Attacking the Linux PRNG on Android & Embedded Devices

David Kaplan, Sagi Kedmi, Roee Hay & Avi Dayan
IBM Security Systems
agenda

• Motivation and Introduction
• Linux Random Number Generator
agenda

• Motivation and Introduction
• Linux Random Number Generator
• Our Attack
• 1\textsuperscript{st} Attack Vector – Local Atk.
• Demo
• 2\textsuperscript{nd} Attack Vector – Remote Atk.
agenda

• Motivation and Introduction
• Linux Random Number Generator
• Our Attack
• 1st Attack Vector – Local Atk.
• Demo
• 2nd Attack Vector – Remote Atk.
• Mitigations
MOTIVATION
• We discovered CVE-2014-3100, a **stack-based Buffer Overflow** in Keystore
  • Service responsible for securely storing crypto related data

• We had privately reported to Google and they provided a patch available in *KITKAT*. [Whitepaper](#).

• Exploit must overcome various defense mechanisms, including **Stack Canaries**.

```c
/* KeyStore is a secured storage for key-value pairs. In this implementation, each file stores one key-value pair. Keys are encoded in file names, and values are encrypted with checksums. The encryption key is protected by a user-defined password. **To keep things simple, buffers are always larger than the maximum space we needed**, so boundary checks on buffers are omitted. */
```
Stack canaries and their enforcement

- On libbionic load:
  ```c
  __stack_chk_guard = *(uintptr_t *)getauxval(AT_RANDOM));
  ```
- Function Prologue:
  Place `__stack_chk_guard` on the stack (before ret).
- Function Epilogue:
  Compare saved stack canary with `__stack_chk_guard`;
  → Crash if mismatch
Stack canaries and their enforcement

- On libbionic load:
  
  ```c
  __stack_chk_guard = *(uintptr_t *)getauxval(AT_RANDOM));
  ```

- Function Prologue:
  
  Place `__stack_chk_guard` on the stack (before ret).

- Function Epilogue:
  
  Compare saved stack canary with `__stack_chk_guard`;
  → Crash if mismatch

Canary origins; *nix process creation model

- `fork()` → `execve()`.
- `execve()` → Auxiliary vector (AUXV)
- `AUXV[AT_RANDOM]` = 16 Random bytes from the PRNG
- `libbionic` assigns canary = first 4 bytes of `AT_RANDOM`
Stack canaries and their enforcement

- On libbionic load:
  \[
  \_stack\_chk\_guard = *(uintptr_t *)getauxval(AT\_RANDOM));
  \]
- Function Prologue:
  Place \_stack\_chk\_guard on the stack (before ret).
- Function Epilogue:
  Compare saved stack canary with \_stack\_chk\_guard;
  \(\rightarrow\) Crash if mismatch

Canary origins; *nix process creation model

- fork() \(\rightarrow\) execve().
- execve() \(\rightarrow\) Auxiliary vector (AUXV)
- AUXV[AT\_RANDOM] = 16 Random bytes from the PRNG
- libbionic assigns canary = first 4 bytes of AT\_RANDOM

Remember this; We'll get back to it
Attacks on the Stack-Smashing Protection:

• Naive Online Bruteforce of the *Canary* Value
• Impractical: $2^{32}$ attempts on average.
Attacks on the Stack-Smashing Protection:

- Naive Online Bruteforce of the *Canary* Value
  - Impractical: $2^{32}$ attempts on average.
- Online Learning of the *Canary* Value
  - By another info leak issue
- Re-forking server:
  - Very efficient: 514 attempts until success on average
Motivation: Keystore Buffer Overflow

**Attacks on the Stack-Smashing Protection:**

- Naive Online Bruteforce of the *Canary* Value
  - Impractical: $2^{32}$ attempts on average.
- Online Learning of the *Canary* Value
  - By another info leak issue
  - Re-forking server:
    - Very efficient: 514 attempts until success on average
- Overwrite `__stack_chk_guard`
  - By overwriting some pointer
Attacks on the Stack-Smashing Protection:

