ATTACKING THE LINUX PRNG ON ANDROID

WEAKNESSES IN SEEDING OF ENTROPIC POOLS AND LOW BOOT-TIME ENTROPY

David Kaplan, Sagi Kedmi, Roee Hay, Avi Dayan
IBM Security Systems
{davidka,sagik,roeeh,avrahamd}@il.ibm.com
25th July, 2014

Abstract

Android is the most prevalent Linux-based mobile Operating System in the market today. Many features of the platform security (such as stack protection, key generation, etc.) are based on values provided by the Linux Pseudorandom Number Generator (LPRNG) and weaknesses in the LPRNG could therefore directly affect platform security. Much literature has been published previously investigating and detailing such weaknesses in the LPRNG. We build upon this prior work and show that - given a leak of a random value extracted from the LPRNG - a practical, inexpensive attack against the LPRNG internal state in early boot is feasible. Furthermore, the version of the Linux kernel vulnerable to such an attack is used in the majority of Android-based mobile devices in circulation. We also present two real-world exploitation vectors that could be enabled by such an attack. Finally, we mention current mitigations and highlight lessons that can be learned in respect to the design and use of future PRNGs for security features on embedded platforms.

I. Introduction

Embedded uses of the Linux kernel are becoming increasingly prevalent. These systems make use of randomness in order to provide platform security features such as ASLR, stack canaries, key generation, and randomized IP packet identification. Such features require that the sources of randomness are robust and therefore cannot be easily predicted by an attacker.

There have been a number of published works on the weaknesses in the Linux Random Number Generator (LPRNG). In their paper, *Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices*, Heninger et al. [1] describe observed catastrophic key generation failures likely due to the fact that embedded systems that generate these keys do so off PRNGs that fail in robustness at the time of the key generation. In their paper they describe an observable deterministic flow of the LPRNG due to low boot-time entropy. Becherer et al. [2] describe an attack against the LPRNG of an Amazon EC2 AMI image which enables a search for an ssh key-pair generated off the LPRNG.

Our contribution **-** Whereas Heninger et al. [1] have previously observed the effects of low boot-time entropy on real-world RSA key-pair generation and have further studied the cause in respect to weaknesses of the LPRNG on embedded devices, in our paper we present a formal mathematical method for modeling these weaknesses. This allows us to quantify the effort required to enable active probabilistic attacks on the LPRNG internal state.

Using an algorithm functionally similar to that of the simulation technique postulated by Becherer et al. [2] in their presentation (modified for our embedded environment), we show how the mapping of a target embedded system's boot flow combined with an active leak of a LPRNG value at some point during a low-entropy part of the boot flow could allow a *remote* attacker to recreate the internal state of the LPRNG. We further demonstrate that it is possible to perform a practical, optimized search for the LPRNG seeding data at boot even when entropy is injected from external sources (on vulnerable kernels).

While work by Ding et al. [3] also describes weaknesses in the Boot-time entropy of Android, we present a Proof of Concept attack against a popular mobile platform in use today. For this Proof of Concept, we have identified a suitable remote active leak in a popular, vulnerable version of the Linux kernel. In addition, we have identified another leak which can be used locally by malware on most versions of Android.

We will demonstrate our attack via two practical resultant attack scenarios:

In the first scenario, we demonstrate an IPv6 Denial of Service attack against the randomly generated fragment identifier in Linux kernel version 3.0.31.

In the second scenario, we show how the Android stack canary protection could be bypassed, thereby enabling exploitation via buffer overflow vulnerability attack vectors.

II. Background

A. Use of Linux Kernel In Android

The Android platform is based on the Linux kernel and as of Linux kernel 3.3/3.4 most Android patches have been merged into the mainline kernel itself. Being a largely open-source platform, albeit developed by Google, many mobile manufacturers have built devices running it. According to Net Applications, Android is continually gaining global market share, holding at around 37.75% at time of writing [4]. Android's main problem however is one of fragmentation. There is no centralized body responsible for maintaining a single build or code-base and manufacturers have carte blanche to do as they wish. Furthermore, devices are generally abandoned by their manufacturers - it is rare that software updates are issued to older devices.

It is obvious that fragmentation leads to a security nightmare. While Google does a good job of patching vulnerabilities in the latest versions of the Android code-base, manufacturers generally don't roll out these releases to their customers. This is true even of the flagship devices once a certain amount of time has passed.

Exacerbating the problem, kernel versions for various Android releases often are incongruent with the Android platform version. For example, we found that some modern devices are running the latest version of Android (4.4 KitKat) on an older kernel which was released with Android 4.3.

III. The Linux Random Number Generator

The Linux Random Number Generator consists of two Pseudorandom Number Generators; a blocking RNG which is exposed to user-space via the `/dev/random` device and blocks requests based on a count of estimated entropy persisted by the kernel and a non-blocking PRNG which is exposed both to user-space via the `/dev/urandom` device and to kernel-space via the exported `get_random_bytes()` function.

The LPRNG maintains three entropy pools. The input pool acts as the main entropy store and feeds both the blocking pool and the non-blocking pool used by the PRNGs. Our research focuses on the non-blocking PRNG – i.e. the non-blocking pool - and its associated pulls (extraction of a value) from the input pool. We do not consider the blocking pool at all as it is generally unused during Kernel boot.

When a value is requested from the non-blocking pool, the PRNG determines whether the prerequisite entropy is preset in the input pool. If sufficient entropy is available in the input pool, the pool is mixed in to the non-blocking pool (i.e. entropy is transferred to the non-blocking pool) via a Twisted Generalized Feedback Shift Register (TGFSR) mixing function. In order to extract the value, the pool is hashed (SHA1), mixed back into itself and hashed once more. The mixing of the hash back into the pool is intended to provide forward security (meaning that nothing can be known about previous values given a compromise of the internal pool state). Data is extracted in blocks of 80 bits and the final bits are truncated to match the requested pull value size.

All pools are initially seeded by mixing in the current system time in nanosecond resolution (`ktime_t`) to their uninitialized pool states. Entropy can be further added to the input pool in a number of ways (relevant to Linux kernels prior to 3.4.9):

Input Randomness – provided by input devices such as mouse, keyboard, touch screen.

Disk Randomness – provided by disk activities.

Interrupt Randomness – provided by triggered interrupts (generally disabled by modern kernels).

Each of the above can trigger entropy generation off the current system cycle count coupled together with data provided by the source event.

The kernel persists a counter for each pool which holds the amount of entropy in the pool as estimated by the kernel. This value is incremented when entropy is mixed into the pool and decremented when values are pulled from the pools. It is important to note that the entropy count as recorded by the kernel does not correlate to the actual amount of entropy of the pools. In this paper, we refer to the entropy counter value as persisted by the kernel as the Kernel Entropy Count (KEC).

IV. Attack

As an attacker, we would like to be able to ascertain a random value sourced from the PRNG at a certain point in the boot flow, meaning that we need to discern the internal state of the entropy pools at that point. The Shannon entropy of an attack at that point can

be generalized as $H(A) = H(p_{in}, p_{nb})$ where p_{in} and p_{nb} are the input pool state and the non-blocking pool state respectively. In order to discern p_{in} and p_{nb}, we must construct an attack which can provide us with knowledge of the following components:

1) Input pool seed
2) Non-blocking pool seed
3) External entropic injections

Our attack consists of performing an optimized search for the values of these components.

