Manuscript. First posted online October 3, 2016.

An Investigation of the FreeBSD r278907 RNG Bugfix

Wilson Lian
UC San Diego

Hovav Shacham
UC San Diego

Stefan Savage
UC San Diego

Abstract

Operating systems and applications rely on random number generators (RNGs) for a number of important tasks, most notably cryptographic key generation. The impact of flawed random number generation practices has been studied extensively in the past [2, 5]. In this paper, we examine the implications of an RNG bug in FreeBSD that was fixed by Subversion commit r278907 [3]. In particular, our analysis seeks to discover uses of weak random numbers that either enable an attacker to discover the internal state of the RNG or use such knowledge to predict security-relevant values.

1 The Bug

In Subversion commit r273872, new code was introduced into FreeBSD for managing the randomness source backing both the `/dev/random` device file and the kernel's `read_random()` interface. Each such randomness source (or "random adaptor" in FreeBSD parlance) exports a function which takes as parameters a pointer to a buffer into which to write random bytes and the number of random bytes to be written.

On system startup, the current random adaptor is set to a dummy adaptor which is backed by the kernel's random() function,[1] which returns consecutive members of the series $x[n+1] = (7^5 \cdot x[n]) mod (2^{31} - 1)$. The underlying 32-bit state of random() is seeded twice during system bootup. The first time, it is seeded with the current cycle count; later during bootup, it is seeded with a function of the current timestamp (`ts.tv_sec ^ts.tv_nsec`, where `ts` is a `struct timespec`).

Later on during bootup, a more sophisticated random adaptor which draws entropy from non-deterministic system events is installed. The code that installs the new random adaptor should overwrite the function pointer used by `read_random()` to acquire random bytes, but does not do so. Consequently, calls to the kernel's `read_random()` interface continue returning weak random numbers from `random()` rather than genuine random numbers from the sophisticated random adaptor. Reading from `/dev/random` acquires random bytes from the randomness source through a different chain of pointers that is not impacted by the bug.

2 Collateral damage: `arc4rand`/`arc4random`

The FreeBSD kernel provides an implementation of the Alleged RC4 stream cipher (like the kernel's `random()` implementation, this one is separate from libc's). The RC4 stream cipher produces a stream of pseudorandom bits and periodically "stirs" in 32 fresh random bits acquired from `read_random()`. Therefore, if an adversary can discern the state of the kernel's `random()` interface, she has a much better

[1] The kernel's `random()` function is separate from libc's implementation and keeps different state. Kernel `random()` is only exposed to kernel and driver code.

chance of predicting the state of the RC4 key stream. This stirring procedure occurs under the following four conditions:

1. A randomness adaptor unblocks itself because it believes it has collected enough entropy to return random values.
2. A caller to the kernel's `arc4rand()` provides an argument explicitly requesting stirring to occur prior to generating the result.
3. More than 64KB of key stream have been produced since the last stirring.
4. More than 300 seconds have passed since the last stirring.

3 Leakage of the state of the kernel's `random()`

The `read_random()` function has numerous call sites in kernel code, and its return values arer often exposed directly. In other cases, the values produced by `read_random()` are combined with other values to produce an adversary-observable value (indirect leakage). If those other values are known or easily guessed by an adversary, the attacker can brute force search the state space of `random()` until she discovers which state (or states) produced the observed value. In this section, we discuss the uses of `read_random()` that an adversary may be interested in predicting or using as a source of direct or indirect state leakage.

3.1 Initialization Vectors

Since `random()` returns its entire state, an attacker that learns a single return value can predict all future return values. We therefore seek places in the FreeBSD codebase where the return value from `read_random()` might be leaked to a remote adversary.

In many cases, values from `read_random()` are used by drivers for hardware cryptographic accelerators as initialization vectors (IVs) for block cipher modes of operation that require them (e.g., cipher block chaining). Depending on how the exact details of each mode of operation, the IV may or not be transmitted in the clear. If an adversary is able to predict IVs, she may be able to launch a chosen plaintext attack. However, since this vulnerability depends on victim machines requiring specific hardware that is not necessarily widely deployed, we did not investigate in depth whether the implemented modes rely on in-the-clear IVs and, more importantly, whether or not an attacker can induce the transmission of these values remotely (i.e., via scanning).