- Naive Online Bruteforce of the *Canary* Value
  - Impractical: $2^{32}$ attempts on average.
- Online Learning of the *Canary* Value
  - By another info leak issue
  - Re-forking server:
    - Very efficient: 514 attempts until success on average
- Overwrite `_stack_chk_guard`
  - By overwriting some pointer
- **Our attack**: *Offline* reconstruction of the PRNG’s internal state
Wrap things up:

- We found a vulnerability in a critical service in Android 4.3.
- In an effort to exploit it, we had to overcome a stack canary, we couldn't do so using known techniques.
- Canaries are 4 random bytes that are extracted from the Linux PRNG.
- Aimed to find a weakness in the PRNG that will allow us to intelligently guess the canary.
- End up with a full-fledged attack on the Linux PRNG.
LINUX PRNG
lprng_overview

Bird's eye view

- Output is hashed twice using SHA1
- Extracts in blocks of 10 bytes and truncates if necessary.
*KEC = Kernel Entropy Count

**entropy_sources**

```
INPUT POOL

NON-BLOCKING-POOL

if KEC >= 192 bits

ERROR

EXTRACTION (PULL)
```
**boot_time_vulnerability**

*KEC = Kernel Entropy Count*
boot_time_vulnerability

```
seconds  nanoseconds
   63    31    0

ktime_t

NON-BLOCKING-POOL

EXTRACTION (PULL)
```
Device powers on → Kernel starts booting → PRNG is initialized → Kernel boot Finished & Platform starts booting → Input Pool mixed into Nonblocking Pool :( → Phone is ready

May occur in different order
OUR WORK
Prior art on weakness in early boot *

Present practical run-time attack

Formalize attack

Demonstrate PoC against current mobile platforms

Given a **LEAK** of a value extracted from the non-blocking pool and **LOW ENTROPY AT BOOT**, the **STATE** of the PRNG can be determined until external entropy is too high.
Using the PRNG against itself

- Recall: Low boot-time entropy degenerates the PRNG and that the output of the PRNG is hashed twice using SHA1.
- Fact: Crypto. hash functions are designed to be collision resistant.
Using the PRNG against itself

- Recall: Low boot-time entropy degenerates the PRNG and that the output of the PRNG is hashed twice using SHA1.
- Fact: Crypto. hash functions are designed to be collision resistant.
- It is highly unlikely that PRNGs that are seeded with different seeds will result in the same output. Regardless of the order of extractions.
attack_leak

Using the PRNG against itself

- Recall: Low boot-time entropy degenerates the PRNG and that the output of the PRNG is hashed twice using SHA1.
- Fact: Crypto. hash functions are designed to be collision resistant.
- It is highly unlikely that PRNGs that are seeded with different seeds will result in the same output. Regardless of the order of extractions.
- Result: Every leak(sequence of random bytes) from the non blocking pool is almost certainly the offspring of one specific seed.
Using the PRNG against itself

- Given a leak from the nonblocking pool of a “Real” PRNG we could simulate offline PRNGs with different seeds and compare extractions with the online leak.
**Using the PRNG against itself**

- Given a leak from the nonblocking pool of a “Real” PRNG we could simulate offline PRNGs with different seeds and compare extractions with the online leak.
- Due to SHA1’s collision resistance, if one of the simulated PRNGs produces a sequence of random bytes that is the same as the leak value – we almost certainly found the seed.
Using the PRNG against itself

- Given a leak from the nonblocking pool of a “Real” PRNG we could simulate offline PRNGs with different seeds and compare extractions with the online leak.
- Due to SHA1's collision resistance, if one of the simulated PRNGs produces a sequence of random bytes that is the same as the leak value – we almost certainly found the seed.
- Once we have the seed we can produce the same outputs of the “Real” PRNG until noise from the Input pool is mixed to the Nonblocking pool.
Even After the mixing, the PRNG is vulnerable

- **Note:** in the whitepaper we demonstrated a more intricate attack flow
attack_overview

Problems we faced:

- The Nonblocking pool seed is 8 bytes long. Say we consider only the nanoseconds and assuming uniform distribution,

\[ 10^9 = 2^{\log_2(10^9)} \approx 2^{30} \]

...
Problems we faced:

- The Nonblocking pool seed is 8 bytes long. Say we consider only the nanoseconds and assuming uniform distribution:
  \[ 10^9 = 2 \log_2(10^9) \approx 2^{30} \]

- Hidden entropy source – Concurrency

**Yellow Path**
- Process A: extract from pool
- Process A: mix into pool
- Process B: extract from pool
- Process B: mix into pool

**Green Path**
- Process A: extract from pool
- Process B: extract from pool
- Process A: mix into pool
- Process B: mix into pool
Problems we faced:

- The Nonblocking pool seed is 8 bytes long. Say we consider only the nanoseconds and assuming uniform distribution:
  \[ 10^9 = 2^{\log_2(10^9)} \approx 2^{30} \]

- Hidden entropy source – Concurrency

- What can be attacked?
- Where can we get the leak value?
Where can we find leaks and attack targets?

- Device powers on
- Kernel starts booting
- PRNG is initialized
- Best Leak/Target
- Kernel boot Finished & Platform starts booting
- Good Leak/Target
- Concurrency Hell
- Input Pool mixed into Nonblocking Pool :(
- Bad Leak/Target
- Phone is ready
Terminology

**Device powers on**
- Kernel starts booting
- PRNG is initialized

**Kernel Boot-time Leak/Target**
- Kernel boot finished & Platform starts booting

**Platform Boot-time Leak/Target**
- Input Pool mixed into Nonblocking Pool :(
- Phone is ready

**Concurrency Hell**
- Best Leak/Target
- Good Leak/Target
- Bad Leak/Target
1st Attack Vector
Malware → PRNG Seed → Keystore's Canary
s4_offline_study
Instrumenting a device

• Samsung Galaxy S4, Android 4.3
Instrumenting a device

- Samsung Galaxy S4, Android 4.3

- `printk()` input and nonblocking pool seeds - find a bias in the seed value
- `printk()` `get_random_bytes()` callers and amount of random bytes requested - find leak and attack targets
s4_offline_study

Instrumenting a device

- Samsung Galaxy S4, Android 4.3
- `printk()` input and nonblocking pool seeds - find a bias in the seed value
- `printk()` `get_random_bytes()` callers and amount of random bytes requested – find leak and attack targets
- Fixed the seeds to see catch some bias in the order of extractions – find bias in conc.
Instrumenting a device

- Samsung Galaxy S4, Android 4.3

- `printk()` input and nonblocking pool seeds - find a bias in the seed value
- `printk()` `get_random_bytes()` callers and amount of random bytes requested – find leak and attack targets

- Fixed the seeds to see catch some bias in the order of extractions – find bias in conc.

- In total, we rebooted(script) the device more than 2000 times, each time we dumped the kernel ring buffer to a file.
s4_attack_leak

Details

- Android designers chose to spawn every app process by **forking** a master process - Zygote
s4_attack_leak

Details

• Android designers chose to spawn every app process by **forking** a master process – Zygote

• Zygote(app_process) is fork'ed and exec'ed by init at platform boot-time
s4_attack_leak

Details

- Android designers chose to spawn every app process by \textbf{forking} a master process – Zygote.

- Zygote(app\_process) is fork\'ed and exec\'ed by init at platform boot-time.

- *nix-like vs. App process creation model. Exec() ?
s4_attack_leak

Details

• Android designers chose to spawn every app process by **fork**ing a master process – Zygote

• Zygote(app_process) is fork'ed and exec'ed by init at platform boot-time

• *nix-like vs. App process creation model. **Exec() ?**

• Recall: exec() enforces ASLR and assigns the **AT_RANDOM**
Result: All Applications in Android has the same Canary value (AT_RANDOM) and largely the same address space layout.
Details

- Result: All Applications in Android have the same Canary value (AT_RANDOM) and largely the same address space layout.