The LPRNG is most susceptible to attack during early boot. An example boot flow can be observed in Algorithm 1.

Algorithm 1 Example Initialization of PRNG Pools and Non-blocking Pulls

```
1:  function INITSTDDATA(pool)
2:      pool.entropycount ← 0
3:      now ← KTIMEGETREAL
4:      MIXPOOLBYTES(pool, now)
5:      MIXPOOLBYTES(pool, utsname())
6:  end function
7:
8:  function RANDINITIALIZE
9:      INITSTDDATA(inputpool)
10:     INITSTDDATA(blockingpool)
11:     INITSTDDATA(nonblockingpool)
12: end function
13:
14: on boot
15: RANDINITIALIZE
16:
17: GETRANDOMBYTES
18: INJECTENTROPY
19: GETRANDOMBYTES
20: INJECTENTROPY
21: INJECTENTROPY
22: GETRANDOMBYTES
```

The kernel makes extensive use of the non-blocking pool and extracts random values via the `get_random_bytes()` function. In order to perform our attack, we require knowledge of a value pulled from the non-blocking pool, ideally as close to the seeding of the pools as possible. By combing through all calls made by the kernel to `get_random_bytes()` in early-boot, an attacker may be able to identify areas which could potentially leak the necessary value to the user.

It is also necessary for us to know the order of the pulls from the non-blocking pool up until the point of our value leak. This will allow us to generate the correct state for the pulls at the point of leak locally during our attack.

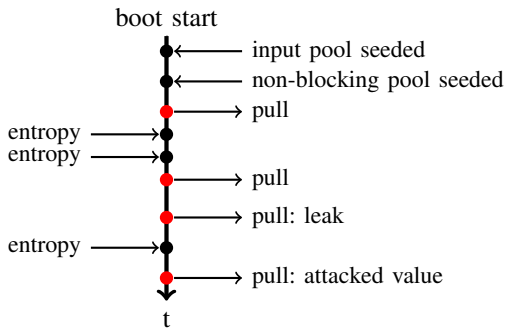

Figure IV.1. Boot flow depicting seeding of pools, injection of external entropy into the input pool and value extraction from the non-blocking pool

The internal state of the input and non-blocking pools are left uninitialized at boot, however on many modern devices, this memory is zeroed. The pools are seeded with an 8-byte `ktime_t` value (Figure IV.2) returned by `ktime_get_real()`. The most significant DWORD (4 bytes) holds the seconds since the epoch from the system Real Time Clock (RTC). The least significant DWORD represents nanoseconds. At first glance, this method of seeding the pools should add 8 bytes of entropy to each pool. In reality, the entropy is significantly less.

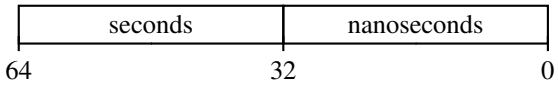

Figure IV.2. ktime_t

The most significant 4 bytes of `ktime_t` is often predictable. For example, an attacker could measure the time taken to reach the RNG initialization function since boot adding it to the `btime` value queried off `/proc/stat` from user-space (a localized attack scenario). Alternatively, as mentioned by Heninger et al., this value can potentially be leaked remotely via TCP timestamps.

As mentioned, the least significant 4 bytes of `ktime_t` represent nanoseconds. Theoretically this should unpredictably hold one of 10^9 possible values to provide just under 30 bits of entropy with each pool being seeded independently from a repeat call to `ktime_get_real()`. However, performing statistical analysis on this value across a range of samples shows a significant bias due to low variance across boots on embedded devices, greatly dropping the effective entropy provided by `ktime_t`. Our attack is predicated on this statistical analysis yielding a significant bias with which we can use to infer likely seed candidates.

As described in Section III, the kernel provides a number of interfaces for injecting entropy into the input pool. However a pull from the non-blocking pool will never cause a pull from the input pool until the input pool has a sufficiently high entropy count to allow such a pull (KEC of at least 192 bits). During early-boot of embedded devices, generally only a small amount of entropy is injected into the input pool and the KEC is low.

Considering a case where there is a sufficiently high KEC of the input pool prior to an extraction from the non-blocking pool, the entropy of the input pool, $H(p_{in})$, is bounded by the entropy mixed in through the seeding of the pool - in addition to the entropy added from external sources; $H(p_{in}) \leq H(s_{in}) + H(ext)$. It follows therefore that the entropy of the attack, $H(A)$, is bounded by to the entropy of the input pool seed combined with the entropy of the non-blocking pool seed - in addition to the entropy added from external sources; $H(A) \leq H(p_{in}) + H(s_{nb})$. This can be optimized further as we can consider the seed of the non-blocking pool to be dependent on the seed of the input pool. This is due to the fact that the non-blocking pool seeding `ktime_t` can be assumed to be an offset from the input pool seeding `ktime_t`. The seed of the non-blocking pool is therefore dependent on the seed of the input pool such that $H(A) \leq H(p_{in}) + H(o_{nb})$ where o_{nb} is the offset of the non-blocking pool seed from s_{in}; $o_{nb} = s_{nb} - s_{in}$. Generally $H(o_{nb}) \ll H(s_{nb})$.

The KEC of the input pool as recorded by the kernel is based off time deltas for each entropy source across calls to `add_timer_randomness()` which is responsible for injecting entropy into the pool (as all injected entropic sources call `add_timer_randomness()`). As the boot process is generally short, it seems unlikely that there will be sufficient time deltas across the available entropy sources in order to accumulate an KEC of 192 bits and therefore the input pool may not be mixed into the non-blocking pool during early boot. The only exception to this might be the `add_interrupt_randomness()` source which holds a state per IRQ; however this source is deprecated in the Linux kernel version under observation in this paper. In such a case, the entropy of the attack, $H(A)$, is exactly equal to the independent entropy of the non-blocking pool seed; $H(A) = H(s_{nb})$.

In order to determine the candidates for s_{in} and o_{nb}, we perform statistical analysis on samples of these values across boots on a target device. This analysis also yields the related Shannon entropies $H(s_{in})$ and $H(o_{nb})$ - giving an indication as to whether our attack would be feasible on the particular target device under investigation.

As $H(A) \leq H(p_{in}) + H(o_{nb}) = H(s_{in}) + H(o_{nb}) + H(ext)$, in order to continue with our attack, we now need to consider entropy injected from external entropy sources, $H(ext)$.

Entropy is injected via the `add_timer_randomness()` call, mixing in a 12 byte `sample` of the current `cycles`, `jiffies` and a value passed from the entropy source, `num`. The sample structure which will be mixed into the input pool looks as follows (kernel 3.0.31):

```
struct {
  cycles_t cycles;
  long jiffies;
  unsigned num;
} sample;
```

The `num` value can be observed by instrumenting the boot flow. One cannot consider the entropy injection to fully be deterministic as there is an observed variance in entropy addition due to `jiffies` and `cycles`. However, both `jiffies` and `cycles` are time-dependent, enabling a probabilistic approach to determine the most likely flow resulting in a particular entropy injection. We generalize `jiffies` and `cycles` as our `time` component.

In order to determine the `time` value most likely to be mixed into the input pool, we map the flow from boot until the point which we require a prediction of the next value pulled from the non-blocking pool. Calls to `add_timer_randomness()` are often called in quick succession and can be considered to be grouped together, forming what we term a *set* of calls. For each set, we record the variance of `time` across each of the calls in the set.

