DESIGN AND ANALYSIS OF RANDOM NUMBER GENERATOR

Student Name: HIMANI
Roll Number: 2013040

Student Name: SHUBHAM SHARMA
Roll Number: 2013100

BTP report submitted in partial fulfillment of the requirements
for the Degree of B.Tech. in Computer Science & Engineering

on 18th April, 2017

BTP Track: Research Track

BTP Advisor
Donghoon Chang

Indraprastha Institute of Information Technology
New Delhi

Student's Declaration

We hereby declare that the work presented in the report entitled **"Design And Analysis of Random Number Generator"** submitted by us for the partial fulfillment of the requirements for the degree of *Bachelor of Technology* in *Computer Science & Engineering* at Indraprastha Institute of Information Technology, Delhi, is an authentic record of our work carried out under guidance of **Prof. Donghoon Chang**. Due acknowledgements have been given in the report to all material used. This work has not been submitted anywhere else for the reward of any other degree.

Himani
Shubham Sharma

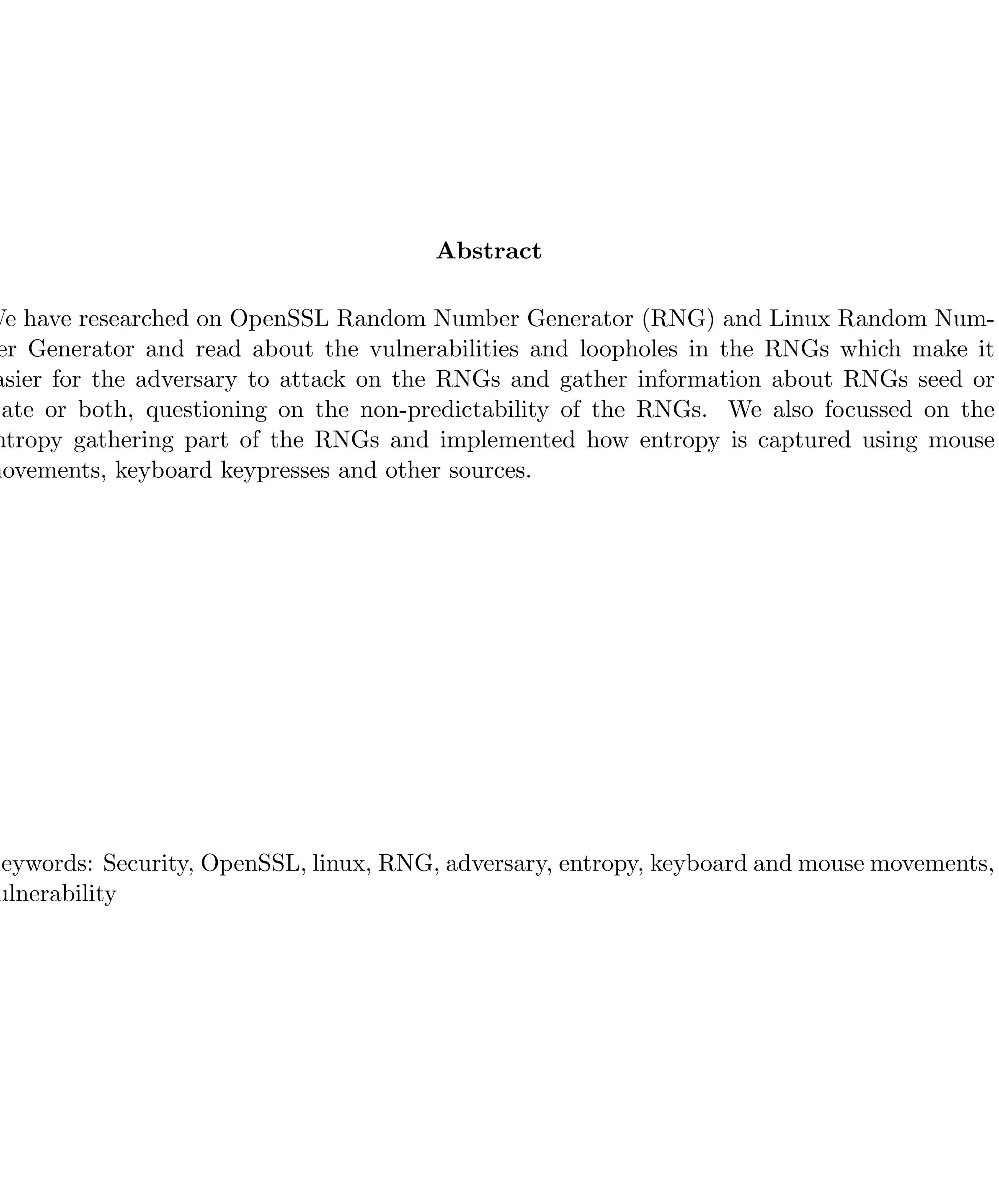
Place & Date: New Delhi, 18th April, 2017

Certificate

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

....
Donghoon Chang

Place & Date: New Delhi, 18th April, 2017



Abstract

We have researched on OpenSSL Random Number Generator (RNG) and Linux Random Number Generator and read about the vulnerabilities and loopholes in the RNGs which make it easier for the adversary to attack on the RNGs and gather information about RNGs seed or state or both, questioning on the non-predictability of the RNGs. We also focussed on the entropy gathering part of the RNGs and implemented how entropy is captured using mouse movements, keyboard keypresses and other sources.