- Leakage occurs due to the reuse of the same seed values in the PRNG for different applications.

- Applications like Zygote, WhatsApp, Contacts, and MALWARE inherit their Canary values from Zygote upon initialisation.

- The leakage is detected at the fork() point where the applications are created and inherit the Canary values.

- Diagram shows the flow of PRNG seeds and their leakage targets.
s4_attack_leak_concurrency

Given a leak, what's the probability of finding the original seed?

- Zygote's AT_RANDOM is our leak. It's a platform boot-time leak, which means it occurs in the 'Concurrency Hell' phase.
s4_attack_leak_concurrency

Given a leak, what's the probability of finding the original seed?

- Zygote's AT_RANDOM is our leak. It's a platform boot-time leak, which means it occurs in the 'Concurrency Hell' phase.

- An offline study of the samples revealed bias towards a specific extraction path from the nonblocking pool.
s4_attack_leak_concurrency

Given a leak, what's the probability of finding the original seed?

- Zygote's AT_RANDOM is our leak. It's a platform boot-time leak, which means it occurs in the 'Concurrency Hell' phase.
- An offline study of the samples revealed bias towards a specific extraction path from the nonblocking pool.
- 20% of the samples had Zygote's AT_RANDOM bytes somewhere in the extraction path.
s4_attack_leak_concurrency

Given a leak, what's the probability of finding the original seed?

- Given a leak and assuming we try all $2^{30}$ possible seeds the chance is $\frac{1}{5}$. 
$H(s_{nb}) = 23.5\text{ bits}$
Given a seed, Probabilities of finding the canary of early boot services
s4_attack_targets

Given a seed, Probabilities of finding the canary of early boot services

![Diagram showing probabilities of finding the canary of early boot services]

- ueventd: 1.000
- e2fsck: 1.000
- sh: 1.000
- servicemanager: 0.403
- void: 0.346
- mDriverDaemon: 0.394
- void: 0.264
- hmvivbed: 0.362
- rtcdd2: 0.212
- neta: 0.268
- debuggerd: 0.209
- s: 0.277
- chmod: 0.260
- chmod: 0.247
- fild: 0.199
- sh_1: 0.121
- kiesexe: 0.091
- smdexe: 0.117
- connfwexe: 0.095
- milex: 0.117
- npsmobex: 0.074
- surfacefinger: 0.069
- app_process: 0.078
- dmserver: 0.013
- media: 0.077
- install: 0.078
- chmod: 0.229
- chmod: 0.229
- sh_2: 0.087
- sh_3: 0.082
- ddxexe: 0.113
- Sh_2: 0.066
- sh_4: 0.091
- known: 0.069
- keystore: 0.051
- s: 0.043
- sensorhubbserver: 0.056

Given a seed, the probabilities of finding the canary of early boot services:
- seed_t_1
- seed_t_k
- seed_t_n

= LEAK?
Given Zygote's AT_RANDOM, the probability of guessing the Keystore's canary value is:

$$\frac{1}{5} \cdot \frac{6}{100} \simeq 0.01 \rightarrow 1\%$$

Remember where we came from... we needed to guess 32 random bits
Given Zygote's ATRANDOM, the probability of guessing the Keystore's canary value is:

\[
\frac{1}{5} \cdot \frac{6}{100} \approx 0.01 \rightarrow 1\% \\
\frac{1}{2^{32}} \approx 0.00000000023 \rightarrow 0.000000023\%
\]
2nd Attack Vector
Ping6 → PRNG Seed → IPv6 Fragment Injection & Getting Keystore's Canary
Instrumenting a device

- Samsung Galaxy S2, Android 4.1.2
s2_offline_study

Instrumenting a device

- Samsung Galaxy S2, Android 4.1.2
- `printk()` input and nonblocking pool seeds - find a bias in the seed value
- `printk()` `get_random_bytes()` callers and amount of random bytes requested - find leak and attack targets
s2_offline_study
Instrumenting a device