The flow from the first call to `add_timer_randomness()` in a set to the last call we term a *path*. This path is described in offsets from the first value of a component of `time` in the set. For example, the path `[00112]` for the `jiffies` component of `time` describes a set of 5 calls where the first two calls have a value of `jiffies`, the next two calls have a value of `jiffies + 1` and the final call, `jiffies + 2`. We map the paths of the call sets across our range of samples to extract the most likely paths for each respective set. Taking, for example, a set of 6 calls, $path_A$, where $path_A = \begin{cases} [000111] & P = 0.7 \\ [000011] & P = 0.3 \end{cases}$, yielding $H(path_A) = 0.88$ bits of entropy. Entropy injected due to all entropy paths is $H(paths) = \sum_{k=1}^{n} H(path_k)$ where n is the number of sets and the paths are assumed to be independent.

Once we have mapped the paths, we consider the variance of `time` of the first call in each path in

respect to the value of `time` of the final call in the previous path (i.e. the dependency of the paths). We term this offset the *distance,* Δ, between the paths. This distance is the therefore the offset between two sets of calls. The distances between two paths, $path_B$ and $path_A$, is Δ_{BA}. Entropy injected due to variance between sets is $H(\Delta s) = \sum_{k=1}^{n} H(\Delta h_k)$ where n is the number of sets.

The only remaining unknown is the initial value of the `time` which could be dependent on s_{in} resulting in a lower entropy for $H(time|s_{in})$.

Therefore, total entropy due to external entropy injection is $H(ext) \leq H(paths) + H(\Delta s) + H(time)$.

If the series of pulls before the leak is constant, the formalization of the attack is now complete. We can generalize further if there is some variance in the pulls before the leak. Let $L = ((n_1, e_1), (n_2, e_2), \ldots, (n_k, e_k))$ be a vector denoting a series of pulls, where n_i is the number of pulled bytes, and e_i denotes the input pool bytes mixed into the non-blocking pool before the pull, $(paths, \Delta s, time)$. The last pair (n_k, e_k) denotes the pull of the leak. In the case of a constant series of pulls before the leak, $H(L) = H(ext)$. A leak resulting from user-space applications will have a $H(L)$ dependent on variance due to concurrency.

A complete attack is therefore yielded in $H(A) \leq H(s_{in}) + H(o_{nb}) + H(L)$. In order to optimize the search, we attempt candidate values (s_{in}, o_{nb}, L) in descending order of inferred probabilities. Let N be the number of attempts until a success, then the expected number of candidates attempted is $\mathbb{E}(N) = \sum_{k=1}^{n} k \cdot p_k$ where $\{p_k\}_{k=1}^{n}$ are the ordered inferred probabilities ($p_1 \geq p_2 \geq \cdots \geq p_n$) of the candidates. Further search optimization follows if one leak path, L_1, is a prefix of another L_2, i.e. $L_1 = ((n_1, e_1), (n_2, e_2), \ldots, (n_{k_1}, e_{k_1}))$ and $L_2 = ((n_1, e_1), (n_2, e_2), \ldots, (n_{k_1}, e_{k_1}), \ldots, (n_{k_2}, e_{k_2}))$.

V. Random Value Leak

In order to perform our attack, we require a leak of a random value pulled from the non-blocking pool during early boot. We were able to identify a number of areas which could potentially leak the necessary value to the user.

A. Stack Canary/Auxilliary Vector of Zygote

On Android, application processes are spawned by forking the Zygote (`app_process`) process (which is executed on boot).

Zygote initializes an instance of the Dalvik Virtual Machine and preloads the necessary Java classes and resources. It also exposes a socket interface for application spawn requests. When it receives such a request, the Zygote process forks itself to create a new app process with a preinitialized address space layout. This method for spawning processes introduces a known weakness described by Ding et al. [3] to the stack canary protection mechanism due to the fact that forked processes are not execve()'d and therefore the stack canary value and auxiliary vector is inherited from parent process.

As detailed in Section VII-B, on Android versions prior to 4.3, stack canaries are generated from a 4-byte pull from the LPRNG and, as all apps are forked() from Zygote, they share the same, parent, canary value. As any application can simply inspect its own memory space and extract this canary value, this constitutes a leak that we can use to attack the LPRNG state.

On Android versions 4.3 and above, canary values are generated directly from the `AT_RANDOM` of the auxiliary vector which can be leaked in a similar fashion. In fact, leaking the AT_RANDOM value is possible on versions prior to 4.3 as well as the auxiliary vector exists within the process memory space regardless of whether or not it is used for the stack canary.

B. IPv6 Fragment Identifier

An IPv6 packet of size greater than Maximum Transmission Unit (MTU) size of the network interface is split up into fragments and transmitted. In order to defend against packet fragmentation attacks, the kernel assigns a random value to the packet identifier [5]. This hinders the ability of an attacker to guess the fragment identifier. In kernel versions $\geq 3.0.2 < 3.1$, the identifier value is calculated off a value pulled from the non-blocking pool in early-boot (simplified in Algorithm 2).

Algorithm 2 IPv6 fragment generation

```
1: on boot persist
2: hashidentrnd ← GETRANDOMBYTES(4)
3:
4: on generate fragment ident                ▷ (simplified)
5: hash ← JHASH2(address, hashidentrnd)
6: ident ← hash + 1
```

In order to leak our random value, we can send an IPv6 packet of size greater than MTU size to an address we control and capture the fragment identifier (see Figure V.1). For the first packet sent to a destination address, the identifier is usually a hash (*jhash2*) of the random value pulled at boot (*hashidentrnd*) incremented

by 1. We therefore can calculate the value of *hashidentrnd* given the ability to reverse the *jhash2* hashing function.

We are able to actively leak this fragment identifier remotely by sending an IPv6 ICMP "Echo request" with data size greater than target device interface MTU. As the "Echo request" requires the target to respond with an "Echo reply" with exactly the same data - and the size is greater than the interface MTU - a fragmented packet will be returned by the target.

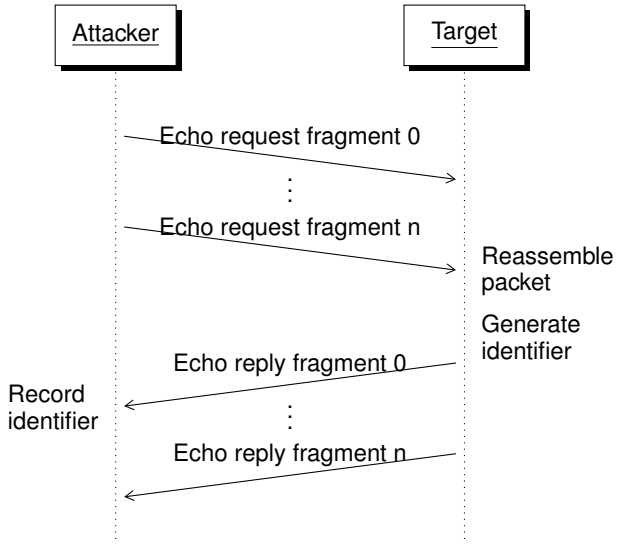

Figure V.1. IPv6 remote fragment leak

C. IPv6 Private Temporary Address

The kernel has the ability to create a private temporary IPv6 address for an interface. The lower 64 bits of the address are randomized (EUI-64) and appended to a valid IPv6 address suffix to create a valid private address. This functionality is enabled via setting `/proc/sys/net/ipv6/conf/all/use_tempaddr` to 2 (the default on Ubuntu 13.10 and 14.04 and potentially on many other modern distributions). Unfortunately, this functionality – while supported by the kernel – is disabled on our Samsung Galaxy S2 target phone.