Keywords: Security, OpenSSL, linux, RNG, adversary, entropy, keyboard and mouse movements, vulnerability

Acknowledgments

We would like to express our gratitude to Prof Donghoon Chang who supported our research and provided insight, expertise and comments on our work and for providing us the right direction. We thank our mentors, Amit Kumar Chauhan, Naina Gupta and Arpan Jati who greatly assisted the research and implementation.

Contents

1 Introduction **1**

1.1 Types of Random Numbers 1

1.2 Measure of Randomness: Entropy 1

1.3 Shannon's Entropy 2

1.4 OpenSSL RNG 2

1.5 Linux RNG 2

2 OpenSSL RNG **3**

2.1 Prerequisites: Description of Functions used in the algorithm . . 4

2.2 Algorithm 4

3 Vulnerabilities in OpenSSL RNG **6**

3.1 Security Analysis of OpenSSL RNG 6

3.2 Identified Vulnerabilities 6

3.3 LESLI 7

3.4 ELO-240,ELO-160,ELO-80 7

3.5 DEJA-SEED 7

3.6 DEJA-STATE 8

4 Linux RNG **9**

4.1 Sources of Entropy 9

5 Sources of Entropy- Keyboard timings, mouse movements, IDE timings, HAVEGE algorithm **11**

5.1 Entropy from Keyboard Timings 11

5.2 Entropy from Mouse Movements 11

5.3 HAVEGE 14

6 Implementation- Gathering entropy from keyboard timings and mouse movements **16**

6.1 Output 16

6.2 Basic Declaration of Variables 17
6.3 Timestamping 17
6.4 Mouse Collector 18
6.5 Keyboard Collector 18
6.6 Function start() 19
6.7 Function stop() 19
6.8 Shannon's Entropy 20
6.9 HTML Code 21

Chapter 1

Introduction

RNG is a computational or physical device to generate random numbers or symbols. "Classic" examples of generating random numbers: the rolling of dice, coin flipping, the shuffling of playing cards. Some of the natural processes used for generating random numbers are:

1. Measuring atmospheric noise, thermal noise, and other external electromagnetic and quantum phenomena.

2. Cosmic background radiation or radioactive decay as measured over short timescales represent sources of natural entropy.

3. The speed at which entropy can be harvested from natural sources is dependent on the underlying physical phenomena being measured.

1.1 Types of Random Numbers

1. **True Random Numbers:** It measures some physical phenomenon that is expected to be random. Then compensates for possible biases in the measurement process.

2. **Pseudo Random Numbers:** It Uses computational algorithms to produce long sequences of apparently random results. Results are determined by a shorter initial value, known as a seed value or key.

1.2 Measure of Randomness: Entropy

In computing, entropy is the randomness collected by an operating system or application for use in cryptography or other uses that require random data. This randomness is often collected from hardware sources, either pre-existing ones such as mouse movements or specially provided randomness generators.

1.3 Shannon's Entropy

Shannons entropy is the expected value of the information contained in each message. In our case, that information is Randomness. So, Shannons entropy gives us a measure of randomness that is contained in our message. It is measured by H = - $\sum p_i log_b p_i$

Where p = Probability of the event happening And b = Base
Consider an example: We have balls in a bin: 4 Red, 3 Green and 2 Yellow. There are three outcomes possible when you choose the ball, it can be either red, yellow, or green (n = 3).
Entropy = - (4/9) log(4/9) + -(2/9) log(2/9) + - (3/9) log(3/9)
= 1.5304755
Therefore, we are expected to get 1.5304755 information each time we choose a ball from the bin.

1.4 OpenSSL RNG

OpenSSL is software library to be used in applications that need to secure communications against eavesdropping or need to ascertain the identity of the party at the other end. OpenSSL provides a number of software based random number generators based on a variety of hardware and software sources. The core library is written in the C programming language.

1.5 Linux RNG

Random number generation from kernel space was implemented for the first time for Linux in 1994. In Unix-like operating systems, **/dev/random** is a special file that serves as a blocking pseudorandom number generator. It allows access to environmental noise collected from device drivers and other sources. A counterpart to /dev/random is **/dev/urandom** which reuses the internal pool to produce more pseudo-random bits. This means that the call will not block, but the output may contain less entropy than the corresponding read from /dev/random.

Chapter 2

OpenSSL RNG

The implementation of the RNG is found in the file md_rand.c. It defines the default RNG to be used in OpenSSL, though in principle the framework allows for switching to different RNG implementations provided by the user. Any purely software-based RNG is based on a pseudo-random number generator (PRNG).

