Microsoft

The Windows 10 random number generation infrastructure

Niels Ferguson

Published: October 2019

For the latest information, please see
https://aka.ms/win10rng

This document describes the Windows 10 random number generation infrastructure. It is intended for readers who are familiar with random number generators and entropy collection terminology.

PRNG infrastructure

The PRNG infrastructure generates random numbers for all areas of the OS based on a single seed in the kernel. We will discuss the seeding of the PRNG infrastructure in the second part of the document.

Basic PRNG

All PRNGs in the system are SP800-90 AES_CTR_DRBG with 256-bit security strength using the df() function for seeding and re-seeding (see SP 800-90 for details).

AES_CTR_DRBG is a cryptographic pseudo-random number generator. That means that given any number of output bytes, it is computationally infeasible to determine the internal state of the PRNG or any other output byte. As the name suggests, AES_CTR_DRBG is based on using the AES block cipher in a counter mode, similarly to the way it is used to create a key stream in AES-GCM.

This PRNG has *backtracking resistance*; after it has produced an output, the updated PRNG state does not contain enough information to recover that output. The backtracking resistance property is maintained throughout the RNG system.

Buffered PRNG

The Basic PRNGs are not used directly, but rather through a wrapping layer that adds several features.

- A small buffer of random bytes to improve performance for small requests.
- A lock to support multi-threading.
- A seed version.

Buffering

The buffering is straightforward. There is a small buffer (currently 128 bytes). If a request for random bytes is 128 bytes or larger, it is generated directly from AES_CTR_DRGB. If it is smaller than 128 bytes it is taken from the buffer. The buffer is re-filled from the AES_CTR_DRBG whenever it runs empty. So, if the buffer contains 4 bytes and the request is for 8 bytes, the 4 bytes are taken from the buffer, the buffer is refilled with 128 bytes, and the first 4 bytes of the refilled buffer are used to complete the request, leaving 124 bytes in the buffer. When bytes are taken from the buffer, their locations are wiped (zeroed). This maintains the backtracking resistance. The bytes in the buffer are discarded (i.e. not used but not wiped) on every reseed; the next time a small request arrives the buffer is overwritten with new random bytes.

(The implementation will also occasionally throw away bytes in the buffer to improve the memory alignment of requests.)

We give a short sketch of the security analysis of this buffer construction:

- Bytes in the buffer are from the DRBG. Bytes are used only once, and wiped when they are given out. Thus, every output byte is a byte from the DRBG; the buffering merely re-orders the bytes. The re-ordering is determined by the sequence of requests for random numbers. These requests can depend on previous random bytes, but not on the random bytes currently in the buffer. As we assume that all output bytes of the DRBG are independent and equivalent, the re-ordering does not change the independence or probability distribution of the output bytes.
- The buffer maintains secrecy. After a call to generate bytes, the buffered RNG state no longer has the data to reconstruct the output it provided. Of course, at any time the RNG state has the information to predict all future outputs until the next reseed.
- Discarded bytes (on reseed or to improve alignment) are not wiped. These bytes are never given as output to a caller. However, it would be perfectly okay for a caller to have requested those bytes, and published them. Thus, there is no security concern with not wiping discarded bytes.

Rationale: Each request from an AES_CTR_DRBG has a significant overhead as it requires an AES key expansion, producing the output bytes, and producing 48 additional bytes for the next DRBG state. For large requests, the overhead is not significant, but for small requests the overhead cost dominates. The buffer allows the system to amortize the overhead over multiple requests, significantly reducing the average cost for small requests.

Locking

All access to the buffered PRNG state is gated on the lock. This ensures that multiple threads do not attempt to read or modify the same state at the same time.

Root PRNG

There is a single (buffered) Root PRNG in kernel mode maintained by the CNG.SYS driver. All random bytes in the system are derived in some way from this PRNG. The Root PRNG is instantiated upon system startup. The entropy system (described below) initializes and reseeds the root PRNG.

Kernel per-processor PRNG

Using a single root PRNG would create a bottleneck on large computers. To avoid a bottleneck, the kernel mode has one (buffered) PRNG state per logical processor. (A 4-core CPU with hyperthreading has 8 logical processors from the OS point of view.) The per-processor PRNG states are also maintained by the CNG.SYS driver.

