MWR Labs Whitepaper

QNX Security Architecture

Alex Plaskett

Contents page

1. Introduction

This paper aims to cover the QNX local attack surface and architecture (practically applied to Blackberry 10). This paper contains a technical overview of QNX in order to allow further security research and exploration of the security features of the platform. This paper will focus primarily on QNX architecture and security features which are novel to QNX (and therefore Blackberry 10).

The first half of this paper will provide the necessary background information into the QNX operating system itself and its key features. The second half of this paper will describe how to attack these QNX features and locate vulnerabilities. This paper builds on previous knowledge about Blackberry 10 (BB10), for example how to unpack the firmware images for static analysis and benefits from some prior knowledge of attack research in this area.

A number of vulnerabilities have been reported to Blackberry, however are currently in different states of remediation, therefore if an issue is still outstanding it has not been included within the paper.

Whilst there has been a low number of sales of Blackberry 10 device in comparison to the other mobile platforms, the QNX operating system itself runs on a large amount of hardware and supports many different industries. These includes Automotive, Industrial and Medical, Networking, Telecoms and the Security and Defence sectors. Many of these platforms are safety or security critical, therefore weaknesses in QNX can have a significant impact to these systems. This research should allow security professionals the necessary background information to start performing in-depth assessment of these systems.

The code produced as part of this research is available at:

https://github.com/alexplaskett/QNXSecurity

Further issues will be released on http://labs.mwrinfosecurity.com once patches are available.

1.1 Previous Research

Currently only a small amount of QNX security research has been publically disclosed. This paper aims to address this by providing a good overall understanding of QNX. However, there has been previous research performed into Blackberry 10 and the Playbook OS.

Please see the appendix for a full list of previous security research performed in these areas.

2. QNX Hardware Platforms

Whilst Blackberry 10 OS (BB10 OS) is a continuation of the Playbook OS, the Playbook runs on different a hardware System on Chip (TI OMAP) than Blackberry 10 (Qualcomm Snapdragon). However, there was an initial development of a BB10 development alpha phone which ran on an OMAP-based platform called the London.

At the time of writing MWR focused on the following devices:

2.1.1 Blackberry 10 Device (Z10)

+ Qualcomm MSM8960 System on Chip
+ Qualcomm ARM Snapdragon S4 dual-core processor at 1.5GHz
+ BB10 OS
+ Adreno 225 GPU and 2GB of RAM.

This research was performed with access to this device, however, the aim was to identify generic methods that could be used when assessing any QNX based device.

2.1.2 Blackberry 10 Simulator

The Blackberry 10 simulator is a para-virtualised BB10 virtual machine with all the hardware dependant features removed. The BB10 simulator runs on X86 architecture whilst physical devices run on the ARM platform. One advantage of running the simulator for security assessment is that it is possible to root the simulator using techniques described in the following QCONN section.

2.1.3 QNX 6.5 Virtual Machine

The QNX 6.5 virtual machine was also used during this research to provide a number of platform tools which had been removed on the BB10 simulator (for example mkifs, dumpifs). The QNX 6.5 virtual machine was available as a trial download from the QNX website.

3. QNX Operating System Introduction

BlackBerry 10 is based off QNX Neutrino 8.0 Real-time Operating System (RTOS). In order to understand the security model of Blackberry 10 one must first understand the architecture of the OS and security features provided by the platform. There is a large amount of non-security information available about QNX including public documentation on-line for developers. This paper relies on the public information to provide the background into QNX and aims to highlight where potential security weaknesses might be found. In comparison there is little security documentation available for QNX or best practices.

The latest version of QNX available as a trial from the website is QNX 6.5. Therefore the Blackberry 10 simulator has been used together with a physical device (Z10). This is primarily due to the ability to root the simulator using techniques described in the following QCONN section of the document.

The Blackberry 10 simulator is currently available at the following location: http://developer.blackberry.com/devzone/develop/simulator/simulator_installing.html.

The Blackberry 10 simulator at the time of writing is:

+ Version: 10.3.1.995

A fully patched Z10 at the time of writing is as follows:

+ Version: 10.3.2.2474

The following CPU info can be found on the Z10 device:

```
cat cpuinfo
Processor : ARMv7 Processor rev 2 (v7l)
BogoMIPS : 162.54
Features : swp half thumb fastmult vfp edsp thumbee neon
CPU implementer: 0x51
CPU architecture: 7
CPU variant : 0x0
CPU part : 0x00f
CPU revision : 2
```

At one point within QNX's lifetime some of the source code was made publically available for QNX 6.5 (http://www.qnx.com/news/pr_2471_1.html). However, this was quickly removed and source code access restricted. However, old QNX source code for version 6.4 was found at the following location on SourceForge (http://sourceforge.net/p/monartis/opengnx/ci/master/tree/).

3.1 QNX Security History

QNX has not had a particularly good security history throughout its lifetime. Problems have been identified previously with setuid binaries (such as http://seclists.org/fulldisclosure/2014/Mar/98), file system paths and even linker problems. Whilst the operating system itself is designed to be highly robust and fault tolerant, mistakes which are often found on other *NIX based platforms can be introduced into QNX too and due to the lack of public review can exist for a large amount of time.

3.1.1 QNX 6.3 - QNX 6.5

Between QNX 6.3 and QNX 6.5 the following vulnerabilities were identified (as reported on NIST vulnerability database) a summary is included below:

CVE-2014-2534 - /sbin/pppoectl in BlackBerry QNX Neutrino RTOS 6.4.x and 6.5.x allows local users to obtain sensitive information by reading "bad parameter" lines in error messages, as demonstrated by reading the root password hash in /etc/shadow.

CVE-2014-2533 - /sbin/ifwatchd in BlackBerry QNX Neutrino RTOS 6.4.x and 6.5.x allows local users to gain privileges by providing an arbitrary program name as a command-line argument.

CVE-2013-2687 - Stack-based buffer overflow in the bpe_decompress function in (1) BlackBerry QNX Neutrino RTOS through 6.5.0 SP1 and (2) QNX Momentics Tool Suite through 6.5.0 SP1 in the QNX Software Development Platform allows remote attackers to cause a denial of service (application crash) or possibly execute arbitrary code via crafted packets to TCP port 4868.

CVE-2011-4060 - The runtime linker in QNX Neutrino RTOS 6.5.0 does not properly clear the LD_DEBUG_OUTPUT and LD_DEBUG environment variables when a program is spawned from a setuid program, which allows local users to overwrite files via a symlink attack.

CVE-2008-3024 - Stack-based buffer overflow in phgrafx in QNX Momentics (aka RTOS) 6.3.2 and earlier allows local users to gain privileges via a long .pal filename in palette/.

CVE-2006-0618 - Format string vulnerability in fontsleuth in QNX Neutrino RTOS 6.3.0 allows local users to execute arbitrary code via format string specifiers in the zeroth argument (program name).

CVE-2006-0619 - Multiple stack-based buffer overflows in QNX Neutrino RTOS 6.3.0 allow local users to execute arbitrary code via long (1) ABLPATH or (2) ABLANG environment variables in the libAP library (libAp.so.2) or (3) a long PHOTON_PATH environment variable to the setitem function in the libph library.

QNX Kernel Issues:

Only a small number of security weaknesses have been identified publically in the QNX kernel itself. One of these is as follows which was in a core system call (ker_msg_sendv) identified by Ilja van Sprundel. This issue was an integer overflow in a system call which leads to heap memory corruption within the kernel.

It should be noted that this weakness was identified prior to Blackberry's acquisition of QNX Software Systems. More information can be found at the following location:

http://www.ioactive.com/pdfs/QNXIntegerOverflow.pdf

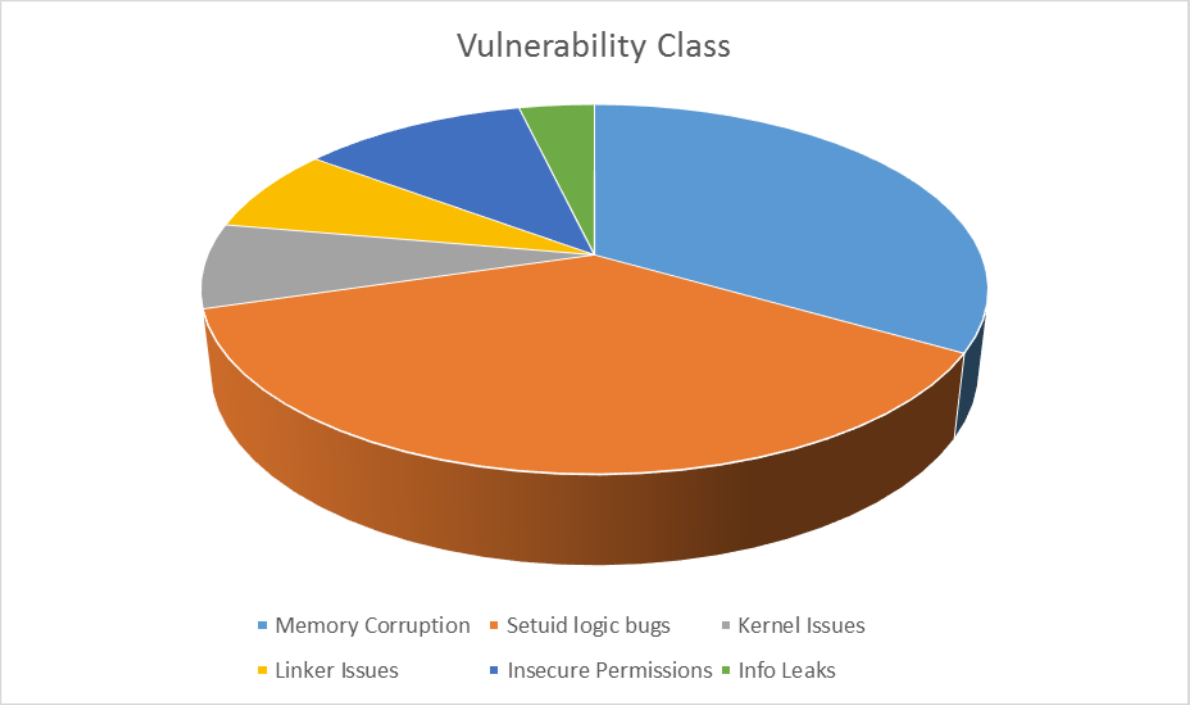
Other previous kernel weaknesses have been identified with the handling of ELF images or process manager calls as follows:

https://gist.github.com/mogigoma/193603

https://www.exploit-db.com/exploits/7823/

These issues were listed as a denial of service and expected could not be turned into a method of privilege escalation, however, full investigation into the root cause of these issues was not performed.

A summary of the issues identified between QNX 6.0 and QNX 6.5 is as follows:



As you can see from this diagram most recent QNX issues fell into the typical vulnerability classes found within a POSIX based operating system. Primarily the use of setuid binaries was a common source of vulnerabilities for prior QNX versions.

