# Dissecting QNX

**Jos Wetzels[1,2,3], Ali Abbasi[3,4]**

[1] *Midnight Blue*
[2] *Eindhoven University of Technology (TU/e)*
[3] *University of Twente (UT)*
[4] *Ruhr-University Bochum (RUB)*

**This work concerns a dissection of QNX: a proprietary, real-time operating system aimed at the embedded market. QNX is used in many sensitive and critical devices in different industry verticals and while some prior security research has discussed QNX, mainly as a byproduct of BlackBerry mobile research, there is no prior work on QNX exploit mitigations and secure random number generators. In this work, carried out as part of the master's thesis of the first author, we present the first reverse-engineering and analysis of the exploit mitigations, secure random number generators and memory management internals of QNX versions up to and including QNX 6.6 and the brand new 64-bit QNX 7.0 released in March 2017. We uncover a variety of design issues and vulnerabilities which have significant implications for the exploitability of memory corruption vulnerabilities on QNX as well as the strength of its cryptographic ecosystem.**

## 1 Introduction

QNX [17] is a proprietary, closed-source, Unix-like real-time operating system with POSIX support aimed primarily at the embedded market. Initially released in 1982 for the Intel 8088 and later acquired by BlackBerry, it forms the basis of BlackBerry OS, BlackBerry Tablet OS and BlackBerry 10 used in mobile devices as well as forming the basis of Cisco's IOS-XR used in carrier-grade routers such as the CRS, the 12000 and the ASR9000 series. QNX also dominates the automotive market [61] (particularly telematics, infotainment and navigation systems) and is found in millions of cars from Audi, Toyota, BMW, Porsche, Honda and Ford to Jaguar and Lincoln. In addition, it is deployed in highly sensitive embedded systems such as industrial automation PLCs, medical devices, building management systems, railway safety equipment, Unmanned Aerial Vehicles (UAVs), anti-tank weapons guidance systems, the Harris Falcon III military radios, Caterpillar mining control systems, General Electric turbine control systems and Westinghouse and AECL nuclear powerplants.

The interest of high-profile actors in QNX-based systems is evidenced by a series of documents from the United States *Central Intelligence Agency (CIA)* obtained and released by *WikiLeaks* under the name *'Vault 7'*. These documents show an interest on part of the CIA's *Embedded Development Branch (EDB)* of the *Engineering Development Group (EDG)* (which develops and tests exploits and malware used in covert operations) in targeting QNX [65].

In this work, we focus primarily on QNX's *'binary security'* ie. its hardening against memory corruption exploitation, as well as the quality of its secure random number generators.
More precisely, this work makes the following novel contributions:

- It presents the first reverse-engineering of the proprietary, closed-source QNX OS to document the internals of its memory manager, exploit mitigations (eg. NX memory, ASLR, stack canaries, RELRO) and secure random number generators (both the kernel PRNG and `/dev/random`), covering all QNX versions as of writing (ie. $\leq$ 6.6 and the newly released QNX 7.0).
- It presents the first analysis of the exploit

mitigations and secure RNGs on QNX $\leq$ 6.6 and 7.0 and uncovers a variety of design issues and vulnerabilities which have significant implications for the exploitability of memory corruption vulnerabilities on QNX as well as the strength of its cryptographic ecosystem.

- As a result of this work, we disclosed the uncovered issues to the vendor and cooperated in drafting patches to help protect system end-users.

Given that there is, as discussed in Section 2.1, no prior work on QNX's mitigations, secure random number generators or memory management internals, we consider this work a significant contribution to the state of the art in understanding QNX security as well as QNX OS internals more broadly.

In Section 2 we present an brief overview of QNX's OS architecture, its security architecture and its memory management internals. We discuss the result of our reverse-engineering and analysis of the exploit mitigations of QNX versions up to and including 6.6 in Section 3 and those of QNX version 7.0 in Section 4. In Section 5 we present the results of our reverse-engineering and analysis of the secure random number generators of QNX versions $\leq$ 6.6 and 7.0. Finally, in Section 6 we present our concluding remarks.

## 2 QNX Overview

### 2.1 Security History

Most of the relatively scarce public research available on QNX security has been the byproduct of research into BlackBerry's QNX-based mobile operating systems such as Tablet OS, BlackBerry OS and BlackBerry 10 [3, 13–15, 66] most of which has not focussed on QNX itself. Recent work by Plaskett et al. [1, 42] has focussed on QNX itself, particularly security of the Inter-Process Communication (IPC), message passing and Persistent Publish Subscribe (PPS) interfaces as well as kernel security through system call fuzzing. When it comes to specific vulnerabilities the work done by Julio Cesar Fort [27] and Tim Brown [19] stands out in particular and the MITRE CVE database [25] reports, as of writing, 34 vulnerabilities most of which are setuid logic bugs or memory corruption vulnerabilities.

### 2.2 OS Architecture

QNX supports a wide range of CPU architectures and features a pre-emptible microkernel architecture with multi-core support ensuring virtually any component (even core OS components and drivers) can fail without bringing down the kernel. QNX itself has a small footprint but support is available for hundreds of POSIX utilities, common networking technologies (IPv4/IPv6, IPSec, FTP, HTTP, SSH, etc.) and dynamic libraries. As opposed to the monolithic kernel architecture of most general-purpose OSes, QNX features a microkernel which provides minimal services (eg. system call and interrupt handling, task scheduling, IPC message-passing, etc.) to the rest of the operating system which runs as a team of cooperating processes as illustrated in Figure 1. As a result, only the microkernel resides in kernelspace with the rest of the operating system and other typical kernel-level functionality (drivers, protocol stacks, etc.) residing in userspace next to regular user applications albeit separated by privilege boundaries. In QNX the microkernel is combined with the process manager in a single executable module called procnto. QNX *libc* converts POSIX function calls into message handling functions which pass messages through the microkernel to the relevant process. As of writing, the latest QNX release is version 7.0.

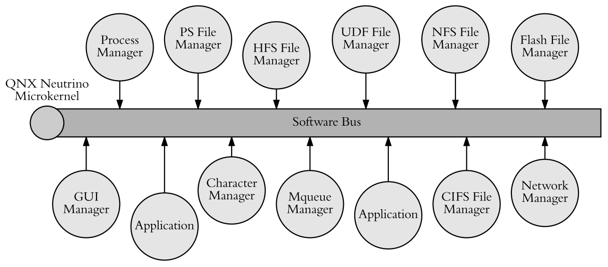


**Figure 1:** *QNX Microkernel Architecture [60]*

Most elements of the QNX system architecture (such as the messaging layer, process & resource management, filesystem and networking functionality, etc.) are well described in prior work [42] and since this work focuses on memory corruption we will only discuss QNX memory management and the security architecture in 'broad strokes'.

### 2.3 Security Architecture

As a Unix-like operating system QNX inherits a large part of the Unix security model, primarily in the form of user groups and associated permission-based access controls. QNX is certified to *Common Criteria* ISO/IEC 15408 *Evaluation Assurance Level (EAL)* 4+. The certification report [21] indicates that the *Target of Evaluation (TOE)* boundary encompasses only the procnto system process (ie. the microkernel and process manager) and libc.

QNX features a strong separation between kernel- and userspace running everything except for the microkernel and process manager in kernelspace by assigning it to the procnto process which runs as root with `PID 1`. Other OS components run as their own root processes in userspace next to non-OS processes. Separation between OS processes and non-OS processes comes down to a combination of enforcement of user permissions and additional sandboxing capabilities [54]. If a non-OS process is run as root, the only way to wall it off from the wider OS is by restricting its capabilities. On the other hand, capabilities can be assigned on a granular level allowing or disallowing access to system actions and resources meaning for many processes there is no need to run as root to perform their functionality. Security separation between userspace and kernelspace is also mediated in this fashion which does mean, however, that there is no 'absolute isolation' of the microkernel and a root user without significant capability restrictions (as is the default for most OS processes) can easily pivot into the microkernel by means of common kernel calls, access to sensitive devices (eg. `/dev/mem`) or installation of *Interrupt Service Routines (ISRs)*.

As such one should not confuse the safety guarantee that the crashing of one component does not lead to a crash of the entire system with a security guarantee that the compromise of one component could not lead to the compromise of the entire system. If no explicit capability restrictions are put in place by system integrators, nothing prevents a compromise of a process with the right privileges or capabilities from leading to arbitrary kernelspace code execution.

## 2.4 Memory Management

QNX offers a full-protection memory model placing each process within its own private virtual memory by utilizing the MMU as shown in Figure 2. On QNX every process is created with at least one main thread (with its own, OS-supplied stack) and any subsequently created thread can either be given a customly allocated stack by the program or a (default) system-allocated stack for that thread. QNX's virtual memory provides permission capabilities and the memory manager ensures inter-process memory access is mediated by privilege as well as capability checks [54]. QNX handles typical memory objects such as stacks, the heap, object memory (eg. video card memory mapped into userspace), shared libraries, etc. and has support for shared- and typed memory [57, 59]. The relevant memory manager internals are described in detail in Section 3.2.

For QNX versions up to and including 7.0, we illustrate QNX user- and kernel-space address boundaries, derived from reverse-engineering, in Tables 1 and 2. On QNX systems where ASLR is not enabled, `libc` is loaded by default at the addresses illustrated in Table 3. For QNX versions up to and including 6.6 on x86, the default user- and kernel-space layouts when ASLR is disabled are illustrated in Figures 3 and 4

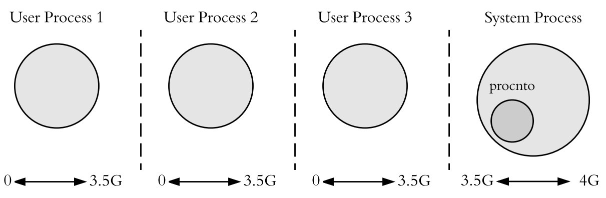


**Figure 2:** *QNX Private Virtual Memory [53]*

| Architecture | Start | End |
|---|---|---|
| **Userspace** | | |
| x86 | `0x00000000` | `0xBFFFFFFF` |
| AArch32 | `0x00000000` | `0x7FFFFFFF` |
| MIPS | `0x00000000` | `0x7FFFFFFF` |
| PPC | `0x40000000` | `0xFFFB0000` |
| SuperH | `0x00000000` | `0x7BFF0000` |
| **Kernelspace** | | |
| x86 | `0xC0000000` | `0xFFFFFFFF` |
| AArch32 | `0x80000000` | `0xFFFFFFFF` |
| MIPS | `0x80000000` | `0xFFFFFFFF` |
| PPC | `0x00000000` | `0x3FFFFFFF` |
| SuperH | `0x80000000` | `0xCFFFFFFF` |

**Table 1:** *QNX $\leq$ 6.6 Address Boundaries*

| Architecture | Start | End |
|---|---|---|
| **Userspace** | | |
| x86 | `0x00000000` | `0xBFFFFFFF` |
| x86-64 | `0x00000000` | `0x0000007FFFFFFFFF` |
| AArch32 | `0x00000000` | `0x7FFFFFFF` |
| AArch64 | `0x00000000` | `0x0000007FFFFFFFFF` |
| **Kernelspace** | | |
| x86 | `0xC0000000` | `0xFFFFFFFF` |
| x86-64 | `0x0000008000000000` | `0xFFFFFFFFFFFFFFFF` |
| AArch32 | `0x80000000` | `0xFFFFFFFF` |
| AArch64 | `0x0000008000000000` | `0xFFFFFFFFFFFFFFFF` |