When a caller asks for random data in kernel mode, the code determines what logical CPU it is running on and checks if a PRNG state for this CPU has already been allocated. If it has, it uses that PRNG state to provide the requested random bytes. If there is no per-CPU state for the current CPU it attempts to allocate a PRNG state and instantiate it with seed material from the root PRNG. If that succeeds, it proceeds as before, servicing the request from the newly-created PRNG. If the allocation fails, the request is serviced from the root PRNG.

In practice, the allocations fail almost never and all requests are served by the per-CPU state. This has a number of very nice properties. The number of PRNG states scales with the size of the machine, providing both low memory footprint on small machines and high throughput on big machines. The memory for each per-processor PRNG state is allocated from that NUMA node which improves memory locality. Using a per-processor state also has good cache-locality, and there is very low contention on the PRNG state lock.

We also have the property that a request for random bytes *never fails*. In the past our RNG functions could return an error code. We have observed that there are many callers that never check for the error code, even if they are generating cryptographic key material. This can lead to serious security vulnerabilities if an attacker manages to create a situation in which the RNG infrastructure returns an error. For that reason, the Win10 RNG infrastructure will never return an error code and always produce high-quality random bytes for every request.

Process base PRNG

For each user-mode process, we have a (buffered) base PRNG maintained by BCryptPrimitives.dll. When this DLL loads it requests a random seed from kernel mode (where it is produced by the per-CPU states) and seeds the process base PRNG. If this were to fail, BCryptPrimitive.dll fails to load, which in most cases causes the process to terminate. This behavior ensures that we never have to return an error code from the RNG system.

Process per-processor PRNG

Just like in kernel mode, we have per-processor buffered PRNG states in each process. These are again created on-demand and seeded from the process base PRNG. The per-processor PRNG states are maintained by BCryptPrimitives.dll.

A running machine can have thousands of PRNG states. If the machine has N logical CPUs, and M processes, then there can be up to (N+1)(M+1) PRNG states. This could grow to a very large number on very large machines, but this has not been a problem in practice. (Very large machines tend to have very large memories, and they tend to run a few very large processes rather than thousands of smaller ones.)

Reseeding the tree

All PRNG states in Windows 10 are reseeded on a regular schedule.

The root PRNG keeps a seed version. This is a 64-bit count of how many times the root PRNG has been seeded since boot. This counter value is published in a memory location that is readable and writable from kernel mode, and readable (but not writeable) in user mode.

All buffered PRNG states, both in user mode and kernel mode, keep track of their current seed version. A seed version of X implies that the PRNG was reseeded with data derived from the root PRNG when the root had seed version X. Any time a PRNG produces seed data for another PRNG, it also provides its current seed version, which the target PRNG stores.

Any time a PRNG is asked for random data, it checks its seed version against the root PRNG's seed version. (This check is very quick as it only involves a memory read and a 64-bit comparison.) If they are not equal, the PRNG will first reseed from its parent before producing output.

As an example, let's assume that we have a full set of PRNGs with seed version 7, and the root PRNG is just being reseeded by the entropy system. The root PRNG will increment its seed version to 8. A few microseconds later an application asks for 13 random bytes.

The application's thread selects the user-mode per-processor PRNG state and asks for 13 random bytes. This state checks its seed version and finds that it is out of date. It asks its parent, the process base PRNG for 32 random bytes to reseed with. The process base PRNG finds that it is out of date and performs an IOCTL request to the kernel for 32 random bytes to reseed itself. The IOCTL request asks the kernel per-processor PRNG state, which finds itself out of date and asks the root PRNG for seed material. The root is always up-to-date (by definition) so it provides reseed data to the kernel per-processor PRNG state. This reseeds, and provides the seed material to the process base PRNG, which reseeds and provides seed material to the user-mode per-processor PRNG state, which reseeds and provides the applications with the requested 13 bytes.

Subsequent requests require much less work. Once a PRNG has been reseeded it does not need to reseed until the next time the root PRNG reseeds. Furthermore, each PRNG in the system caches up to 128 bytes of output, so the first time a PRNG is asked for seed material it will generate 128 bytes and hand out 32 bytes. The next time it is asked for 32 bytes of seed material, it can be provided from the buffer. For example, if another thread on another CPU in the same process asks for random data, it will only reseed the per-processor state for that CPU. The seed material will be copied from the buffer of the process base CPU, making this a much faster operation.