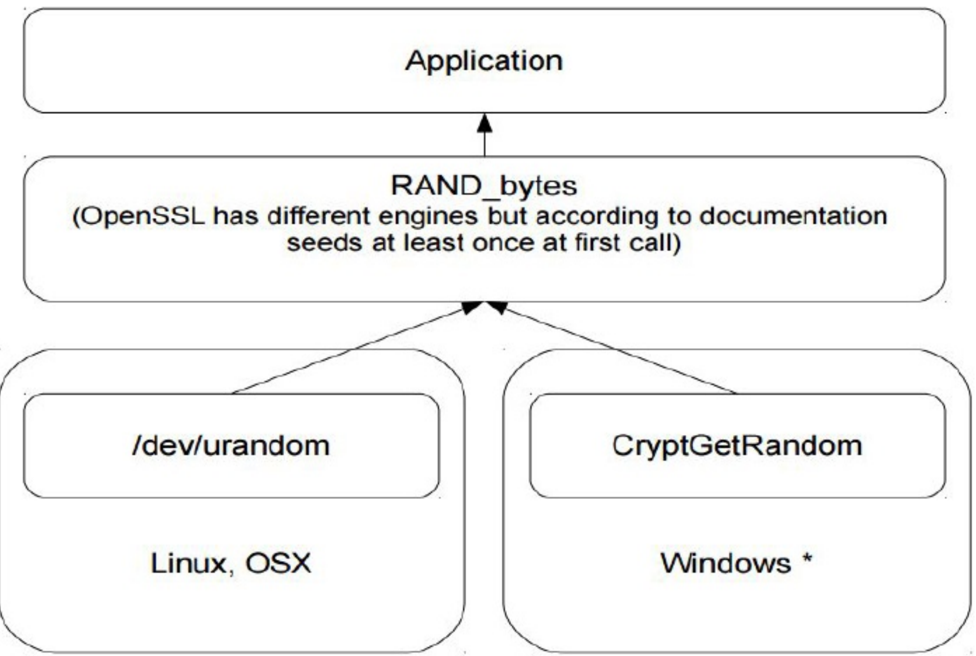


Figure 2.1: OpenSSL RNG

2.1 Prerequisites: Description of Functions used in the algorithm

1. **void RAND_add(const void *buf, int num, double entropy):** It adds the entropy contained in buf of length num to the RNGs internal state. Here, entropy shall be an estimate of the actual entropy contained in buf. The newly fed data modifies the RNGs state such that any subsequently generated random output will be affected by it.

2. **int RAND_bytes(unsigned char *buf, int num):** It outputs num random bytes into buf. If the entropy level of the RNG, computed as the sum of the estimates provided by the calls to RAND_add, is less than the specified minimum of 256 bits, this function returns with an error code.

3. **int RAND_poll():** It draws entropy from the operating systems randomness sources, e.g. /dev/random on Linux, and feeds it to the RNG.

4. **int RAND_load_file(const char *filename, long max_bytes):** It just loads a file and feeds up to max_bytes from that file to the RNG using RAND_add. Depending on the operating system, it feeds some further data to the RNG.

5. **void RAND_seed(const void *buf, int num):** It is a wrapper for RAND add. It calls that function with entropy = num, i.e. it expects the seed data to have maximal entropy.

6. **int RAND_pseudo_bytes(unsigned char *buf, int num):** It performs the same operation for the random output generation as RAND_bytes. But it also generates output if the minimum entropy level has not been reached.

2.2 Algorithm

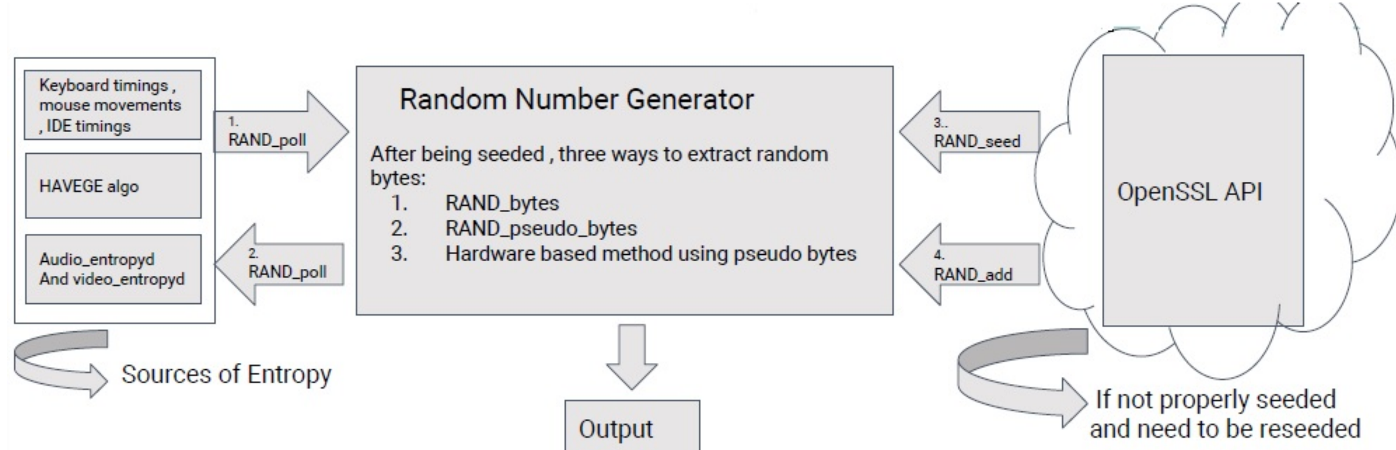


Figure 2.2: Working of OpenSSL Random Number Generator

Fig. 2.2 explains the basic algorithm and working of OpenSSL RNG.