**Table 2:** *QNX 7.0 Address Boundaries*

| Architecture | Libc Addr. |
|---|---|
| **QNX $\leq$ 6.6** | |
| x86 | `0xB0300000` |
| AArch32 | `0x01000000` |
| MIPS | `0x70300000` |
| PPC | `0xFE300000` |
| SuperH | `0x70300000` |
| **QNX 7.0** | |
| x86 | `0xB0300000` |
| x86-64 | `0x0000000100000000` |
| AArch32 | `0x01000000` |
| AArch64 | `0x0000000100000000` |

**Table 3:** *QNX Default Libc Load Addresses*

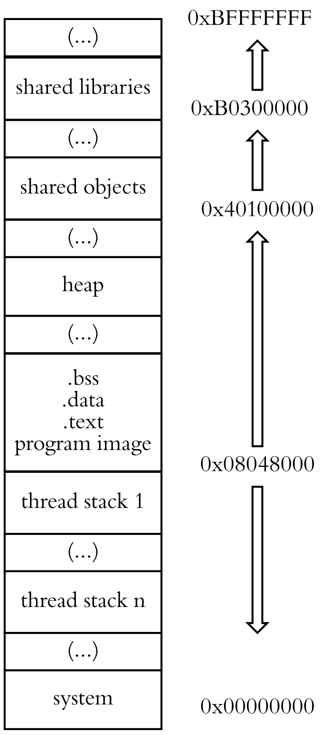


**Figure 3:** *QNX ≤ 6.6 Userspace Memory Layout (x86)*

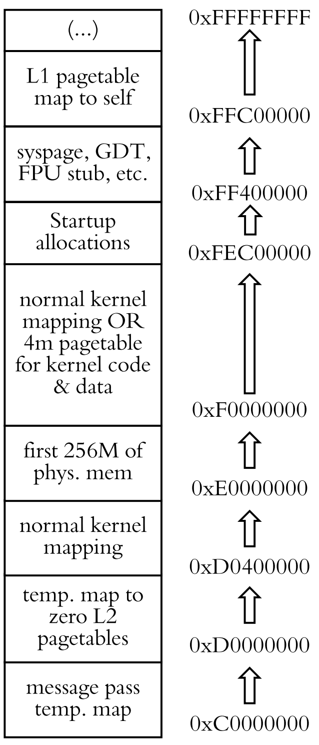


**Figure 4:** *QNX ≤ 6.6 Kernelspace Memory Layout (x86)*

## 3 QNX ≤ 6.6 Exploit Mitigations

In this section we will present the results of our reverse-engineering and subsequent analysis of QNX's exploit mitigations and secure random number generator as they are implemented in QNX versions up to and including 6.6. QNX supports a variety of exploit mitigations as outlined in Table 4 and the compiler- and linker parts of these mitigations rely on the fact that the *QNX Compile Command (QCC)* uses *GCC* as its back-end [16]. On the operating system side of things, however, the mitigation implementations are heavily customized as we will see in this section.

We can also see from Table 4 that while basic mitigations (ESP, ASLR, SSP, RELRO) are supported, this is not the case for more modern ones (eg. CFI, Kernel Data & Code Isolation, etc.) which are becoming the norm in *general purpose* operating systems such as Windows or Linux. While some of these mitigations (eg. CFI, CPI, Vtable Protection) are mostly implemented in the compiler and several libraries, it is currently not clear to what degree they are (in)compatible with QNX's design.

| Mitigation | Support | Since | Default |
|---|---|---|---|
| ESP | ✓ | 6.3.2 | × |
| ASLR | ✓ | 6.5 | × |
| SSP | ✓ | 6.5 | ×[1] |
| RELRO | ✓ | 6.5 | ×[1] |
| NULL-deref Protection | × | n/a | n/a |
| Vtable Protection | × | n/a | n/a |
| CFI | × | n/a | n/a |
| CPI | × | n/a | n/a |
| Kernel Data Isolation[2] | × | n/a | n/a |
| Kernel Code Isolation[3] | × | n/a | n/a |

**Table 4:** *QNX ≤ 6.6 Exploit Mitigation Overview*
[1] *Default QNX Momentics IDE Settings*, [2] *eg. UDEREF / SMAP / PAN*, [3] *eg. KERNEXEC / SMEP / PXN*

We disclosed all discovered issues to the vendor and as a result fixes and improvements based on our suggestions were included in QNX 7.0 as documented in Section **??**.

### 3.1 Executable Space Protection

Executable Space Protection (ESP), also referred to as Data Execution Prevention (DEP), NX memory or W^X memory, is a mitigation that seeks to prevent attackers from executing arbitrary injected payloads through a *Harvard*-style code and data memory separation on *Von Neumann* processors by rendering data memory non-executable and ensuring code memory is non-writable. ESP can be implemented by either relying on hardware support (eg. the x86 NX bit or ARM XN bit) or by means of software emulation. QNX has support for hardware-facilitated ESP among most of the architectures which support it since version 6.3.2 as shown in Table 5.

**Insecure ESP Default Policy (CVE-2017-XXXX):** While QNX supports ESP for several architectures, its

| Architecture | Support |
|---|---|
| x86 | ✓ (requires PAE on IA-32e) |
| ARM | ✓ |
| MIPS | × |
| PPC 400 | ✓ |
| PPC 600 | ✓ |
| PPC 900 | ✓ |

**Table 5:** *QNX $\leq$ 6.6 Hardware ESP Support*

implementation is dangerously weakened due to insecure default settings. As a result, the stack (but not the heap) is always executable regardless of the presence of hardware ESP support. As the documentation [55] states, the QNX microkernel, and process manager executable (PROCNTO) has a memory management startup option relating to stack executability (available as of QNX 6.4.0 or later):

- **-mx**: (Default) Enable the PROT_EXEC flag for system-allocated threads (the default). This option allows gcc to generate code on the stack - which it does when taking the address of a nested function (a GCC extension).

- **-m˜x**: Turn off PROT_EXEC for system-allocated stacks, which increases security but disallows taking the address of nested functions. You can still do this on a case-by-case basis by doing an mprotect() call that turns on PROT_EXEC for the required stacks.

Since the first option is the default, any QNX system which starts PROCNTO without explicit -M˜X settings will have an executable stack, regardless of hardware ESP support or individual binary GNU_STACK [35] settings. The rationale behind this decision seems to have been a desire for backwards compatibility with binaries which require executable stacks which has caused similar issues on Linux in the past [33]. This backwards compatibility is enforced on a system-wide (rather than on an opt-out, per-binary basis) as confirmed by the fact that the QNX program loader does not parse the GNU_STACK header of binaries. The problem with the QNX approach here is that this setting is applied on a system-wide basis and has an insecure default, putting the secure configuration burden on system integrators.

## 3.2 Address Space Layout Randomization

When developing exploits, attackers rely on knowledge of the target application's memory map for directing write and read operations as well as crafting code-reuse payloads. ADDRESS SPACE LAYOUT RANDOMIZATION (ASLR) [31] is a technique which seeks to break this assumption by ensuring memory layout secrecy via randomization of addresses belonging to various memory objects (eg. code, stack, heap, etc.) and rendering them hard to guess.

QNX has ASLR support since version 6.5 (not supported for QNX Neutrino RTOS Safe Kernel 1.0) but it's disabled by default. QNX ASLR can be enabled on a system-wide basis by starting the PROCNTO microkernel with the -MR option [55] and disabled with the -M∼R option. A QNX child process normally inherits its parent's ASLR setting but as of QNX 6.6 ASLR can also be enabled or disabled on a per-process basis by using the ON utility [51] (with the -AE and -AD options respectively). Alternatively, one can use the SPAWN_ASLR_INVERT or POSIX_SPAWN_ASLR_INVERT flags with the SPAWN and posix_spawn process spawning calls. To determine whether or not a process is using ASLR, one can use the DCMD_PROC_INFO [49] command with the DEVCTL [50] device control call and test for the _NTO_PF_ASLR bit in the flags member of the procfs_info structure.

As shown in Table 6, QNX ASLR randomizes the base addresses of userspace and kernelspace *stack*, *heap* and *mmap'ed* addresses as well as those of userspace *shared objects* (eg. loaded libraries) and the *executable image* (if the binary is compiled with PIE [16]). It does not, however, have so-called KASLR support in order to randomize the kernel image base address. The QNX Momentics Tool Suite development environment (as of version 5.0.1, SDP 6.6) does not have PIE enabled by default and indeed after an evaluation with a customized version of the CHECKSEC [34] utility we found that none of the system binaries (eg. those in /BIN, /BOOT, /SBIN directories) are PIE binaries in a default installation.

| Memory Object | Randomized |
|---|---|
| **Userspace** | |
| Stack | ✓ |
| Heap | ✓ |
| Executable Image | ✓ |
| Shared Objects | ✓ |
| mmap | ✓ |
| **Kernelspace** | |
| Stack | ✓ |
| Heap | ✓ |
| Kernel Image | × |
| mmap | ✓ |

**Table 6:** *QNX $\leq$ 6.6 ASLR Memory Object Randomization Support*

We reverse-engineered QNX's ASLR implementation (as illustrated in Figure 5) and found that it is ultimately implemented in two function residing in the microkernel: stack_randomize and map_find_va (called as part of mmap calls). QNX uses the *Executable and Linking Format (ELF)* binary format and processes are loaded from a filesystem using the `exec*`, `posix_spawn` or `spawn` calls which invoke the program loader implemented in the microkernel. If the ELF binary in question is compiled with PIE-support, the program loader will randomize the program image base address as part of an `mmap` call. When a loaded program was linked against a shared object, or a shared object is requested for loading dynamically, the runtime linker (contained in `libc`) will load it into memory using a series of `mmap` calls. A stack is allocated automatically for the main thread (which involves an allocation of stack space using `mmap`) and has its base address (further) randomized by a call to `stack_randomize`. Whenever a new thread is spawned, a dedicated stack is either allocated (and managed) by the program itself or (by default) allocated and managed by the system in a similar fashion. Userspace and kernelspace heap memory allocation, done using functions such as `malloc`, `realloc` and `free`, ultimately relies on `mmap` as well. In *kernelspace*, a dedicated stack is allocated for each processor using a call to `_scalloc` and thus relies on `mmap`. As such, all ASLR randomization can be reduced to analysis of `stack_randomize` and `map_find_va`:

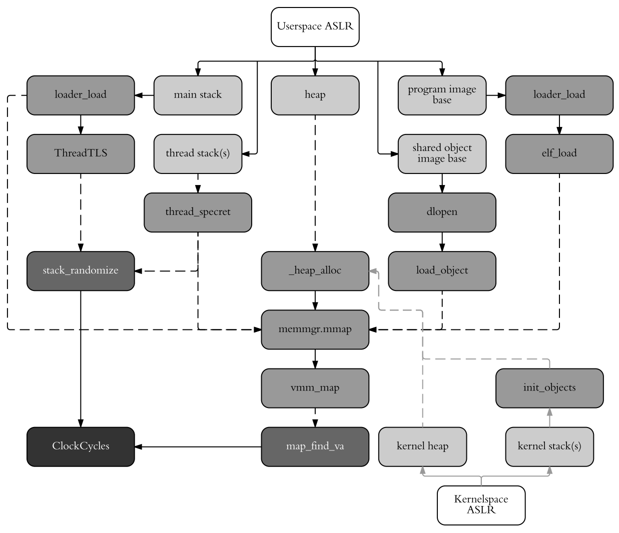


**Figure 5:** *QNX $\leq$ 6.6 ASLR Memory Object Graph*

`map_find_va`: As shown in Listings 1, 2 and 3, the QNX memory manager's `vmm_mmap` handler function invokes `map_create` and passes a dedicated mapping flag (identified only as `MAP_SPARE1` in older QNX documentation) if the ASLR process flag is set. `map_create` then invokes `map_find_va` with these same flags, which randomizes the found virtual address with a randomization value obtained from the lower 32 bits of the result of the `ClockCycles` [46] kernel call. This 32-bit randomization value is then bitwise left-shifted by 12 bits and bitwise and-masked with 24 bits resulting in a value with a mask form of `0x00FFF000`, ie. a randomization value with at most 12 bits of entropy.