VI. Experiment

The attack was built against our target device – a Samsung Galaxy S2 running Android 4.2 (Jelly Bean) and kernel 3.0.31.

We make use of the IPv6 Fragment Identifier leak as described in Section V-B.

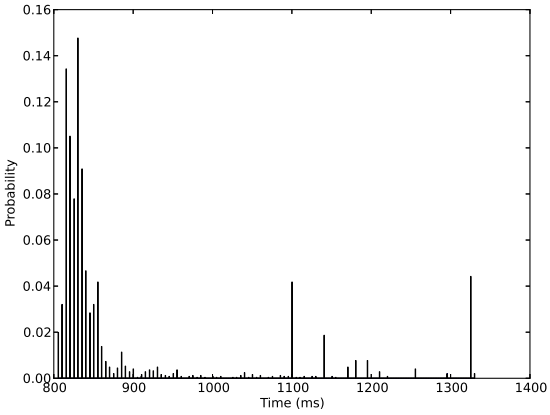

Figure VI.1. PMF depicting sampled input pool seed value (showing a clear bias) on Galaxy S2. 1000 bins of 19 bits

A. Input- and non-blocking pool entropy

In order to define the parameters for the optimized search, we sampled approximately 2,500 input/non-blocking seed pairs, by adding `printk()` debug statements to the RNG initialization function, booting the device, dumping the output of kernel ring buffer with `dmesg` (which we enabled by modifying the `initramfs`) and rebooting (the effect of the `printk()` would need to be compensated for when calculating the search parameters in order to perform a blind, real-world attack).

Interestingly, we observed that the most-significant 4 bytes of `ktime_t` on the ARM systems we tested were always zero. We therefore only needed to consider the least significant 4 bytes (nanoseconds). Probability mass functions were generated off the sampled seeds to determine the search ranges for both the input pool seed, s_{in}, and the offset of the non-blocking pool seed, o_{nb}.

Input pool seed – by experimentation we determined the optimum bin size to be 11.4 bits. The Shannon entropy of the pool, $H(s_{in}) \cong 18$ bits (See Figure VI.1).

Non-blocking pool seed – again by experimentation, the bin size was selected to be 3.3 bits. The Shannon entropy of the pool, $H(o_{nb}) \cong 10$ bits (See Figure VI.2).

Total Shannon entropy due to pool seeds is therefore $H(s_{in}) + H(o_{nb}) \cong 28$ bits.

B. External entropy injection

On our S2 device we observed that the input pool will never be mixed into the non-blocking pool due to the fact that insufficient entropy is injected at

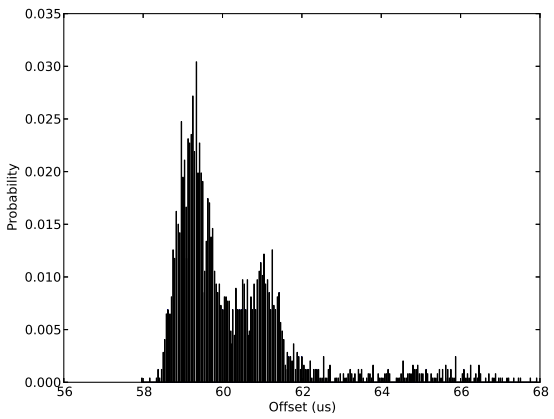

Figure VI.2. PMF depicting sampled non-blocking pool seed offset from input seed on Galaxy S2. 1000 bins of 3.3 bits

boot and the KEC 192-bit threshold is never reached before the leak of the random value. As an attack based on the non-blocking pool only is trivial ($H(A) = H(p_{nb}) = H(s_{nb})$), we modified the source code in order to artificially increase the KEC prior to the leak in order to make sure that the input pool had a KEC of at least 192 bits. We did this by adding 16 calls to `rand_initialize_irq()` and `add_interrupt_randomness()` to the initialization function of the *s3c-udc* USB driver which is executed by the kernel early in the boot flow. This modification allows us to demonstrate an attack on the input pool seed as well. Whilst not necessary for a real-world attack against the Galaxy S2 in early-boot, we wished to prove that an optimized search for the seeds is feasible even with additional external entropic addition - together with the entropy added via the seeding of the input pool itself. The modified flow, including external entropy sources can be seen in Algorithm 3.

Algorithm 3 Modified Flow Including Injected Entropy

1: on boot
2: RANDINITIALIZE
3: ADDINTERRUPTRANDOMNESS ▷ kernel modification
4: GETRANDOMBYTES(8)
5: ADDDISKRANDOMNESS
6: ADDINPUTRANDOMNESS ▷ called by module A
7: GETRANDOMBYTES(4) ▷ called by module B
8: *hashidentrnd* ← GETRANDOMBYTES(4) ▷ called by the ipv6 subsystem

To continue with our attack, we now need to consider entropy injected from external entropic sources via the `sample` structure of the `add_timer_randomness()` call.

On our target device, the 32 bit `sample.cycles` value is always zero. The 32 bit `num` variable is a deterministic value set by the caller of `add_timer_randomness()` which we obtain by instrumenting the kernel boot flow. We therefore only need to consider `sample.jiffies` as our generalized `time` value. Interestingly, the 16 most significant bits are always `0xffff`. As a result, there is an upper bound of 16 bits of entropy per call to `add_timer_randomness()` to which there are 28 calls in our flow. In practice, however, the entropy is significantly less due to the fact that `add_timer_randomness()` is called in quick succession in three separate groupings (sets); the first (which we artificially added by modifying the kernel) with 16 calls, the second with 10 calls and the third with 2 calls. As the calls are executed in quick succession in each set, we observe that the variance of `jiffies` across each of the calls in each respective set is small; i.e. each call in a set has $jiffies + x$ where $x \in \{0,1\}$.

We mapped the paths of the call sets across our range of 2,500 samples to extract the most likely paths for each respective set:

The first set consisting of our artificially injected 16 calls has $Pr_{path_A}[00000000000000] = 1$, therefore entropy $H(path_A) = 0$.

The second set of calls provides some variance:

$$path_B = \begin{cases} [0000000111] & P = 0.822 \\ [0000000011] & P = 0.155 \\ [0000000001] & P = 0.022 \\ others & P = 0.001 \end{cases}$$

with $H(path_B) \cong 0.79$.

The third set is observed to be $Pr_{path_C}[00] = 1$, therefore $H(path_C) = 0$.

Total additional entropy due to injected run-time entropy paths is therefore $H(paths) = H(path_B) \cong 0.79$ bits.