3.1.2 QNX 6.6 Onwards

QNX 6.6 onwards introduced a number of compiler and operating system level mitigations to increase the difficulty of exploitation on the platform. This included the following:

- User space ASLR
- Stack Cookies
- Non-Executable Memory Protection (NX)
- Position Independent Code (PIE) / RELocation Read Only (RELRO)
- Process Manager Abilities (procmgr_ability see process manager section following).

Blackberry 10 OS took this further and removed all setuid binaries from the build. However, a number of setgids still exist. This has significantly hardened the device against privilege escalation vulnerabilities and therefore more in-depth investigation into the OS is required in order to identify possible

weaknesses. Blackberry have also performed significant hardening of the stock QNX source, which can be supported by identifying vulnerabilities in QNX 6.5 and determining if they affect Blackberry 10 OS.

3.1.3 Playbook Vulnerabilities

One issue was identified with the Blackberry Playbook which led to the initial root of this device. This jailbreak was named Dingleberry by the creators (@cmwdotme, @xpvqs and @neuralic). The following information was included in the patch information released from Blackberry which addressed the weakness using in Dingleberry:

"The BlackBerry PlayBook service on the Research in Motion (RIM) BlackBerry PlayBook tablet with software before 1.0.8.6067 allows local users to gain privileges via a crafted configuration file in a backup archive"

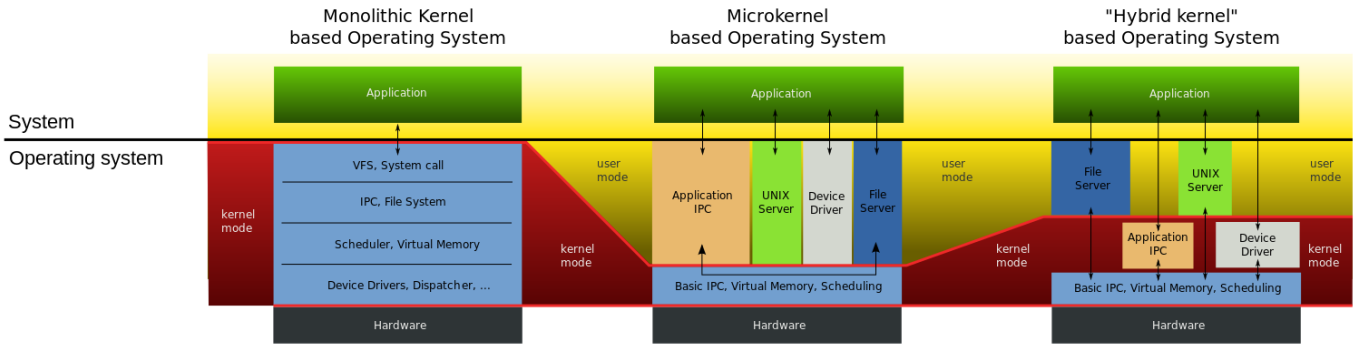
Dingleberry exploited a weakness with Blackberry backup's not being cryptographically signed to replace Samba.conf file with a modified version. This modified version was then restored to device and used to gain root access.

Blackberry addressed this weakness and after that date there was no publically known jailbreak of the Blackberry Playbook.

3.2 QNX Microkernel Design Overview

A microkernel is typically defined as the minimum of software which can provide the mechanisms to implement an operating system. Typically these are address space management, thread management and inter-process communication which are part of the microkernel itself.

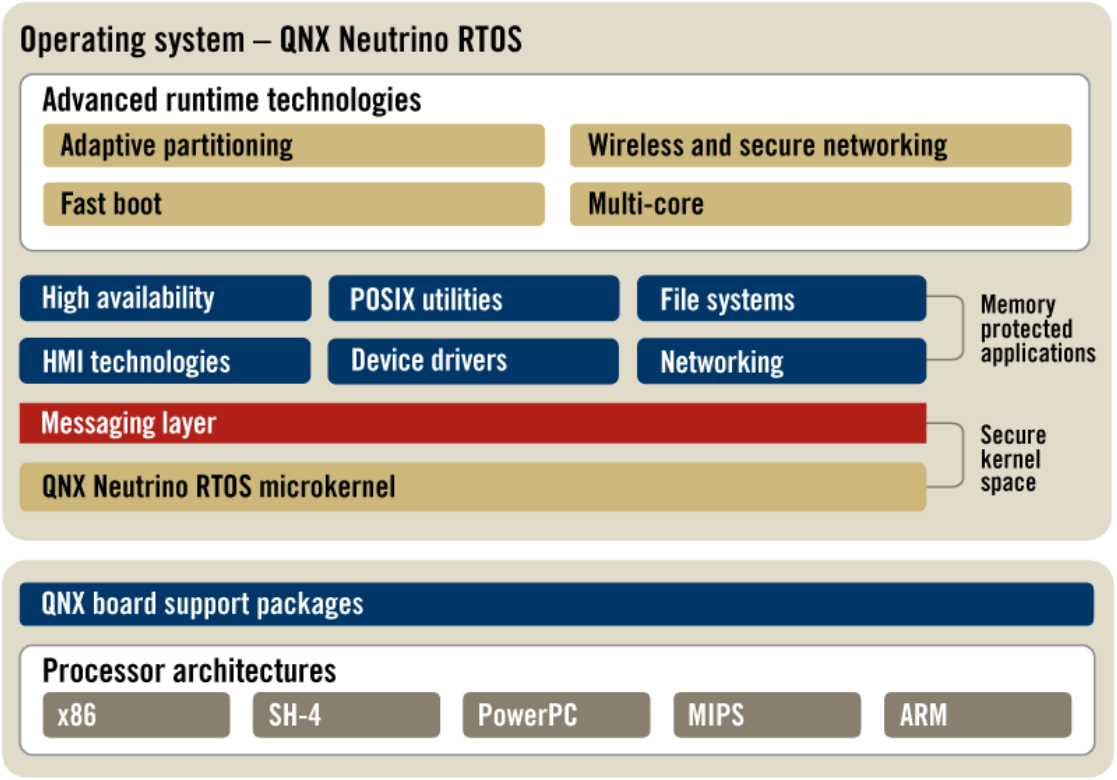
In comparison a monolithic operating system is one in which the whole operating system is executing within kernel space in supervisor mode (drivers, file system etc.). This can be visualised as follows:



Taken from: https://upload.wikimedia.org/wikipedia/commons/thumb/d/d0/OS-structure2.svg/800px-OS-structure2.svg.png

QNX is based on the microkernel design approach to limit the amount of code within the microkernel itself. Android on the other hand is based on the monolithic Linux kernel model. In QNX, the microkernel is combined with the process manager in a single module (called procnto). This process can be considered the main critical process for OS operation.

The QNX Neutrino RTOS is structured as follows:



Taken from: http://www.qnx.org.uk/images/products/rtos/Neutrino_RTOS_3_10_D1_480.gif

This model has some interesting concepts compared to other operating systems:

+ The drivers, protocol stack and file system all run outside of the kernel. Therefore if any crashes occur then these processes can be restarted and not result in a full kernel panic.
+ Whilst this is designed for reliability, this also increases the security by reducing the attack surface of the kernel itself by moving code into other components. However, as we will see in future this increases the IPC attack surface.
+ QNX architecture is based on message passing between processes on the system.

Another popular microkernel is the L4 kernel (https://www.l4ka.org/) which also shares a similar architectural approach as QNX.

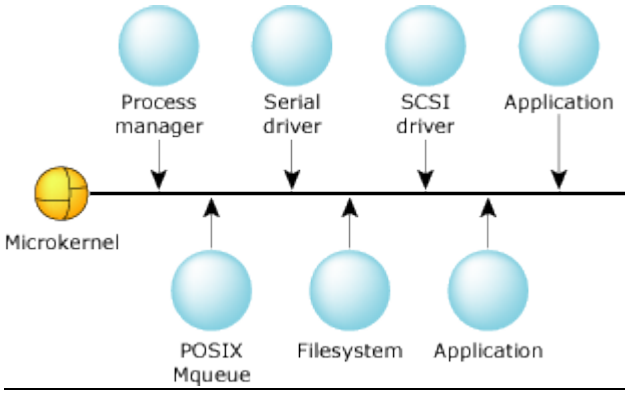
Whilst there are pros and cons of the microkernel approach, one of the advantages of microkernel architecture is that it can allow better compartmentalisation of resources and reduces the attack surface of the kernel itself. This can be also be seen by compartmentalisation applied to Blackberry 10. It is argued that using this approach inherently leads to the principles of least privilege needed to provide the functionality offered.

Therefore microkernel have typically been used in systems which are designed for high security applications. In certain cases of third generation microkernels formal method verification of the implementation has been performed (https://sel4.systems/). However, in QNX's case this has not been subject to such a high level of formal assurance.

3.3 QNX Messaging Layer

In order for the microkernel to function QNX relies on message passing between the OS kernel and processes on the system for inter-process communication. This messaging model can be conceptualised as a typical client / server architecture based messaging. The client sends a message to the server, server receives the messages, handles the message and replies to the client.

The message passing architecture can be visualised as follows:



Taken from: **http://www.qnx.org.uk/developers/docs/6.4.1/neutrino/technotes/managing_mq_mqueue.html**

This can be demonstrated with the following simple messaging passing code:

```
// establish a connection
coid = ConnectAttach (0, 77, 1, 0, 0);
if (coid == -1) {
    fprintf (stderr, "Couldn't ConnectAttach to 0/77/1!\n");
    perror (NULL);
    exit (EXIT_FAILURE);
}

// send the message
if (MsgSend (coid,
             smsg,
             strlen (smsg) + 1,
             rmsg,
             sizeof (rmsg)) == -1) {
    fprintf (stderr, "Error during MsgSend\n");
    perror (NULL);
    exit (EXIT_FAILURE);
}
```

This code uses ConnectAttach to establish a connection between a process and a channel (http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/lib_ref/c/connectattach.html) and then uses MsgSend to send to message over the channel. The process id of 77 of the owner of the channel is used together with the channel id of the channel in which the connection is to be established too.