**Listing 1:** *QNX 6.6 vmm_mmap Routine*

```
int vmm_mmap(PROCESS *prp, uintptr_t
    vaddr_requested, size_t size_requested,
        int prot, int flags, OBJECT *obp,
            uint64_t boff, unsigned
            alignval,
        unsigned preload, int fd, void **
            vaddrp, size_t *sizep,
            part_id_t mpart_id)
{
    ...

    create_flags = flags;

    ...

    if ( prp->flags & _NTO_PF_ASLR )
        create_flags |= MAP_SPARE1;

    r = map_create(..., create_flags);
}
```

**Listing 2:** *QNX 6.6 map_create Routine*

```
int map_create(struct map_set *ms, struct
    map_set *repl, struct mm_map_head *mh,
        uintptr_t va, uintptr_t size,
            uintptr_t mask, unsigned flags)
{
    ...

    if(flags & (MAP_FIXED|IMAP_GLOBAL)) {
        ...
    } else {
        repl->first = NULL;

        va = map_find_va(mh, va, size,
            mask, flags);
        if(va == VA_INVALID) {
            r = ENOMEM;
            goto fail1;
        }
    }

    ...
}
```

**Listing 3:** *QNX 6.6 map_find_va Routine*

```
uintptr_t map_find_va(struct mm_map_head *
    mh, uintptr_t va, uintptr_t size,
        uintptr_t mask, unsigned flags)
{

    sz_val = size - 1;

    ...

    if ( flags & MAP_SPARE1 )
    {
        uint64_t clk_val = ClockCycles();
    unsigned int rnd_val = ((_DWORD)
        clk_val << 12) & 0xFFFFFF;

    if ( flags & MAP_BELOW )
    {
        start_distance = start -
            best_start;
        if ( start != best_start )
        {
            if ( rnd_val >
                start_distance )
                rnd_val %=
                    start_distance;
            start -= rnd_val;
        }
    }
    else
    {
        end_distance = best_end -
            sz_val - start;
        if ( best_end - sz_val !=
            start )
        {
            if ( rnd_val >
                end_distance )
                rnd_val %=
                    end_distance;
            start += rnd_val;
        }
    }
}
```

`stack_randomize`: *Userspace* processes have a main stack and a dedicated stack for each subsequently spawned thread. The main stack is allocated by the program loader using an `mmap` call and subsequently has its start address randomized by `stack_randomize` called as part of a `ThreadTLS` call. Dedicated thread stacks are spawned by the `thread_specret` routine which allocates them with an `mmap` call (invoked as part of a call to `procmgr_stack_alloc`) and subsequently randomizes their start address with `stack_randomize`. This function, as shown in reverse-engineered form in Listing 4, checks whether a process has the ASLR flag set and if so it generates a sizemask (between 0x000 and 0x7FF). A randomization value is drawn from the lower 32 bits of the result of a ClockCycles kernel call which are then bitwise left-shifted by 4 bits and have the sizemask applied to them. The resulting value is subtracted from the original stack pointer value and bitwise and-masked with 28 bits. This results of a randomization value with a mask form of 0x000007F0, ie. an upper limit of 7 bits of entropy for the maximum value of size_mask = 0x7FF.

**Listing 4:** *QNX 6.6 stack_randomize Routine*

```
uintptr_t stack_randomize(const THREAD *
    const thp, uintptr_t new_sp)
{
    uintptr_t rnd_sp;
    size_t stack_size;
    unsigned int size_mask;

    rnd_sp = new_sp;

    if ( thp->process->flags &
        _NTO_PF_ASLR )
    {
        stack_size = thp->un.lcl.stacksize
            >> 4;
        if ( stack_size )
        {
            size_mask = 0x7FF;
            if ( stack_size <= 0x7FE )
                do { size_mask >>= 1; }
                    while ( size_mask >
                    stack_size );

            rnd_sp = (new_sp - ((
                ClockCycles() << 4) &
                size_mask)) & 0xFFFFFFF0;
        }
    }
    return rnd_sp;
}
```

**Weak ASLR Randomization (CVE-2017-3892):** As observed above, the randomization underlying `mmap` has a theoretical upper limit of 12 bits of entropy and the additional randomization applied to *userspace* stacks introduces at most 7 bits of entropy, combining into at most 19 bits of entropy with a mask of form 0x00FFF7F0. Addresses with such low amounts of entropy can be easily bruteforced (especially locally) and while ASLR on 32-bit systems is generally considered inherently limited [7] one should remember that these are *upper bounds*, ie. they express the maximum possible introduced entropy. Given that these upper bounds already compare rather unfavorably against the measurements of *actual* ASLR entropy in eg. Linux 4.5.0, PaX 3.14.21 and ASLR-NG 4.5.0 as per [7], this does not bode well.

All QNX ASLR randomization draws upon ClockCycles as the sole source of entropy. The QNX ClockCycles [46] kernel call returns the current value of a free-running 64-bit cycle counter using a different implementation per architecture as outlined in Table 7. Even though QNX's usage of ClockCycles seems to provide 32 bits of 'randomness', it is an ill-advised source of entropy due to its inherent regularity, non-secrecy, and predictability.

| **Architecture** | **Implementation** |
|---|---|
| x86 | *RDTSC* |
| ARM | Emulation |
| MIPS | *Count Register* |
| PPC | *Time Base Facility* |
| SuperH | *Timer Unit (TMU)* |

**Table 7:** *QNX* ClockCycles *Implementations*

In order to demonstrate this, we evaluated the entropic quality of QNX ASLR randomized addresses of several userspace memory objects. We did this with a script starting 3000 ASLR-enabled PIE processes per boot session and running 10 boot sessions, collecting 30000 samples per memory object in total. We used the *NIST SP800-90B* [41] *Entropy Source Testing (EST) tool* [38] in order to evaluate the entropic quality of the address samples by means of a *min entropy* estimate, illustrated in Table 8. *Min entropy* is a conservative way of measuring the (un)predictability of a random variable $X$ and expresses the number of (nearly) uniform bits contained in $X$, with 256 bits of uniformly random data corresponding to 256 bits of min entropy.

| **Memory Object** | **Min Entropy (8 bits per symbol)** |
|---|---|
| Stack | 1.59986 |
| Heap | 1.00914 |
| Executable Image | 0.956793 |
| Shared Objects | 0.905646 |

**Table 8:** *QNX 6.6 ASLR Userspace Memory Object Min Entropy*

From Table 8 we can see that, on average, a QNX randomized userspace memory object has a min entropy of 1.11785975. This means that it has a little more than 1 bit of min entropy per 8 bits of data. If we extrapolate this to the full 32 bits of a given address this means that the stack, heap, executable image and shared object base addresses have min entropy values of 6.39944, 4.03656, 3.827172 and 3.622584 respectively, with an average of 4.471439 bits of min entropy. This compares very unfavorably with the entropy measurements for various Linux-oriented ASLR mechanisms in [7].

On QNX, as is the case with many operating systems, child processes inherit the memory layout of parent processes. As a result when attacking forked or pre-forked applications an attacker can guess an ASLR address, after which the target child crashes and is restarted with an identical memory layout allowing the attacker to make another guess and so on. This facilitates both brute-force attacks and malicious child processes attacking siblings in *Android Zygote*-style models [23]. Given this memory layout inheritance, the fact that QNX ASLR provides only limited entropy and has no active relocation (ie. memory object locations are never re-randomized), QNX ASLR is highly susceptible to brute-force attacks.

Finally it should be noted that ClockCycles is not a secure random number generator and by drawing directly from its output the clock cycle counter value acts analogous to a random number generator's internal state. Contrary to a secure random number generator's internal state, however, the clock cycle counter value is not considered *secret* and in fact it leaks everywhere (both to local users as well as via network services). As a result, an attacker in posession of the current clock cycle counter value could reconstruct the clock cycle counter value (in a fashion analogous to the work in [36]) at the time of memory object randomization. Given the current clock cycle counter value and an estimate on memory object initialization times, an attacker can deduce the clock cycle counter value at randomization time for a given memory object and reconstruct it as:

$$clock_t = clock_c - ((time_c - time_t) * cycles_s)$$

where $clock_t, clock_c, time_t$ and $time_c$ are the target and current clock cycle counter and timestamp values and $cycles_s$ is the number of cycle increments per second.

**procfs Infoleak (CVE-2017-3892)**: The proc filesystem (procfs) is a pseudo-filesystem on Unix-like operating systems that contains information about running processes and other system aspects via a hierarchical file-like structure. This exposure of process information often includes ASLR-sensitive information (eg. memory layout maps, individual pointers, etc.) and as such has a history as a source for local ASLR infoleaks [12, 44, 62] with both *GrSecurity* and mainline Linux [26, 64] seeking to address procfs as an infoleak source. On QNX procfs [48] is implemented by the process manager component of procnto and provides the following elements for each running process:

- cmdline: Process command-line arguments.
- exefile: Executable path.
- as: The virtual address space of the target process mapped as a pseudo-file.

These procfs entries can be interacted with like files and subsequently manipulated using the devctl [50] API to operate on a file-descriptor resulting from opening a procfs PID entry. Since process entries in QNX's procfs are world-readable by default, this means a wide range of devctl-based information retrieval about any process is available to users regardless of privilege boundaries. For example, the QNX pidin [52] utility, which makes use of procfs to provide a wide range of process inspection and debugging options, easily allows any user to obtain stackframe backtraces, full memory mappings and program states for any process. This effectively constitutes a system-wide local information leak allowing attackers to circumvent ASLR. It should be noted this issue is not due to the availability of any particular utility (such as pidin) but rather results from a fundamental lack of privilege enforcement on

PROCFS.

**LD_DEBUG Infoleak (CVE-2017-9369)**: The LD_DEBUG [37] environment variable is used on some Unix-like systems to instruct the dynamic linker to output debug information during execution. On QNX there are no cross-privilege restrictions on this environment variable, leading to a (local) information leak that can be used to circumvent ASLR. Since there are no privilege checks (akin to the *'secure-execution mode'* [63] offered by some Linux distributions) on this environment variable, a local attacker can execute setuid binaries with higher privileges using dynamic linker debugging settings (eg. `LD_DEBUG=all`) in order to output sensitive information (eg. memory layout, pointers, etc.). This issue is similar to CVE-2004-1453 [22] affecting certain versions of GNU glibc.

**ASLR Correlation Attack (CVE-2017-3892)**: ASLR randomization of memory object base addresses can prove to be insufficient if different memory objects are correlated. In such a case an attacker in the posession of one address can determine the location of others by means of applying a static (or minimally varying) offset, rendering even the most limited information leaks very powerful.