This on-demand reseeding has the same effect as if we were to forcefully reseed every PRNG in the system whenever the root PRNG was reseeded, but it is far more efficient, and it avoids updating PRNG states that are not being used at all.

There is, however, a significant cost to reseeding the root PRNG. With hundreds of processes the system can contain thousands of PRNG states. Thus, the total cost of a reseed is quite high. Therefore, frequent root PRNG reseeding are avoided.

Random number generation APIs

The following are the primary APIs for random number generation in Windows 10.

SystemPrng

This is a function exported by CNG.SYS and available to all code in kernel mode. It is the primary interface to the kernel-mode per-processor PRNG system.

ProcessPrng

This is the primary interface to the user-mode per-processor PRNGs. It is implemented in BCryptPrimitives.dll.

BCryptGenRandom

The BCRYPT_RNG_ALGORITHM produces random data by calling the per-processor AES-CTR-DRBG PRNGs. This works both in kernel mode and user mode.

The BCRYPT_RNG_FIPS186_DSA_ALGORITHM and BCRYPT_RNG_DUAL_EC_ALGORITHM are no longer supported. For compatibility, these algorithms still work, but they produce output from the same per-processor AES-CTR-DRBG states as the default RNG algorithm.

CryptGenRandom

The Microsoft CSPs use ProcessPrng function to produce random bytes.

RtlGenRandom

RtlGenRandom uses the ProcessPrng function to produce random bytes.

Entropy system

The entropy system consists of several components:

- Entropy sources
- Entropy pools that store entropy from the entropy sources
- Reseed logic that determines when and how to reseed the root PRNG from the pools

Entropy sources

There is a kernel API for creating new entropy sources. Windows 10 supports three types:

- Low pull
- High pull
- High push

There are minor differences in how the entropy provided by different entropy source types is handled. In general, low sources are assumed to provide unconditioned entropy events; such as mouse movements. High sources are assumed to provide high-quality random bytes. All sources can provide entropy data at any time, but a pull source is additionally notified whenever the system reseeds the root PRNG from the entropy pools. This allows an on-demand source to provide entropy for every reseed without having its own timer logic.

For any source, the first time it provides entropy the data is used to directly reseed the root PRNG. The entropy pools are bypassed. The primary reason is to ensure that as soon as an entropy source provides data, PRNG output depends on the data from that source. Furthermore, it allows an external caller to force a reseed of the PRNG tree.

All subsequent times that a source provides entropy, the entropy is put in the entropy pools.

As mentioned above, reseeding all PRNGs in the system is expensive; we recommend against frequent creation of new entropy sources.

Reseeding the root

The root is reseeded from the pools on a time schedule. After the initial seeding at boot time (documented below) the first reseed is scheduled 1 second later. The interval between reseeds is tripled every time, so subsequent reseeds happen 3, 9, 27, etc. seconds after the previous one. The interval keeps increasing until it hits the limit, which is currently at 3600 seconds (1 hour). Detail: All timers used for this schedule have a 33% variability; this allows the OS to trigger the timer at a point in time when the CPU is already awake and avoids having to wake the CPU to only process a reseed. (Waking up the CPU is expensive in terms of power.)

Rationale: At early boot we want to get as much entropy into the system as soon as we can. On the other hand, reseeds are very expensive (both in time and power consumption), and if we are in a situation where the flow of entropy is low, we want to collect more entropy between reseeds. The progressive slow-down of the reseed schedule is a compromise between these goals.

Multiple entropy pools

Each entropy pool is implemented as a SHA-512 computation; all data provided to the entropy pool is appended, and when the pool is used the SHA-512 of all the data is used as output of the pool.

The system can have multiple entropy pools. When there are N pools, entropy events from each source are distributed round-robin fashion over the N different pools. This ensures that each pool has approximately the same inflow of entropy.

The N pools are numbered 0,…,N-1. Pool 0 is used on every reseed. Pool 1 is used (in addition to pool 0) every 3rd reseed, pool 2 is used every 9th reseed (in addition to pools 0 and 1), etc. In general, pool k is used every 3^kth reseed (in addition to all the lower numbered pools).