As described in our generalized attack, we now consider the distance between each respective path. The distances between $path_B$ and $path_A$, and $path_C$ and $path_B$ are defined as Δ_{BA} and as Δ_{CB} respectively.

$$\Delta_{BA} = \begin{cases} 79 & P = 0.178 \\ 80 & P = 0.75 \\ 81 & P = 0.069 \\ others & P = 0.003 \end{cases}$$

$$\Delta_{CB} = \begin{cases} 0 & P = 0.507 \\ 1 & P = 0.493 \end{cases}$$

Shannon entropy $H(\Delta_{BA}) \cong 1.05$ bits and $H(\Delta_{CB}) \cong 1$ bit. Total entropy due to distance across the 2,500 samples is therefore $H_\Delta \cong 2.05$ bits. External entropy injected thus far in our attack is therefore $H(path_B) + H(\Delta_{BA}) + H(\Delta_{CB}) \cong 2.84$ bits.

The only remaining unknown is the initial value

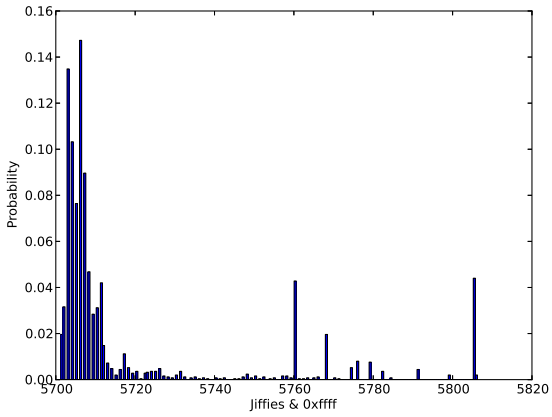

Figure VI.3. PMF depicting sampled jiffies

of the `jiffies` itself. In order to determine this value, we generated a PMF across our sample set (which can be seen in Figure VI.3). Considering this PMF independently, its entropy would be $H(jiffies) = 4.3$ bits, however one can clearly observe a direct correlation between the `jiffies` value and the input seed value (Figure VI.1). Experimentally, we found that `jiffies` increments every 5ms. Furthermore, we observed that the time between the seeding of the input pool (s_{in}) and the first addition of external entropy is fairly constant with a drift of < 5 ms yielding a deterministic relative function $jiffies = f(s_{in})$. Therefore we can consistently predict ($H(jiffies|s_{in}) = 0$) the `jiffies` value for any given input seed according to:

$$\begin{aligned} jiffies &= base + \|s_{in}/t_{tick}\| \\ &= 0x15a4 + \|s_{in}/5M\| \end{aligned}$$

We investigated whether we could perhaps further optimize by considering the relation between the distances and the paths - and between the initial `jiffies` value - in order to determine whether there is a statistically significant dependency which could allow us to determine a likelihood of a certain distance/initial `jiffies` value being present for any specific configuration of $\texttt{path}_\texttt{B}$. Looking at the distance and `jiffies` offset for incidences of the most likely path (`[0000000111]`) only, we notice that the total entropy is improved by a mere ~ 0.09 bits which is statistically insignificant over our sample set.

As the leak in this experiment occurs in kernel-space and before execution concurrency, $H(L) = H(ext)$.

C. Constructing the attack

Based on the above, total Shannon entropy for our attack is $H(A) = 30.84$ bits. The KEC is greater than 192 bits and therefore one can clearly see that the actual entropy is significantly less than that recorded by the kernel. We compute the ranges of candidates to attempt. Each candidate is comprised of s_{in}, o_{nb}, $path$, and Δ. In order to optimize the search, we attempt candidate values in descending order of inferred probabilities. The resultant ordered list of candidate ranges is further split into 400 sets of 50 candidate ranges each in order to assist with parallelization of the search. Each candidate range is 14.82 bits in size, giving a coverage of $\approx 80\%$ (see Figure VI.4).

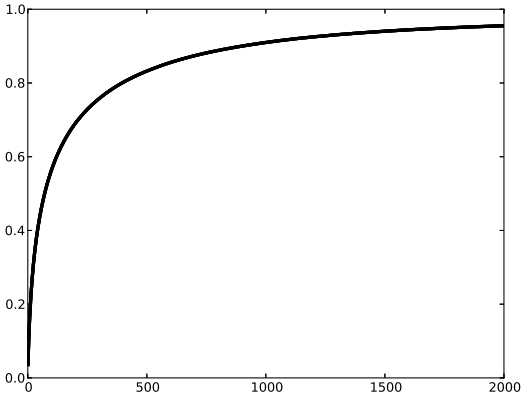

Figure VI.4. Probability of a match within 2000 sets of 50 candidate ranges of size $2^{15.8}$

Let N^* be the number of attempts of our search algorithm until it stops, then the expected number of candidates attempted is $\mathbb{E}(N^*) = \sum_{k=1}^{\psi} k \cdot Pr[N^* = k]$ where ψ is the 8th decile of N:

$$N^* = \begin{cases} n < \psi & Pr[N = n] \\ \psi & Pr[N \geq \psi] \end{cases}$$

Therefore the expected number of attempts $\mathbb{E}(N^*) \cong 2^{26.5}$.

The flow of the search can be seen in Algorithm 4. For each candidate, the algorithm generates an *ident* which is compared against the *expected* (i.e. leaked) value, forming the 'test'.

Algorithm 4 Seed Search with IPv6 Fragment Identifier Leak

```
1:  function TESTCANDIDATE(hashidentrnd)
2:      ident ← JHASH2(address, hashidentrnd)
3:      ident ← ident + 1
4:      found ← ident == expected
5:      return found
6:  end function
7:
8:  for all s_in, o_nb, path_B, Δ_BA, Δ_CB do
9:      RESETPOOLS
10:     INITPRNG(s_in, o_nb)
11:     ADDINTERRUPTRANDOMNESS
12:     GETRANDOMBYTES(8)
13:     ADDDISKRANDOMNESS(path_B, Δ_BA)
14:     ADDINPUTRANDOMNESS(Δ_CB)
15:     GETRANDOMBYTES(4)
16:     hashidentrnd ← GETRANDOMBYTES(4)
17:     found ← TESTCANDIDATE(hashidentrnd)
18:     if found then                    ▷ match found
19:         break
20:     end if
21: end for
```

We performed the search with the sets distributed over 8 1.6 GHz virtual CPU cores in 10 Windows Azure XL Virtual Machine instances (for a total of 80 concurrent executions). We observed that each core executes $\sim 86,000$ ($2^{16.4}$) tests per second with time to cover all 400 sets < 3 minutes (accounting for overhead in range distribution instructions to cloud servers). The cost of running the 10 8-core Windows Azure VM instances is 5¢per core per hour yielding a total cost of attack (given 5 minutes of up-time) of a mere 34¢.

D. Newer devices

The Samsung Galaxy S2 - while still present in large quantities in the market (3.8% share at time of writing [6]) - is now an aging device. However our attack can most likely be applied to a majority of Android devices in the market today (given a suitable information leak). This is due to the fact that over 90% of all Android devices at the time of writing [7, 8] are running Android 4.3 or older and therefore (while we have no concrete data to support this) could be running a vulnerable kernel. Furthermore, we investigated whether our attack could work on some of the newer devices as well (with the latest version of Android - 4.4 KitKat).

The PRNG kernel code of the stock Samsung Galaxy S4 (Exynos version) running Android 4.4 may still be vulnerable to attack using the method described above. Shannon entropy $H(s_{in}) = 16$ bits (Figure VI.5) and $H(o_{nb}) = 12$ bits (Figure VI.6) over 647 samples. However we noticed that the init process on our 4.4 ROM writes to the LPRNG thereby adding entropy to the non-blocking pool (as described in Section VIII). It should however be possible to perform an attack against a S4 device running Android 4.3 using the `AT_RANDOM` leak as described in Section V-A.

