WHITEWOOD

ENCRYPTION SYSTEMS, INC.

an AMFI Company

UNDERSTANDING AND MANAGING ENTROPY

As Moore's Law marches on, the Internet of Things continues to gain traction, and personal privacy is an above the fold news story, the importance of cryptography on the modern Internet continues to increase. While once the realm of mathematicians and hard core computer scientists, cryptography has gone mainstream. Vulnerabilities such as FREAK and POODLE are reported by mainstream media and cause discussions not just in SOC's but dinner tables around the world.

With all this attention on the quality of our protective encryption capabilities, encryption is still a very complicated and not well-understood process. The ecosystem of crypto components that is used in a conventional enterprise is dizzying and even those who try to understand what's going on are stymied by the diversity, age, and code complexity of the various software components.

While cryptographic core algorithms have been well studied, other components in enterprise cryptosystems are less understood. Nearly every crypto system relies heavily on access to high quality random numbers. These random numbers are used for long term key creation, ephemeral key creation, and nonces that prevent replay attacks and various types of cryptanalysis. If the random numbers used in these crypto systems aren't truly random (or at least random enough to withstand cryptanalytic scrutiny), then the security the algorithm provides can be completely compromised.

WHAT IS ENTROPY?

Historically, true random numbers have been hard to come by. True random numbers typically come from chaotic physical processes such as thermal noise, atmospheric noise, and even Lava Lamps. While these processes are chaotic and generate truly random numbers, they are relatively low bitrate and can't keep up with the needs of most crypto systems.

Provisioning of random numbers in both computers and dedicated crypto systems is typically driven not by quality of randomness, but by a desire to build the cheapest

possible circuit that seems random enough. For example, linear congruential generators (LCGs) have known statistical properties, and are insufficiently random for almost any purpose. However, they are still used by every operating system as the default rand() function.

In hardware, chips with a cryptographic core will often use ring oscillators, odd-length of NOT gates whose output feeds back into its input. It can be tricky to guarantee randomness of these circuits, especially as chip feature size decreases, but they are incredibly cheap to build into a chip, and are self-clocking.

To compensate for the lack of high speed true random numbers, system developers turned to Pseudo Random Number Generators (PRNG). These PRNG's conventionally consist of an internal state that updates itself through a fixed rule, and a hash function applied on output to hide the internal state. The data that comes from the hash function looks random and as long as an attacker can't find the internal state of the PRNG and the seeding data is random enough, then the data that comes from the PRNG is cryptographically strong enough for use.

Truly random data has a measurable characteristic called *entropy*. The term was coined in 1865 to describe a measure of the disorder of a thermodynamical system, and adapted by Shannon in 1948 as a measure of the unpredictability of information content. So right from its inception, entropy was always supposed to be measurable. Unfortunately, there is no one objectively good way to measure the entropy of real data. For example, in 2012 NIST published a suite of statistical tests for entropy sources. In 2015, they reported that those tests had been found to seriously underestimate entropy [3].

The difference between entropy and randomness can be difficult to understand. Probably the easiest way to think about the difference is entropy is the uncertainty of an outcome yet to happen; Randomness is the quality of the uncertainty from a historical perspective. While we can agree on the abstract concept of randomness, the only thing we can quantitatively measure is our ability to distinguish some piece of data from abstract random data, and that depends on our perspective, background knowledge, and the quantity of data. In short, it is highly subjective. For a good discussion on entropy in PRNG's and PRNG construction in general, see [4].

Since entropy can be quantified, estimation of entropy can be useful when attempting to understand the quality of seed data fed in to a PRNG. In general, the more entropy that is fed in to the PRNG, the more secure the output of the PRNG is presumed to be. If the data that is fed in to the PRNG contains little (or no) entropy, attackers may have the ability to guess the inputs and therefore may be able to divine the output. While low entropy is not an guarantee of compromise, it does weaken the entire crypto system. For example, the Debian OpenSSL PRNG debacle of 2008 caused all keys generated on a Debian system to be easily breakable, and all DSA keys used on a Debian system to be breakable also [5]. In 2015, an audit of Github SSH keys found that many were still vulnerable [6].

THE ENTROPY LIFECYCLE

Entropy can (and should) be viewed in the context of a lifecycle. In Figure 1, data moves from the land of pure entropy on the left and gets diluted through a series of PRNG's and other activities as entropy is eventually consumed by applications on the right.