On the server side a channel has previously been created and is listening for messages:

```
chid = ChannelCreate (0);

    // this is typical of a server:  it runs forever
    while (1) {

        // get the message, and print it
        rcvid = MsgReceive (chid, message, sizeof (message),
                            NULL);
```

QNX message passing is documented well at the following location:

http://www.qnx.co.uk/developers/docs/6.4.1/neutrino/getting_started/s1_msg.html

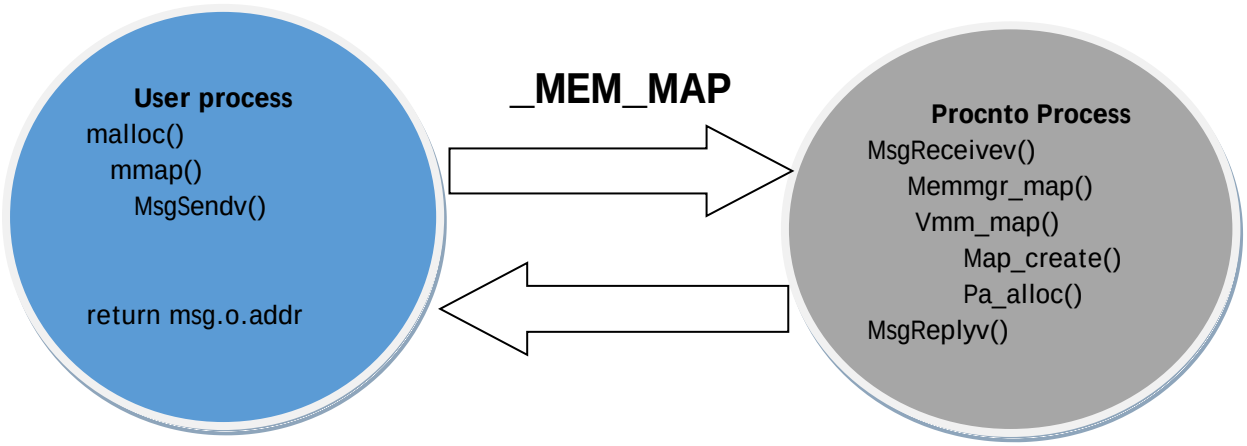
3.3.1 LIBC Implementation

To implement message handling, the libc library converts POSIX function calls into message handling functions internally. If this did not occur it would increase program development time significantly and introduce the complexity of writing programs on this architecture. This approach allows POSIX code to be ported to QNX with a minimal amount of modifications by the developer.

An example of this is as follows:

```
mmap(void *addr, size_t len, int prot, int flags, int fd, off_t off)
{
..
       MsgSend(..);
}
```

The mmap function creates a message structure, initialises the message structure and calls the messaging kernel functions to deliver the message. This can be visualised as follows:



Whilst this is conceptually how the low level message passing works within QNX a more abstracted API is provided (using POSIX calls) to make this more developer friendly. More information about QNX messaging can be found in the following section on how this is implemented.

4. QNX Key OS Components

In order to understand QNX for exploitation, it is necessary to determine what the key components of the OS are and their purposes. Attempts were made to examine the attack surface of each of these components to determine if it could be used to escalate privileges on the device.

4.1 QNX Process Manager

The first of these is the process manager and is one of the most critical processes on QNX. The process manager is in charge of creating new processes for the system. Spawn, fork, exec functions are implemented within the process manager. The process manager receives messages from other processes to spawn new processes, terminate processes and handle the processes lifetime. In QNX the process manager is typically referred to as “procmgr” as the component name.

The process manager is included into the procnto process (combined with the microkernel code itself) and has the following attributes:

- First process on the system and part of the microkernel itself.
- Has permission to use _ring0 kernel call (_NTO_PF_RING0 flag) and runs under the root user.

Therefore from a security perspective, any compromise of the process manager could lead to ring 0 code execution.

The header file uses to form messages for delivery to the process manager is as follows:

/bbndk/target_<version>/qnx6/usr/include/sys/procmsg.h

This header file is included within the Blackberry 10 SDK and can be used to determine the structure of messages sent to the process manager.

An example of sending a message to the process manager is as follows:

```
proc_fork_t                             msg;
pid_t                                   pid;
uintptr_t frame;

msg.i.type = _PROC_FORK;
msg.i.zero = 0;
msg.i.flags = 0;                // _FORK_ASPACE
msg.i.frame = 0;

MsgSendnc(PROCMGR_COID, &msg.i, sizeof msg.i, 0, 0);
```

PROCMGR_COID and PATHMGR_COID are hardcoded channel IDs. This is used to bootstrap a processes communication with the operating system. This shows a client program sending a fork command to the process manager.

4.1.1 Process Manager Abilities

One feature of QNX 6.6 is the ability to restrict what operations a process can perform. This feature is implemented using a feature called process manager abilities. This allows a process running as root to restrict its capabilities to only that in which it needs. This concept is similar to Linux's capabilities implementation. For example the following capabilities can be used with the procmgr_ability call:

- PROCMGR_AID_SPAWN_SETUID – Allows setting of the lower and upper bounds of user ids that a process can set the child process to.
- PROCMGR_AID_IO – Prevents the process from requesting I/O privileges

More information can be found about process manager abilities at the following location:

http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.lib_ref%2Ftopic%2Fp%2Fprocmgr_ability.html

4.1.2 Process Manager Attack Surface

As an attacker the process manager is an interesting target due to the highly privileged access it has within the QNX system.

The process manager exposes the following main attack surface:

- Message Handling (procmgr)
- Parsing of ELF binaries (the ELF loader code is located here)

As part of Blackberry 10 a new API has been added to further restrict access and limit the ability in what activities a process can perform. More information can be found in the API documents at:

https://developer.blackberry.com/native/reference/playbook/com.qnx.doc.neutrino.lib_ref/topic/p/procmgr_ability.html

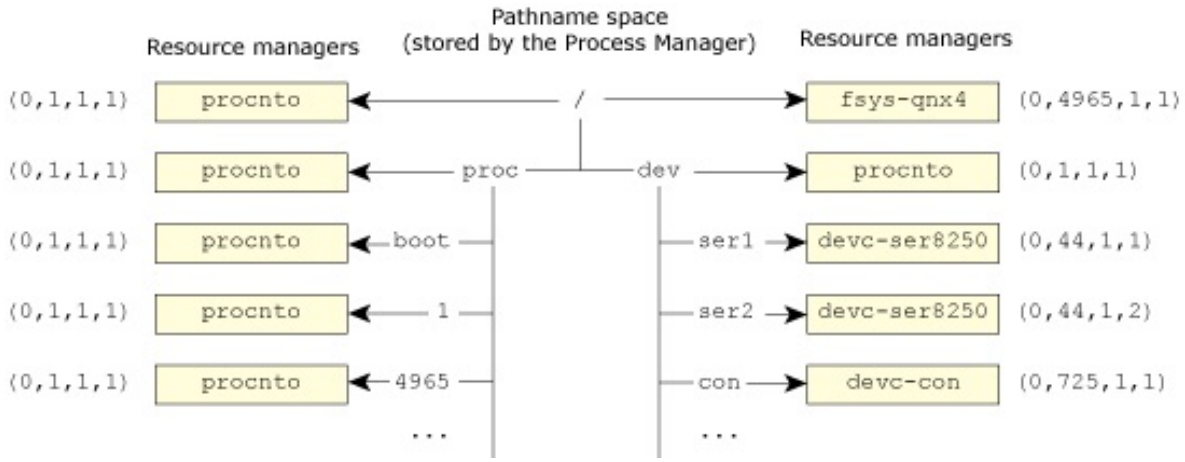
More information can be found about the process manager at the following location:
http://www.qnx.org.uk/developers/docs/6.3.0SP3/neutrino/sys_arch/proc.html

4.2 QNX Path Manager

The path manager owns the entire namespace. Within QNX the namespace is represented logically as paths within the system, however, these are backed by resource managers at these locations. The path manager is one of the most novel QNX concepts coming from a traditional *NIX background. The best way to understand how the path managers is implemented is in the form of an example. The path manager is in case of maintaining a prefix tree which maps paths within the path namespace to the associated node id, process id, channel ids.

For example, when a call is made to open(), it's the path manager that compares the path against the prefix tree to determine which resource manager the open message should be directed to.

This can be conceptually visualised as follows:



Pathname resolution is well documented in the following location:

http://community.qnx.com/sf/docman/do/downloadDocument/projects.core_os/docman.root.articles/doc1166/1

The implementation of path manager messages can be found in the following location:

```
\target_<version>\qnx6\usr\include\sys\pathmsg.h
```

An example of sending a path manager message is as follows:

```
fd = _connect(PATHMGR_COID, path, 0, O_CREAT | O_EXCL | O_NOCTTY, SH_DENYNO, subtype,
testcancel, 0, filetype,0, 0,0, 0, 0, 0);
```

This is making use of the _connect API function to pass a message to the path manager. As mentioned above the PATHMGR_COID is a hardcoded channel id used to bootstrap processes to locate the path manager channel.

4.2.1 Path Manager Attack Surface

The path manager is interesting due to the ability to map system calls to resource managers. The attack surface of the path manager is typically a conduit to delivering messages to the appropriate resource manager (process). However, the path manager itself can be attacked in the following ways:

- Path / Namespace Squatting
- Path Name fuzzing
- Device file paths (IOCTL fuzzing)

4.3 QNX Memory Manager

The memory manager on QNX is implemented to handle virtual and physical memory.

The messages format for the memory manager can be found in the following header:

/target_<version>/qnx6/usr/include/sys/memmsg.h

The memory manager in an OS is responsible for the memory management functionality. QNX implements a memory manager process to handle virtual, physical and architectural specific features.

Memory management is a critical function of the operating system and is important to maintain the stability and safety of processes executing on the system. The memory management is designed in a typical way such that any process crash is limited to the process itself and that memory from one process is segment from another process on the system.

This is implemented using typical virtual memory segmentation, where each address within a program is a virtual address. These virtual addresses are then mapped by the memory manager to actual physical memory locations.

A number of additional memory protections such as ASLR have been implemented in BB10 OS starting from QNX 6.6.

More information can be found on the QNX wiki:

http://community.qnx.com/sf/wiki/do/viewPage/projects.core_os/wiki/Memmgr_source_guide

4.3.1 Memory Manager Attack Surface

Like the process and the path manager the memory manager itself has a number of exposed attacks surfaces which make for an interesting place to audit, these are as follows:

+ Memory Mapping Message Call Handlers
+ Memory Segmentation Protections

4.4 QNX Resource Managers

Another core concept within QNX architecture is that of a resource manager. QNX resource managers are essentially processes which provide a level of abstraction of some service. The pathname is used as their domain of authority as described in the path manager section above. A resource manager typically uses the resmgr_attach (http://www.qnx.com/developers/docs/6.4.1/neutrino/lib_ref/r/resmgr_attach.html) function to register itself within the path namespace. For example:

```
if ( (id = resmgr_attach
          ( dpp, &resmgr_attr, "/somepath", _FTYPE_ANY, 0,
          &connect_funcs, &io_funcs, &attr)) == -1 ) {
        fprintf( stderr, "%s: Unable to attach name.\n", \
                 argv[0] );
        return EXIT_FAILURE;
    }
```

This attaches the resource manager to the /somepath within the path namespace. Therefore in order for a client process to communicate with the resource manager it can then call open on /somepath to obtain a valid file descriptor for communication with the resource manager providing the path. The path name manager contains a mapping of pathnames to nodes id, PID and Channel id. Typically the first parameter node id is used by a QNX when configured in a distributed architecture. Qnet is the networking protocol which is used to communicate between nodes in when configured in a distributed architecture. On Blackberry 10 the node id is not used and is therefore always 0 because there is no Qnet support within BB10 OS.