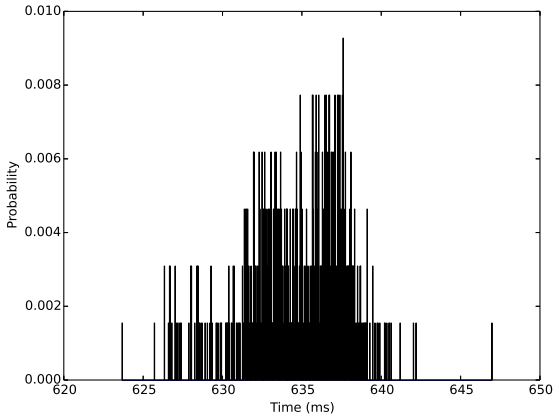

Figure VI.5. PMF depicting sampled input pool seed value on a Galaxy S4. 1000 bins of 14.5 bits

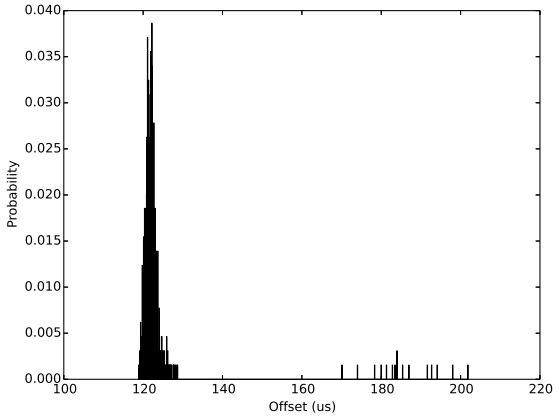

Figure VI.6. PMF depicting sampled non-blocking pool seed offset from input seed on a Galaxy S4. 1000 bins of 6.4 bits

Looking at the available kernel sources for the Motorola G (Android 4.3), we noticed that it too uses a vulnerable version of the PRNG code. We did not investigate whether there is a leak that could enable the attack.

The latest version of the kernel sources available in the git repository (at the time of writing) for the Google Nexus 5 (Hammerhead) include a newer version of the LPRNG code which includes some of the mitigations detailed in Section VIII. We did not spend significant time investigating the resultant PRNG flow and therefore cannot comment on the impact of these mitigations on our attack.

VII. Exploitation Vectors

A. IPv6 fragmentation attacks

Atlasis [9] describes a number of potential attacks using IPv6 fragmentation. Our search algorithm allows us to generate an expected fragment identifier for any destination address. We can use these identifiers to perform an off-path Denial of Service (DoS) against a target (specifically the S2 running vulnerable kernel 3.0.41) as will be described below.

For our Proof of Concept, we perform a DoS against a fragmented "Echo reply" packet (ping reply). Two parties, our target T (S2 mobile phone) and peer X wish to communicate over IPv6. Attacker, A, wishes to disrupt the communication. The nature of the communication is such that IPv6 packets will be fragmented (size > MTU). In this case, X wishes to send an "Echo request" (ping) to T and A wishes to disrupt the reply. The full flow of the attack is as follows (Figure VII.1):

1) Prior to communication between T and X, Attacker A sends an "Echo request" with size > MTU to target T
2) T responds with a fragmented "Echo reply"
3) A uses the fragment identifier from the reply to perform a search for seed data as described in Section VI
4) A then calculates the fragment identifier for a packet from T to destination X
5) A spoofs an "Echo reply" fragment from T and sends it to X with invalid data, IPv6 fragment offset and M flag set to 0 (last fragment)
6) X sends a fragmented "Echo request" to T
7) T responds with an "Echo reply"
8) X does not reassemble the fragmented packets correctly as the invalid fragment from attacker A is spliced in thereby causing the ICMPv6 "Echo reply" checksum to fail; X therefore drops the "Echo reply."

Theoretically, should the attacker be able to construct a fragment with data such as to cause the checksum to be valid, the above attack flow could be used to perform a fragment injection attack.

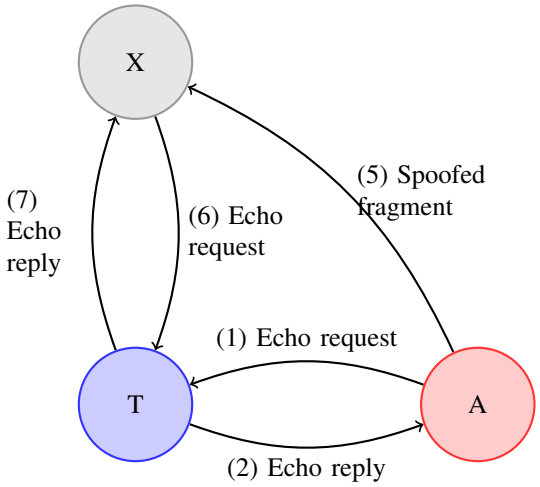

Figure VII.1. IPv6 ping DoS

B. Stack canary bypass

A stack canary is a protection mechanism to mitigate buffer overflow vulnerabilities. A stack canary consists of a persisted random value which is placed on the stack at a point where a buffer could potentially harm the integrity of the data or code execution. Code then checks that the value on the stack matches the expected persisted value. Should a mismatch occur, execution is halted or transferred to an appropriate error handling routine. The GNU Compiler Collection (GCC) compiler generates code which pushes the canary value (`__stack_chk_guard`) onto the stack after the return address but before any local variables (see Figure VII.2). This value is checked when the function returns.

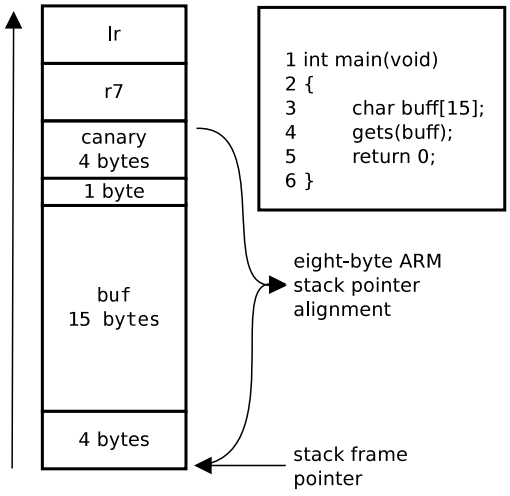

Figure VII.2. Example stack layout depicting canary placement

On Android 4.2.2 and below (relevant to our Galaxy S2 running Android 4.1.2), the canary value is con-

structed using a value pulled from the /dev/urandom device exposed by the LPRNG module.

When Android's libc.so (Bionic) is loaded, the dynamic linker (/system/bin/linker), calls __guard_setup() which assigns the stack canary pointer as follows:

```
fd = open("/dev/urandom", O_RDONLY);
if (fd != -1) {
  ssize_t len = read(fd, &
      __stack_chk_guard, sizeof(
      __stack_chk_guard));
  ...
```

On Android 4.3 and above (relevant to the Galaxy S4 stock ROM which we tested), the stack canary value is constructed using the auxiliary vector. The auxiliary vector is a list of key-value pairs that the kernel's ELF binary loader (/fs/binfmt_elf.c in the kernel source) populates when a new executable image is loaded into a process. AT_RANDOM is one of the keys that is placed in process's auxiliary vector; its value is a pointer to a 16 random byte value provided by the kernel. The dynamic linker uses this value to initialize the stack canary protection.