During our evaluation we found a *partial* correlation attack on QNX's ASLR implementation, affecting both PIE and non-PIE binaries. The offset from the program image base to the base address of the first loaded shared library (`libc`) is of the mask form 0x00FFF000 with at most 12 bits being randomized. We evaluated the entropic quality of this offset value in order to determine the feasibility of correlation attacks by collecting 300 offset samples per boot session and running 10 boot sessions, making for 3000 samples total. Using the *NIST Entropy Source Testing (EST) tool* [38] we determined the min entropy of these offset values to be 0.918311, making for less than 1 bit of min entropy per 8 bits of data, which corresponds to 1.3774665 bits of min entropy for the 12 affected variable bits in the offset. Given that this is well below an exhaustive search, this makes a variation of the *offset2lib* [6] attack feasible.

## 3.3 Stack Smashing Protector

STACK SMASHING PROTECTOR (SSP) [43] is a so-called *stack canary* scheme which seeks to prevent the exploitation of stack buffer overflows by inserting a *secret* and *unpredictably random* canary value in between the local stack variables and the stackframe metadata (eg. saved return address, saved frame pointer). Any attempt at stack smashing which seeks to overwrite such metadata also ends up corrupting the canary value which is, upon function return, compared against the original master value so that when a mismatch is detected the SSP will invoke a failure handler.

QNX's QCC implements the GCC SSP scheme [43] and supports all the usual SSP flags (*strong*, *all*, etc.). Since the compiler-side of the QNX SSP implementation is identical to the regular GCC implementation, the *master canary* is stored accordingly and canary violation invokes the `__stack_chk_fail` handler.

**Insecure User Canary Generation (CVE-2017-XXXX)**: For *userspace applications*, this handler is implemented in QNX's LIBC. On the OS-side, reverse-engineering of LIBC shows us that violation handler (shown in cleaned-up form in Listing 5) is a wrapper for a custom function called `_SSP_FAIL` which writes an alert message to the `/dev/tty` device and raises a `SIGABRT` signal. QNX generates its master canary value once upon program startup (during loading of LIBC) and it is not renewed at any time. Instead of the regular LIBSSP function `__guard_setup`, QNX uses a custom function called `__init_cookies` (shown in Listing 6) invoked by the `_init_libc` routine in order to (among other things) generate the master canary value.

**Listing 5:** *QNX 6.6 Stack Canary Failure Handler (Userspace)*

```
void __stack_chk_fail(void)
{
    if ((fd = open("/dev/tty", 1)) != -1)
        write(fd, "***␣stack␣smashing␣
            detected␣***");
    raise(SIGABRT);
}
```

**Listing 6:** *QNX 6.6 Userspace Canary Generation*

```
void _init_cookies(void)
{
    void* stackval;

    ts0 = (ClockCycles() & 0xffffffff);
    can0 = (ts0 ^ (((&stackval) ^ (
        _init_cookies)) >> 8));
    _stack_chk_guard = can0;

    ts1 = (ClockCycles() & 0xffffffff);
    can1 = (((&stackval) ^ can0) >> 8);
    _atexit_list_cookie = (can1 ^ ts1);

    ts2 = (ClockCycles() & 0xffffffff);
    _atqexit_list_cookie = (can1 ^ ts2);

    _stack_chk_guard &= 0xff00ffff;
}
```

As shown in Listing 6, QNX canaries have a terminator-style NULL-byte in the middle and are generated using a custom randomization routine (slightly resembling the "*poor man's randomization patch*" included in some Linux distributions for performance purposes [28, 29]) rather than drawing it from a secure random number generator. The custom randomization routine draws upon three sources:

- **_init_cookies**: This is the function's own address and as such, the only randomization it introduces is derived from ASLR's effect on shared library base addresses which means that if ASLR is disabled (or circumvented) this is a static value.
- **stackval**: This is an offset to the current stack pointer and as such, the only randomization it introduces is derived from ASLR's effect on the stack base which means that if ASLR is disabled (or circumvented) this is simply a static value.
- **ClockCycles**: The lower 32 bits of the result of a ClockCycles() call are used to construct the master canary value.

Since it includes a terminator-style NULL-byte, the QNX master canary value has a theoretical upper limit of 24 bits of entropy. However, all entropy in QNX stack canaries is ultimately based on invocations of the ClockCycles kernel call. If ASLR is enabled, the stackval address gets randomized when the main thread is spawned during program startup and the `_init_cookies` address gets randomized when libc gets loaded by the runtime linker. The ts0 value is generated when `_init_cookies` is called by `_init_libc` which is invoked upon application startup (but after libc is loaded).

The first problem with using ClockCycles as a source of randomness is the limited entropy provided and thus the degree to which the canary is *unpredictable* and the size of the search space. In order to evaluate the entropic quality of QNX's canary generation mechanism, we collected canary values for three different process configurations: No ASLR, ASLR but no PIE and ASLR with PIE. We used a script starting 785 instances of each configuration per boot session and repeated this for 10 boot sessions, collecting 7850 samples per configuration in total. We then used the *NIST Entropy Source Testing (EST) tool* [38] in order to obtain *min entropy* estimates for the sample sets as illustrated in Table 9. Based on these observations we can conclude that a) QNX canary entropy is far less than the hypothetical upper bound of 24 bits, being on average 7.78981332 bits for a 32-bit canary value and b) ASLR plays no significant contributing role to the overall QNX canary entropy.

Due to the absence of any canary renewal functionality [5], regular as well as *byte-for-byte* [67] brute-force attacks are feasible against QNX, especially considering the low entropic quality of the canaries.

| Settings | Min Entropy (8 bits per symbol) |
|---|---|
| No ASLR | 1.94739 |
| ASLR, no PIE | 1.94756 |
| ASLR + PIE | 1.94741 |

**Table 9:** *QNX 6.6 Stack Canary Min Entropy*

On top of entropy issues, ClockCycles is not a secure random number generator, as we discussed before with ASLR, and as such similar reconstruction attacks could be mounted against QNX canaries.

**Absent Kernel Canary Generation (CVE-2017-XXXX)**: When it comes to *kernelspace stack canary protection*, the QNX microkernel (in the form of the procnto process) also features an SSP implementation covering a subset of its functions. Since the kernel neither loads nor is linked against libc (and canary violations need to be handled differently), SSP functionality is implemented in a custom fashion here. Reverse-engineering of the microkernel showed that it has a custom `__stack_chk_fail` handler (illustrated in Listing 7) but no master canary initialization routine. As a result, QNX never actually initializes the kernel master canary value and hence its value is completely static and known to the attacker (0x00000000), rendering QNX kernel stack canary protection trivial to bypass.

**Listing 7:** *QNX 6.6 Stack Canary Failure Handler (Kernelspace)*

```
void __stack_chk_fail(void)
{
    kprintf("***␣stack␣smashing␣detected␣
        in␣procnto␣***");
    __asm{ int 0x22 };
}
```

## 3.4 Relocation Read-Only

QNX supports Relocation Read-Only (RELRO), a mitigation that allows for marking relocation sections as read-only after they have been resolved in order to prevent attackers from using memory corruption flaws to modify relocation-related information (such as Procedure Linkage Table (PLT) related Global Offset Table (GOT) entries using *GOT-overwriting* [20]). RELRO comes in two variants: *partial* and *full*, with the former protecting non-PLT entries and the latter protecting the entire GOT. RELRO is implemented partially in the toolchain and partially in the operating system.

In most RELRO implementations, the compiler first stores constants requiring dynamic relocation in a dedicated section (typically named .data.rel.ro) before the linker creates a `PT_GNU_RELRO` program segment (itself enclosed in a `PT_LOAD` segment) to cover the sections in question. For *full* RELRO the linker also emits the `BIND_NOW` flag. On the operating system side of the implementation the dynamic linker will, upon encountering a `PT_GNU_RELRO` segment, mark the relevant pages as *read-only* after dynamic relocation. If the `BIND_NOW` flag is specified all relocations are applied immediately upon program startup. A proper RELRO implementation requires at least the following:

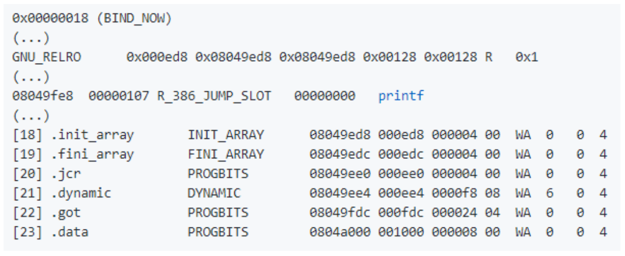
- A linker which reorders relocation sections so that they are grouped together, properly aligned for memory permission marking and precede the program data sections.
- A linker which emits a GNU_RELRO segment covering all relocation sections as well as (for full RELRO) a BIND_NOW flag.
- A dynamic linker which parses a binary for the `GNU_RELRO` segment and upon encountering it properly marks contained sections as read-only after applying relocations as well as immediately applying all relocations upon encountering a BIND_NOW flag.
- Any disabling of RELRO functionality should be mediated by privilege checks.

We uncovered the following issues violating the above:

**Broken RELRO (CVE-2017-3893)**: The GNU_RELRO segment emitted by QNX's QCC linker only covers relocation up until the .DATA section but mistakenly the section order is not properly adjusted so that internal data sections (eg. .GOT) precede program data sections (eg. .DATA). As a result, the most security-critical relocation section (the PLT-related elements of the GOT) are not included in the GNU_RELRO segment and are thus not made *read-only* after relocation.

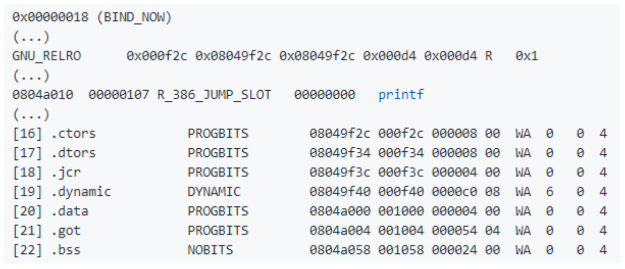
For example, on Debian Linux a full RELRO binary will look as pictured in figure 6. There we can see the GNU_RELRO segment covers the area from 0x08049ED8 to 0x0804A000 which includes .GOT. On QNX, however, the same full RELRO binary looks as pictured in figure 7. There we can see the GNU_RELRO segment covers the area from 0x08049F2C to 0x804A000 which does *not* include .GOT and thus allows us to violate RELRO with eg. a *GOT-overwriting* attack.