- Samsung Galaxy S2, Android 4.1.2
- `printk()` input and nonblocking pool seeds - find a bias in the seed value
- `printk()` `get_random_bytes()` callers and amount of random bytes requested – find leak and attack targets
- Fixed the seeds to see catch some bias in the order of extractions – find bias in conc.
Instrumenting a device

- Samsung Galaxy S2, Android 4.1.2
- `printk()` input and nonblocking pool seeds - find a bias in the seed value
- `printk()` `get_random_bytes()` callers and amount of random bytes requested - find leak and attack targets
- Fixed the seeds to see catch some bias in the order of extractions - find bias in conc.
- In total, we rebooted(script) the device more than 2000 times, each time we dumped the kernel ring buffer to a file.
**s2_attack_leak**

Details

- **Device powers on**
- **Kernel starts booting**
- **PRNG is initialized**
- **Kernel boot finished & Platform starts booting**
- **Input Pool mixed into Nonblocking Pool :(**
- **Concurrency Hell**
- **Phone is ready**
s2_attack_leak

Details

• While the kernel is brought up, an IPv6 module initializes and extracts 4 random bytes. Let's call them rand.
While the kernel is brought up, an IPv6 module initializes and extracts 4 random bytes. Let’s call them \texttt{rand}.

- IPv6 packet fragment identifier is computed by a deterministic function.

\[ f(\texttt{rand, ipv6\_dst\_addr}) = \texttt{ipv6\_frag\_id} \]
While the kernel is brought up, an IPv6 module initializes and extracts 4 random bytes. Let's call them **rand**.

IPv6 packet fragment identifier is computed by a deterministic function.

The pair \((\text{ipv6\_dst\_addr}, \text{ipv6\_frag\_id})\) is our leak. Why?

\[
f(\text{rand}, \text{ipv6\_dst\_addr}) = \text{ipv6\_frag\_id}
\]
While the kernel is brought up, an IPv6 module initializes and extracts 4 random bytes. Let's call them **rand**.

- IPv6 packet fragment identifier is computed by a deterministic function.

\[
f(\text{rand, ipv6\_dst\_addr}) = \text{ipv6\_frag\_id}
\]

- The pair \((\text{ipv6\_dst\_addr}, \text{ipv6\_frag\_id})\) is our leak. Why?

- We simulate PRNGs up to **rand**, and feed it to the deterministic function \(f\).
s2_attack_leak

Details

- While the kernel is brought up, an IPv6 module initializes and extracts 4 random bytes. Let's call them \texttt{rand}.
- IPv6 packet fragment identifier is computed by a deterministic function.

\[
f(\text{rand, ipv6\_dst\_addr}) = \text{ipv6\_frag\_id}
\]
- The pair \((\text{ipv6\_dst\_addr, ipv6\_frag\_id})\) is our leak. Why?

- We simulate PRNGs up to \texttt{rand}, and feed it to the deterministic function \(f\).
- OK, fine, but how did you get \texttt{ipv6\_dst\_addr}?
IPv6 fragmentation & ICMPv6 Echo Req.

- IP packets that exceed the path MTU, are divided into fragments which are sent and then reassembled by receiver.
IPv6 fragmentation & ICMPv6 Echo Req.

- IP packets that exceed the path MTU, are divided into fragments which are sent and then reassembled by receiver.

- Each fragment of the packet contains the same fragment id. Which is used by the receiver to identify fragments of a packet.
IPv6 fragmentation & ICMPv6 Echo Req.

- IP packets that exceed the path MTU, are divided into fragments which are sent and then reassembled by receiver.

- Each fragment of the packet contains the same fragment id. Which is used by the receiver to identify fragments of a packet.

- IPv6 fragmentation doesn't happen very often. How do we make it happen?

---

s2_attack_leak

---
s2_attack_leak

IPv6 fragmentation & ICMPv6 Echo Req.

- Ping6 – a utility for sending ICMPv6 Echo Requests which requires the target to send an ICMPv6 Echo Replay with the exactly the same data.
IPv6 fragmentation & ICMPv6 Echo Req.