On startup, only one pool exists, and none of the multi-pool logic is enabled until the reseed interval hits the maximum. Once the reseed interval hits the maximum, the creation of additional entropy pools is enabled. A new pool is created whenever it would have been used in a reseed if it had existed. Thus, once the reseed interval hits the maximum, there are two more reseeds with just one pool. On the third reseed a second pool would have been used, so it is created. From that point forward entropy events are divided between the pools, and pool 1 is used every 3rd reseed. Similarly, 9 reseeds later the 3rd pool is enabled.

The design rationale of the multiple entropy pool system is given in an appendix.

Distributing entropy over the pools

By default each entropy source distributes its entropy events over the pools in a round-robin fashion. If there are 3 pools, consecutive entropy events are put in pools 0, 1, 2, 0, 1, 2, ….

The only difference between the handling of entropy from low and high entropy sources is that the first 32 bytes produced by a high entropy source after a reseed from the pools is always put in pool 0. Thus, a high entropy source will preferentially affect the next reseed.

A typical high-pull source will be active just after every reseed, and it will produce 64 bytes of high-quality random data. The first 32 bytes go into pool 0; the remaining bytes go into the next pool in the round-robin schema.

Initial seeding

Initial seeding starts in Winload before the Ntoskrnl has been loaded. Winload retrieves the following items

- Seed file
- External entropy
- TPM randomness
- RDRAND randomness
- ACPI-OEM0 table
- Output from the UEFI entropy provider
- The current time

The individual entropy sources are discussed later.

The data from these sources is hashed together (using SHA-512) and winload uses the result to seed a SP 800-90 AES-CTR-DRBG. This DRBG produces 48 bytes of output that is passed on to CNG to be used at startup. (Other output bytes are passed to the kernel for future use.) Winload also passes the per-source results to CNG to be included in diagnostic event logs.

When CNG loads, it instantiates the root PRNG using the 48 bytes passed by winload. It then starts each of its own entropy sources (described below). Any entropy source that provides data will of course trigger a root PRNG reseed. Note that the AES-CTR-DRBG reseed function combines the existing state of the PRNG with the input so that the existing state is not lost.

Entropy sources

Windows 10 has many entropy sources; these work together to ensure that the OS has good entropy. Different entropy sources guarantee good entropy in different situations; by using them all the best coverage is attained.

Interrupt timings

The primary entropy source in Windows 10 is the interrupt timings. On each interrupt to a CPU the interrupt hander gets the Time Stamp Count (TSC) from the CPU. This is typically a counter that runs on the CPU clock frequency; on X86 and X64 CPUs this is done using the RDTSC instruction.

Let TSC[i] be the TSC value for interrupt i on a 64-bit CPU. To reduce the amount of data, multiple TSC values are first combined using the following rule:

```
n := floor(i/64) mod 32
B[n] <- (B[n] >>> 19) xor TSC[i]
```

Here B is an array of 32 words of 64 bits each. An equivalent way to describe this is that 64 consecutive TSC values are each rotated by a different rotation amount, the results xorred together, and the result xorred into the B[n] word.

Every 1024 interrupts half of the B array has been updated and the interrupt handler schedules a DPC that calls back to the interrupt timing entropy source code. This code copies out the 128 bytes that were just updated and feeds them to the entropy system as an entropy event. While this is ongoing, the interrupt handler routine is updating the other half of the B array.

The total amortized cost of gathering the interrupt timings is around 30 cycles/interrupt; about 25 in the actual interrupt handler and about 5 in the further processing.

On a 32-bit CPUs the same system is used, but the B array consists of 64 words of 32 bits and the rotation constant is 5.

Startup

On startup the interrupt handlers start collecting data before the CNG driver is loaded. When the CNG driver registers its callback with the kernel, the kernel calls the callback once for every logical CPU. The CNG entropy source code combines all that information (containing the collected TSC data from all the interrupts that have occurred so far) and provides that as the first entropy event (which then bypasses the pool and is used to reseed the root). This ensures that as much entropy as is available is pushed into the root PRNG on startup.

On a typical boot cycle there are hundreds of interrupts available at startup.

Analysis

The interrupt handler code has a feature that allows a special driver to collect the raw TSC data. Essentially the driver can set up a second array in which the interrupt handler will store the raw TSC data (in addition to the normal rotate-and-xor buffer operation). The driver can gather the data from the buffer during the callback and write the raw TSC data to disk.