**Local RELRO Bypass (CVE-2017-3893)**: On Unix-like systems the LD_DEBUG environment variable is used to pass debugging settings to the dynamic linker. QNX has a custom debugging option dubbed 'IMPOSTER' which, among other things, disables RELRO. Since there are no privilege checks on this debug setting, a low-privileged user could leverage it to target setuid binaries belonging to higher-privileged users in order to bypass any RELRO protections and thus ease exploitation.



```
0x00000018 (BIND_NOW)
(...)
GNU_RELRO      0x000ed8 0x08049ed8 0x08049ed8 0x00128 0x00128 R   0x1
(...)
08049fe8  00000107 R_386_JUMP_SLOT   00000000   printf
(...)
[18] .init_array       INIT_ARRAY      08049ed8 000ed8 000004 00  WA  0   0  4
[19] .fini_array       FINI_ARRAY      08049edc 000edc 000004 00  WA  0   0  4
[20] .jcr              PROGBITS        08049ee0 000ee0 000004 00  WA  0   0  4
[21] .dynamic          DYNAMIC         08049ee4 000ee4 0000f8 08  WA  6   0  4
[22] .got              PROGBITS        08049fdc 000fdc 000024 04  WA  0   0  4
[23] .data             PROGBITS        0804a000 001000 000008 00  WA  0   0  4
```

**Figure 6:** *Debian Linux RELRO Example*



```
0x00000018 (BIND_NOW)
(...)
GNU_RELRO      0x000f2c 0x08049f2c 0x08049f2c 0x000d4 0x000d4 R   0x1
(...)
0804a010  00000107 R_386_JUMP_SLOT   00000000   printf
(...)
[16] .ctors            PROGBITS        08049f2c 000f2c 000008 00  WA  0   0  4
[17] .dtors            PROGBITS        08049f34 000f34 000008 00  WA  0   0  4
[18] .jcr              PROGBITS        08049f3c 000f3c 000004 00  WA  0   0  4
[19] .dynamic          DYNAMIC         08049f40 000f40 0000c0 08  WA  6   0  4
[20] .data             PROGBITS        0804a000 001000 000004 00  WA  0   0  4
[21] .got              PROGBITS        0804a004 001004 000054 04  WA  0   0  4
[22] .bss              NOBITS          0804a058 001058 000024 00  WA  0   0  4
```

**Figure 7:** *QNX 6.6 RELRO Example*

## 4 QNX 7.0 Exploit Mitigations

QNX 7, released in January 2017, is the successor to QNX 6.6 and comes with support for 64-bit architectures such as ARM v8 or Intel x86-64. It comes with a host of new security features such as secure boot, integrity measurement, mandatory access control, host-based anomaly detection, granular sandboxing and secure software updates.

In this section we will document the results of our reverse-engineering and analysis of QNX 7.0's exploit mitigations and secure random number generators and the degree to which they differ from and improve upon their predecessors in QNX 6.6 and earlier. Table 10 provides an overview of QNX 7.0's exploit mitigations and their default settings.

### 4.1 Executable Space Protection

Despite our disclosure of the insecure default settings, it turns out they have not been fixed in QNX 7 and as such, the stack (but not the heap) is executable by default. In order to enable non-executable stacks system integrators have to start PROCNTO with the `-m˜x` flag. Unfortunately there is no way to guarantee per-binary backwards compatibility in this case as the

| Mitigation | Support | Default |
|---|---|---|
| ESP | ✓ | × |
| ASLR | ✓ | × |
| SSP | ✓ | ✓[1] |
| RELRO | ✓ | ✓[1] |
| NULL-deref Protection | × | n/a |
| Vtable Protection | × | n/a |
| CFI | × | n/a |
| CPI | × | n/a |
| Kernel Data Isolation[2] | × | n/a |
| Kernel Code Isolation[3] | × | n/a |

**Table 10:** *QNX 7 Exploit Mitigation Overview*
[1] *Default QNX Momentics IDE Settings*, [2] *eg. UDEREF / SMAP / PAN*, [3] *eg. KERNEXEC / SMEP / PXN*

GNU_STACK ELF header is not parsed.

## 4.2 Address Space Layout Randomization

QNX 7 ASLR remains disabled by default and does not provide KASLR support for kernel image base randomization.

**ASLR Randomization**: As shown in Listing 8, QNX 7's stack_randomize routine remains identical to that of QNX 6.6 save for replacing ClockCycles as the entropy source with random_value. However, the issue of the entropy being theoretically limited to an upper bound of 7 bits as a result of the application of the bitmasking remains.

**Listing 8:** *QNX 7 stack_randomize Routine*

```
uintptr_t stack_randomize(const THREAD *
    const thp, uintptr_t new_sp)
{
  uintptr_t rnd_sp;
  size_t stack_size;
  unsigned int size_mask;

  rnd_sp = new_sp;
  stack_size = thp->un.lcl.stacksize >> 4;
  if ( stack_size )
  {
    size_mask = 0x7FF;
    if ( stack_size <= 0x7FE )
    {
      do
        size_mask >>= 1;
      while ( stack_size < size_mask );
    }

    rnd_sp = (new_sp - ((random_value() <<
         4 & size_mask)) & 0
        xFFFFFFFFFFFFFFF0;
  }
  return rnd_sp;
}
```

The ASLR randomization underlying calls to mmap is now being handled by the kerext_valloc and vm_region_create functions as part of a rewritten memory manager. As shown in Listings 9 and 10 in both cases all entropy is drawn from random_value. In the former case the full 32 bits of entropy could be absorbed on a 64-bit system while in the latter case the bitmasking imposes a theoretical upper limit of 12 bits.

**Listing 9:** *QNX 7 kerext_valloc Snippet*

```
void kerext_valloc(void *data)
{
    ...

    if ( size_val != obj->size )
    {
        randomized_addr = (obj->addr + (
            random_value() << 12) % (obj->
            size - size_val)) & 0
            xFFFFFFFFFFFFF000;
    }

    ...
}
```

**Listing 10:** *QNX 7 vm_region_create Snippet*

```
signed __fastcall vm_region_create(
    vm_aspace_t *as, vm_mmap_attr_t *attr)
{
    ...

    rnd_val = (random_value() << 12) & 0
        xFFF000;
    start_distance = start - best_start;
    if ( start != best_start )
    {
        if ( start_distance < rnd_val )
            rnd_val %= start_distance;
        start -= rnd_val;
    }

    ...
}
```

Curiously, however, QNX 7's memory manager restricts initial randomized mapping of userspace *stack*, *heap* and *executable image* objects to the lower 32 bits of the address space while restricting *shared libraries* to the lower 36 bits.

**Information Leaks**: While the LD_DEBUG Infoleak (CVE-2017-9369) has been fixed in QNX 7, the procfs Infoleak (CVE-2017-3892) is still very much present with only some restrictions imposed on it. In QNX 7 procfs has been slightly modified so that interaction with processes now goes through the /proc/*/ctl pseudo-file. While the /proc/* directories have stronger permission settings and the PIDIN tool no longer allows for direct disclosure of sensitive address information from higher-privileged processes, `/proc/*/ctl` remains world-readable for all process entries and accessible to the `devctl` API. As such, a local attacker is still able to disclose sensitive address information across privilege boundaries. While capability-based sandboxing might limit the exposure of certain processes this is not configured to be the case by default.

We have left correlation attack evaluation for QNX 7 to future work.

## 4.3 Stack Smashing Protector

SSP is enabled by default in *QNX Momentics 7.0.0* and generates 64-bit canaries on 64-bit systems.

**Userspace Canary Generation**: The new `_init_cookies` routine in QNX 7 is shown in Listing 11 where we can see that `_stack_chk_guard` is formed by the XOR sum of `rdtsc`, the code address of `_init_cookies`, the stack address of `stackval` and the value stored in `auxil_val`. This approach is similar to the one in QNX 6.6 save for the introduction of `auxil_val` which is a 64-bit value drawn from the `AT_RANDOM` ELF auxiliary vector entry. ELF auxiliary vectors [30] are a mechanism to transfer OS information to user processes via the program loader. This approach was integrated into QNX 7.0 based on our suggestions to the vendor.

**Listing 11:** *QNX 7 Userspace Canary Generation*

```
void _init_cookies()
{
  auxv_t *auxil;
  int auxil_type;
  __int64 auxil_val;
  unsigned __int64 c0;
  unsigned __int64 c1;
  char stackval;

  auxil = auxv;
  auxil_type = auxil->a_type;
  if ( auxil_type )
  {
    while ( auxil_type != AT_RANDOM )
    {
      ++auxil;
      auxil_type = auxil->a_type;
      if ( !auxil->a_type )
        goto END_AUXV;
    }
    auxil_val = auxil->a_un.a_val;
  }
  else
  {
    END_AUXV:
        auxil_val = 0LL;
  }
  c0 = __rdtsc() ^ ( ((unsigned __int64)
      _init_cookies ^ (unsigned __int64)&
      stackval) >> 8) ^ auxil_val;
  _stack_chk_guard = (void *)c0;
  c1 = ((unsigned __int64)&stackval ^ c0)
     >> 8;
  _atexit_list_cookie = (void *)(c1 ^
      __rdtsc());
  BYTE_OFFSET_6(_stack_chk_guard) = 0;
  _atqexit_list_cookie = (void *)(c1 ^
      __rdtsc());
}
```

Upon reverse-engineering the `loader_load` routine in the QNX microkernel as shown in Listing 12 we can see `AT_RANDOM` is filled with a concatenation of two 32-bit values drawn from the `random_value` kernel PRNG (discussed below in Section 5.2).

**Listing 12:** *QNX 7 AT_RANDOM Generation*

```
auxil_pointer->a_type = AT_RANDOM;
auxil_pointer->a_un.a_val = (unsigned
    int)random_value();
if ( interp_name[7] & 8 )
{
    auxil_pointer->a_un.a_val |=
        random_value() << 32;
    ...
}
```

**Kernelspace Canary Generation**: The absent kernelspace canary generation vulnerability affecting QNX 6.6 and prior has been fixed in QNX 7. During early boot in `kernel_main`, prior to kernel kickoff, the kernelspace master canary is drawn from a concatenation of two 32-bit random values drawn from the `random_value` kernel PRNG as shown in Listing 13.

**Listing 13:** *QNX 7 Kernelspace Canary Generation*

```
callin_init();
mdriver_check();
*(_DWORD *)&inkernel = 0xD00;
c0 = random_value();
c1 = random_value();
_stack_chk_guard = (void *)((c1 << 32)
    | c0);
ker_exit_kickoff(percpu_ptr->data.
    ker_stack);
```

## 4.4 Relocation Read-Only

The RELRO vulnerability we reported has been fixed in QNX 7 with QCC observing proper ELF section ordering and full RELRO being enabled by default in *QNX Momentics 7.0.0*.

## 5 QNX Secure Random Number Generators

### 5.1 /dev/random in QNX $\leq$ 6.6

Many mitigations require a source of secure randomness and ideally this is provided by the operating system itself. As such, the security of the OS random number generator is of crucial importance to the security of exploit mitigations as well as the overall cryptographic ecosystem. As prior work has shown [2, 4, 9, 10], embedded random number generation suffers from a variety of issues with far-reaching consequences and as such we reverse-engineered and analyzed the QNX OS random number generator.

QNX provides an *Operating System Cryptographically Secure Random Number Generator (OS CSPRNG)* exposed through the Unix-style `/dev/random` and `/dev/urandom` interfaces, both of which are non-blocking in practice. The OS CSPRNG is implemented as the random service [56] which runs as a userspace process started by the `/etc/rc.d/startup.sh` script. On QNX versions up to and including 6.6 the CSPRNG underlying the random service is based on the Yarrow CSPRNG [8] (which is also used by iOS, Mac OS X, AIX and some BSD descendants) rather than its recommended successor Fortuna [11].