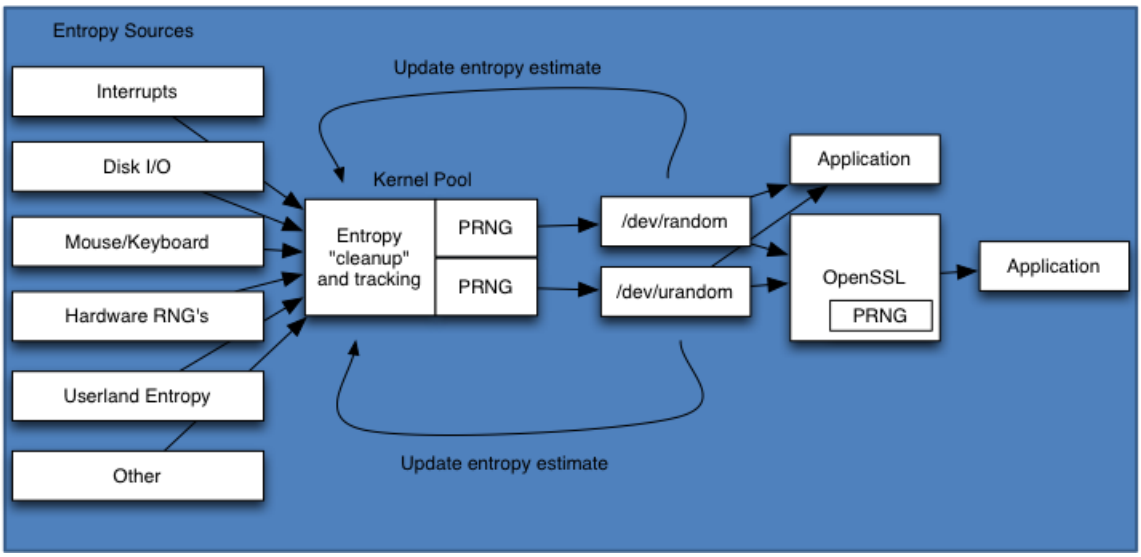

Figure 1 – Entropy Lifecycle

In modern Unix-variants and Linux, there are a numerous sources of entropy available to the kernel. From interrupt timing and network inter-arrival time on the low end, to hardware RNG's such as Intel's Ivy Bridge RNG and Whitewood's Quantum Random Number Generator on the high end, entropy is produced at various rates. In Linux, this entropy is then fed in to the kernel entropy pool where it is hashed for use in two separate interfaces. /dev/random provides random data that is nearly 100% entropy. If there is no entropy in the pool, /dev/random will block until the OS generates more entropy. /dev/urandom, on the other hand, does not block and will hand out data from its PRNG regardless of the amount of entropy in the entropy pool. The default size of the kernel entropy pool is 4096 bits, but can be modified via a kernel configuration option.

Applications can pull directly from /dev/random or /dev/urandom to get random numbers as needed. One of the largest consumers of random numbers is OpenSSL. OpenSSL provides cryptographic capabilities both through a command line interface and through a library that is linked in to a wide variety of software packages, including Apache and OpenSSH. OpenSSL pulls data from /dev/urandom only. OpenSSL has its own PRNG that it uses to perform its own cryptographic operations. This PRNG is seeded by a call to /dev/urandom.

Entropy Production

Entropy production in Linux can come from a variety of sources. Without the assistance of external hardware RNG's, Linux uses naturally occurring chaotic events on the local system to generate entropy for the kernel pool. These events, such as disk IO events, network packet arrival times, keyboard presses, and mouse movements, are the primary sources of entropy on many systems. The amount of entropy generated from these sources is directly proportional to the amount of activity on each source.

From Whitewood's research, the growth of the pool due to events on the system is relatively slow. Table 1 shows growth rates for various use cases. It should be noted that even conservative hardware RNG's will fill the entropy pool in <1 second. High speed RNG's such as Intel's Ivy Bridge RNG and Whitewood's quantum solution can produce entropy in excess of 100Mbps.