Having gathered the raw data, statistical analysis by us and some of our customers indicate that TSC values have more than 1 bit of entropy each. As a typical machine will see hundreds of interrupts per second, the total entropy flow is quite high.

TPM

During boot, Winload gets 40 random bytes from the TPM (if present) and uses them as one of the entropy sources. This use of the TPM can be disabled through a BCDEDIT setting, and is also disabled during a safe boot. (This allows the OS to boot even on a machine where the BIOS can’t find the TPM and hangs when it is accessed.)

Once the OS is booted, the TPM driver loads, registers as an entropy source, and provides 64 bytes of entropy for each reseed. Due to TPM limitations, the TPM entropy source provides entropy at most once every 40 minutes.

RDRAND/RDSEED

The Intel RDRAND instruction is an on-demand high quality source of random data.

If the RDRAND instruction is present, Winload gathers 256 bits of entropy from the RDRAND instruction. Similarly, our kernel-mode code creates a high-pull source that provides 512 bits of entropy from the RDRAND instruction for each reseed. (As a high source, the first 256 bits are always put in pool 0; providing 512 bits ensures that the other pools also get entropy from this source.)

Due to some unfortunate design decisions in the internal RDRAND logic, the RDRAND instruction only provides random numbers with a 128-bit security level. The Win10 code tries to work around this limitation by gathering a large amount of output from RDRAND which should trigger a reseed of the RDRAND-internal PRNG to get more entropy. Whilst this solves the problem in most cases, it is possible for another thread to gather similar outputs form RDRAND which means that a 256-bit security level cannot be guaranteed.

Based on our feedback about this problem, Intel implemented the RDSEED instruction that gives direct access to the internal entropy source. When the RDSEED instruction is present, it is used in preference to RDRAND instruction which avoids the problem and provides the full desired guarantees. For each reseed, we gather 128 output bytes from RDSEED, hash them with SHA-512 to produce 64 output bytes. As explained before, 32 of these go into pool 0 and the others into the 'next' pool for this entropy source.

Seed file

The seed file is a registry entry in the system hive. It is written by the OS for use during the next reboot. This is the primary source of entropy for the OS during boot, except in the first boot after installation.

To write a new value for the seed file, the system requests 64 bytes from the system PRNG (this is the kernel-mode per-processor PRNG) and writes it to the registry.

A new seed file is written at startup, and during shutdown of the operating system. Additionally, if the machine did not shut down cleanly during the previous boot cycle, a new seed file is written periodically. Periodic writes start about 4 minutes after a reboot; subsequent writes are progressively slowed down (by factors of 3) until they reach once every 8 hours. This schedule is modified by delaying each seed file write until the next time the pools reseed the root PRNG; this avoids running an extra timer and waking up the CPU from a low-power state needlessly. The seed file write schedule is based on a clock that stops when the system is sleeping or hibernating.

On shutdown, the system reseeds the root PRNG with all the entropy in all the pools and writes a new seedfile. The seedfile is read on startup by winload.

Rationale: If the machine is shut down cleanly, all the entropy collected during the boot cycle is used to generate the seed file, and the next boot cycle will have good entropy at startup. However, if a machine is always shut down badly (e.g. powered down without an OS shutdown) then it would gather very little entropy. This mechanism detects that and ensures that the next boot cycle will regularly save fresh entropy to the seed file.

As the seedfile is updated during the boot process, it will be different for the next boot cycle even if the other entropy collected during the boot cycle is lost in an unclean power down. Furthermore, entropy collected by the interrupt timings between the initialization of the kernel and the writing of the seed file during startup is always used for the next boot.

The reason to disable periodic seedfile writes during normal operations is to avoid periodic disk activity which has a high cost in battery-powered devices that are always on.

External entropy

External entropy is a registry entry in the system hive. It is typically written by the setup program. When setup runs (typically from a WIN-PE image from the DVD or network) the underlying OS gathers entropy as usual. Towards the end of the setup, the setup process takes entropy from the PRNG system of the OS it is running on and writes it to the external entropy registry key in the newly installed OS. Winload consumes this entropy and it affects all subsequent RNG operations. If an external entropy value is present, it is deleted during the boot process.