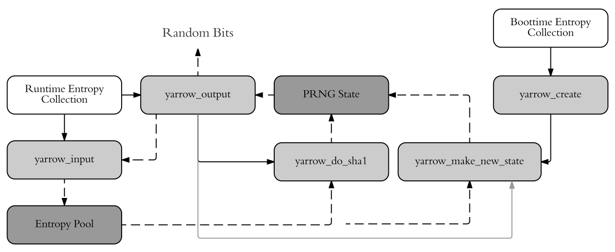


**Figure 8:** *Simplified QNX 6.6 Yarrow Design*

The QNX Yarrow implementation (as illustrated in Figure 8), however, is not based on the reference Yarrow-160 [8] design but instead on an older Yarrow 0.8.71 implementation by *Counterpane* [24] which has not undergone the security scrutiny Yarrow has seen over the years and differs in the following key aspects:

- **Single Entropy Pool**: While Yarrow-160 has separate fast and slow entropy pools, Yarrow 0.8.71 only has a single entropy pool. The two pools were introduced so that the fast pool could provide frequent reseeds of the Yarrow key to limit the impact of state compromises while the slow pool provides rare, but very conservative, reseeds of the key to limit the impact of entropy estimates which are too optimistic. Yarrow 0.8.71's single pool does not allow for such security mechanisms.
- **No Blockcipher Applied To Output**: As opposed to Yarrow-160, Yarrow 0.8.71 does not apply a block cipher (eg. in CTR mode) to the Yarrow internal state before producing PRNG output and instead simply outputs the internal state directly which results in a significantly weaker design than that of Yarrow-160.

In addition, the QNX Yarrow implementation diverges from Yarrow 0.8.71 as well in the following aspects:

- **Mixes PRNG Output Into Entropy Pool**: As part of its various entropy collection routines, QNX Yarrow mixes PRNG output back into the entropy pool. For example in the high performance counter entropy collection routine (as per the snippet in Listing 14) we can see PRNG output is drawn from QNX Yarrow, used as part of a delay routine and subsequently mixed (via a xor operation with the result of a `ClockCycles` call) back into the entropy pool. This construction deviates from all Yarrow designs and is ill-advised in the absence of further security analysis or justification.

**Listing 14:** *QNX Yarrow HPC Entropy Collection Snippet*

```
if( Yarrow )
{
    yarrow_output( Yarrow, (uint8_t *)&
        rdata, sizeof( rdata ) );
    timeout = ( rdata & 0x3FF ) + 10;
}

delay( timeout );
clk = ClockCycles();
clk = clk ^ rdata;

if( Yarrow )
    yarrow_input( Yarrow, (uint8_t *)&
        clk, sizeof( clk ), pool_id, 8
        );
```

- **Absent Reseed Control (QNX < 6.6)**: In all QNX versions prior to 6.6 reseed control is completely absent. While the required functionality was implemented, the responsible functions are *never actually invoked*, which means that while entropy is being accumulated during runtime it is never actually used to reseed the state and thus only boottime entropy is actually ever used to seed the QNX Yarrow state in versions prior to 6.6.
- **Custom Reseed Control (QNX 6.6)**: In QNX 6.6 there is a custom reseeding mechanism integrated into the `yarrow_do_sha1` and `yarrow_make_new_state` functions (as illustrated in Listings 15 and 16) which are called upon PRNG initialization and whenever output

is drawn from the PRNG (which means it is also constantly called during entropy accumulation due to the above mentioned output mixing mechanism). In both cases, a permutation named `IncGaloisCounter5X32` is applied to the entropy pool before the pool contents are mixed into a SHA1 state which eventually becomes the Yarrow internal state. Contrary to YARROW design specifications, no entropy quality estimation is done before reseeding.

**Listing 15:** *QNX 6.6 yarrow_do_sha1 function*

```
void yarrow_do_sha1( yarrow_t *p,
    yarrow_gen_ctx_t *ctx )
{
    SHA1Init(&sha);

    IncGaloisCounter5X32(p->pool.state)
        ;
    sha.state[0] ^= p->pool.state[4];
    sha.state[1] ^= p->pool.state[3];
    sha.state[2] ^= p->pool.state[2];
    sha.state[3] ^= p->pool.state[1];
    sha.state[4] ^= p->pool.state[0];

    SHA1Update(&sha, ctx->iv, 20);
    SHA1Update(&sha, ctx->out, 20);
    SHA1Final(ctx->out, &sha);
}
```

**Listing 16:** *QNX 6.6 yarrow_make_new_state function*

```
void yarrow_make_new_state(yarrow_t *
    p, yarrow_gen_ctx_t *ctx, uint8_t
     *state )
{
    for(i = 0; i < 20; i++)
      ctx->iv[i] ^= state

    SHA1Init(&sha);

    IncGaloisCounter5X32(p->pool.state)
        ;
    sha.state[0] ^= p->pool.state[4];
    sha.state[1] ^= p->pool.state[3];
    sha.state[2] ^= p->pool.state[2];
    sha.state[3] ^= p->pool.state[1];
    sha.state[4] ^= p->pool.state[0];

    SHA1Update(&sha, ctx->iv, 20);
    SHA1Final(ctx->out, &sha);
}
```

While all the above discussed divergences are at the very least ill-advised, the reseeding control issues constitute a clear security issue. In the case of absent reseeding control, it eliminates YARROW's intended defense against state compromise as well as greatly increasing system susceptibility to the so-called "*boottime entropy hole*" [10] that affects embedded systems. In the case of the QNX YARROW 6.6 custom reseeding control no entropy quality estimation is done before reseeding the state from the entropy pool thus potentially allowing for low-quality entropy to determine the entire state.

In order to evaluate the QNX YARROW PRNG output quality we used two test suites: DIEHARDER [18] and the NIST SP800-22 [40] STATISTICAL TEST SUITE (STS) [39]. DIEHARDER is a random number generator testing suite, composed of a series of statistical tests, "*designed to permit one to push a weak generator to unambiguous failure*" [18]. The NIST STATISTICAL TEST SUITE (STS) consists of 15 tests developed to evaluate the 'randomness' of binary sequences produced by hardware- or software-based cryptographic (pseudo-) random number generators by assessing the presence or absence of a particular statistical pattern. The goal is to "*minimize the probability of accepting a sequence being produced by a generator as good when the generator was actually bad*" [40]. While there are an infinite number of possible statistical tests and as such no specific test suite can be deemed truly complete, they can help uncover particularly weak random number generators.

QNX YARROW passed both the DIEHARDER and NIST STS tests but this only tells us something about the quality of PRNG output, leaving the possibility open that raw noise / source entropy is (heavily) biased which can result in predictable PRNG outputs as well as attackers being able to replicate PRNG internal states after a reasonable number of guesses. As such we reverse-engineered and evaluated the QNX RANDOM service's boot- and runtime entropy sources.

**Boottime Entropy Analysis**: When RANDOM is initialized it gathers initial boottime entropy from the following sources (as illustrated in Figure 9) which are fed to the SHA1 hash function to produce a digest used to initialize the PRNG initial state:

- **ClockTime** [47]: The current system clock time.
- **ClockCycles** [46]: The current value of a free-running 64-bit clock cycle counter.
- **PIDs**: The currently active process IDs by walking the `/proc` directory.
- **Device Names**: The currently available device names by walking the `/dev` directory.

In order to evaluate RANDOM's boottime entropy quality we used the NIST SP800-90B [41] ENTROPY SOURCE TESTING (EST) TOOL [38] to evaluate boottime entropy by means of a *min entropy* estimate. We collected RANDOM's boottime entropy from 50 different reboot sessions on the same device (by instrumenting `yarrow_init_poll` and logging the collected raw noise) and using NIST EST determined that the average min-entropy was `0.02765687`, which is far less than 1 bit of min-entropy per 8 bits of raw noise. In addition to the boottime entropy of individual boot sessions being of low quality, the static or minimally variable nature of many of the boottime noise sources (identical processes and devices available upon reboot, real-time nature of QNX limiting jitter between kernel calls thus reducing `ClockCycles` entropy, etc.) results in predictable and consistent patterns across reboots.

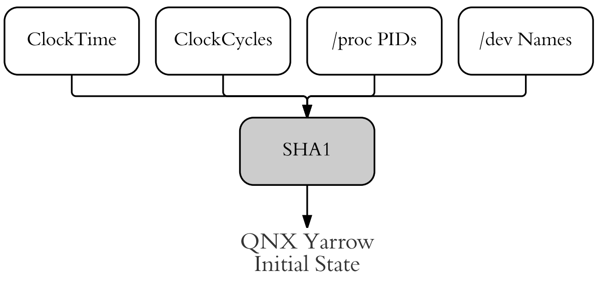


**Figure 9:** *QNX Yarrow Boottime Entropy Collection*

Another boottime entropy issue with QNX's RANDOM service is the fact that the service is started as a process by `startup.sh`. As a result, the CSPRNG is only available quite late in the boot process and many services which need it (eg. sshd) start almost immediately after. Since RANDOM only offers non-blocking interfaces, this means that one can draw as much output from the CSPRNG as one wants immediately upon availability of the device interface. Hence, many applications which start at boot and require secure random data have their 'randomness' determined almost completely by the (very low quality) boottime raw noise since there is little time for the QNX RANDOM service to gather runtime entropy before being queried thus amplifying the impact of the "*boot-time entropy hole*" [10].

**Runtime Entropy Analysis**: The QNX *random* service leaves the choice and combination of runtime entropy sources (as illustrated in Figure 10) up to the person configuring the system with the following options:

- **Interrupt Request Timing**: Up to 32 different interrupt numbers may be specified to be used as an entropy source. The entropy here is derived from interval timing measurements (measured by the `ClockTime` kernel call) between requests to a specific interrupt.
- **System Information Polling**: This source collects entropy from system information via the PROCFS [48] virtual filesystem in `/proc`. This information is composed of process and thread information (process and thread IDs, stack and program image base addresses, register values, flag values, task priority, etc.) for every currently active process.
- **High-Performance Clock Timing**: This source draws entropy from the PRNG (using the `yarrow_output` function), initiates a delay (in milliseconds) based on the PRNG output, invokes `ClockCycles` and xors the result against the earlier obtained PRNG output and feeds this into the entropy pool.
- **Library Hardware Entropy Source (Undocumented)**: This undocumented entropy source (invoked using command-line parameter *-l*) allows a user to specify a dynamic library to supply entropy collection callback functions named `entropy_source_init` and `entropy_source_start`. In order to be used the library has to export a symbol named `cookie` with the NULL-terminated value `RNG (0x524E4700)`. Based on debugging information it seems this is to allow for drawing from a hardware random number generator as an entropy source.
- **User-Supplied Input (Undocumented)**: In QNX 6.6 the RANDOM service has a write-handler made available to users via the kernel resource manager (in the form of handling write operations to the `/dev/(u)random` interfaces) which takes arbitrary user inputs of up to 1024 bytes per write operation and feeds it directly into the entropy pool by passing it to the `yarrow_input` operation. Write operations of this kind are restricted to the root user only.

After initialization, RANDOM starts a thread for each entroy source which will gather entropy and store it in the entropy pool. Contrary to our analysis of QNX RANDOM's boottime entropy, we did not perform a runtime entropy quality evaluation because during our contact with the vendor they had already indicated the current design would be overhauled in upcoming patches and future releases as a result of our findings. In addition, in all QNX versions except for 6.6 runtime entropy is accumulated but not used due to the previously mentioned absent reseeding control. We did have the following observations however:

- **Entropy Source Configuration**: Configuring runtime entropy sources is entirely left to system integrators. Since the entropic quality of certain sources (eg. interrupt request timings or system information polling) varies depending on the particular system, it is non-trivial to pick suitable sources.
- **System Information Entropy Source**: System information polling gathers raw noise from

currently running processes (in the form of process and thread debug info). A significant number of the fields in the process and thread debug info structures, however, are largely static values (eg. uid, flags, priority, stack and program base in the absence of ASLR, etc.) with most randomness derived from time-based fields (starttime, runtime) or program state (ip, sp).