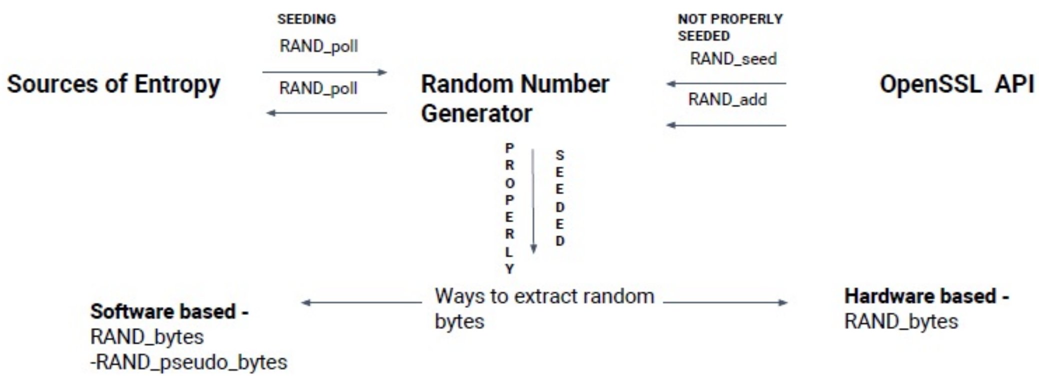


Figure 2.3: Flow diagram: OpenSSL Random Number Generator

Lifecycle States of OpenSSL Random Number Generator: It always starts in the state unseeded. In this state, it has zero entropy. If on the system a random device such as /dev/random on Unix is available, a call to RAND_bytes or RAND_pseudo_bytes will transfer the RNG automatically into the state seeded or falsely seeded by drawing 32 bytes of randomness from that device with a presumed entropy of 256 bits and feeding them to the RNG through a call to RAND_add. The distinction between seeded and falsely seeded is not reflected by the RNG state directly, but only implicitly through the quality of the seed. In case of a seed with considerably lower entropy than 256 bits drawn from the random device, we identify the resulting state as falsely seeded. Another possibility of entering this state is through a compromise of the seeded state through a break-in into the system. Recovery from the falsely seeded state is attempted by feeding the RNG with a high entropy seed. The resulting state is referred to as Reseeded. The RESEEDED stage is distinguished from the seeded state by the fact that the so-called **stirring operation**. The stirring operation distributes the entropy within the RNG state, is never carried out in this state. The Stirring operation is executed in the first call to RAND_bytes in the seeded state and never again after that.

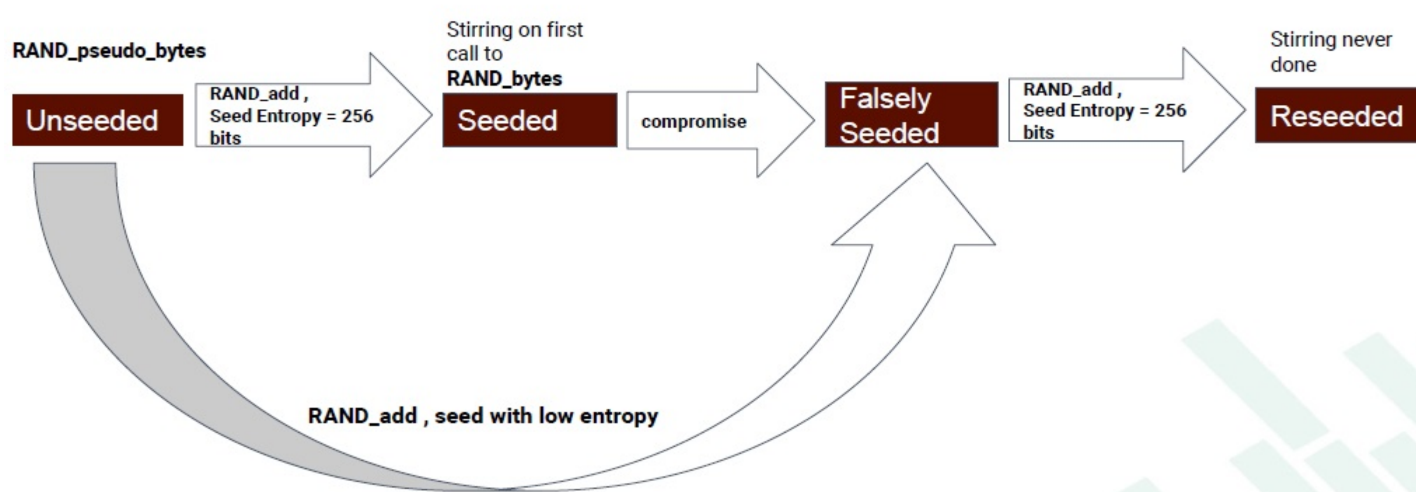


Figure 2.4: Lifecycle of OpenSSL Random Number Generator

Chapter 3

Vulnerabilities in OpenSSL RNG

3.1 Security Analysis of OpenSSL RNG

1. OpenSSL RNG solely relies on reseeding by the client application.Even when seeded initially with a 256 bit entropy seed, the RNG output may only have an entropy level of 240 bits for up to several hundreds of output bytes.

2. In a low entropy state, various functions of OpenSSL silently feed data to the RNG that is potentially secret and of low entropy.As a consequence, the RNGs function for outputting low entropy random numbers is prone to leak these low entropy secrets.

3. The potentially leaked values are identified to be the keys of weak ciphers such as DES and the previous contents of buffers overwritten with random bytes.

4. The previous contents of buffers overwritten with random bytes.It is problematic because wiping secret data by overwriting them with (pseudo) random data is an established practice.

3.2 Identified Vulnerabilities

The identified vulnerabilities are listed below :

1. LESLI

2. ELO-240,ELO-160,ELO-80

3. DEJA-SEED

4. DEJA-STATE

3.3 LESLI

LESLI - "Low Entropy Secret Leakage Issue" is a problem in the management of the RNG rather than in its cryptographic design.The RNG features a function for outputting random numbers before being sufficiently seeded -RAND_pseudo_bytes.LESLI attack puts low entropy secret seed data like user passwords at risk to be leaked through the RNG's output .OpenSSL also silently feeds other data to the RNG as seed data .This data includes the initial contents of the buffers to be filled with random data through RAND_pseudo_bytes .If these buffers contain low entropy secrets, these are also at risk to be leaked through subsequent RNG output .This is a rather realistic assumption, since overwriting secrets in memory with random values is an established practice.

