QNX SOFTWARE SYSTEMS

Secure by Design: Using a Microkernel RTOS to Build Secure, Fault-Tolerant Systems

Paul Leroux
QNX Software Systems
paull@qnx.com

Strong boundaries

Virtually every embedded system today is connected, either physically or wirelessly, to the outside world. This network connectivity allows users to perform remote monitoring and control, and enables systems to download new software features or content on the fly. Unfortunately, it also makes systems vulnerable to infiltration by a growing cadre of cyber terrorists and extortionists. In fact, malicious hackers have already compromised a variety of SCADA systems, HVAC control systems, networking routers, mobile devices, and nuclear safety systems, using viruses, denial-of-service (DoS) attacks, and other networked-based exploits.

To thwart such attacks, many companies and organizations surround their systems with a protective barrier that consists of network security, cryptographic security, and even physical security. But as experience shows, malicious hackers can often break through this barrier to attack the system within. Consequently, the system itself must also be designed to survive assaults, without loss of service or corruption of data. In other words, developers must implement security not only around the system, but also *within* the system.

As the software that provides centralized access to the CPU, memory, and other resources, the realtime operating system (RTOS) can play a major role in achieving this goal of building secure, survivable embedded systems. In particular, it can enforce strong boundaries between software processes to prevent any process from affecting the performance, behavior, or data of other processes. Processes can damage one another intentionally (via malware) or unintentionally (via bugs); a well-designed RTOS will provide mechanisms to prevent such damage and to keep the system in a healthy state.

The reference monitor

James Anderson established the core principles of computer security in his *Computer Security Technology Planning Study*, published in 1972. Two years later, Jerome Saltzer and Michael Schroeder expanded upon these principles in *The Protection of Information in Computer Systems.*

In his study, Anderson introduced the concept of the *reference monitor,* a mechanism implemented in the OS kernel that validates every request for data, peripherals, and other resources. The reference monitor ensures that every resource is accessed not only by the appropriate software process, but also by the right process operating against the correct data in the correct context.

To fulfill this role, the reference monitor must possess three key attributes:

- Tamper resistant
- Always invoked
- Small and simple enough to be easily verifiable

Tamper resistant

If users can breach the reference monitor's integrity, either intentionally or accidentally, then the system cannot be trusted to behave correctly. Consequently, the OS kernel, which contains the reference monitor, must resist any attempts at modification. For example, the QNX® Neutrino® microkernel is read-only and cannot be modified at run time. It also performs integrity checks at startup to ensure that it is undamaged. If an attempted attack or unintended action has modified the kernel image, the microkernel won't boot up into operational mode. Since the microkernel cannot be modified while running, and won't start if modified, it can be trusted to carry out its operations correctly.

Always invoked

The reference monitor must evaluate all access requests consistently and explicitly, with no exceptions. Consequently, it must be invoked every time a process attempts to access a resource. By implementing the reference monitor within the kernel, the RTOS can ensure that every access request goes through the exact same mechanism for verification.

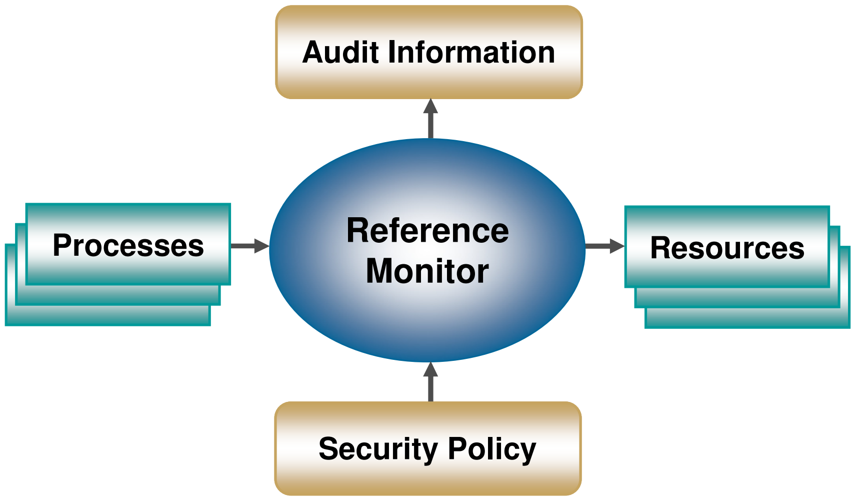

Figure 1 — The reference monitor, which resides within the OS kernel, validates all attempts to access system resources.