Activity	Average Production Rate
Unloaded system	1.94 bits/sec
Lightly loaded webserver	3.85 bits/sec
Heavily loaded webserver	5.56 bits/sec
Receiving 10 pings/sec	2.22 bits/sec
Receiving 1000 pings/sec	3.77 bits/sec

Table 1 – Entropy Production

Entropy Consumption

Entropy consumption can vary wildly depending on the application and specifics of the cryptographic operation. For this research, we examined common use cases involving OpenSSL including Apache and nginx webservers as well as common OpenSSL crypto operations such as key generation.

OpenSSL Concerns

When it is initialized, OpenSSL attempts to pull 32 bytes of random data from the non-blocking interface, /dev/urandom. OpenSSL does not verify how much entropy is in the pull and assumes the data is acceptable. From our research, if there are less than 384 bits of entropy in the kernel pool, /dev/urandom will not return data with 32 bytes of entropy. In fact, if the kernel pool has less than 192 bits of entropy, the kernel entropy estimation isn't decremented at all. This means that depending on the state of the kernel entropy pool when OpenSSL is started, little to no new entropy will be contained in the random numbers provided from /dev/urandom.

OpenSSL only seeds its internal PRNG once per runtime. This is not an issue for short lived executions, such as command line invocations to create a single key. However, for long running processes, such as servers that link to the OpenSSL libraries, it means that all PRNG operations performed for the duration of the server only have as much entropy as was available at the invocation of the OpenSSL library.

MOD_SSL CONCERNS

Mod_SSL, the primary mechanism for servicing TLS connections in Apache, attempts to add its own entropy to OpenSSL to address this problem. For every request, Mod_SSL attempts to add entropy by concatenating the PID, time, and 256 bytes from the stack and then stirring that value in to the internal OpenSSL PRNG.

This operation adds very little entropy. The PID for webservers is long running and does not change over time, and the date value used in this calculation is only down to the second resolution so it changes very slowly (and regularly). The value from the stack could be a source of large amounts of entropy, but that does not seem to be the case. Based on Whitewood's analysis, this stack value only changes ~8% of the time it is queried. All other times, the value from the query is the same as the previous value. While 256 bytes is clearly a lot to guess, the goal of adding entropy to the PRNG is usually not successful.

NGINX CONCERNS

While mod_SSL at least attempts to add entropy to OpenSSL's PRNG, nginx does not. By default it never reseeds the PRNG with any entropy at all. It can be configured to add entropy directly through OpenSSL by calling OpenSSL's reseed function. However since this is not the default behavior, it is unclear how often administrators chose to perform this reseed.

LIBWES – AN OPENSSL ENTROPY MANAGEMENT LIBRARY

There are already several options, such as egd and haveged, available to provide more entropy to the kernel. However, from Whitewood's experience, the providing more entropy to the kernel can be satisfied with hardware RNG's and still does not address the issues regarding OpenSSL's lack of reseeding in long running processes.

To address this concern, Whitewood has developed libWES. libWES has two major components as shown in Figure 2. The first part is the libWES library itself, which is a replacement of the OpenSSL random number engine. When installed and configured, it replaces the default engine and handles all requests for random numbers.