So for example the following code could be used:

```
fd = open ("/somepath", O_WRONLY);
```

Once the path name manager has received the message and determined within the prefix tree if there is a match then the ND, PID and Channel ID is provided back to the client code (in open()). The open() function then creates another message which is then destined to the ND,PID and Channel ID returned initially. The resource manager then gets the connect message and performs its validation to make sure the caller is allowed to communicate with it. The client's *open()* then returns to the client with a valid file descriptor and this can be used for subsequent operations on the resource manager.

Within Blackberry 10 a number of services are implemented as resource managers and are executing with varying privilege levels. A concrete example of a resource manager is the networking stack which handles socket communication.

More information can be found about implementing resource managers at the following locations:

http://www.qnx.org.uk/developers/docs/6.4.1/neutrino/getting_started/s1_resmgr.html

http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/prog/resmgr.html#RESMGR_COMPONENTS

4.4.1 Resource Manager Attack Surface

A resource manager itself exposes a significant amount of attack surface. For example, when the path manager has performed and determined a valid file description, this file description can then be used to communicate with the resource manager. As mentioned above resource managers can register custom IO handler functions using the iofunc_func_init method. The code below demonstrates the registration of a read system call handler:

```
int main (int argc, char **argv)
{
...
    iofunc_func_init (_RESMGR_CONNECT_NFUNCS, &my_connect_functions,
                      _RESMGR_IO_NFUNCS, &my_io_functions);
    my_io_functions.io_read = my_io_read;
...
}
```

The my_io_read function will be called when a read() call is made on a file description provided by the resource manager. Therefore it is important that all functions registered by resource managers are suitably robust code. An attacker who is able to send malformed data to a function registered by a resource manager may be able to achieve code execution within that resource manager's context if sufficient validation is not being performed.

4.5 QNX Processing Listing

On a Blackberry 10 device the devuser is prevented from determining what UID/GID an executable is running under (using pidin) unless it is running as the same UID as devser.

However, it is possible to determine process permissions on a rooted simulator but these permissions may not necessarily be the same on a real device (however in the majority of instances this will be the same).

QNX uses the same concept as Unix where every user must be member of at least one group. This is called the primary group which the user is assigned. A user can be then be a member of additional groups called supplementary groups.

There are two additional ways to determine what privilege a program is running under:

1) By examining the unpacked firmware image and the scripts which are used to start the process.

An example of this is as follows (taken from startup.sh):

```
CMN_ON ${VAR_PLAT_APP_LIBC_STRINGS} -d -p 9 -u
203:203,0,200,320,1000,1002,1004,1200,1202,1204,1005,1007,1008,1205,750,751,753,1207,1208
${BASEFS}/usr/bin/emalauncher > /dev/null 2>&1
```

This demonstrates the ON utility called 'CMN_on' in this example (http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.utilities%2Ftopic%2Fo%2Fon.html) being used to launch the /usr/bin/emalauncher with the user and groups specified.

2) Exploiting an information leak with the sysctl handler (sysctl -a) to display the UID/GIDs of processes. The output of this command shows the UID/GID of processes which have performed network communication on the device and is not limited to only the devuser.

```
qnx.kern.nws.table =      UID      GID      PID NAME          INTERFACE        SENT          RECEI
VED
         0          0  15654949 sshd              tiw_sta0          70/       1
147/       1
        43          0  15642719 ping              lo0               28/       1
28/        1
        43          0  15642719 ping              tiw_sta0         168/       2
168/       2
        42          0   6729845 cpdd              tiw_sta0         404/       6
1344/       5
       202        200   5734476 python3.2         lo0              494/       6
392/       4
       810        810  12730609 app_process       lo0              508/       6
558/       7
        22          0  14938363 nmbd              tiw_sta0           0/       0
156/       2
         0          0   5296205 inetd             ncm0            7980/      35
5562/      29
       205        205  14528607 bozohttpd         tiw_sta0         280/       4
588/       4
         0          0   8556706 mdnsd             bptp0              0/       0
```

There are limitations to this information leak due to only being processes which perform network communication, however, providers a little more visibility when performing security assessment.

4.6 QNX Persistent Publish Subscribe Architecture

QNX Persistent Publish Subscribe (PPS) is a service which offers a way for processes to publish or subscribe to. PPS uses an object based system to implement this functionality. Clients can subscribe for update to the object and receive notifications when the object changes (i.e. a publisher change it). Typically all that is required to publish to an object all that is needed is to call open() on the path and then write() to perform the update. Subscribers on the other hand can call open() and then read() to query the object.

There are lots of small details associated with PPS and the documentation is fairly conclusive on this: http://www.qnx.org.uk/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_pps%2Fpps.html

However, to communicate with a PPS endpoint typically JSON is crafted. For example, the following displays a message in the background of the application.

```
echo "msg::lockDevice\ndat::Backup Interrupted at $(date)" >> /pps/system/navigator/background
```

By using echo to write to this PPS endpoint, the PPS resource manager's write function will be called internally. This will then notify all the subscribers that an update has occurred and the subscribers will be able to act on this. One important thing to note is that PPS messages allow parameter passing which needs to be handled by the subscriber. Therefore it is important that all data passed into a PPS message is sufficiently validated.

One previous issue identified was with the use of the special file .all by Zach Lanier and Ben Nell. Whilst the file system and ACLs lock down access to the PPS objects themselves, the .all aggregates contents of files which would have not been readable due to restricted file permissions.

More information can be found in: https://speakerdeck.com/quine/voight-kampffing-the-blackberry-playbook-v2

4.7 QNX Firmware Analysis

In order to perform static reversing of the Blackberry 10 firmware it is first necessary to acquire it for ARM architecture. Sachesi tool has been created to aid this process and allows both the download of firmware updates and the ability to unpack firmware images into their relevant components. Using the advanced option it is possible to extract both the QNX6 file system partition and the boot image file system (ifs) for security research. Sachesi can be obtained from:

https://github.com/xsacha/Sachesi

Once the firmware has been extracted into its relevant partitions it is possible to access the files in these partitions either by mounting the image (QNX6) or using dumpifs tool to extract the files from the image

filesystem. Intrepidus group created a tool called ifs_parse.py which can automate this extraction process:

https://github.com/intrepidusgroup/pbtools

More information can be found at the following URLs:

https://github.com/alexplaskett/QNXSecurity/tree/master/FWAnalysis

http://www.qnx.com/developers/docs/6.3.2/neutrino/utilities/d/dumpifs.html

http://www.qnx.com/developers/docs/6.3.2/neutrino/utilities/m/mkifs.html

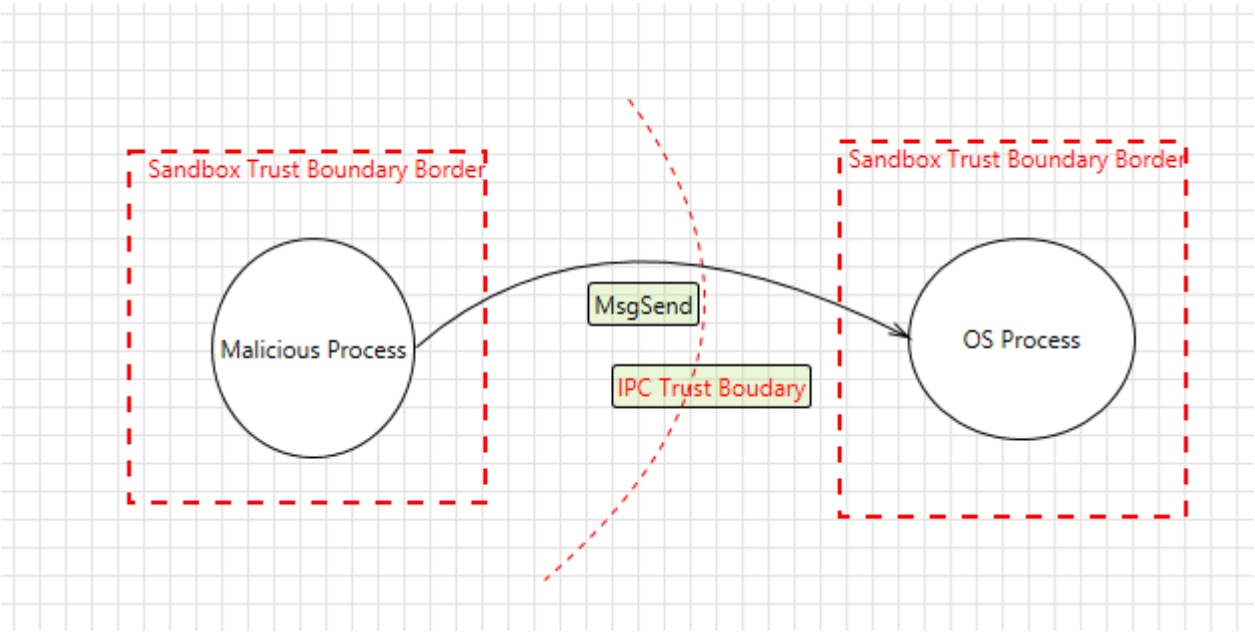
4.8 QNX Overview Summary

Now that an understanding has been gained of how the QNX architecture is composed it is then possible to examine the ways in which an attacker could attempt to circumvent any security controls implemented within the architecture.

5. Attacking QNX Messaging

This section will build on the understanding of the QNX architecture and determine how an attacker would go about attempting to elevate their access.

As mentioned previously QNX messages flow across a trust boundary and can be received by processes with higher privileges. If a malicious process is able to deliver messages which are handled by a legitimate process then it may be possible to perform malicious actions. Conceptually this can be pictures as follows:



On Blackberry 10 the applications are running at multiple different privilege levels, therefore attacks can be made from a low privilege process to a higher privilege process.

5.1 Obtaining a side channel connection ID

In order for applications to be able to look up the connection ID of a channel the name_open function can be used.

http://www.qnx.com/developers/docs/6.3.2/neutrino/lib_ref/n/name_open.html

```
int name_open( const char * name, int flags );
```

In order to obtain names then the application must either know the name passed to the name_attach function http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/lib_ref/n/name_attach.html or enumerate them from the /dev/name/local/ namespace.