Simple and easily verifiable

The kernel and its reference monitor must operate correctly at all times. Thus, the kernel design must be as simple and verifiable as possible. If it becomes too complex, security holes might escape notice and be exploited, either intentionally or accidentally.

A microkernel OS offers an ideal architecture for addressing this requirement, since the microkernel implements only the basic necessities of an operating system, such as signals, timers, and thread scheduling. All the other system services — device drivers, networking stacks, file systems — run outside of the microkernel as separate, user-space programs. See Figure 2. In traditional operating systems, all these services run within the kernel, adding significantly to kernel complexity.

Because the microkernel is small and comprehensible, it is less prone to errors and allows for easier validation than conventional kernels. Also, because drivers and other system services run as separate, user-space processes, a fault in any service won't damage the microkernel or other running processes. For instance, if a device driver attempts to access memory outside of its process container, a user-space process called a *software watchdog* can quickly terminate the driver, reclaim the resources the driver was using, and then restart the driver, without interrupting other services. This ability to clean up resources and restart components from a known good state helps address the security requirement for fault tolerance.

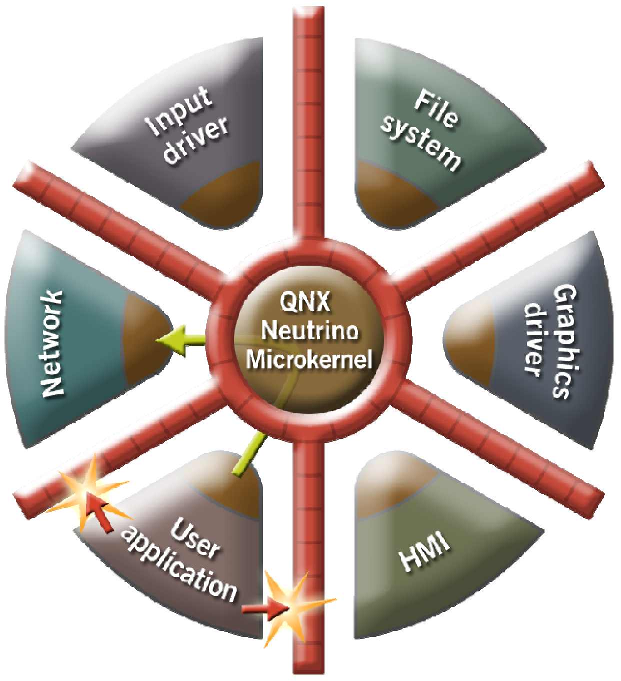

Figure 2 — Microkernel architecture satisfies the security requirement for a small, simple, and comprehensible OS kernel. It also prevents faults in device drivers and other system services from damaging the kernel or other processes.

Salter & Schroeder’s eight principles

In 1974 Salter and Schroeder expanded upon Anderson’s three principles with eight of their own. See Table 1.

Eight principles of a security design	
Economy of mechanism	Reduce complexity to eliminate unexpected side effects or behavior.
Fail-safe defaults	Deny a subject access to an object, unless the subject has been given explicit permission.
Complete mediation	Check every access to an object to ensure that the access is allowed.
Open design	Don’t rely on security through obscurity.
Separation of privilege	Grant access to an object only if the subject satisfies multiple conditions.
Least privilege	Provide a process or user with the least set of privileges possible to complete a given job.
Least common mechanism	Prevent resources from being shared implicitly.
Psychological acceptability	Ensure the system is understandable by users.

Table 1 — Salter & Schroeder’s eight principles for preventing security flaws.

The first principle, economy of mechanism, complements Anderson’s notion of a small, verifiable kernel. An OS kernel must be simple and efficient, and do only what it is supposed to do; otherwise, unexpected side effects or behavior are more likely to occur.