3.4 ELO-240,ELO-160,ELO-80

The first design flaw of OpenSSLs core RNG, which we refer to as ELO-240, leads to the possibility that generated keys are limited to 240 bits of security instead of 256 bits .In contrast to other weaknesses ,this issue arises even when the RNG is initially seeded with correctly estimated high entropy data before any output is generated .This problem does not allow any practical attacks but it shows us a design flaw of the RNG .From a strictly formal point of view, it is commonly agreed that keys with 256 bit security shall be used for applications where long term security matters .It is also reflected by the standardized key sizes for AES and elliptic curve keys, and that thus an RNG should produce output with the corresponding entropy level .In ELO-160, there is an entropy limitation of the output of RAND_bytes to 160 bits and the attacker has access to output after the reseeding .Whereas,in the ELO-80 attack there is an entropy limitation of the output of RAND_bytes to 80 bits. The attacker has access to some output from the same call to RAND_bytes as that which he wishes to predict.

3.5 DEJA-SEED

State Recovery Attacks happen against the RESEEDED state .These attacks are labelled after the deja-vu effect since they exploit the re-entering of the state bytes .This re-entering occurs where the high entropy seed was added by exploiting the wraparound at the end of the state bytes in the RNG .In the same scenario, two different attacks are possible: DEJA-SEED and DEJA-STATE .The RNG is in the state FALSELY SEEDED with zero entropy .Then, a 160-bit entropy seed v is fed to the RNG using RAND_add .Afterwards, a 160-bit key is generated .Fixed positions of the state bytes are used .Let the 160- bit entropy seed data be fed to the RNG when $p = 40$.

Assume the added seed data v is of the following form:

- A publicly known 10-byte constant part followed by the 20 byte seed data with full entropy .

After the seed has been added, p points to s[70] .Now, the 160- bit key k is generated.

- Afterwards, either through further additions of seed data or through output generation, p reaches the value 0 again.

- In this situation, a 90 byte output is generated with a single call to RAND_bytes, which is accessible to the attacker.The attacker uses this output to recover the seed v and subsequently the 160-bit key k.

3.6 DEJA-STATE

It is similar to the DEJA- SEED attack .The difference being that not the high entropy seed value is recovered, but only the RNG state .There is no requirement on the seed data v to be prepended with constant data .This is because here we attack the state bytes in blocks of 80 bits, as they are processed by RAND_bytes .This attack allows the prediction of all output of the RNG after the high entropy reseeding like in the DEJA-SEED attack.

The scenario can be explained as follow:

- We assume that a 160-bit seed with full entropy was fed to the RNG through a call to RAND_add when p pointed to s[40] .

- Accordingly, the state bytes s[40 : 59] are affected by this update .The attack starts in the same way as the DEJA-SEED attack.

- This includes the recovery of md1 through the first output blocks, up to the point where in the DEJA-SEED attack the values for v0 and v1 would be guessed.

Chapter 4

Linux RNG

A PRNG is a deterministic algorithm that produces numbers whose distribution is indistinguishable from uniform.It usually involves an internal state from which a cryptographic function outputs random-looking numbers.In UNIX-like operating systems, a PRNG with input was implemented for the first time for Linux 1.3.30 in 1994.The entropy source comes from device drivers and other sources such as latencies between user-triggered events (keystroke, disk I/O, mouse clicks). Pseudo-random bits are created from the special files /dev/random and /dev/urandom. It is gathered into an internal state called the entropy pool.The internal state keeps an estimate of the number of bits of entropy in the internal state.

4.1 Sources of Entropy

The sources of entropy in Linux RNG are /dev/random and /dev/urandom files. They have been explained further below :

- /dev/random on Linux collects information from the variation in timing of hardware interrupts from sources such as hard disks returning data, keypresses and incoming network packets.

- This approach is secure provided that the kernel does not overestimate how much entropy it has collected.

- Linux /dev/random adds randomness from the environment and estimates the entropy it gains by doing so.

- Entropy is a measure of the uncertainty which an outside observer would have over the internal RNG state (called the "random pool" in /dev/random) .It is assumed that the observer does not have detailed knowledge of the events which are occurring, but he may have some general idea of system activity and characteristics.

- The character special files /dev/random and /dev/urandom provide an interface to the kernel's random number generator .The random number generator gathers environmental noise from

device drivers and other sources into an entropy pool .The generator also keeps an estimate of the number of bits of noise in the entropy pool. From this entropy pool random numbers are created.

- When read, the /dev/random device will only return random bytes within the estimated number of bits of noise in the entropy pool. /dev/random should be suitable for uses that need very high quality randomness such as one-time pad or key generation. When the entropy pool is empty, reads from /dev/random will block until additional environmental noise is gathered.

- A read from the /dev/urandom device will not block waiting for more entropy .As a result, if there is not sufficient entropy in the entropy pool, the returned values are theoretically vulnerable to a cryptographic attack on the algorithms used by the driver.

- Writing to /dev/random or /dev/urandom will update the entropy pool with the data written, but this will not result in a higher entropy count. This means that it will impact the contents read from both files, but it will not make reads from /dev/random faster.