- Ping6 – a utility for sending ICMPv6 Echo Requests which requires the target to send an ICMPv6 Echo Replay with the exactly the same data.

- Result: Sending ICMPv6 Echo Request with data > MTU will make the receiver send a fragmented reply.
s2_attack_get_leak

Amsterdam Schiphol Airport
s2_attack_get_leak

Amsterdam Schiphol Airport

REAL PRNG = ipv6_frag_id ?

seed_t_k

LEAK

SIM. PRNG

seed_t_1

rand_t_1

f(rnd,dst)

= ipv6_frag_id ?

SIM. PRNG

seed_t_k

rand_t_k

f(rnd,dst)

SIM. PRNG

seed_t_n

rand_t_n

f(rnd,dst)
s2_attack_get_leak

Amsterdam Schiphol Airport

SSID= Schiphol Free

A

REAL PRNG

LEAK

seed_t_k

SIM. PRNG

f((rnd,dst))

rand_t_k

= ipv6_frag_id?
**s2_attack_get_leak**

**Amsterdam Schiphol Airport**

SSID = "Schiphol Free"

---

**REAL PRNG**

**LEAK**

**seed_t_k**

**SIM. PRNG**

**seed_t_1**

**seed_t_k**

**seed_t_n**

**rand_t_1**

**rand_t_k**

**rand_t_n**

\( f(rnd, dst) \)

\( = \text{ipv6_frag_id} ? \)
s2_attack_get_leak

Amsterdam Schiphol Airport

SSID=Schiphol Free

Fragmented ICMPv6 Echo Request

A

V

REAL PRNG

seed_t_k

LEAK

SIM. PRNG

seed_t_1

rand_t_1

f(rnd,dst)

= ipv6_frag_id ?

SIM. PRNG

seed_t_k

rand_t_k

f(rnd,dst)

SIM. PRNG

seed_t_n

rand_t_n

f(rnd,dst)
s2_attack_get_leak

Amsterdam Schiphol Airport

SSID=Schiphol Free

Fragmented ICMPv6 Echo Request

Fragmented ICMPv6 Echo Reply

A

V

f(rnd,dst) f(rnd,dst) f(rnd,dst)

rand_t_1 rand_t_k rand_t_n

seed_t_1 seed_t_k seed_t_n

REAL PRNG

LEAK

SIM. PRNG

SIM. PRNG

SIM. PRNG

= ipv6_frag_id?
Attacker got the leak:

- V computed ipv6_frag_id with A's ipv6_src_addr
- A knows ipv6_frag_id and ipv6_dst_addr.
Given the leak we find the seed

$$H(s_{nb}) = 18.4 \text{ bits}$$
s2_attack_targets

Given the seed what can we attack?

- IPv6 Fragment injection – We can derive the exact fragment id V will use for any destination address.
s2_attack_targets

**Given the seed what can we attack?**

- **IPv6 Fragment injection** – We can derive the exact fragment id V will use for any destination address.

- **Canary value of early boot services.** For instance, with a probability of 1/20 we can compute Keystore's canary value, given the seed.
s2_attack_targets