An executable image is loaded into a process using the execve system call. The entry point of the execve system call is the sys_execve() function. This function delegates its work to the do_execve() routine. Figure VII.3 illustrates the flow from do_execve() to the point where the 16 random bytes of AT_RANDOM are generated and placed in the memory of the user-space process.

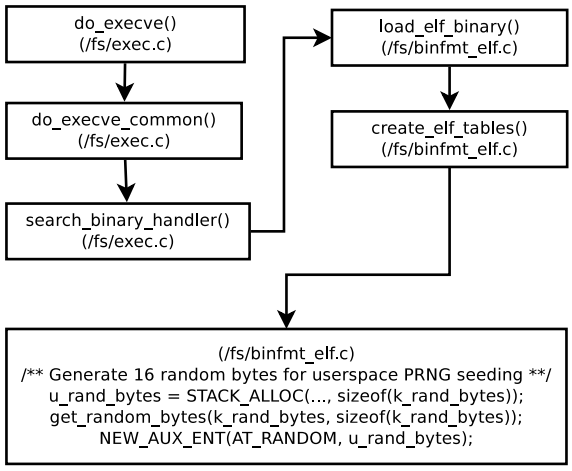

Figure VII.3. Auxiliary vector AT_RANDOM generation

On libc.so load, the dynamic linker calls the C runtime initializer __libc_preinit() which then calls __libc_init_common(). The latter is responsible for assigning the stack canary pointer as follows:

```
__stack_chk_guard = *reinterpret_cast<
    uintptr_t*>(getauxval(AT_RANDOM));
```

As can be seen from the above, the stack canary is the 4 upper bytes of AT_RANDOM. The kernel generates an AT_RANDOM for each process, therefore each process will have a unique stack canary value.

Our attack allows us to generate a likely candidate for the stack canary of a process executed on boot. Once we have performed a successful search for the random pool seeds (and therefore can re-create the pool states locally), we can then extract bytes from our local non-blocking pool in the order that they are extracted on boot. This allows us to call get_random_bytes() the prerequisite amount of times up until the point where the target process's canary is to be pulled from the pool. For the initial processes executed (such as /sbin/ffu and /system/bin/e2fsck) we can generate canary values with a probability that tends to determinism. As further processes are executed in time however, the probability that we are able to generate the correct value is impacted. This is due to the variance in process execution order due to concurrency - which affects us in two ways: Firstly, the scheduler is able to schedule execution concurrently across a number of logical threads and therefore it's difficult to consistently predict the order in which each process will perform an extraction from the non-blocking pool. Secondly, there is a race condition that can occur within the extraction of AT_RANDOM from the non-blocking pool itself [1]. The race condition is due to the fact that entropy is extracted in 10 byte blocks. AT_RANDOM is 16 bytes long, so two extractions of 10 bytes each need to be performed in order to pull the 16 bytes from the non-blocking pool. As two processes may be scheduled to execute concurrently, the extraction from the non-blocking pool by the second process could conceivably take place before the extraction of the final 10 bytes of AT_RANDOM but after the extraction of the first 10 bytes. Any attack that is predicated on prediction of the canary value for a process will therefore need to take this variance into consideration.

In order to determine the effect of concurrent execution on a process in early boot on the S2, we recorded canary values for fixed seeds & entropy across 503 samples. The probability that the most likely canary value occurs for each process can be seen in Figure VII.4.

As an example of an early boot service that has been found vulnerable, in September 2013 we privately disclosed a buffer overflow vulnerability in Android 4.3's keystore service to Google (CVE-2014-3100) (Hay and Dayan [10]). This vulnerability was fixed

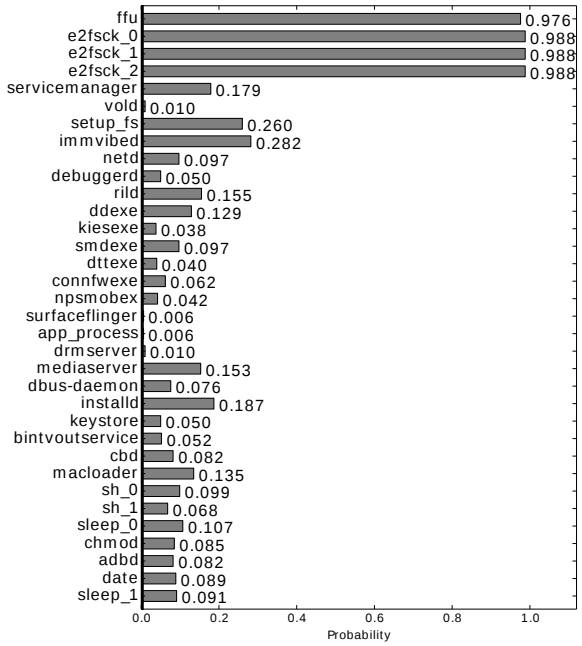

Figure VII.4. Probability of most likely candidate for boot processes on Galaxy S2

in Android 4.4 and subsequently publicly disclosed in June 2014. For the practical exploitation of this vulnerability, one would need to bypass the stack canary protection. As described above, our attack could potentially be used to ascertain the stack canary value (with a certain probability due variance as a result of concurrency) for the `keystore` process as it is launched in early boot.

VIII. Mitigation

A. State persistence

It is widely recommended that entropy is persisted across boots by pulling prior to device shutdown and mixing the pulled value on next boot. We observed that numerous Linux distributions do in fact do this, however this measure is of limited effectiveness. This is the due to fact that this value is usually persisted on the filesystem and is pulled from a user-space script. Therefore, the PRNG can still be attacked prior to the entropy being mixed in in user-space; for example in an attack such as our fragment injection attack as described in Section VII-A, or by attacking the stack canary of the processes executed prior to script execution. There has been some discussion regarding applying the persisted state in kernel-space (perhaps passed by the bootloader) [11]. In our opinion such a move could provide good mitigation in certain use-cases.

B. Trusted external entropy injection

There is a move to use trusted external sources such as web-based sources in order to add entropy to the pools on boot (such as Ubuntu's Pollinate service, or random.org's randbyte api) and doing so is to be encouraged. However the kernel network stack is usually brought up only after the kernel RNG code and the external entropy request is usually mixed in from user-space.

C. Hardware RNG inputs

Hardware Random Number Generators (HWRNGs) are becoming increasingly available on modern platforms. In 2012 Intel added support for a new instruction, RDRAND, in their Ivy Bridge CPUs. RDRAND is the output of a 128-bit HW PRNG that is compliant to NIST SP 800-90A. Furthermore, other device vendors provide HWRNGs as IP blocks on their SoCs.

The Linux kernel supports the `RDRAND` Intel instruction (via `ARCH_RANDOM`) which the kernel RNG code uses to mix values into the entropy pools.

On devices such as those using Qualcomm's MSM, Samsung's Exynos, and other SoCs with RNG support, the HWRNG is provided via a kernel device driver and exposed over `/dev/hw_random`. A user-space daemon/script is often used to mix in entropy pulled from the HWRNG on these devices into the LPRNG by reading from `/dev/hw_random` and writing to `/dev/random` at regular intervals [12].

