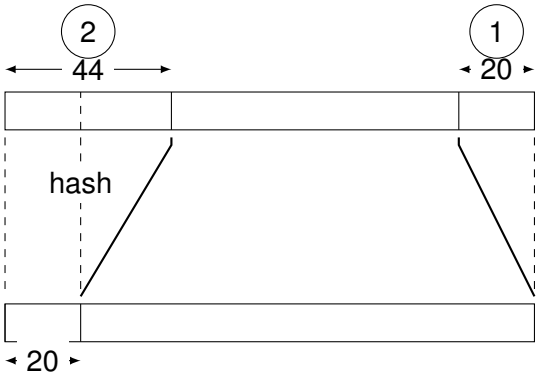
Mixing Function Proposal by Gutmann [USENIX '98]



Mixing Function Implemented in Libgcrypt (1)



Mixing Function Implemented in Libgcrypt (2)



Conclusion


GNU Privacy Guard
@gnupg
Follow
@dguido FWIW, this small function had external audits several times and nobody spotted the algorithmic problem.
8:39 PM - 17 Aug 2016
1

Conclusion

Audits are essential—
but so are technical assurance measures!

Thanks!

Questions?

Why is a Partial Predictability Bad?

If an attacker has access to the first 580 byte, wouldn't they have access to the following 20 as well?

- Random data used for different purposes
- Help for a brute-force attacker

What is the Impact?

For practical impact, more than 600 bytes have to be requested in one chunk (as the output pool is emptied after each request).

Impact on GnuPG RSA keys:

- GnuPG requests random data for RSA keys in several small(er) chunks
- Keys shorter than 4096 bits are probably fine

Impact on other applications using the Libgcrypt PRNG:

- Impossible to tell

Just Use /dev/urandom!

Fixing flaws does not take away other options.

How do you know the kernel PRNG is bug-free?

All entropy-processing applications are susceptible.

Just Use /dev/urandom!

Fixing flaws does not take away other options.

How do you know the kernel PRNG is bug-free?

All entropy-processing applications are susceptible.

- Linux ASLR bug (CVE-2015-1593)
- Early German debit card system flaw (“EC-Karte”)
- ASF Software Inc. online poker software flaw

Entropy Loss in Linux ASLR (CVE-2015-1593)

```
1  static unsigned long randomize_stack_top(
2                           unsigned long stack_top)
3  {
4    unsigned int random_variable = 0;
5
6    if ((current->flags & PF_RANDOMIZE) &&
7      !(current->personality & ADDR_NO_RANDOMIZE)) {
8      random_variable = get_random_int() & STACK_RND_MASK;
9      random_variable <<= PAGE_SHIFT;
10   }
11 #ifdef CONFIG_STACK_GROWSUP
12   return PAGE_ALIGN(stack_top) + random_variable;
13 #else
14   return PAGE_ALIGN(stack_top) - random_variable;
15 #endif
16 }
```