Probabilities of finding the canary of early boot services

<table>
<thead>
<tr>
<th>Service</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffu</td>
<td>0.976</td>
</tr>
<tr>
<td>e2fsck_0</td>
<td>0.988</td>
</tr>
<tr>
<td>e2fsck_1</td>
<td>0.988</td>
</tr>
<tr>
<td>service manager</td>
<td>0.178</td>
</tr>
<tr>
<td>void</td>
<td>0.010</td>
</tr>
<tr>
<td>setup_fs</td>
<td>0.026</td>
</tr>
<tr>
<td>immvibed</td>
<td>0.280</td>
</tr>
<tr>
<td>netd</td>
<td>0.098</td>
</tr>
<tr>
<td>debugger</td>
<td>0.050</td>
</tr>
<tr>
<td>nil</td>
<td>0.156</td>
</tr>
<tr>
<td>ddexe</td>
<td>0.128</td>
</tr>
<tr>
<td>kiesene</td>
<td>0.038</td>
</tr>
<tr>
<td>smdexe</td>
<td>0.098</td>
</tr>
<tr>
<td>dtexe</td>
<td>0.040</td>
</tr>
<tr>
<td>connfwexe</td>
<td>0.062</td>
</tr>
<tr>
<td>npsmobex</td>
<td>0.042</td>
</tr>
<tr>
<td>surfaceflinger</td>
<td>0.006</td>
</tr>
<tr>
<td>app_process</td>
<td>0.006</td>
</tr>
<tr>
<td>drmservice</td>
<td>0.010</td>
</tr>
<tr>
<td>mediaserver</td>
<td>0.154</td>
</tr>
<tr>
<td>dbus-daemon</td>
<td>0.076</td>
</tr>
<tr>
<td>installd</td>
<td>0.188</td>
</tr>
<tr>
<td>keystore</td>
<td>0.050</td>
</tr>
<tr>
<td>bintvoutservice</td>
<td>0.052</td>
</tr>
<tr>
<td>dbd</td>
<td>0.082</td>
</tr>
<tr>
<td>macloader</td>
<td>0.136</td>
</tr>
<tr>
<td>sh_0</td>
<td>0.098</td>
</tr>
<tr>
<td>sh_1</td>
<td>0.068</td>
</tr>
<tr>
<td>sleep_0</td>
<td>0.108</td>
</tr>
<tr>
<td>chmod</td>
<td>0.086</td>
</tr>
<tr>
<td>adb</td>
<td>0.082</td>
</tr>
<tr>
<td>date</td>
<td>0.090</td>
</tr>
<tr>
<td>sleep_1</td>
<td>0.092</td>
</tr>
</tbody>
</table>

f(rnd,dst) = ipv6_frag_id?

seed_t_k

SIM. PRNG

RANDOM VALUE
Mitigations
mitigations
Current mitigations

- Save entropy across boots
Current mitigations

- Save entropy across boots
- Trusted external entropy injection – web service / HWRNG
mitigations

Problem with those mitigations

Device powers on → Kernel starts booting → PRNG is initialized → Kernel boot Finished & Platform starts booting → Input Pool mixed into Nonblocking Pool :

Entropy

Injecting Entropy to Pools

Concurrency Hell

Best Leak/Target → Good Leak/Target → Bad Leak/Target → Phone is ready

Good Leak/Target

Bad Leak/Target

Injecting Entropy to Pools

Input Pool mixed into Nonblocking Pool :(
Problem with those mitigations

- **Entropy injection occurs after the kernel boots up**
- **Input Pool mixed into Nonblocking Pool :(
- **Phone is ready**

**Kernel Boot-time Leak/Target**

**Injecting Entropy to Pools**

**Concurrency Hell**

**Best Leak/Target**

**Good Leak/Target**

**Bad Leak/Target**

Device powers on

Kernel starts booting

PRNG is initialized

Kernel boot Finished & Platform starts booting

Entropy
Current mitigations

- Initialize the seeds using a hardware RNG
  - RDRAND,RDSEED Intel's ISA
  - Early random, Qualcomm
Current mitigations

- Initialize the seeds using a hardware RNG
  - RDRAND, RDSEED Intel's ISA
  - Early random, Qualcomm

- Mix device-specific data to nonblocking and blocking pools
Current mitigations

- Initialize the seeds using a hardware RNG
  - RDRAND, RDSEED Intel's ISA
  - Early random, Qualcomm
- Mix device-specific data to nonblocking and blocking pools
- Changes to newer kernels allow for more boot time entropy
Talk wrap up

- Linux-based devices with low boot time entropy may allow a practical, low-cost attack on the PRNG
- The attack requires an offline study of a device and an online leak
- Allows the attacker to predict a random number which is generated by the victim's PRNG
- Two manifestations - Local/Remote Atk.
- Mitigations
Thank you

Thanks Nadja Kahan for the illustrations!
http://www.nadjakahan.com