To satisfy this requirement, an RTOS can use a well-designed microkernel. It can also use a standard privilege mechanism: POSIX. This industry standard is well-known and understood, and provides users with an explicit understanding as to how the system works, without any surprises. By using POSIX, the RTOS can satisfy the requirement of doing only what it needs to do, without adding complexity that can harbor unknown holes and exploits.

According to the principle of fail-safe defaults, access to a resource must be explicitly given and implicitly denied. POSIX has an inherent concept of fail-safe defaults: the developer must explicitly give processes permission to access a resource; otherwise, they are denied access. Likewise, the security-related attributes of an RTOS must be set to a “known good default” during initialization. Hence, whenever new objects or subjects are created, the RTOS assigns their security attributes to “known good” initial default values.

This approach ensures that processes behave in a known fashion. It also ensures that the owner of a new object cannot access information from previous objects that belonged to other owners. In the QNX Neutrino RTOS, for example, every new object (process, memory allocation, etc.) is set to a known, neutral state to prevent unintentional transmission of information. Because this approach ensures that every process begins from known good defaults, QNX Neutrino can roll back a failed process (for instance, a device driver) to its initial state, without a system reset.

Complete mediation is simply the notion that the OS kernel mediates every access to a given resource. The kernel evaluates every access request consistently and explicitly, with no special handling for any request. By making strategic use of the POSIX guidelines and system interlock hardware (via the MMU), an RTOS like QNX Neutrino can ensure that these access controls are in place.

In the nineteenth century, the Dutch cryptographer Auguste Kerckhoffs postulated that a cryptosystem should be secure even if everything about the system, except the key, becomes public knowledge. Ever since then, security experts, including Salter and Schroeder, have warned against security through obscurity and have promoted the advantages of open design. An open design allows the system to inspected by many reviewers and makes it easier for the user to evaluate whether the system is appropriate for an intended application.

To satisfy this requirement, the RTOS should adhere to POSIX and other standard APIs. Moreover, the RTOS vendor should ensure that the APIs are well-defined, well-documented, and behave in known ways. That way, users, system builders, and system architects can readily determine how the RTOS should behave.

To ensure strong security, a system should grant access to a resource only if the process requesting the resource satisfies multiple conditions — a concept known as separation of privilege. According to Salter & Schroeder, “a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key.” Separation of privilege ensures a more resilient system and prevents accidental or intentional system disruption. Table 2 illustrates how the QNX Neutrino RTOS implements separation of privilege to control access to certain resources.

	User thread (uid !=-0)	Root thread (uid=0)	Root thread with I/O privilege
Maximum thread priority	User max (64 by default)	255	255
Map shared memory	Based on permissions	**Yes**	**Yes**
Map physical memory	**No**	**Yes**	**Yes**
I/O operation	**No**	**No**	**Yes**
Attach to IRQ	**No**	**No**	**Yes**
Disable interrupts	**No**	**No**	**Yes**

Table 2 — Separation of privilege in the QNX Neutrino RTOS. For instance, to perform I/O operations, a thread must not only have root permissions, but also ask for I/O privilege.

Every software component must also satisfy least privilege; that is, it must operate with only the security privileges and system resources that it needs to function correctly. This approach minimizes unintentional or intentional damage to the system and ensures that the system behaves in an understandable and consistent fashion. In the QNX Neutrino RTOS, for instance, every request for privileges or resources must go through the microkernel, which, prior to granting access, ensures that the requesting process has the appropriate permissions.

To satisfy the “least common mechanism” principle, the RTOS must prevent implicit sharing of information. For instance, if an application relinquishes control over a resource, the QNX Neutrino microkernel will regain control of that resource and clear it of any residual information. The developer can choose whether this clearing occurs immediately after the process relinquishes the resource or when the kernel reallocates the resource for subsequent use. This approach can prevent processes from “spying” on protected resources. It can also help ensure that no information is ever reclaimable, even within the same process.