- **Interrupt Request Timing Entropy Source**: Interrupt request timing gathers raw noise from interrupt invocation timings. As such this means that if integrators choose to specify interrupts that are rarely or never invoked, barely any runtime entropy is gathered using this source. Interrupt invocation frequency can be very system specific and picking the right interrupts is not trivial. The QNX documentation explicitly recommends to "*minimize the impact of [interrupt request timing overhead] by specifying only one or two interrupts from low interrupt rate devices such as disk drivers and input/serial devices*" [56], an advice which would result in less entropy being accumulated from this source. Furthermore, it seems that if for whatever reasons the RANDOM service cannot attach to an interrupt, the interrupt entropy gathering thread fails silently and no entropy is gathered for that interrupt at all.

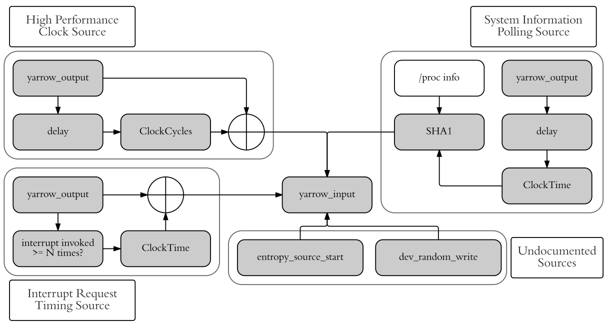


**Figure 10:** *QNX Yarrow Runtime Entropy Collection*

## 5.2 QNX 7.0 Kernel PRNG

QNX 7 has a new kernel PRNG for generation of secure random numbers implemented in the microkernel's `random_value` function. As shown in Listing 17 and illustrated in Figure 11, the kernel PRNG consists of a 256-bit seed block fed through SHA256 to produce a digest from which 32-bit random numbers are drawn iteratively before reseeding after exhausting the entire digest.

**Listing 17:** *QNX 7 Kernel PRNG*

```
unsigned int random_value()
{
  unsigned int new_dig_idx;
  unsigned int result;
  uint32_t keypad[8];
  sha256_t shout;

  if ( dig_idx > 7 )
  {
    keypad[0] = salt ^ ClockCycles();
    keypad[1] = actives[get_cpunum()];
    keypad[2] = salt ^ qtimeptr->nsec;
    keypad[3] = pid_unique ^ salt;
    keypad[4] = wakeup_timer;
    keypad[5] = kernel_exit_count;
    keypad[6] ^= random_seed;
    sha256_init(&shout);
    sha256_add(&shout, keypad, 0x20u);
    sha256_done(&shout, digest);
    result = digest[0];
    if ( !salt )
      salt = digest[0];
    new_dig_idx = 1;
  }
  else
  {
    new_dig_idx = dig_idx + 1;
    result = digest[dig_idx];
  }
  dig_idx = new_dig_idx;
  return result;
}
```

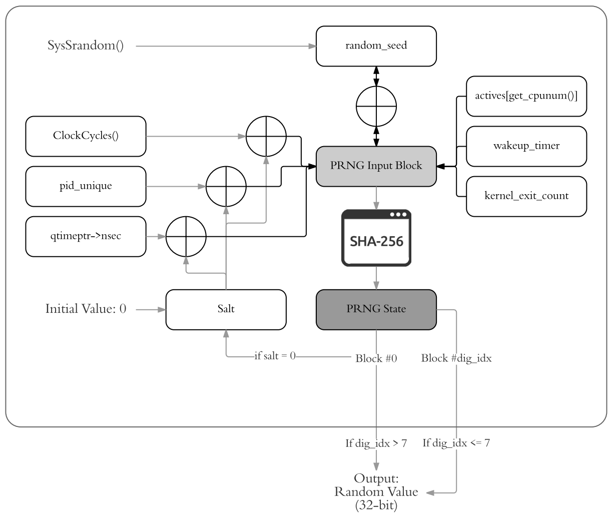


**Figure 11:** *QNX Kernel PRNG*

Kernel PRNG entropy is drawn from a combination of the following values:

- `salt`: A salt value which starts out as 0 and then gets filled with the first non-zero 32 bits of every newly generated digest.
- `ClockCycles`: The current clock cycle counter value.

- `actives[get_cpunum()]`: The currently active thread on this CPU.
- `qtimeptr->nsec`: The current time in nanoseconds.
- `pid_unique`: The currently active PID.
- `wakeup_timer`: The timer wakeup value [45].
- `kernel_exit_count`: Counter keeping track of the number of kernel exit operations.
- `random_seed`: Random seed user-supplied via `SysSrandom` [58] kernel calls. This kernel call can only be made by processes with the `PROCMGR_AID_SRANDOM` ability.

Of these sources, `pid_unique` and `actives[get_cpunum()]` have a limited range of possible values and none of the sources except for `wakeup_timer`, `kernel_exit_count` and `random_seed` can be considered *secret*. Some sources (eg. `ClockCycles`, `kernel_exit_count`) are also likely to have greatly reduced ranges during boot-time.

Finally, note that all sources are truncated to 32-bit values when stored in the seed block, that `random_seed` is only initialized when system integrators utilize it and that the final block (`keypad[7]`) is never initialized. As such, in many cases the theoretical maximum of the entropy contained within the seed block would be reduced to 192 bits. A full evaluation of the entropic quality of the QNX 7 kernel PRNG is left to future work.

## 5.3 /dev/random in QNX 7.0

Following our advisory on the QNX YARROW PRNG, the QNX 7 `random` service was redesigned to use FORTUNA instead. While the design and interface of the `random` service remains mostly the same, QNX 7 uses a customized version of the HEIMDAL [32] FORTUNA implementation as illustrated in Figure 12.

The QNX 7 FORTUNA implementation no longer has dedicated boot- and runtime entropy collection routines and draws upon the following entropy sources:

- **Interrupt Request Timing**: This source is identical to the one in the QNX 6.6 RANDOM implementation.
- **System Information Polling**: This source is identical to the one in the QNX 6.6 RANDOM implementation.

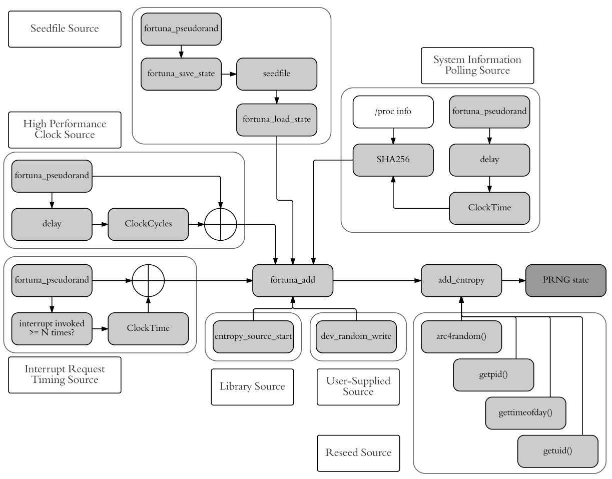


**Figure 12:** *QNX 7 Fortuna Entropy Collection*

- **High-Performance Clock Timing**: This source is identical to the one in the QNX 6.6 RANDOM implementation.
- **Library Hardware Entropy Source**: This source is identical to the one in the QNX 6.6 RANDOM implementation.
- **User-Supplied Input**: Anything written to the `/dev/(u)random` device is immediately absorbed into the PRNG state and, if seedfile state persistence is enabled, the state is saved as well. It is possible to shield this functionality with the `-m mode` option specifying permissions but by default the interface is world-writable which could possibly present an avenue for reseeding attacks.
- **Seedfile Source**: If specified, QNX 7 FORTUNA can load and save entropy from and to a 128-byte seedfile (done after writing to `/dev/(u)random` or automatically after 8192 reseedings). This file is owned by root with user read/write permissions only.
- **Reseed Source**: Reseed control is integrated into the `fortuna_bytes` and `fortuna_init` routines and thus checked periodically. It is implemented as shown in Listing 18. This routine is not likely to provide high quality reseed entropy considering that `pid` and `uid` do not change for the RANDOM service and that `arc4random` reads from `/dev/random` (creating a circular reseed loop) and uses the broken RC4 cipher.

**Listing 18:** *QNX 7 fortuna_reseed*

```
#define INIT_BYTES 128

int fortuna_reseed()
{
    uint32_t buf[INIT_BYTES / sizeof(
        uint32_t)];
    int i;
    int entropy_p = 0;

    if ( !init_done )
        abort();

    for (i = 0; i < sizeof(buf)/sizeof(buf
        [0]); i++)
        buf[i] = arc4random();

    add_entropy(&main_state, (void *)buf,
        sizeof(buf));
    entropy_p = 1;

    pid_t pid = getpid();
    add_entropy(&main_state, (void *)&pid,
         sizeof(pid));

    struct timeval tv;
    gettimeofday(&tv, NULL);
    add_entropy(&main_state, (void *)&tv,
        sizeof(tv));

    uid_t u = getuid();
    add_entropy(&main_state, (void *)&u,
        sizeof(u));

    return entropy_p;
}
```

Due to the elimination of dedicated boottime entropy harvesting and its rapid startup time, QNX 7 FORTUNA is likely to suffer from the "*boottime entropy hole*" (unless system integrators explicitly enable seedfiles / state persistence) but we leave a full analysis of entropic quality to future work.

## 6 Conclusion

We reverse-engineered and analyze the exploit mitigations and secure random number generators of QNX $\leq$ 6.6 and 7.0 and found and reported a myriad of issues of varying degrees of severity. Table 11 presents an overview of the analyzed mitigations and RNGs, their issues and what versions are affected by them. Note that we have left proper RNG entropy quality and ASLR correlation attack evaluation of QNX 7's to future work and as such we can neither confirm nor rule out issues in this regard.

| Component | Issues | Affected |
|---|---|---|
| ESP | Disabled by default | $\leq$ 7.0 |
| ASLR | Disabled by default[1] | $\leq$ 7.0 |
| ASLR | No KASLR[1] | $\leq$ 7.0 |
| ASLR | Weak Randomization[1] | $\leq$ 6.6 |
| ASLR | No Re-Randomization[1] | $\leq$ 7.0 |
| ASLR | `procfs` Infoleak[1] | $\leq$ 7.0 |
| ASLR | `LD_DEBUG` Infoleak[3] | $\leq$ 6.6 |
| ASLR | Correlation Attack[1] | $\leq$ 6.6 |
| SSP | Disabled by default | $\leq$ 6.6 |
| SSP | Weak Randomization | $\leq$ 6.6 |
| SSP | No Re-Randomization | $\leq$ 7.0 |
| SSP | No Kernel Canaries | $\leq$ 6.6 |
| RELRO | Disabled by default[2] | $\leq$ 6.6 |
| RELRO | Broken RELRO[2] | $\leq$ 6.6 |
| RELRO | `LD_DEBUG` Bypass[2] | $\leq$ 6.6 |
| OS CSPRNG | Ill-Advised Design | $\leq$ 6.6 |
| OS CSPRNG | Absent Reseed Control | < 6.6 |
| OS CSPRNG | Low Boottime Entropy | $\leq$ 6.6 |

**Table 11:** *QNX Mitigation & RNG Issues Overview*
[1] *CVE-2017-3892*, [2] *CVE-2017-3893*, [3] *CVE-2017-9369*

We can see that despite our disclosure of the issues affecting QNX 6.6 and subsequent fixes being drafted for the bulk of them, some of them remained in QNX 7.0. Regardless, General Availability (GA) patches are available for all issues affecting QNX $\leq$ 6.6 in Table 11 (naturally excluding those affecting QNX 7.0).