Chapter 5

Sources of Entropy- Keyboard timings, mouse movements, IDE timings, HAVEGE algorithm

5.1 Entropy from Keyboard Timings

There can be a reasonable amount of entropy in key presses. The entropy comes not simply from which key is pressed, but from when each key is pressed. In fact, measuring which key is pressed can have very little entropy in it, particularly in an embedded environment where there are only a few keys. Most of the entropy will come from the exact timing of the key press.
Process: The basic methodology is to mix the character pressed, along with a timestamp, into the entropy pool.If you can easily get information on both key presses and key releases (as in an event-driven system like Windows), it is recommended to mix such information in as well. The big issue is in estimating the amount of entropy in each key press. The first worry is what happens if the user holds down a key. The keyboard repeat may be so predictable that all entropy is lost. Solution: It does not measure any entropy at all, unless the user pressed a different key from the previous time.Ultimately, the amount of entropy estimate getting from each key press should be related to the resolution of the clock used to measure key presses. It reads data from the keyboard, hashes it into a SHA1 context, and repeats until it is believed that the requested number of bits of entropy has been collected.

5.2 Entropy from Mouse Movements

The entropy estimation is done on the basis of event timing. There are separate timers maintained for mouse, keyboard, block device, and other interrupts, respectively. When an event occurs the entropy is estimated based on the timing of that event compared to the timings of

earlier events of that same class. When a key is pressed, for example, the entropy is estimated based on the timing of that keyboard press compared to the timing of earlier keyboard presses. The actual data of the event does not contribute to the estimate. The data is added to the random pool (along with the event time), but it is not taken into consideration in estimating the entropy. For a mouse interrupt, the mouse position and interrupt time is added to the pool, but only the time is used to estimate the entropy contribution, and similarly for the other interrupt classes. The entropy estimate is done on the basis of first, second, and third differences from the timing of previous events of the same class. The first difference is the delta of the new event time minus the previous event time. The second difference is the delta of first differences of this event and the previous; the third difference is the delta of second differences. The code actually uses the absolute values of the deltas.

Example For the latest event being added, at time 1041 above, the first difference is 11, the

Mouse event times	**1004**		**1012**		**1024**		**1025**		**1030**		**1041**
1st differences		**8**		**12**		**1**		**5**		**11**	
2nd differences			**4**		**11**		**4**		**6**		
3rd differences				**7**		**7**		**2**			

second difference is 6, and the third difference is 2. The entropy estimator takes the minimum delta value, which in this case would be 2, as its measure of how unexpected the new data point is. Given this minimum delta, the log base 2 of that value is then used as the entropy estimate. In this case the new value would be credited as adding 1 bit of entropy.

This use of deltas is approximately the same as attempting to fit an n degree polynomial to the previous n+1 points, then looking to see how far the new point is from the best prediction based on the previous n points. The minimum of the deltas is used, which has the effect of taking the best fit of a 0th, 1st or 2nd degree polynomial, and using that one. With this model of how the entropy estimation works, we can evaluate how suitable it is for the purposes in which it is used. The estimator seems to be implicitly based on a physical model in which the event timings would be expected to occur on a low degree polynomial. It is conceivable that an outside observer might be able to guess this fact and, making some simple assumptions about the initial state and polynomial coefficients, he could predict a series of timing values. The entropy estimator takes that into consideration by only including departures from low degree polynomials as a source of entropy. The assumption is that such departures are unpredictable and random and could not be anticipated by the observer. This model would seem better suited to some kinds of data than others. For example, mouse positions (measured at constant time intervals) would produce data series which are likely to be on low degree polynomials. The mouse is at rest for much of the time, then it is accelerated by hand and moved at constant or

slowly varying speed, then it is decelerated. If the mouse is being moved around vigorously it will often trace out something like a Lisajous pattern, where the two coordinates are simple trig functions, which can be well approximated locally by second degree polynomials.
However, the model is not applied to mouse positions. It is only applied to event timings. Here it seems less certain that the assumptions are valid. The timing of mouse interrupts, for example, does depend to some extent on mouse speed, but not necessarily in a simple way. It is not clear that mouse interrupt timing would be expected to fit on a low degree polynomial. In fact the model can be seen to be deficient in one significant respect, in that it does not take into consideration quantization effects. Event times may, because of various system characteristics, all occur at some multiple of a high speed timer.

Consider the example above,
and suppose that all event times were multiplied by 10:

Mouse event times	**10040**		**10120**		**10240**		**10250**		**10300**		**10410**
1st differences		**80**		**120**		**10**		**50**		**110**	
2nd differences			**40**		**110**		**40**		**60**		
3rd differences				**70**		**70**		**20**			

Now the minimum difference is 20 instead of 2, and the entropy added will be 4+ bits rather than 1. But the actual randomness of the data has not increased, from the point of view of an outside observer who is aware of this quantization effect. The system is overestimating the entropy because it does not detect the quantization. A better model for timing based entropy would be designed to look specifically for quantization effects. If all the deltas and the new value are close to a multiple of some common factor, this factor should be removed before the entropy is estimated. The polynomial fitting can also become confused by some simple variations. Suppose that due to either hardware or software characteristics, all events are passed to the entropy estimator twice. This leads to a pattern like:

Mouse event times	**1024**	**1025**	**1025**	**1030**	**1030**	**1041**
1st differences	**1**	**0**	**5**	**0**	**11**	
2nd differences	**1**	**5**	**5**	**11**		
3rd differences	**4**	**0**	**6**			