On Blackberry 10 the following names are attached by default:

```
nrw-rw-rw-   1 root      nto             0 May 31 14:06 MercuryComponent    <- Qualcomm JPEG
stuff
nrw-rw-rw-   1 root      nto             0 May 31 14:06 QMUX_SERVER    <- Qualcomm Radio
libqm*
nrw-rw-rw-   1 root      nto             0 May 31 14:06 VideoCore
nrw-rw-rw-   1 root      nto             0 May 31 14:06 _tracelog        <- libtracelog.so.1
nrw-rw-rw-   1 root      nto             0 May 31 14:06 battmgr
       <- battmgr
nrw-rw-rw-   1 root      nto             0 May 31 14:06 bide             <- bide-msm8x60
dr-xr-xr-x   2 root      nto             0 May 31 14:06 csm
nrw-rw-rw-   1 root      nto             0 May 31 14:06 dashboard_channel  <- dashinfod
nrw-rw-rw-   1 root      nto             0 May 31 14:06 dsi_server_primary
dr-xr-xr-x   2 root      nto             0 May 31 14:06 gltracelogger        <- libegl.so /
libwfd.so
nrw-rw-rw-   1 root      nto             0 May 31 14:06 io-asr-bb10          <- io-asr-bb10
(speech recognision)
nrw-rw-rw-   1 root      nto             0 May 31 14:06 led_control         <- led_control
nrw-rw-rw-   1 root      nto             0 May 31 14:06 mmcore
nrw-rw-rw-   1 root      nto             0 May 31 14:06 mtouch_ctl                <- mtouch
nrw-rw-rw-   1 root      nto             0 May 31 14:06 muxClient_13664519        <- rimusb
nrw-rw-rw-   1 root      nto             0 May 31 14:06 muxClient_5980248         <- rimusb
nrw-rw-rw-   1 root      nto             0 May 31 14:06 persist_mgr               <- QNX
persist_mgr
nrw-rw-rw-   1 root      nto             0 May 31 14:06 phone-service             <- phone-
service
nrw-rw-rw-   1 root      nto             0 May 31 14:06 powerauth                 <-
/proc/boot/powerauth
nrw-rw-rw-   1 root      nto             0 May 31 14:06 publisher_channel         <- device-
publisherd
nrw-rw-rw-   1 root      nto             0 May 31 14:06 qpmic-intsvr              <- qpmic-
intsvr
nrw-rw-rw-   1 root      nto             0 May 31 14:06 rim_usb
       <- rimusb
nrw-rw-rw-   1 root      nto             0 May 31 14:06 slogger2
nrw-rw-rw-   1 root      nto             0 May 31 14:06 usbmgr                    <-
/proc/boot/usbmgr (no perms)
nrw-rw-rw-   1 root      nto             0 May 31 14:06 vpeCore
```

These names can then be mapped back to their associated binary in which the name_attach registration occurs.

Unfortunately, there is no easy way to identify the binary which registers the name, however, the approach taken was to identify the binary from the firmware image which use name_attach and then cross reference them with process listing and file system listing.

On a Blackberry 10 device it is not possible to list the names of running processes with their PIDs, however, there are a number of information leaks which can help correlate this information. For example: sysctl –a displays networked process id's.

5.1.1 /proc/mount

It is also possible to obtain the information necessary to communicate with another process from the /proc/mount directory. This directory is a hidden directory, however, it is possible to cd into it or list the information.

```
$ ls -al /proc/mount (Simulator)
ls: No such file or directory (/proc/mount/0,1,1,2,-1)
total 25
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 0,1,1,18,11 <- procnto-smp-instr
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 0,1,1,19,8
drwxr-xr-x   2 root      nto              10 Jun 12  2014 0,1,1,21,0
dr-xr-xr-x   2 root      nto               0 Jul 22 08:39 0,1,1,3,-1
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 0,1007641,4,0,11 <- IO-HID
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 0,1196065,1,0,11 <- IO-AUDIO
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 0,1310756,2,8,11 <- io-pkt-v6-hc
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 0,61450,4,0,11 <- devb-eide
drwxr-xr-x  10 root      nto            4096 Jul 22 08:29 0,61450,4,6,0
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 0,8195,1,1,4 <- pipe
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 accounts
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 apps
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 base
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 dev
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 enterprise
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 pps
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 proc
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 sys
dr-xr-xr-x   2 root      nto               1 Jul 22 08:39 var
```

These tuples show above in the file system listing are as follows:

3) Node ID
4) Process ID
5) Channel ID
6) Handle
7) File Type

The difficulty here is establishing the process which has created the channel and constructing messages using the correct format.

5.2 Reverse Engineering Message Handlers

Once the process has been identified which uses name_attach() then it is necessary to locate the message handling functionality.

Typically the pattern to look for is as follows:

```
if ((attach = name_attach(NULL, ATTACH_POINT, 0)) == NULL) {
        return EXIT_FAILURE;
    }

    /* Do your MsgReceive's here now with the chid */
    while (1) {
        rcvid = MsgReceive(attach->chid, &msg, sizeof(msg), NULL);
```

However, message_attach is commonly used to register a message handler function and allows messages of different types to be handled by different code. (http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/lib_ref/m/message_attach.html)

As a practical example the phone-service process can be examined. As you can see below firstly it registers the name phone-service which in /dev/name/local/ path.

```
.text:0002BC58 loc_2BC58                               ; CODE XREF: sub_2BBB8+82□j
.text:0002BC58                 LDR.W           R1, =(aPhoneService - 0x2BC62)
.text:0002BC5C                 MOVS            R2, #0
.text:0002BC5E                 ADD             R1, PC  ; "phone-service"
.text:0002BC60                 BLX             name_attach
```

Once this occurs the then message_attach is used as follows to register the message handler:

```
.text:0002BC6C                 BLX             memset
.text:0002BC70                 MOVS            R3, #1
.text:0002BC72                 STR             R3, [SP,#0x50+var_38]
.text:0002BC74                 MOV.W           R3, #0x1000                     ; low
.text:0002BC78                 STR             R3, [SP,#0x50+var_34]
.text:0002BC7A                 MOV.W           R2, #0x8000                     ; high
.text:0002BC7E                 LDR             R3, =(sub_2B930+1 - 0x2BC88)
.text:0002BC80                 MOV             R0, R5
.text:0002BC82                 ADD             R1, SP, #0x50+s
.text:0002BC84                 ADD             R3, PC ; sub_2B930              ; func
.text:0002BC86                 STMEA.W         SP, {R3,R4}
.text:0002BC8A                 MOV             R3, R2
.text:0002BC8C                 BLX             message_attach
```

The function sub_2B930 in this case is the message handling function which is called when the process receives a message and it is within the given range. Message_attach therefore can be used multiple times to register different message handlers which are called based on the range of messages which the handler is interested in. In the above case the message handler sub_2B930 will be called when messages are within the range 0x1000 to 0x8000.

Therefore in order to communicate with this from a client perspective, the following code can be used:

```
typedef struct
{
       uint16_t msg_no;
       char msg_data[255];
} client_msg_t;

client_msg_t msg;
memset( &msg, 0, sizeof( msg ) );
msg.msg_no = 0x8000
snprintf( msg.msg_data, 254, "client %d requesting reply.", getpid() );
```

An example of this is as follows:

http://support7.qnx.com/download/download/9881/client.c

ConnectClientInfo (http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/lib_ref/c/connectclientinfo.html) can be used to validate that the calling process has the appropriate UID/GID to be allowed to communicate with the channel.

This can be performed by examining the cred struct UID and GID obtained http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/lib_ref/c/connectclientinfo.html#InfoStructure

5.3 Fuzzing Message Handlers

The code used to fuzz IPC message handlers is available at: https://github.com/alexplaskett/QNXSecurity/tree/master/IPCFuzz

5.3.1 Dumb Fuzzing

Initially a fuzzer was constructed which generated malformed data without attempting to craft messages within the correct format for the handlers. This identified a number of issues and lead to a significant process crashes and device reboots by triggering these crashes. The primary issue with this is that due to targeting higher privilege process the only way to determine why the crash occurred was using static reverse engineering. Debugging of the higher privileged process however could not be performed. More information can be found in the debugging section following which describes the issues with debugging on QNX.

Using a dumb fuzzing method on MsgSend() it was possible to trigger a number of crashes within the IPC endpoints exposed from the phone. In a number of cases this lead to a full reboot of the phone, therefore it was determined that this fuzzing process located a number of issues and demonstrated that it may be a profitable vector for an attacker. However, it should be noted that a reboot of the phone does not necessarily mean an exploitation condition and may be due to a watchdog process executing.

5.3.2 Smarter Fuzzing

The message fuzzer was then improved to ensure that the messages sent by the client were the expected length for the receiver and that the structure was close to what was expected. This lead to an expected increased code coverage and more issues were identified.

5.3.3 In-Line Fuzzing and Logging

Scripts were written to man in the middle the MsgSend function and mutate the data which is being sent to the receiver. Unfortunately, this can only be used from processes launched from the devuser, therefore using this method it was not possible to gain significant code coverage.

5.3.4 Endpoint Blocking

A number of IPC endpoints were found to block the MsgSend(), therefore it was necessary to create a blacklist of these endpoints to prevent the blocking of the fuzzer.

5.3.5 Determining if a process crash had occurred

Whist on the simulator it is possible to perform debugging and access the crash dumps, on a Blackberry 10 device it is not possible to do this (without a vulnerability). Therefore, the primary way to perform crash monitoring on the device was to check to see if the pathname was still accessible. If a process crash has occurred in the process which calls name_attach() then the endpoint would no longer to present, demonstrating that a crash had occurred.

5.4 Message Handling Weaknesses

As mentioned previously, a number of issues were identified through this fuzzing process demonstrating that the reliability of these processes was not of a sufficiently high standard. By performing targeted fuzzing on an endpoint, it was typically not very long before a crash would occur or the device would reboot due to the watchdog process. However, this does not indicate that the issue is an exploitable condition, for example the watchdog process can reboot the phone if it gets too hot to prevent damage to the hardware.

Static reverse engineering was needed to determine the class of bug due to the lack of debug visibility on a real Blackberry 10 device.

6. Attacking QNX PPS

The first step in attacking the PPS subsystem was to map out the attack surface exposed to the user or an application. This was achieved using the following process:

6.1 Identifying writable PPS endpoints

In order to communicate with a PPS endpoint it is necessary to have write access to the endpoint. The following script can be used to determine which PPS endpoints provide write access (from the account running the script):

```
dir="/pps"
files="$(/usr/bin/find "$dir" -type f)"
echo "Count: $(echo -n "$files" | /usr/bin/wc -l)"
echo "$files" | while read file; do
       if [ -w "$file" ]
       then
              echo "Write permission is granted on $file"
       fi
done
```

A script was produced to automate some of the initial device review tasks that an assessor would typically need to perform when a new build of the operating system is produced or assessing a different QNX device. The code for this script can be found at the following location:

https://github.com/alexplaskett/QNXSecurity/blob/master/devrev.sh

6.2 Reverse Engineering PPS Messages

Once the PPS endpoint has been identified then it is necessary to determine which process creates the PPS endpoint and the message format it expects.

Typically PPS messages will either be created by scripting languages or native programs. In the case of a scripting language the following code can typically be found within the shell scripts:

```
       echo "msg::lockDevice\ndat::Backup $1 $process_count of $total" \
              >> /pps/system/navigator/background
```

By performing a grep across all shell scripts it was possible to identify a number of messages crafted in this way. However, not all of the attack surface can be identified using this technique and it is necessary to examine the binaries in which handle the PPS messages.