If the SoC supports the `CONFIG_ARCH_RANDOM` kernel flag (meaning that there is either support in the instruction set or via a HWRNG IP block), the LPRNG module will mix in HWRNG values during initialization of the entropy pools. This solves the problem of low boot-time entropy.

Though a number of modern, commonly used ARM-based SoCs do have an HWRNG, random values are generally not pulled from the HWRNG by the LPRNG device itself (rather, it is exposed through a device as explained above). This is true for the kernels of the Google Nexus 4/5 and Samsung Galaxy S4/S5 which we investigated.

The LG G2 and G3 kernels includes code (`/arch/arm/mach-msm/early_random.c`) which utilizes hardware-generated random numbers via a call to TrustZone during early boot. This could also mitigate issues with low boot-time entropy.

We believe that our research may still be relevant in a world of HWRNGs. It is conceivable that some HWRNGs may be susceptible to external manipulation (such as by changing their physical properties to influence the robustness of their randomness) [13, 14]. Also, some implementations might make use of microcode

as part of their implementation and it is conceivable that there might be instances of microcode that could be vulnerable to exploitation (the impact of which is demonstrated by Hornby [15]). There has been some discussion on whether the risk of instruction compromise should be mitigated via hashing the RDRAND value into the entropy pools instead of XOR-ing it in [16, 17].

D. Changes to the PRNG in latter Kernels

1) Device randomness entropic source: In Kernel 3.4.9 a new entropic source for adding **device randomness** was added. The `add_device_randomness()` function calculates a time value ($jiffies \oplus cycles$) and mixes it - together with a value provided by the caller - in to both the input and non-blocking pools. The time value may be deterministic or predictable using the same techniques as described in our attack above, however the use of the call to mix in an unpredictable value to the pools could be interesting. According to the Kernel developers, this entropy source was added in order to provide a method for differentiating entropy flows across multiple devices and not to specifically add entropy itself. The effectiveness of this measure depends on how it's used. Any use which adds a value which can be discerned by an attacker would not prevent an attack. For example, adding the MAC address of a specific device would only be effective if an attacker could not discerned that MAC address (which might be possible both remotely and locally). On the flip side using an applications processor chip-id of reasonable length which might only be determinable locally by a privileged process could potentially provide a higher level of protection; or alternatively adding the value of an uninitialized device register could also be useful (provided that there is no way to externally predict or influence the value of the register).

2) Improvements to add_timer_randomness(): The `add_timer_randomness()` function now mixes in early boot entropy into the non-blocking pool if there is insufficient entropy in the pool (< 128 bits). This removes the effect of completely ignoring external entropy sources prior to the input pool having accumulated sufficient entropy in order to have been mixed in to the non-blocking pool. Nevertheless the resultant effect on entropy may still be predictable or deterministic for the same reasons as before.

3) Return of interrupt randomness entropic source: The interrupt randomness addition routine has been rewritten and now can actively be used by the kernel. We have not investigated the effectiveness of the new implementation.

4) x86 RDSEED: `RDSEED` is an x86 instruction set extension which will be introduced in Intel Broadwell CPUs [18]. It is guaranteed to always pull from a True Random Number Generator (TRNG) and is therefore useful for seeding PRNGs. Patches have been submitted to Linux kernel 3.15 to support this instruction [19].

IX. Conclusion

In this paper we demonstrated a practical remote attack on the LPRNG as used in Android in the majority of Android devices in use today. We modified the kernel in order to show that an attack at early boot is feasible even when considering limited entropy sources. Additionally, we briefly commented on current/future mitigations.

Even though weaknesses in the LPRNG are well known and have been discussed in various publications, we believe that our research is helpful in quantifying the issue and demonstrating how such an attack could be built. We hope that our work will encourage device vendors and PRNG developers alike to give thought as to how their chosen random number generators actually function and highlight the risks of ineffective implementations.

References

[1] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Mining your Ps and Qs: detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, page 35. USENIX Association, 2012. URL http://dl.acm.org/citation.cfm?id=2362793.2362828.

[2] Andrew Becherer, Alex Stamos, and Nathan Wilcox. Cloud Computing Security: Raining on the Trendy New Parade, 2009. URL https://www.isecpartners.com/media/12952/cloud-blackhat-2009-isec.pdf.

[3] Yu Ding, Zhuo Peng, Yuanyuan Zhou, and Chao Zhang. Android Low Entropy Demystified. In *IEEE International Conference on Communications (ICC)*, 2014.

[4] NetMarketShare. Market share for mobile, browsers, operating systems and search engines, 2014. URL http://marketshare.hitslink.com/.

[5] Gregkh. Patch "ipv6: make fragment identifications less predictable" has been added to the 3.0-stable tree, 2011. URL http://permalink.gmane.org/gmane.linux.kernel.stable/16086.

[6] AppBrain. Android phone market share, 2014. URL http://www.appbrain.com/stats/top-android-phones.

[7] AppBrain. Android SDK version market shares, 2014. URL http://www.appbrain.com/stats/top-android-sdk-versions.

[8] Google. Dashboards, 2014. URL https://developer.android.com/about/dashboards/index.html?utm_source=ausdroid.net.

[9] Antonios Atlasis. Fragmentation (Overlapping) Attacks One Year Later..., 2013. URL https://www.troopers.de/wp-content/uploads/2013/01/TROOPERS13-Fragmentation_Overlapping_Attacks_Against_IPv6_One_Year_Later-Antonios_Atlasis.pdf.

[10] Roee Hay and Avi Dayan. Android Keystore Stack Buffer Overflow, June 2014. URL http://www.slideshare.net/ibmsecurity/android-keystorestackbufferoverflow.

[11] Jason Cooper. Re: [RFC/PATCH 0/3] Add devicetree scanning for randomness, 2014. URL https://lkml.org/lkml/2014/2/12/508.

[12] Code Aurora Forum. android_external_qrngd. URL https://github.com/CyanogenMod/android_external_qrngd/blob/cm-10.2/qrngd.c.

[13] A Theodore Markettos and Simon W Moore. The Frequency Injection Attack on True Random Number Generators. *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 317–331, 2009. doi: 10.1007/978-3-642-04138-9_23.

[14] M Soucarros, C Canovas-Dumas, J Clediere, P Elbaz-Vincent, and D Real. Influence of the temperature on true random number generators. In *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, pages 24–27, 2011. doi: 10.1109/HST.2011.5954990.

[15] Taylor Hornby. Prototyping an RDRAND Backdoor in Bochs, 2014. URL https://defuse.ca/files2/poc/pocorgtfo03.pdf.

[16] Ingo Molnar. [PATCH 07/10] random: add new get_random_bytes_arch() function, 2012. URL http://thread.gmane.org/gmane.linux.kernel/1323386/focus=1332780.

[17] Theodore Ts'o. [PATCH] random: use the architectural HWRNG for the SHA's IV in extract_buf(), 2013. URL https://groups.google.com/forum/#!topic/linux.kernel/QgrTQMMufaM.

[18] John Mechalas. The Difference Between RDRAND and RDSEED, 2012. URL https://software.intel.com/en-us/blogs/2012/11/17/the-difference-between-rdrand-and-rdseed.

[19] Michael Larabel. Linux 3.15 Random To Support Intel's RDSEED, 2014. URL http://www.phoronix.com/scan.php?page=news_item&px=MTY1NDY.