This now adds log(6) or 2+ bits of entropy for this event. But if the observer knows this characteristic of the system, there is no more entropy in this series than in the original one. There may be circumstances where the event code (key press, mouse position, disk or other event data) is useful in estimating entropy contributions. Auto-repeat key presses, for example, might have timing characteristics sufficiently different from user typing that it is hard to have one model capture both, and the event code could help to detect those. Presently this data is ignored in the entropy estimate. A few other problems can be mentioned briefly. All classes of events are being dealt with by the same model. It is not clear that it is reasonable to assume that such disparate kinds of events can be dealt with in this one framework. The use of exactly three levels of deltas is also questionable. Has it been determined that a fourth level is not necessary, for any of the event classes handled by /dev/random? Do all classes of events need exactly the same three levels? In summary, the /dev/random entropy estimates are based on fitting low degree polynomials to event times. All classes of events are modelled using the same approach. More justification is needed to show that thisis a reasonable method for estimating entropy for distinctly different classes of events. Some specific problems (quantization, repeated events) will cause this model to overestimate entropy.

5.3 HAVEGE

Hardware Volatile Entropy Gathering and Expansion: Random bits produced by peripherals events (e.g. by Entropy Gathering Daemons), or by HAVEGE, are not, strictly speaking, truly random bits nor pseudo-random bits. They are not truly random because the process which produces them is deterministic. One could theoretically reproduce the sequence if he/she was able to reproduce all the past events on the machine. They are not pseudo-random either since there is no (short) seed which would allow an exact reproduction of the random sequence. The randomness results instead from an inability to control or predict with sufficient accuracy the events involved in the generation process.
Parameters influencing on the execution time of a sequence of instructions: In microprocessor systems built around modern superscalar processors, the precise number of cycles needed to execute a(very short) sequence of instructions is dependent on many internal states of hardware components inside the microprocessor as well as on events external to the process.

Let us consider a simple sequence of instructions:

1. Read of the hardware clock counter.
2. Conditional branch.
3. Load.
4. Read of the hardware clock counter.

The number of cycles for executing this short sequence will depend on:

1. Correct or incorrect branch prediciton (both direction and target)
2. Hit or miss of the instructions in the ITLB
3. Hit or miss of the instructions on the instruction cache
4. Hit or miss of the data load on the data cache
5. Hit or miss of the data load on the data TLB
6. In case of miss on one of those caches , hit or miss on the L2 cache

Chapter 6

Implementation- Gathering entropy from keyboard timings and mouse movements

The implementation part consists of two files:

- entropy-collector.js

- HTML file using the js file

6.1 Output

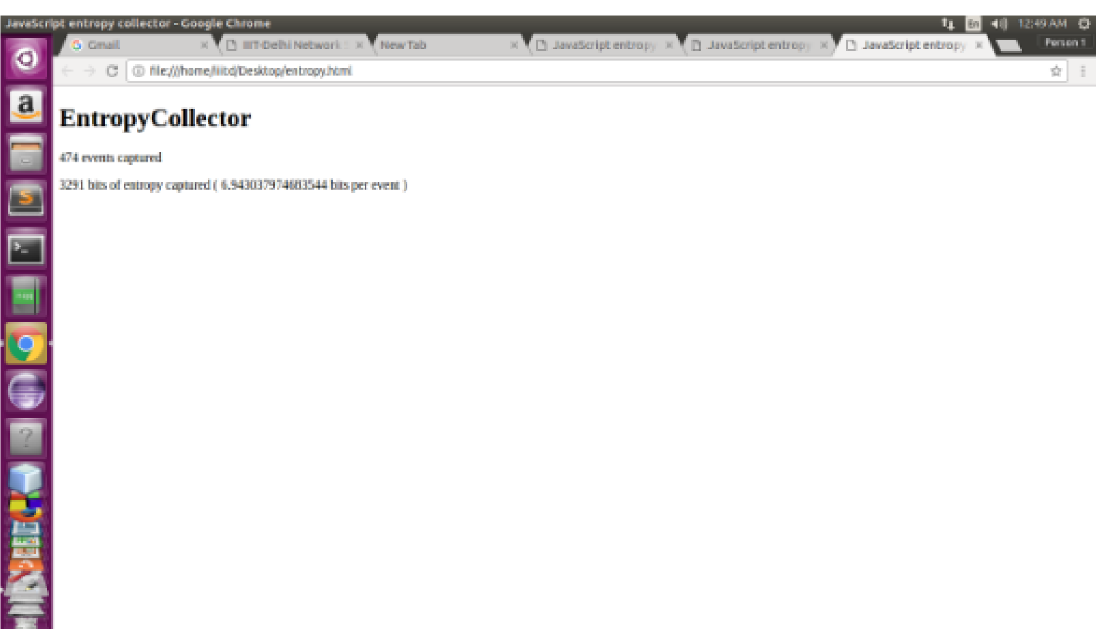


Figure 6.1: Output

6.2 Basic Declaration of Variables

```
var EntropyCollector = ( function ( global ) {
    'use strict';

    var min = Math.min,
        max = Math.max,
        abs = Math.abs,
        floor = Math.floor,
        log = Math.log,
        pow = Math.pow,
        atan2 = Math.atan2,
        sqrt = Math.sqrt,
        LN2 = Math.LN2,
        PI = Math.PI;

    var _date_now = global.Date.now,
        _perf = global.performance,
        _perf_timing,
        _perf_now;

    if ( _perf ) {
        _perf_timing = _perf.timing;
        _perf_now = _perf.now;
    }
```