A service was identified which handled PPS messages to perform certain UI actions. This service is called the Navigator process and it is used to control how applications appear on the device. For example, if a swipe event occurs the Navigator process handles this event. More information on the Navigator process can be found at the following URL:

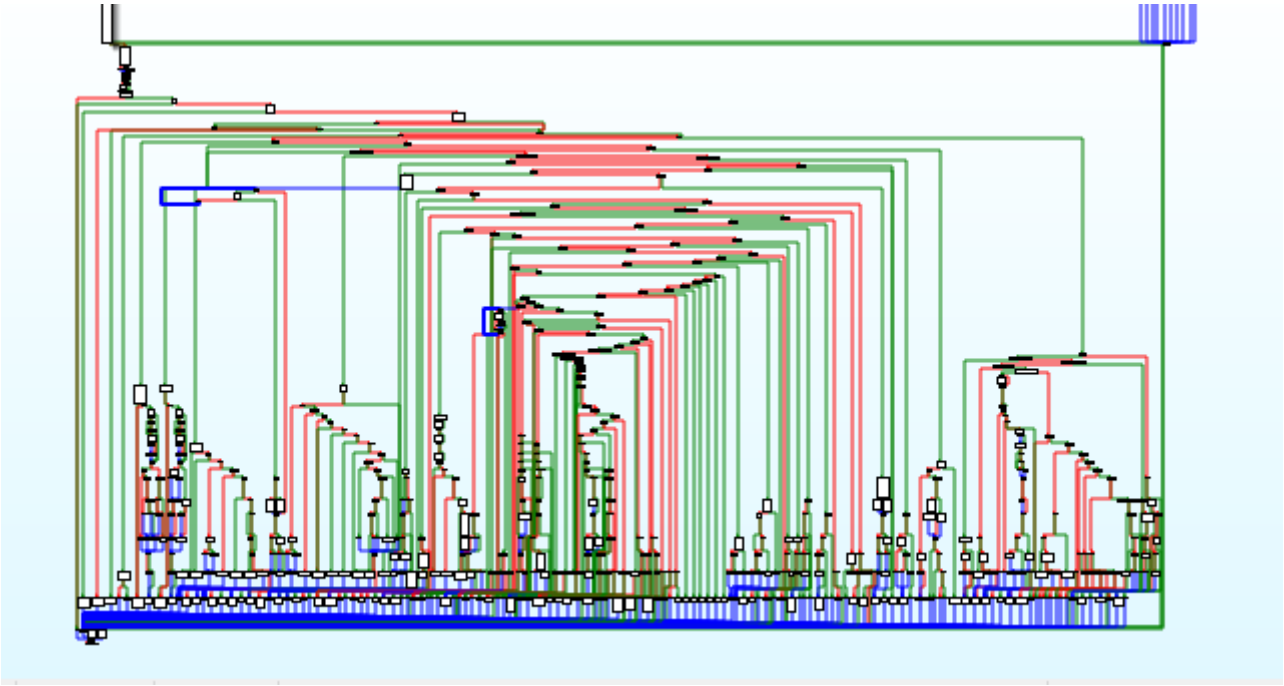
https://developer.blackberry.com/native/reference/core/com.qnx.doc.bps.lib_ref/topic/about_navigator_8h.html

As a concrete example we will identify the correct message format for the Navigator service. If the PPS command is known then a cross-reference can be performed from the string name to determine the function handler.

For example, using the known launchApp string it was possible to determine the message handling function by searching for the string and then looking at the call graph.

```
.text:000F6E00                LDR             R1, =(aLaunchapp - 0xF6E08)
.text:000F6E02                MOV             R0, R9  ; s1
.text:000F6E04                ADD             R1, PC  ; "launchApp"
.text:000F6E06                BLX             strcmp
```

From this it was possible to determine that the complexed function was used as a switch statement to handle the incoming commands.



Using this it was then possible to reverse out all commands that the Navigator service supported.

6.3 PPS Fuzzing

After the endpoints which were writable were identified, then these endpoints were examined to determine any vectors which could be used to escalate privileges. Fuzzing was also performed to locate any memory corruptions which could help in this endeavour.

As the PPS messages are essentially JSON objects, then the libraries which were responsible for handling the parsing of these JSON messages were subject to fuzzing to identify weaknesses.

The code used for fuzzing the PPS endpoints can be found at the following location:

https://github.com/alexplaskett/QNXSecurity/tree/master/PPSFuzz

6.4 PPS Weaknesses

6.4.1 PPS Permissions and Logic Issues

The PPS object (/pps/system/navigator) has the following permissions when

```
-rw-rw----+  1 apps      nto             12 Jun 27 14:20 /pps/system/navigator/control
```

The devuser is not able to interact with this endpoint due to not being part of the apps group. However, a Blackberry 10 Android application is able to interact directly with this endpoint (and the Blackberry Android shell user). This allows certain functions to be triggered (a sample is provided here):

1) terminateAll
2) launchApp
3) launchBkgdApp
4) launchRemoteApp
5) launchService
6) debugApp
7) terminateDname
8) termimateAll
9) isAppRunning
10) openFile

6.4.2 Dangerous Functionality (Logic Issues)

As mentioned above dangerous functionality can be exposed through PPS messages (such as the <redacted> function) which can be triggered as follows:

```
echo "<redacted>" > /pps/system/navigator/control
```

This demonstrates that one application on Blackberry 10 can perform a denial of service against the device and prevent other applications on the device from executing. This issue is not currently patched therefore for more information please see the advisory document in future with the issue details.

6.4.3 Memory Corruption Issues

Memory corruption issues can also be triggered through the use of malformed PPS messages. An example of a malformed message is as follows:

```
echo "msg::launchService\ndat::sys.firstlaunch.gYABgE1L_lY.sjW85E1SCBQsrco,personal,dev_mode"
> /pps/system/navigator/control
echo "msg::prepareForDataLock\ndat::/var/test.sh" > /pps/system/navigator/control
```

Whilst these two commands just trigger null pointer exceptions and are therefore not exploitable, this demonstrates that care needs to be taken when validating PPS messages.

A list of common PPS commands was created and radamsa (https://github.com/aoh/radamsa) used to generate mutations for these commands.

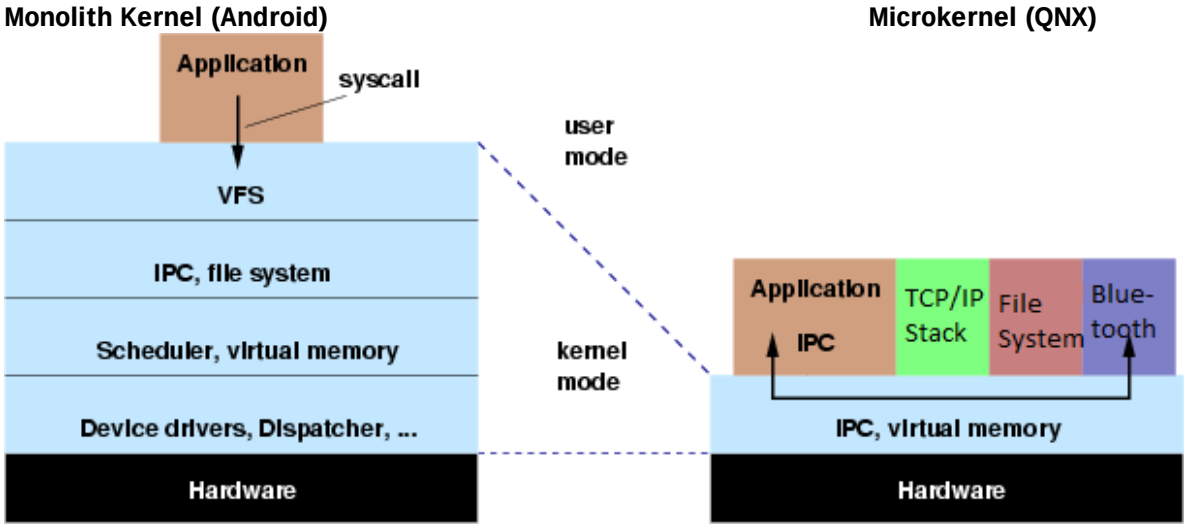
7. QNX Kernel Architecture

7.1 Kernel Introduction

The Microkernel on BB10 OS is implemented in the procnto-smp-instr binary in /proc/boot/.

Due to the Microkernel architecture there is only a small number of system call defined, thus reducing the attack surface significantly in the kernel itself. However, it should be noted that due to messages being routed to other processes, there is the potential for messaging based attacks on other processes.

In BB10 at the time of writing there are currently 106 system calls defined, whilst Linux has 338+. This significantly reduces the attack surface of the microkernel itself.



Taken from: https://upload.wikimedia.org/wikipedia/commons/thumb/6/67/OS-structure.svg/450px-OS-structure.svg.png

The interface to the kernel is defined in neutrino.h (part of the Blackberry 10 Native SDK) in the following location (\target_10_1_0_1020\qnx6\usr\include\sys).

Kercalls.h also contains the system call's number in if the libc abstraction layer (libc.so) is not required. The libc layer does not perform any validation before making the actual transition into kernel more, therefore this can be used directly for testing purposes.

For example the following code is used to make a ChannelCreate syscall:

```
LOAD:00042F38                 EXPORT ChannelCreate
LOAD:00042F38 ChannelCreate                           ; CODE XREF: j_MsgPause_r+8□j
LOAD:00042F38                                         ; j_ld_imposter_exit+8□j ...
LOAD:00042F38                 STMFD           SP!, {LR}
LOAD:00042F3C                 MOV             R12, #0x23 ; '#'
LOAD:00042F40                 SVC             0x51 ; 'Q'
LOAD:00042F44                 LDMFD           SP!, {PC}
```

Which as you can see from the header file kercalls.h matches:

```
__KER_CHANNEL_CREATE,                   /* 35        0x23 */
```

7.2 Kernel Research

The kernel research was performed from both a static reverse engineering perspective and a dynamic debugging perspective. More information on the methods used to perform kernel debugging are included within the following section on debugging. However, the techniques used to build and modify existing QNX images are documented here.

7.2.1 Booting an existing IFS image

In order to boot an existing IFS image QEMU can be used. For example an IFS image build for a Toyota Infotainment system (Ifs-extbox.bin) can be extracted from a firmware update.

Once this has been performed it is then possible to create a NAND image which combines the IFS and bootloaders. The following steps can be used to perform this process:

```
mknand.sh car.img ifs-extbox.bin
nand_ecc car.img 0x0 0xe80000
qemu-system-arm –M beagle –m 256 –mtdblock car.img –nographic
nand read 0x80100000 0x280000 0x400000; go 0x8010000
```

This allows booting of the IFS image as can be seen by the output below:

```
== Toyota Extension Box Boot IFS ==
== Variant:EU-Low   HWRev:Gamma   Arch:armle-v7 ==
== built by DMiller on Wed Dec  7 19:45:38 2011 ==
```

At this stage it has not been possible to boot a Blackberry 10 IFS image on QEMU due to architectural support. It is understood that bootloader modifications and platform support may need to be added to QEMU so is expected to be a non-trivial task. The next stages of this research will attempt this and document the progress made.