One striking observation is that while QNX clearly attempts to keep up with at least basic exploit mitigations as they have evolved in the general purpose world, the fact that it is a proprietary OS outside of the Linux, Windows and BSD lineages means that they cannot trivially port mitigations, patches and improvements from these operating systems. In addition, the relative lack of attention to QNX by outside security researchers is evident from the degree to which certain vulnerabilities and issues (such as the local information leaks or the "*poor man's randomization patch*" design for ASLR/SSP) resemble older vulnerabilities on other Unix-like systems. Finally, our findings re-confirm the notion that secure random number generation and especially integrating suitable entropy sources is an issue that continues to plague the embedded world. The impact of this goes beyond affecting the quality of exploit mitigations and has consequences for the wider security ecosystem as a whole.

It is our hope that this work inspires other security researchers to further investigate the security and OS internals of QNX and other closed-source embedded operating systems.

## Bibliography

[1] Alex Plaskett et al. *QNX: 99 Problems but a Microkernel ain't one!* 2016.

[2] Daniel J. Bernstein et al. "Factoring RSA keys from certified smart cards: Coppersmith in the wild". In: *ASIACRYPT* (2013).

[3] Daniel Martin Gomez et al. *BlackBerry PlayBook Security: Part one*. 2011.

[4] David Kaplan et al. “Attacking the Linux PRNG on Android & Embedded Devices”. In: *Black Hat Europe* (2014). URL: https://www.blackhat.com/docs/eu-14/materials/eu-14-Kedmi-Attacking-The-Linux-PRNG-On-Android-Weaknesses-In-Seeding-Of-Entropic-Pools-And-Low-Boot-Time-Entropy.pdf.

[5] Hector Marco-Gisbert et al. “Preventing brute force attacks against stack canary protection on networking servers”. In: *12th IEEE International Symposium on Network Computing and Applications (NCA)* (2013).

[6] Hector Marco-Gisbert et al. “On the Effectiveness of Full-ASLR on 64-bit Linux”. In: *DeepSec* (2014).

[7] Hector Marco-Gisbert et al. “Exploiting Linux and PaX ASLR’s weaknesses on 32- and 64-bit systems”. In: *BlackHat Asia* (2016).

[8] John Kelsey et al. “Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator”. In: *Sixth Annual Workshop on Selected Areas in Cryptography* (1999).

[9] Keaton Mowery et al. “Welcome to the Entropics: Boot-Time Entropy in Embedded Devices”. In: *IEEE Security and Privacy* (2013).

[10] Nadia Heninger et al. “Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices”. In: *USENIX Security Symposium* (2012).

[11] Niels Ferguson et al. *Practical Cryptography*. Wiley, 2003.

[12] Tavis Ormandy et al. *Linux ASLR Curiosities*. 2009. URL: https://www.cr0.org/paper/to-jt-linux-alsr-leak.pdf.

[13] Zach Lanier et al. *Voight-Kampff’ing The BlackBerry PlayBook*. 2012.

[14] Zach Lanier et al. *No Apology Required: Deconstructing BB10*. 2014.

[15] Alexander Antukh. *Dissecting Blackberry 10 – An initial analysis*. 2013.

[16] BlackBerry. *Using compiler and linker defenses (BlackBerry Native SDK for PlayBook OS)*. URL: http://developer.blackberry.com/playbook/native/reference/com.qnx.doc.native_sdk.security/topic/using_compiler_linker_defenses.html.

[17] BlackBerry. *QNX*. 2017. URL: http://www.qnx.com/content/qnx/en.html.

[18] Robert G. Brown. *Dieharder: A Random Number Test Suite*. URL: https://www.phy.duke.edu/\~rgb/General/dieharder.php.

[19] Tim Brown. *QNX Advisories*. URL: https://packetstormsecurity.com/files/author/4309/.

[20] c0ntex. *How to hijack the Global Offset Table with pointers for root shells*. URL: http://www.infosecwriters.com/text_resources/pdf/GOT_Hijack.pdf.

[21] Communications Security Establishment Canada. *EAL 4+ Evaluation of QNX Neutrino® Secure Kernel v6.4.0*. 2009. URL: https://www.commoncriteriaportal.org/files/epfiles/neutrino-v640-cert-eng.pdf.

[22] Silvio Cesare. *CVE-2004-1453*. 2004. URL: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-1453.

[23] Colt. *Android OS - Processes and the Zygote!* URL: http://coltf.blogspot.nl/p/android-os-processes-and-zygote.html.

[24] Counterpane. *Yarrow 0.8.71*. URL: https://www.schneier.com/code/Yarrow0.8.71.zip.

[25] CVE Details. *QNX CVEs*. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-436/QNX.html.

[26] Jake Edge. *proc: avoid information leaks to non-privileged processes*. 2009. URL: https://patchwork.kernel.org/patch/21766/.

[27] Julio Cesar Fort. *QNX Advisories*. URL: https://packetstormsecurity.com/files/author/3551/.

[28] Hagen Fritsch. *Buffer overflows on linux-x86-64*. 2009.

[29] Hagen Fritsch. *Stack Smashing as of Today*. 2009.

[30] Manu Garg. *About ELF Auxiliary Vectors*. URL: http://articles.manugarg.com/aboutelfauxiliaryvectors.

[31] Hector Marco Gisbert. “Cyber-security protection techniques to mitigate memory errors exploitation”. In: (2015). URL: https://riunet.upv.es/bitstream/handle/10251/57806/Marco%20-%20Cyber-security%20protection%20techniques%20to%20mitigate%20memory%20errors%20exploitation.pdf?sequence=1&isAllowed=y.

[32] Heimdal. *The Heimdal Kerberos 5, PKIX, CMS, GSS-API, SPNEGO, NTLM, Digest-MD5 and, SASL implementation*. URL: http://www.h5l.org/.

[33] Alejandro Hernandez. *A Short Tale About executable_stack in elf_read_implies_exec() in the Linux Kernel*. URL: http://blog.ioactive.com/2013/11/a-short-tale-about-executablestack-in.html.

[34] Tobias Klein. *checksec*. URL: http://www.trapkit.de/tools/checksec.html.

[35] Gentoo Linux. *Hardened/GNU stack quickstart*. URL: https://wiki.gentoo.org/wiki/Hardened/GNU_stack_quickstart.

[36] Matt Miller. "Reducing the Effective Entropy of GS Cookies". In: *Uninformed Vol. 7* (2007).

[37] Bojan Nikolic. *The LD_DEBUG environment variable*. URL: http://www.bnikolic.co.uk/blog/linux-ld-debug.html.

[38] NIST. *NIST Entropy Source Testing (EST) tool*. URL: https://github.com/usnistgov/SP800-90B_EntropyAssessment.

[39] NIST. *NIST Statistical Test Suite (STS)*. URL: http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html.

[40] NIST. "NIST SP800-22: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications". In: *NIST* (2010).

[41] NIST. "NIST SP800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation". In: *NIST* (2016).

[42] Alex Plaskett. *QNX Security Architecture*. 2016.

[43] Paul Rascagneres. *Stack Smashing Protector*. 2010.

[44] Fermin J. Serna. "The info leak era on software exploitation". In: *Black Hat US* (2012).

[45] QNX Software Systems. *Clock and timer services*. URL: http://www.qnx.com/developers/docs/7.0.0/index.html#com.qnx.doc.neutrino.sys_arch/topic/kernel_CLOCKANDTIMER.html.

[46] QNX Software Systems. *ClockCycles()*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fc%2Fclockcycles.html.

[47] QNX Software Systems. *ClockTime(), ClockTime_r()*. URL: http://www.qnx.com/developers/docs/660/topic/com.qnx.doc.neutrino.lib_ref/topic/c/clocktime.html.

[48] QNX Software Systems. *Controlling processes via the /proc filesystem*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Fprocess_proc_filesystem.html.

[49] QNX Software Systems. *DCMD_PROC_INFO*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.cookbook%2Ftopic%2Fs3_procfs_DCMD_PROC_INFO.html.

[50] QNX Software Systems. *devctl*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fd%2Fdevctl.html.

[51] QNX Software Systems. *On*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.utilities%2Ftopic%2Fo%2Fon.html.

[52] QNX Software Systems. *pidin*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=/com.qnx.doc.neutrino.utilities/topic/p/pidin.html.

[53] QNX Software Systems. *Private virtual memory*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.sys_arch%2Ftopic%2Fproc_Private_virtual_memory.html.

[54] QNX Software Systems. *Procmgr abilities*. URL: http://www.qnx.com/developers/docs/6.6.0.update/#com.qnx.doc.neutrino.prog/topic/process_Procmgr_abilities.html.

[55] QNX Software Systems. *Procnto*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.utilities/topic/p/procnto.html.

[56] QNX Software Systems. *Random*. URL: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.utilities/topic/r/random.html.

[57] QNX Software Systems. *Shared memory*. URL: http://www.qnx.com/developers/docs/7.0.0/index.html#com.qnx.doc.neutrino.sys_arch/topic/ipc_Shared_memory.html.

[58] QNX Software Systems. *SysSrandom(), SysSrandom_r()*. URL: http://www.qnx.com/developers/docs/7.0.0/index.html#com.qnx.doc.neutrino.lib_ref/topic/s/syssrandom.html.

[59] QNX Software Systems. *Typed memory*. URL: http://www.qnx.com/developers/docs/7.0.0/index.html#com.qnx.doc.neutrino.sys_arch/topic/ipc_Typed_memory.html.

[60] QNX Software Systems. *QNX Neutrino RTOS: System Architecture*. 2014. URL: http://support7.qnx.com/download/download/26183/QNX_Neutrino_RTOS_System_Architecture.pdf.

[61] QNX Software Systems. *50 Million Vehicles and Counting: QNX Achieves New Milestone in Automotive Market*. 2015. URL: http://www.qnx.com/news/pr_6118_3.html.

[62] Julien Tinnes. *Local bypass of Linux ASLR through /proc information leaks*. 2009. URL: http://blog.cr0.org/2009/04/local-bypass-of-linux-aslr-through-proc.html.

[63] Ubuntu. *ld.so, ld-linux.so - dynamic linker/loader*. 2017. URL: http://manpages.ubuntu.com/manpages/xenial/man8/ld.so.8.html.

[64] Ubuntu. */proc/pid/maps protection*. 2017. URL: https://wiki.ubuntu.com/Security/Features#proc-maps.

[65] Gerrit De Vynck. *CIA Listed BlackBerry's Car Software as Possible Target*. 2017. URL: https://www.bloomberg.com/news/articles/2017-03-08/cia-listed-blackberry-s-car-software-as-possible-target-in-leak.

[66] Ralf-Philipp Weinmann. *BlackberryOS 10 From a Security Perspective*. 2013.

[67] Adam 'pi3'Zabrocki. "Scraps of notes on remote stack overflow exploitation". In: *Phrack* (2010).