Other applications can use this registry key to inject new entropy into an OS before it is booted. For diagnostic purposes the OS keeps a counter (in the registry) of how often it has encountered external entropy. This provides an easy diagnostic signal to ensure that the external entropy has been properly processed.

The external entropy is read by winload, and deleted by cng.sys upon startup.

This source ensures that the OS has good entropy if it was installed from a DVD or the network.

ACPI-OEM0

The ACPI-OEM0 is an ACPI table with the name OEM0. The Hyper-V hypervisor will create this table with 64 bytes of random data. The random data is derived from the host RNG infrastructure, and a fresh value is passed every time a guest is powered up. OEMs could provide this table on physical machines too, but we are not aware of any OEM doing that.

The ACPI-OEM0 table is read by winload. It ensures that any VM guest on Hyper-V has good entropy at startup. (This is especially important as many VMs are often launched from the same disk image, so they all get the same seedfile data.)

We do not know whether other hypervisors provide this ACPI table.

Firmware data

When the RNG system is first initialized, the CNG driver queries a number of system firmware table providers. It enumerates all tables, hashes all the data together, and provides that as a one-shot entropy source to the RNG system.

Currently the ‘ACPI’, ‘RSMB’ and ‘FIRM’ firmware table providers are queried.

Rationale: There is no particular reason to assume that these tables contain good entropy, but this data is cheap to gather and might contain machine-specific information which at least ensures that two machines that boot from the same disk image will produce different random numbers.

UEFI protocol

On UEFI machines there is a defined protocol to ask for random data from the UEFI drivers. Most ARM designs have a dedicated entropy source on the chipset. OEMs can implement this UEFI protocol to provide entropy to Windows at startup. We also recommend that they also expose the entropy source as a high-pull source to the entropy system to reseed the OS while it is running.

Time at startup

Winload gets the current time during every boot, and uses it as an entropy input.

Rationale: there is no reason to believe this provides any good entropy, but it greatly reduces the chance of two machines producing the same RNG state if they boot from the same disk image.

Virtual machine rewind

If the OS is running in a VM, there is a problem that most hypervisors can snapshot the state of the machine and later rewind the VM state to the saved state. This results in the machine running a second time with the exact same RNG state, which leads to serious security problems.

To reduce the window of vulnerability, Windows 10 on a Hyper-V VM will detect when the VM state is reset, retrieve a unique (not random) value from the hypervisor, and reseed the root RNG with that unique value. This does not eliminate the vulnerability, but it greatly reduces the time during which the RNG system will produce the same outputs as it did during a previous instantiation of the same VM state.

We do not know whether other hypervisors support this feature.

Appendix: Design rationale for multiple entropy pools

Of all the design aspects, the use of multiple pools has created the most discussion. This appendix explains the design rationale for using multiple pools.

Multiple entropy pools allow the system to reseed securely even if the rate of entropy is very low. The purpose of reseeds is to inject fresh entropy into the system so that an attacker that happens to know the current PRNG state will not know the future PRNG state. To achieve this goal, a reseed needs at least S bits of entropy, where S depends on the requirements but is typically in the range 128-256. (We'll use 128 in our examples.)

The traditional approach to ensuring a secure reseed is to run entropy estimation on the entropy provided by the sources. The system then collects entropy in a pool until it has at least S bits of entropy, and then uses that to reseed the PRNG system.

This approach is sound for dedicated entropy sources where the source can be characterized and analyzed. However, it fails for ad-hoc entropy sources such as those used by Windows. For example, the behavior of interrupt timings can be analyzed on existing hardware, but future hardware might have completely different properties so any entropy estimator tested on current hardware might be completely wrong on future hardware. (This is especially relevant if you consider that some of our customers consider the designers and implementers of future hardware as their opponents.) The danger is that a single bad entropy estimator might result in a system that consistently reseeds with, say, S/2 bits of true entropy and never achieves a secure state.

For this reason, Windows no longer uses entropy estimates to drive the reseed schedule. Rather, reseeds are done on a clock schedule; initially reseeds are done quickly to get entropy into the system quickly, but the reseeds are slowed down progressively to allow larger amounts of entropy to be collected between reseeds. The multi-pool system merely extends this behavior whilst at the same time retaining a minimum reseed speed.