An RTOS can further control allocation of resources through resource partitioning. Briefly stated, this technique allows the system designer to group software processes into virtual compartments, or partitions, and to allocate a predetermined budget of memory and/or CPU time to each partition. The RTOS can then enforce these budgets and thereby prevent processes in any partition from erroneously or maliciously monopolizing memory or CPU time needed by processes in other partitions.

According to Salter and Schroeder’s last principle, psychological acceptability, a system must behave in a well-documented manner that the user expects; there should be no surprises. In other words, the system must operate in a *deterministic* fashion. To satisfy this requirement, the RTOS must use well-known and accepted standards as a basis for its security model. It must also provide a definition of how it will behave during initialization, self-test, and recovery operations.

Modern enhancements to security

Since Anderson, Saltzer, and Schroeder presented their seminal papers on secure systems, the security community has devised a few enhancements, including accountability, priority of subjects and priority of operations, self-tests, and fault tolerance.

Accountability

Any system that manipulates information must meet at least rudimentary accountability requirements. For instance, in the QNX Neutrino RTOS, all events can be time stamped. The high-resolution time stamps can't be modified, ensuring that no clock-skew exploits are possible with respect to audit subsystems. This approach provides added security since system events that are recorded and audited cannot be repudiated with respect to the timing of their activities.

Priority of subjects and operations

High-priority threads or operations must proceed without undue interference or delay caused by low-priority subjects or operations. By its very design, an RTOS like QNX Neutrino incorporates a strict protocol to ensure that if resources become sparse (i.e. if there are more requests than available kernel threads), only higher-priority requests will float to the top of the processing queue. A well-designed RTOS also implements a prioritization protocol to ensure that higher-priority system-level requests never get "starved out" by lower-priority events. Higher-priority requests are always processed first.

Self-tests

A system must also test itself and verify the integrity of its stored executable code and objects. That way, it can detect corruption caused by failures that don't necessarily stop system operation. Such failures may occur either because of malicious corruption, unforeseen failure modes, or oversights in the design of hardware, firmware, or software.

Fault tolerance

A system must operate correctly even in the event of a failure. If a failure does occur, the system must recover to a "known good" state. For instance, because the QNX Neutrino microkernel employs self-test mechanisms, it can, upon recovery, ensure that it has entered into a known good state. Also, if a process fails, the microkernel can clean up resources used by the process and restart the process from a known good state.

Fault tolerance by design: high availability framework

To simplify the goal of achieving fault tolerance, developers can use a high availability (HA) framework, such as the one supported by the QNX Neutrino RTOS. Properly designed, an HA framework provides easy-to-use tools that allow programmers to implement fault tolerance without having to reinvent the wheel or spend time designing their own process monitoring schemes.

An HA framework can provide:

- facilities for automatically restarting failed processes, without rebooting the system
- a smart watchdog that can monitor for process failures and take appropriate recovery actions
- a programming interface that allows the system designer to select failure conditions and to specify which actions the watchdog will perform when those conditions occur
- a mirror process, or guardian, that perpetually stands ready to take over the watchdog role, if necessary

Figure 3 shows the architecture of an HA framework in a microkernel runtime environment. The framework can also provide fault-tolerant interprocess communication, allowing connections between processes to be reestablished after a process failure.