7.2.2 Rebuilding an IFS image

The tools necessary to rebuild an IFS image are part of the QNX VMWare image (mkifs and dumpifs). From firmware extraction we have all the necessary binaries to rebuild the IFS image. It is possible to create a .build script which contains references to the BSP startup, kernel and shell binaries to create a minimal bootable IFS image.

The following command can be used to build a QNX IFS from a .build script:

mkifs -r stage -vv custom.build qnx.ifs

After this has been performed it is necessary to append the bootloaders and create the NAND image for booting. The mknand.sh script mentioned above can be used to automate this process.

Binaries build using the BB10 toolchain can then be referenced within the .build script and will be integrated into the IFS image. This custom image can then be booted using QEMU as shown above.

7.3 Kernel Attack Surface

In order to perform more kernel security research it was necessary to understand the attack surface exposed to a user land program.

Two main entry points are of interest to an attacker are:

1) ARM Exception Vector Table

Within the old kernel sources the ARM exception vector table can be located within the Kernel.S source code file (http://sourceforge.net/p/monartis/openqnx/ci/master/tree/trunk/services/system/ker/arm/kernel.S). An audit of the ARM vector table handling was performed to identify any weaknesses which could be present in this area.

This consisted of looking at the following exception handlers:

- Fast Interrupt Request Handler
- Interrupt Request Handler
- Software Interrupt and Reset Handler
- Prefetch and Data Abort
- Undefined instruction Handler

2) Syscall Entry Table

The system call table for procnto ARM can be found in the appendix section of this document. A system call fuzzer was written to test the syscall interface.

The code for the QNX syscall fuzzer can be found at the following location:

https://github.com/alexplaskett/QNXSecurity/tree/master/SyscallFuzz

7.4 Kernel Weaknesses

At this stage there are a number of BB10 kernel weaknesses which are still being investigated or with the vendor, this information will be published on labs at a later stage. Therefore it is not possible to include information on these issues.

However, which examining old source code for QNX 6.4 an arbitrary kernel read and write primitive was identified. This bug was found to not affect Blackberry 10 kernel and therefore is of limited use. However, may be useful when targeting older unpatched QNX systems. No information of this vulnerability exists online and it is not tracked in any vulnerability databases.

7.4.1 __emu_cmpxchg ARM Vulnerability

```
__ker_entry:
```

```
/*
* First check for "special" calls:
*
* 0xff000000 - smp_cmpxchg emulation
* 0xfe000000 - ClockCycles emulation
*/
tst ip, #0xff000000
bne __emu_decode

__emu_decode:
teq ip, #0xff000000
beq __emu_cmpxchg
teq ip, #0xfe000000
bhs __emu_clockcycles
b __ker_entry_save


/*
 * ------------------------------------------------------------------------
 * Emulate the x86 cmpxchg to support libc sync primitives.
 * On entry, interrupts are disabled:
 * r0   - ptr to word being tested
 * r1   - value to compare
 * r2   - value to set
 * lr   - saved return pc
 * spsr - saved cpsr
 *
 * On return:
 * r0 - original value in dest
 * r1 - preserved
 * r2 - preserved
 * ip - trashed
 * Z - set if successful
 * ------------------------------------------------------------------------
 */
__emu_cmpxchg:
ldr ip, [r0]
teq ip, r1
streq r2, [r0]
mov r0, ip
mrs ip, spsr
bicne ip, ip, #ARM_CPSR_Z
orreq ip, ip, #ARM_CPSR_Z
msr spsr, ip
movs pc, lr
```

This would allow an attacker from user space to arbitrary write kernel memory with a controlled value since there is no bounds checking on the R0 pointer passed in to ensure that this falls within user space (00000000-7fffffff).

8. QNX Debugging

Currently the main problem with any memory corruption issue identified on Blackberry 10 is that there is no way to perform debugging of a process in which the devuser does not have access to. The limiting of debug capability severely increases the bar for an attacker to identify privilege escalation issues. This includes both attached debugging and access to the core dumps.

In order to perform our ongoing attack research the following approach was taken. The X86 simulator with root access was used to perform debugging on X86. However, due to differences in binaries (X86 vs ARM) it was not possible to confirm certain issues this way and it was necessary to perform static reverse engineering on the ARM binaries. The paravirtualised VM used in the BB10 simulator was also found to be missing certain components found on physical BB10 device firmware.

8.1 Core Dump Architecture

An investigation into other methods of debugging was also performed to determine the feasibility of obtaining crashing information from higher privileged processes. On *NIX architecture typically core dumps are produced when a processes crashes (if no ulimits are set). This is similar on QNX and the dumper process is used to produce core dumps.

On a Blackberry 10 device core dumps are written to the /var/log/ directory by dumper when a process crashes. However, the devuser account has no permissions to that directory and therefore cannot gain access to the core dumps. After a phone restart these coredumps are copied into the /tmp/ directory, however, the permissions prevent access from the devuser. After reboot these coredumps are removed if they are not uploaded to Blackberry via their reporting tools (QUIP).

An investigation into dumper was performed to attempt to identify any weaknesses with the core dump mechanisms:

Dumper is executed in the start-up script using the following command line arguments:

```
CMN_ON -d -e VAR_DUMPER_ARGUMENTS="${VAR_DUMPER_ARGUMENTS}" ${BASEFS}/bin/ksh -c "
        slay -fQ dumper
        qon -d -p9 mv -f /tmp/*.core /var/log > /dev/null 2>&1
        dumper $VAR_DUMPER_ARGUMENTS
"
```

`VAR_DUMPER_ARGUMENTS="-U 27:412 -S -F -d ${LOGDIR} -m"` where `LOGDIR=/var/log`

The dumper process is typically accessed using the /proc/dumper interface.

nrw------- 1 root nto 0 Jul 22 10:44 /proc/dumper

The devuser does not have access to this interface and therefore cannot use this interface.

However, it is possible to communicate with the dumper channel connection:

```
$ ls -al /proc/mount/proc/dumper/
total 0
nrw-------   1 root      nto              0 Jul 22 08:29 0,999452,1,0,10
```

```
0 – node id
999452 – process id
1 – channel id
0 – handle
10 – file type
```

http://www.qnx.com/developers/docs/6.3.2/neutrino/prog/resmgr.html

It is therefore possible to connect to the dumper channel using the following code:

```
ConnectAttach(0,999452,1,0,0);
```

The binaries associated with crash dumping are as follows:

3) Dumper
4) Crash_monitor.so
5) Preservedmem.so

At this time no weaknesses were identified which could allow an attacker to gain direct access to core dumps on a physical BB10 device.

8.2 GDB Debugging

The implementation of QNX debugging was also investigated to determine any weaknesses in the debug subsystem. In the past weaknesses have arisen due to the support for ptrace() syscall. However, this functionality has been removed from BB10 OS.

On QNX debugging is implemented using the procfs.

http://www.qnx.com/developers/docs/6.5.0_sp1/index.jsp?topic=%2Fcom.qnx.doc.neutrino_cookbook%2Fs3_procfs.html

The proc resource manager is responsible for the pathname /proc namespace. The proc pathname namespace is typically used for process level debugging, querying process statistics etc.

There are a number of special pathname's managed by the proc resource manager.

/proc/PID/as

/proc/PID/as can be used to access the process address space of another program on the system. However, due to security implications, QNX 8.0 protects against one process accessing another's process address space using UNIX ACLs. It is not possible to open a file descriptor to (even read only) another processes address space.

```
-rw-------   1 root      nto        15253504 Mar 29 06:01 /proc/1/as
```

/proc/self/

/proc/self is a link to the currently running process. /proc/self/as can be used as a link to the currently running processes address space.

```
-rw-------   1 devuser   devuser      675840 Mar 29 06:02 /proc/self/as
```

In order for a debugger (such as GDB to debug a process the following call is made):

```
(gdb) target qnx <device_IP_address>:8000
(gdb) attach <pid>
(gdb) file /path/to/app/executable/on/the/host
(gdb) set solib-search-path
$HOST_QTDIR/lib:$HOST_QTDIR/plugins/xyz/:$BBNDK/target_<version>/qnx6/armle-
v7/lib/:$BBNDK/target_<version>/qnx6/
armle-v7/usr/lib
(gdb) attach <pid>
(gdb) b main
(gdb) c
```

This process was used to automate fuzzing against an ARM device. The code used to do this is available at the following location:

https://github.com/alexplaskett/QNXSecurity/blob/master/blackberry_monitor.py

8.3 QCONN

QCONN is used for QNX Momentics to remotely interface with the device for debugging. QCONN is comprised of two components qconnDoor and qconn itself. The QCONN service listens on port 8888 when debug mode is enabled on the device.

On the simulator this can be used to gain root access to the device. The following script can be used:

```
service launcher
start/flags run /bin/sh -
cp /bin/ksh /tmp
chmod u+s /tmp/ksh
```

QCONN has been examined by other security researchers and has led to a number of vulnerabilities being identified previously. The first of these is launching a binary as root:

https://www.rapid7.com/db/modules/exploit/unix/misc/qnx_qconn_exec

This is not applicable to BB10 devices because QCONN runs as a low privilege user devuser (on a real device). However, this technique may prove fruitful on other QNX targets.

Memory corruption has also be identified with the QCONN service leading to remote code execution. Whilst this is a serious vulnerability, it requires a BB10 device running an old firmware version and having developer options enabled.

http://www.cvedetails.com/cve/CVE-2014-2389/

8.4 WebKit Debugging

As part of browser exploitation it is necessary to be able to debug the web browser and perform crash analysis. Whilst it is not possible to use GDB to attach to the browser process itself (due to the different privileges), it is possible to use a Webview embedded within an application to perform native debugging of WebKit.

An old version of WebKit has been released at the following URL:

https://github.com/blackberry/WebKit-BB10

However, this version of WebKit is significantly out of date compared to the version running on the device and therefore of little value due to modern public Webkit sources being publically available (unless examination of the vendor specific modifications is required).

8.5 Kernel Debugging

Static reverse engineering of memory corruption vulnerabilities is a time consuming process and dynamic debugging can help identify and confirm issues more easily. An attempt was made to perform kernel debugging of QNX.

On all the versions of QNX which it was possible to research the kernel debugging had been removed from the IFS image. Therefore the process was to recreate the kernel debugger from the old source code tree. On QNX the kernel debugger is called KDEBUG. The old source code for this is on sourceforge (http://sourceforge.net/p/monartis/opengnx/ci/master/tree/), however, due to the lack of the software development kit being available it was necessary to use the BB10 tool chain to integrate this into a custom IFS image.