3.2 TCP Initial Sequence Numbers

In order for an off-path network attacker to launch a TCP injection attack, she must be able to predict or discover the sequence numbers used by both end points in the TCP connection under attack (in addition to the IP addresses of the endpoints and the port numbers being used). The initial sequence number (ISN) that FreeBSD includes in the TCP SYN packet sent during the establishment of an outbound TCP connection is derived from the local and remote port numbers, the local and remote IP addresses, and 32 consecutive bytes from the `random()`-backed `read_random()` interface (i.e., 4 values from `random()`). The initial sequence number is the first 32 bits of the MD5 digest of the concatenation of those values.

If the attacker is able to observe the TCP ISN, port numbers, and IP addresses in a SYN packet from a bug-afflicted FreeBSD host, she can attempt to discern state of the host's `random()` interface by computing the MD5 digest for all possible states and comparing the results against the observed ISN.

There is a chance of false positives since the ISN is only a prefix of the MD5 digest, but confirming multiple ISNs can provide additional confidence.

An attacker who is merely scanning the Internet will not be able to observe an outbound SYN packet, making this vulnerability of little use to her. A web attacker, on the other hand, is able to induce outbound connections to her own server from the victim's machine via JavaScript or some other means of client-side programming. This vulnerability alone provides only a subset of the information required by an adversary to launch a successful TCP injection attack. In order for the attacker to inject packets into a third party TCP stream, the attacker must also predict the port numbers in use. Often the server port is trivially predictable, but the client port is often more difficult. We discuss an adversary's ability to exploit the randomness bug to predict client port numbers in §4.1.1.

3.3 IPSec

3.3.1 Encapsulating Security Payload Padding

Encapsulating Security Payload (ESP) is a protocol in the IPSec suite that provides "confidentiality, data origin authentication, connectionless integrity, an anti-replay service[. . .], and limited traffic flow confidentiality" [4]. ESP packets may contain up to 255 bytes of padding, which by default are 1-byte integers in the sequence $1, 2, 3, \ldots$. There does, however, exist a code path in FreeBSD which populates the padding with bytes from `read_random()`. We are currently unaware of how to coerce a FreeBSD host into using random bytes instead of consecutive bytes for ESP padding.

3.3.2 Security Parameters Index

In IPSec, the Security Parameters Index (SPI) is a 32-bit field used to associate an incoming packet with a Security Association (SA), which is simply the notion of a one-way channel with specific IPSec security services. Values from `read_random()` are coerced into a certain range to generate random SPIs using the formula $min + (\text{read\_random}() \ mod (max - min + 1))$. An adversary may not always know the values of *min* and *max*, except when they take their default values. Further investigation is required to determine when this is the case.

4 Predicting the state of the kernel's `arc4rand()`

Unlike `random()`, `arc4rand()` only reveals a single byte of its state at a time; each time a byte from the key stream is returned, the state is mutated. The easiest way to predict `arc4rand()`'s state is therefore to predict the values that are used to periodically stir its state. This is possible because `arc4rand()`'s state always starts with the same value before the first stir. As we mentioned in §2, the hidden state in the kernel's RC4 implementation is periodically stirred with bytes generated by `read_random()`, and therefore `random()`. If the attacker is able to learn `random()`'s hidden state, she must also keep track of when `arc4rand()`'s state is stirred.

An adversary can use one of the leakage vectors described in §3 or attempt to indirectly infer `random()`'s state using one of the vectors described later in this section. However, the closer to bootup this happens, the easier it is for the adversary. This is because each time the `arc4rand()` state is stirred (which could occur between the production of any two bytes of the key stream), the key stream branches. The search space for discovering the internal states of both `random()` and `arc4rand()` grows with the number of stirs that have occured. The procedure for doing so when `arc4rand()`'s state has only been stirred once with `read_random()` is as follows:

Identify a mechanism *M* through which key stream bytes from victim `arc4rand()` implementation can be requested on-demand.
Use *M* to generate a moderate-sized pool of key stream bytes.
for all Internal `random()` states **do**
Initialize local implementation of `arc4rand()` state to common initial value found at bootup.
Stir local implementation of `arc4rand()` with `random()` values starting at this state.
Generate a large number of key stream bytes using local implementation of `arc4rand()`
if non-empty substring of locally-generated key stream matches non-empty prefix of victim-generated key stream **then**
Candidate `random()` state identified
Offset in victim key stream where locally-generated key stream stops matching is likely stirring point
end if
end for

To track the internal states of `random()` and `arc4rand()`, the attacker can identify stirring points and test subsequent `random()` values as stirring values until one is found which produces the `arc4rand()` key stream following the stirring point. The attacker can periodically poll *M* for `arc4rand()` key stream bytes to keep tabs on the key stream and identify stirring points.

4.1 Usage of the kernel's `arc4rand()`

4.1.1 Local transport layer port numbers

When choosing a local port for TCP or UDP, FreeBSD has the option to use `arc4rand()` to select a port from a range using the formula $min + (\texttt{arc4rand()}\ mod(max - min)) + 1$. The values of *min* and *max* may take either default or system-defined (via sysctl) values. If the chosen local port number is in use, a linear probing sequence is used to find a free port.

As with TCP ISNs, this vector is a poor source of `arc4rand()` key stream bytes since it relies on the victim machine taking the inititative to contact the adversary. However, if an adversary is able to track `arc4rand()`'s state through other means, predicting client port numbers, coupled with predicting TCP ISNs opens the door to TCP injection attacks. Even if the attacker is not able to predict ISNs, an attack such as the one presented by Gilad and Herzberg [1] could be used to discover sequence numbers.

4.1.2 IPv4 Identification Field

The Identification (ID) field in IPv4 packet headers is used whenever an IP datagram's size exceed's the MTU of the underlying link layer. The IP datagram is fragmented, and each fragment derived from that datagram is assigned a common 16-bit ID used by the recipient to reassemble the datagram. FreeBSD directly populates IP ID fields using `arc4rand()`, gathering more bytes from `arc4rand()` in the case of a collision.

The IP ID field is perhaps the most promising source for `arc4rand()` key stream bytes since it can be invoked remotely and on-demand by any network action which elicits a response. However, there may be gaps in the recovered key stream since it only returns 2 bytes at a time, may encounter collisions, and may be interleaved with other calls to `arc4rand()`.

4.1.3 Initialization Vectors

The `arc4rand()` key stream is used to generate IVs. See §3.1 for a discussion of using IVs to leak or infer the state of RNGs.

4.1.4 TCP SYN-ACK sequence numbers

On a FreeBSD system with SYN cookies disabled, the sequence number on a SYN-ACK packet is derived directly from `arc4rand()`. This would be a convenient source for `arc4rand()` key stream bytes, but by default FreeBSD uses SYN cookies at all times, even if the machine is not under heavy connection load.

Using SYN cookies to confirm the state of the RNGs is a foreboding task. SYN cookies are generated by computing a keyed MAC over various information about the incoming connection and a secret 16-byte key derived directly from `arc4rand()`. There are actually two MAC keys. Each key is replaced every 30 seconds, and the two key expirations are 15 seconds out of phase. New SYN cookies are derived from the most-recently generated key. This design allows for high key turnover while still allowing SYN cookies generated with the previous key to be accepted.

Acknowledgements

This material is based in part upon work supported by the U.S. National Science Foundation under award CNS-1410031.

References

[1] GILAD, Y., AND HERZBERG, A. Off-path attacking the web. In *WOOT* (2012), pp. 41–52.

[2] HENINGER, N., DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (2012), pp. 205–220.

[3] JOHN-MARK GURNEY. Freebsd svn commit: r278907 - head/sys/dev/random. https://lists.freebsd.org/pipermail/svn-src-head/2015-February/068405.html, 2015.

[4] KENT, S. IP Encapsulating Security Payload (ESP). RFC 4303 (Proposed Standard), Dec. 2005.

[5] YILEK, S., RESCORLA, E., SHACHAM, H., ENRIGHT, B., AND SAVAGE, S. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In *Proceedings of IMC 2009* (Nov. 2009), A. Feldmann and L. Mathy, Eds., ACM Press, pp. 15–27.