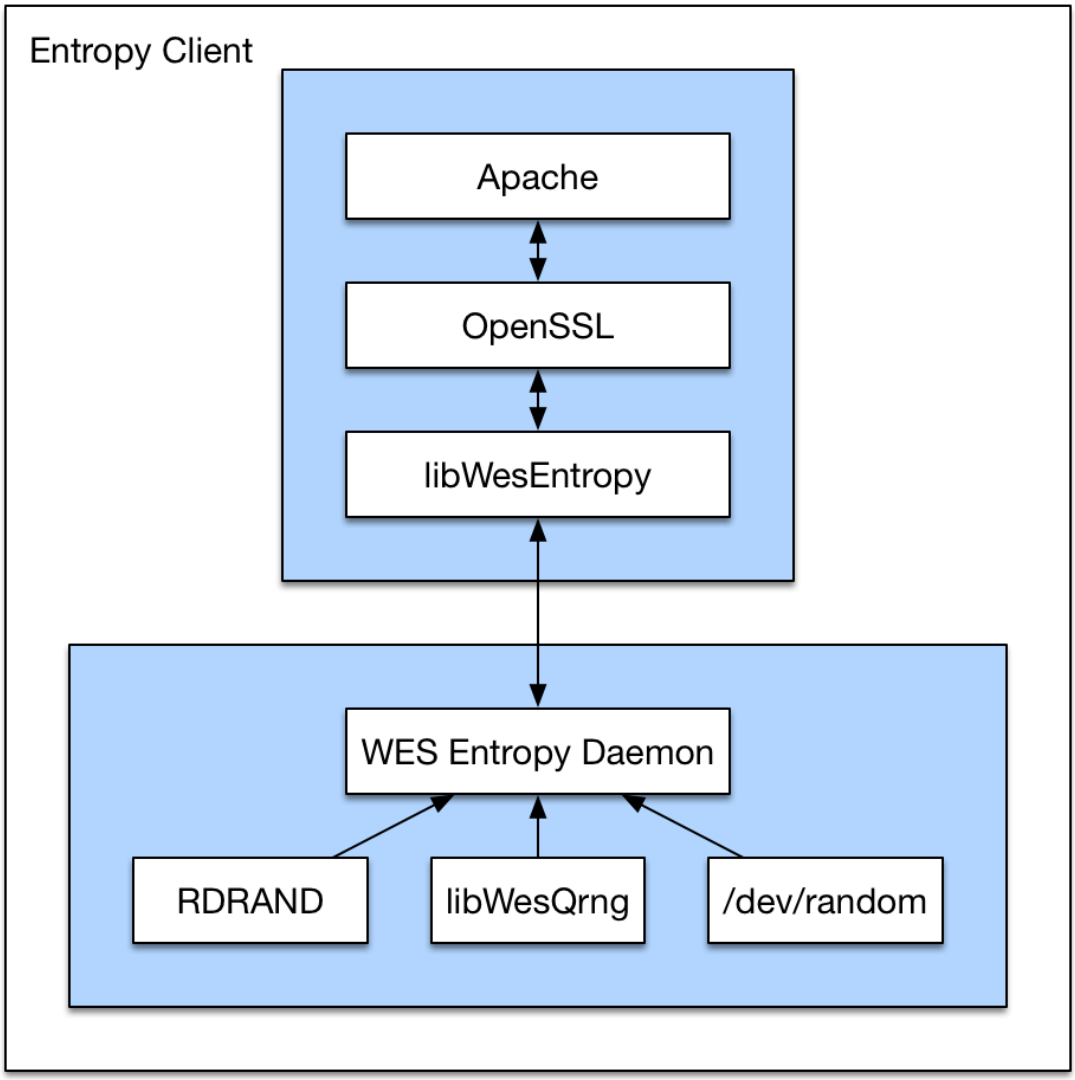

Figure 2 – libWES architecture

The second component is the WES entropy daemon. The daemon can be configured to pull entropy from a variety of entropy sources including hardware RNG's. The daemon continuously feeds entropy into the libWES library to reseed the PRNG. This system addresses the reseed concerns with OpenSSL and allows Apache and nginx to be operated securely for long durations.

libWES has been released as an open source component and is available for download from http://www.whitewoodencryption.com/

CONCLUSION

Random number generation has always been a dark art, as has the practice of designing robust entropy processes to capture and assess entropy. And yet, these same random numbers underwrite the security of every modern communications system, including the Internet.

Crypto designers and security professionals alike are expected to know about cryptographic algorithms and key sizes, and their impact on security. But most will not know how keys are generated for those algorithms, or how entropy is accumulated, even though the underlying algorithms and technology use well-known software and hardware components.

This paper highlights some truly surprising discrepancies between what people think is happening in commonly used PRNGs, and what is actually happening. It is our hope that this research will encourage others to take a look at their PRNGs, and finally bring this subject into the light where it belongs.

REFERENCES

[1] FREAK - https://www.us-cert.gov/ncas/current-activity/2015/03/06/FREAK-SSLTLS-Vulnerability

[2] POODLE https://www.us-cert.gov/ncas/alerts/TA14-290A

[3] How Random is Your RNG https://archive.org/details/How_Random_Is_Your_RNG_SC2015

[4] Pseudorandom Number Generation, Entropy Harvesting, and Provable Security in Linux http://www.blackhat.com/presentations/bh-europe-04/bh-eu-04-hardy/bh-eu-04-hardy.pdf

[5] http://rdist.root.org/2009/05/17/the-debian-pgp-disaster-that-almost-was

[6] https://threatpost.com/audit-of-github-ssh-keys-finds-many-still-vulnerable-to-old-debian-bug/113117