This allowed kernel debugging under X86 QNX 6.* images, however at this stage it was not possible to get ARM support enabled. Under BB10 it is possible to do basic kernel debugging by enabling the VMWare GDBStub (such as inspect memory of the guest), however due to the lack of support in the client OS then it is not possible to do target debugging. This can be achieved using the following setting:

```
debugStub.listen.guest32 = “true”
```

In an ideal situation we want to be able to perform ARM kernel debugging of a BB10 IFS image. This task is on-going and requires a significant amount of effort to reach at this stage. More information can be found in the next steps section of the conclusion.

8.6 Conclusion

This paper provides the initial groundwork for QNX exploitation and can be used to identify further issues with the platform.

It was determined that a lot of design effort has gone into building a secure system architecture from the design choices made. However, due to not many researchers looking at the operating system and lack of external security review then it is still possible to find issues within the OS. QNX is certainly not vulnerability free, however, a lot of effort has gone into prioritising the removal of security issues over general stability issues. For an OS that is touted as being safety critical it did not stand up well to an attacker intentionally providing malicious input.

Restricting the debug capability of the platform increases the difficulty for an attacker to identify and determine security weaknesses within the platform. The watchdog process used by QNX has interesting side effects that lead to the OS failing in a safe way rather than memory corruption occurring first and the attacker being able to continue execution.

The vulnerabilities identified during this research are relatively benign demonstrating that significant error has been put into creating a hardening operating system for a mobile device. However, it is expected that due to the obscurity of the architecture and the knowledge required for the initial groundwork takes some time to obtain. It is expected with this information available in the public domain, other QNX weaknesses will be identified and the platform security increased in future.

More information on the vulnerabilities identified will be published when the vendor has performed any remediation of the issues.

8.6.1 Next Steps

+ QEMU BB10 OS
+ Blackberry PRIV

9. Appendix

9.1 Previous Research and Credits

- @mwrlabs - https://labs.mwrinfosecurity.com/system/assets/410/original/mwri_blackberry-10-security_2013-06-03.pdf
- @esizkur - https://www.youtube.com/watch?v=z5qXhgqw5Gc
- @quine / @bnull - https://cansecwest.com/slides/2014/NoApologyRequired-BB10-CanSecWest2014.pdf
- @alexanderantukh - https://www.sec-consult.com/fxdata/seccons/prod/downloads/sec_consult_vulnerability_lab_blackberry_z10_initial_analysis_v10.pdf
- @juliocesarfort - https://packetstormsecurity.com/files/author/3551/
- @timb_machine - http://seclists.org/fulldisclosure/2014/Mar/98
- @0xcharlie / @nudehaberdasher - http://illmatics.com/Remote%20Car%20Hacking.pdf
- Ken Matthews (Blackberry Incident Response) for timely handling of the issues reported.

9.2 Github Information

The code produced as part of this research can be found in the following GitHub repository:

https://github.com/alexplaskett/qnxsecurity

9.3 Procnto ARM Syscalls

```
LOAD:FE11B2CC ker_call_table  DCD __KER_NOP           ; DATA XREF: __ker_entry+80□o
LOAD:FE11B2CC                                         ; LOAD:off_FE0B66CC□o ...
LOAD:FE11B2D0                 DCD __KER_TRACE_EVENT
LOAD:FE11B2D4                 DCD __KER_RING0
LOAD:FE11B2D8                 DCD __KER_CACHE_FLUSH
LOAD:FE11B2DC                 DCD __KER_SPARE
LOAD:FE11B2E0                 DCD __KER_SPARE
LOAD:FE11B2E4                 DCD __KER_SPARE
LOAD:FE11B2E8                 DCD __KER_SYS_CPUPAGE_GET
LOAD:FE11B2EC                 DCD __KER_SYS_CPUPAGE_SET
LOAD:FE11B2F0                 DCD __KER_MSG_PAUSE
LOAD:FE11B2F4                 DCD __KER_MSG_CURRENT
LOAD:FE11B2F8                 DCD __KER_MSG_SEND
LOAD:FE11B2FC                 DCD __KER_MSG_SEND
LOAD:FE11B300                 DCD __KER_MSG_ERROR
LOAD:FE11B304                 DCD __KER_MSG_RECEIVEV
LOAD:FE11B308                 DCD __KER_MSG_REPLYV
LOAD:FE11B30C                 DCD __KER_MSG_READV
LOAD:FE11B310                 DCD __KER_MSG_WRITEV
LOAD:FE11B314                 DCD __KER_MSG_READWRITEV
LOAD:FE11B318                 DCD __KER_MSG_INFO
LOAD:FE11B31C                 DCD __KER_MSG_SEND_PULSE
LOAD:FE11B320                 DCD __KER_MSG_DELIVER_EVENT
LOAD:FE11B324                 DCD __KER_MSG_KEYDATA
LOAD:FE11B328                 DCD __KER_MSG_READIOV
LOAD:FE11B32C                 DCD __KER_MSG_RECEIVEV
LOAD:FE11B330                 DCD __KER_MSG_VERIFY_EVENT
LOAD:FE11B334                 DCD __KER_SIGNAL_KILL
LOAD:FE11B338                 DCD __KER_SIGNAL_RETURN
LOAD:FE11B33C                 DCD __KER_SIGNAL_FAULT
LOAD:FE11B340                 DCD __KER_SIGNAL_ACTION
LOAD:FE11B344                 DCD __KER_SIGNAL_PROCMASK
LOAD:FE11B348                 DCD __KER_SIGNAL_SUSPEND
LOAD:FE11B34C                 DCD __KER_SIGNAL_WAITINFO
LOAD:FE11B350                 DCD __KER_SPARE
LOAD:FE11B354                 DCD __KER_SPARE
LOAD:FE11B358                 DCD __KER_CHANNEL_CREATE
LOAD:FE11B35C                 DCD __KER_CHANNEL_DESTROY
LOAD:FE11B360                 DCD __KER_CHANCON_ATTR
LOAD:FE11B364                 DCD __KER_SPARE
LOAD:FE11B368                 DCD __KER_CONNECT_ATTACH
LOAD:FE11B36C                 DCD __KER_CONNECT_DETACH
```

```
LOAD:FE11B370                DCD __KER_CONNECT_SERVER_INFO
LOAD:FE11B374                DCD __KER_CONNECT_CLIENT_INFO
LOAD:FE11B378                DCD __KER_CONNECT_FLAGS
LOAD:FE11B37C                DCD __KER_SPARE
LOAD:FE11B380                DCD __KER_SPARE
LOAD:FE11B384                DCD __KER_THREAD_CREATE
LOAD:FE11B388                DCD __KER_THREAD_DESTROY
LOAD:FE11B38C                DCD __KER_THREAD_DESTROYALL
LOAD:FE11B390                DCD __KER_THREAD_DETACH
LOAD:FE11B394                DCD __KER_THREAD_JOIN
LOAD:FE11B398                DCD __KER_THREAD_CANCEL
LOAD:FE11B39C                DCD __KER_THREAD_CTL
LOAD:FE11B3A0                DCD __KER_THREAD_CTLEXT
LOAD:FE11B3A4                DCD __KER_SPARE
LOAD:FE11B3A8                DCD __KER_INTERRUPT_ATTACH
LOAD:FE11B3AC                DCD __KER_INTERRUPT_DETACH_FUNC
LOAD:FE11B3B0                DCD __KER_INTERRUPT_DETACH
LOAD:FE11B3B4                DCD __KER_INTERRUPT_WAIT
LOAD:FE11B3B8                DCD __KER_INTERRUPT_MASK
LOAD:FE11B3BC                DCD __KER_INTERRUPT_UNMASK
LOAD:FE11B3C0                DCD __KER_INTERRUPT_CHARACTERISTIC
LOAD:FE11B3C4                DCD __KER_SPARE
LOAD:FE11B3C8                DCD __KER_SPARE
LOAD:FE11B3CC                DCD __KER_SPARE
LOAD:FE11B3D0                DCD __KER_CLOCK_TIME
LOAD:FE11B3D4                DCD __KER_CLOCK_ADJUST
LOAD:FE11B3D8                DCD __KER_CLOCK_PERIOD
LOAD:FE11B3DC                DCD __KER_CLOCK_ID
LOAD:FE11B3E0                DCD __KER_SPARE
LOAD:FE11B3E4                DCD __KER_TIMER_CREATE
LOAD:FE11B3E8                DCD __KER_TIMER_DESTROY
LOAD:FE11B3EC                DCD __KER_TIMER_SETTIME
LOAD:FE11B3F0                DCD __KER_TIMER_INFO
LOAD:FE11B3F4                DCD __KER_TIMER_ALARM
LOAD:FE11B3F8                DCD __KER_TIMER_TIMEOUT
LOAD:FE11B3FC                DCD __KER_SPARE
LOAD:FE11B400                DCD __KER_SPARE
LOAD:FE11B404                DCD __KER_SYNC_CREATE
LOAD:FE11B408                DCD __KER_SYNC_DESTROY
LOAD:FE11B40C                DCD __KER_SYNC_MUTEX_LOCK
LOAD:FE11B410                DCD __KER_SYNC_MUTEX_UNLOCK
LOAD:FE11B414                DCD __KER_SYNC_CONDVAR_WAIT
LOAD:FE11B418                DCD __KER_SYNC_CONDVAR_SIGNAL
LOAD:FE11B41C                DCD __KER_SYNC_SEM_POST
LOAD:FE11B420                DCD __KER_SYNC_SEM_WAIT
LOAD:FE11B424                DCD __KER_SYNC_CTL
LOAD:FE11B428                DCD __KER_SYNC_MUTEX_REVIVE
LOAD:FE11B42C                DCD __KER_SCHED_GET
LOAD:FE11B430                DCD __KER_SCHED_SET
LOAD:FE11B434                DCD __KER_SCHED_YIELD
LOAD:FE11B438                DCD __KER_SCHED_INFO
```

```
LOAD:FE11B43C                DCD __KER_SCHED_CTL
LOAD:FE11B440                DCD __KER_NET_CRED
LOAD:FE11B444                DCD __KER_NET_VTID
LOAD:FE11B448                DCD __KER_NET_UNBLOCK
LOAD:FE11B44C                DCD __KER_NET_INFOSCOID
LOAD:FE11B450                DCD __KER_NET_SIGNAL_KILL
LOAD:FE11B454                DCD __KER_SPARE
LOAD:FE11B458                DCD __KER_SPARE
LOAD:FE11B45C                DCD __KER_POWER_PARAMETER
LOAD:FE11B460                DCD __KER_POWER_ACTIVE
LOAD:FE11B464                DCD __KER_SCHED_WAYPOINT
LOAD:FE11B468                DCD __KER_SPARE
LOAD:FE11B46C                DCD __KER_SPARE
LOAD:FE11B470                DCD __KER_SPARE
```