Figure 6.2: Variable Declaration

6.3 Timestamping

```
    var now, _time_start, _time_precision;

    if ( _perf_now ) {
        now = function () { return 1000 * _perf.now() | 0 };
        // FIXME spec says `performance.now()` SHOULD be µs-precise, though
is't not guaranteed
        // FIXME required a way to reliably determine the precision at run
time
        _time_precision = 1;
    }
    else {
        _time_start = ( _perf_timing ) ? _perf_timing.navigationStart :
_date_now();
        now = function () { return 1000 * ( _date_now() - _time_start ) | 0 };
        _time_precision = 1000;
    }

    // EventTarget to bind to

    var _event_target = global.document || global;

    // Buffer for events

    var _buffer_size = 1024;
    var _buffer = new Int32Array( 2*_buffer_size );
```

Figure 6.3: Timestamping

6.4 Mouse Collector

```
    var _last_t = 0, _last_x = 0, _last_y = 0;

    function _mouse_collector ( e ) {
        var i = _event_counter % _buffer_size,
            t = now(), x = e.screenX, y = e.screenY;

        if ( _event_counter ) {
            _time_events[i] = max( t - _last_t, 0 );
            _coord_events[i] = ( x - _last_x << 16 ) | ( y - _last_y &
0xffff );
        }

        _last_t = t, _last_x = x, _last_y = y;
        _event_counter++;
    }

    function _touch_collector ( e ) {
        for ( var i = 0; i < e.touches.length; i++ ) {
            _mouse_collector( e.touches[i] );
        }
    }
```

Figure 6.4: Mouse Collector

6.5 Keyboard Collector

```
function _keyboard_collector ( e ) {
    var i = _event_counter % _buffer_size,
        t = now();

    if ( _event_counter ) {
        _time_events[i] = max( t - _last_t, 0 );
    }

    _last_t = t;
    _event_counter++;
}
```

Figure 6.5: Keyboard Collector

6.6 Function start()

```
function start () {
    _event_target.addEventListener( 'mousemove', _mouse_collector );
    _event_target.addEventListener( 'mousedown', _mouse_collector );
    _event_target.addEventListener( 'mouseup', _mouse_collector );

    _event_target.addEventListener( 'touchmove', _touch_collector );
    _event_target.addEventListener( 'touchstart', _touch_collector );
    _event_target.addEventListener( 'touchend', _touch_collector );

    _event_target.addEventListener( 'keydown', _keyboard_collector );
    _event_target.addEventListener( 'keyup', _keyboard_collector );
}
```

Figure 6.6: Function start()

6.7 Function stop()

```
function stop () {
    _event_target.removeEventListener( 'mousemove', _mouse_collector );
    _event_target.removeEventListener( 'mousedown', _mouse_collector );
    _event_target.removeEventListener( 'mouseup', _mouse_collector );

    _event_target.removeEventListener( 'touchmove', _touch_collector );
    _event_target.removeEventListener( 'touchstart', _touch_collector )
    _event_target.removeEventListener( 'touchend', _touch_collector );

    _event_target.removeEventListener( 'keydown', _keyboard_collector )
    _event_target.removeEventListener( 'keyup', _keyboard_collector );
}
```

Figure 6.7: Function stop()

6.8 Shannon's Entropy

```
var _estimated_entropy = 0;

function _shannon0 ( A, W ) {
    var n = 0;
    for ( var k in A ) n += A[k];
    if ( !n ) return 0;

    var h = 0;
    for ( var k in A ) {
        var p = A[k] / n, w = W(k);
        if ( !w ) continue;
        h -= p * log( p / w );
    }

    return h;   }
```

Figure 6.8: Shannon's entropy

```
function _shannon1 ( A, B, W ) {
    var n = 0;
    for ( var k in A ) n += A[k];
    if ( !n ) return 0;

    var h = 0;
    for ( var k in A ) {
        if ( !B[k] ) continue;
        h += ( A[k] / n ) * _shannon0( B[k], W );
    }

    return h;
}
```

Figure 6.9: Shannon's entropy

6.9 HTML Code

```
<!DOCTYPE html>
<html>
<head>
<title>JavaScript entropy collector</title>
<script src="/Users/mac/Desktop/btp/entropy .js"></script>
</head>
<body>
<h1>EntropyCollector</h1>
<p><span id="events">0</span> events captured</p>
<p><span id="bits-total">—</span> bits of entropy captured ( <span id="bits-
per-event">—</span> bits per event )</p>
<script type="text/javascript">
function print () {
    var n = EntropyCollector.eventsCaptured,
        e = EntropyCollector.estimatedEntropy;
    document.getElementById('events').textContent = n;
    document.getElementById('bits-total').textContent = e;
    document.getElementById('bits-per-event').textContent = e/n;
}
EntropyCollector.start();
setInterval( print, 300 );
EntropyCollector.stop();

</script>
</body>
```

Figure 6.10: HTML Code

Bibliography

[1] Falko Strenzke. *An Analysis of OpenSSL's Random Number Generator*, 2016.

[2] *http://www.mail-archive.com/cryptography@c2.net/msg01708.html*, 2016.

[3] Seth Hardy. *The e2random Entropy Harvester and PRNG for Linux*, 2004.

[4] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergnaud, Daniel Wichs *Analysis of the Linux Pseudo-Random Number Generators*, 2013.

[5] *https://cr.yp.to/talks/2011.09.28/slides.pdf*, 2011.

[6] *https://www.cryptolux.org/images/7/7f/RNG$_S$urvey.pdf*, 2013.

[7] *http://www.dis.uniroma1.it/ alberto/didattica/cns-slides/random-number-gener.pdf*, 2013.