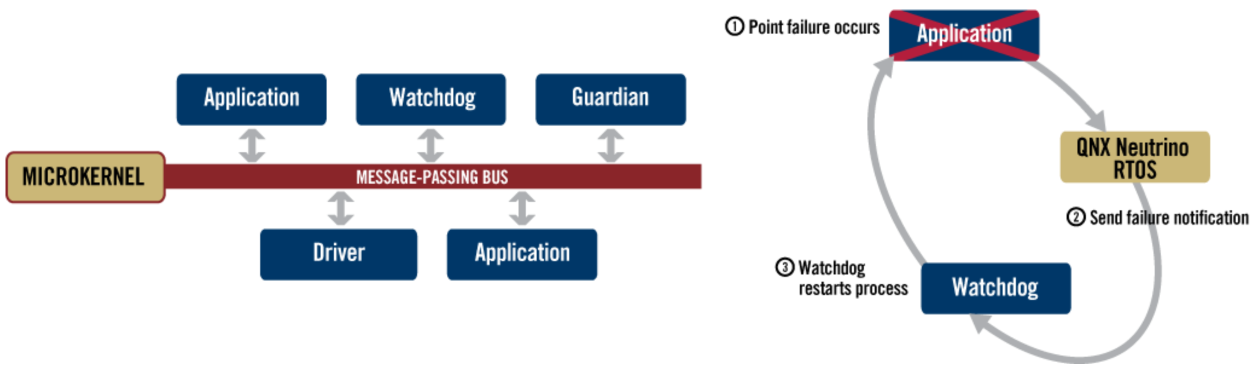

Figure 3 — An HA framework in which a software watchdog restarts failed processes. If the watchdog itself fails, the guardian process, which mirrors the watchdog, can immediately take over the watchdog's role.

Security with adaptive partitioning

A key technique for designing security into a system is to clearly partition system components that are likely to be compromised. This is particularly true for systems open to external access through a network connection, USB file system, or user input. Partitioning becomes even more important if a system allows users to download applications and other content — content that can potentially harbor malware.

By partitioning system resources, such as CPU time, the system designer can ensure that critical system components have access to those resources at all times. Traditional partitioning schemes use fixed partitions, an inefficient approach that forces system designers to allocate partition budgets according to peak resource needs. In this approach, each partition has a fixed CPU budget that corresponds to the peak CPU usage of the processes contained in that partition. As a result, any partition that isn't busy will consume its allocated CPU cycles in an idle state. Meanwhile, other partitions can't access those unused cycles, even if those partitions are busy and could benefit from the extra processing time.

Adaptive partitioning, a new form of time partitioning pioneered by QNX Software Systems, takes advantage of these unused cycles: If a partition doesn't consume all of its allocated CPU cycles, the partitioning scheduler dynamically reallocates those idle cycles to partitions that could benefit from the extra processing time. This approach offers the best of both worlds: it can enforce CPU guarantees when the system runs out of excess cycles (for guaranteed availability of applications and services) and can dispense free CPU cycles when they become available (for maximum CPU utilization and performance). Adaptive partitioning makes time partitioning an option for virtually any type of system; fixed partitioning, in comparison, is typically used in safety-critical systems where the inefficiency is tolerated.

Adaptive partitioning allows system designers to partition critical parts of the system from potentially vulnerable components. For example, an Adobe Flash player that runs downloadable content can run in one partition while a critical control process runs in another partition. If, for some reason, a downloaded Adobe Flash application caused the Flash player to consume all CPU cycles, the adaptive partitioning scheme would prevent the Flash player from exceeding its assigned CPU budget and would ensure that every other partition continues to operate within its own CPU budget. This approach can contain typical denial of service attacks and is simple to configure — partitions can be assigned, monitored, and adjusted at runtime.

Importantly, developers can overlay the adaptive partitioning scheduler onto existing systems without code changes. Applications and system services can simply be launched in a partition, and the scheduler will ensure that partitions receive their allocated budget. Within a partition, the adaptive partitioning scheduler can continue to schedule threads according to the traditional rules of a preemptive, priority-based scheduler.