For example: let's assume we have a machine whose entropy sources produce 30 bits of true entropy per hour. With one pool and one reseed each hour, each reseed gets 30 bits of entropy which is not a secure reseed. After 3 hours the second pool is enabled. Each pool now gets 15 bits of entropy per hour, so the next two reseeds (which use only pool 0) have 15 bits of entropy each. The reseed after that uses pools 0 and 1; pool 1 has collected 45 bits of entropy (3 hours of 15 bits each) and pool 0 has 15 bits, so the reseed has 60 bits of entropy. Essentially, the second pool has siphoned off some of the entropy from two reseeds and bunched it together into the third. Of course, 60 bits is not quite a secure reseed, so this pattern repeats three times after which the 3rd pool is enabled. Now each pool gets 10 bits per hour, and the reseeds will have in turn 10, 10, 40, 10, 10, 40, 10, 10, 130 bits of entropy. Finally, after about 21 hours the system has reseeded with 130 bits of entropy and is in a secure state.

It would of course be more efficient to collect exactly 128 bits of entropy (in just over 4 hours) and use that to reseed. We can look at the efficiency of a reseed system by looking at how much actual entropy it needs to achieve a secure reseed.

A reseed schedule based on entropy estimation assumes that each source has an entropy estimate that *never* overestimates the entropy. As mentioned above, this is infeasible for ad-hoc entropy sources. In practice, implementations like Win7 use entropy estimators that deliberately underestimate the amount of entropy in an attempt to reduce (but not eliminate) the risk of overestimating. The Win7 entropy estimator for the entropy scavenger is based on statistical analysis of the scavenging results in the most pessimistic scenario, and then reports an estimate that is 5 times lower. On the one hand, there is no guarantee that this estimator will never overestimate the entropy. On the other hand, the resulting reseed schedule is inefficient. Even in the most pessimistic scenario the system uses 5 times more entropy than necessary for the reseed, and in all other scenarios (when more entropy is available) the system is even less efficient. Compare this to the multi-pool example above. It is about 5 times slower than an ideal schedule, so in the worst-case scenario it is as good as the Win7 scavenger entropy estimation. In all other scenarios, it is actually much better.

We can show that the multi-pool system is within a constant factor of optimal under a very wide range of scenarios. Windows has a maximum of 8 entropy pools. Let's assume we get E bits of entropy per hour, and all 8 pools are in use. (Pool 7 will be used in a reseed once every 2187 hours, or about once every 3 month.) Each pool receives E/8 bits per hour. The worst-case inefficiency (in this rather simple model) is computed in the table below:

Time between reseeds	Entropy used in reseed	Entropy needed by MP	Inefficiency
1	1/8 * E	8S	8
3	4/8 * E	¾ * 8S	¾ * 8
9	13/8 * E	9/13 * 8S	9/13 * 8
27	40/8 * E	27/40 * 8S	27/40 * 8

The first column shows the time T between two reseeds that are secure. The second column shows how much entropy the multi-pool system uses in that reseed, and the third column how much entropy it needs to collect in that time to ensure that the reseed is secure. The final column shows the inefficiency of the multi-pool system (the amount of entropy it needs compared to the ideal schedule if the entropy estimators were exactly right.) As shown, the inefficiency is almost constant, and in fact it is bounded. (With fewer pools, the system is more efficient.) Thus, the multiple pools provide a simple guarantee: assuming the flow of entropy is uniform and distributed over the pools, the system is guaranteed to reseed securely after having received no more than

K*S bits of entropy, for a modest constant K. (With the obvious limits that the secure reseed is never faster than 1 hour and can't be slower than 2187 hours.)

This guarantee holds irrespective of any entropy estimators. If multiple attackers are attacking the same system at the same time, then the efficiency guarantee holds with respect to all attackers simultaneously even though the actual entropy of each entropy events can be a completely different value for each attacker.

In short, the multi-pool construction provides simple security guarantees, is within a constant factor of optimal, and dispenses with the vulnerable entropy estimators whose validity cannot be guaranteed.

Of course, the advantage of multi-pool is only apparent when the rate of entropy is extraordinary low. This is by itself an undesirable scenario, but not one that can be ignored for an OS used in as many scenarios as Windows.

Though the description of the multi-pool system is rather complicated, the implementation is actually rather simple, adding fewer than 200 lines of code to the system.