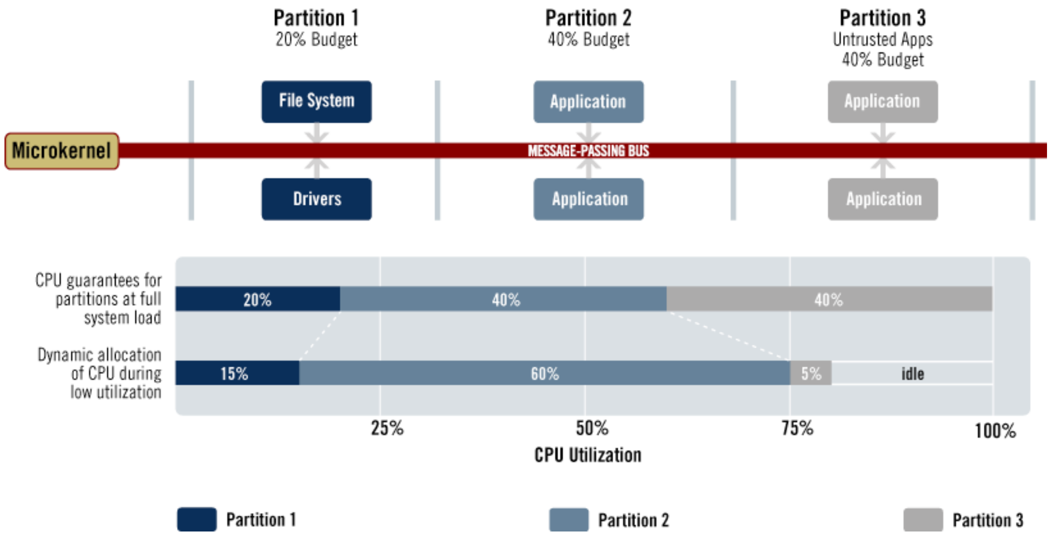

Figure 4 — Adaptive partitioning enforces CPU budgets under full system load and dispenses free CPU cycles during periods of lower processor utilization. For instance, processes and threads in Partition 2 can use CPU cycles allocated to the other partitions when those partitions don't consume their entire CPU budget.

Common Critera ISO/IEC 15408 certification

The principles defined by Anderson, Salter, and Schroeder continue to form the basis of modern security standards and policies, including the *Common Criteria,* an international standard (ISO/IEC 15408) for the development of security specifications. Upon successful completion of a Common Criteria evaluation, an IT product is assigned an evaluation assurance level, such as EAL 4. To achieve a given EAL, a product must meet specific assurance requirements, including design documentation, design analysis, and functional testing.

The process of certifying a product requires a rigorous analysis not only of the product but also of the development processes used to create the product. The process of evaluating and certifying a product under the Common Criteria standard includes:

- Definition of the Target of Evaluation (TOE) — The TOE specifies the exact product and configuration to be evaluated.

- Definition of the Security Target — The Security Target specifies the security objectives for the TOE, the functionality that achieves those objectives, and the threats to this achievement. Within each of these broad categories are specific Security Functional Requirements (SFRs). Examples of SFRs include security audits, user data protection, and identification and authentication.

- Submission of the TOE and the Security Target to the Common Criteria Testing Laboratory (CCTL) — Several of these laboratories exist worldwide, including Communications Security Establishment Canada, which evaluated and certified the QNX Neutrino RTOS Secure Kernel. A vendor submitting a product for evaluation typically works with a consulting partner who helps prepare the submission to the CCTL.

- Preparation of the Validation Oversight Review (IVOR) — The CCTL prepares the IVOR, which reports the initial results of the evaluation. It then completes the testing by implementing the plan outlined in the IVOR. Upon successful testing, the TOE is submitted for certification.

- The CCTL lists the product defined by the TOE as a product certified to the appropriate Evaluation Assurance Level. For instance, the QNX Neutrino RTOS Secure Kernel is certified to EAL 4+, which indicates that the product is methodically designed, tested, and reviewed to highest commercial standards. The "+" designation indicates the certification of QNX Software Systems' security vulnerability remediation process. This process defines how QNX deals with reported security vulnerabilities and how the fixes are designed and tested and built in to an update to the QNX Neutrino RTOS Secure Kernel.

QNX SOFTWARE SYSTEMS

About QNX Software Systems

QNX Software Systems is the industry leader in realtime, embedded OS technology. The component-based architectures of the QNX® Neutrino® RTOS, QNX Momentics® Tool Suite, and QNX Aviage® middleware family together provide the industry's most reliable and scalable framework for building innovative, high-performance embedded systems. Global leaders such as Cisco, Daimler, General Electric, Lockheed Martin, and Siemens depend on QNX technology for security and defense systems, network routers, medical instruments, vehicle telematics units, industrial robotics, and other mission- or life-critical applications. Founded in 1980, QNX Software Systems is headquartered in Ottawa, Canada, and distributes products in over 100 countries worldwide.

www.qnx.com