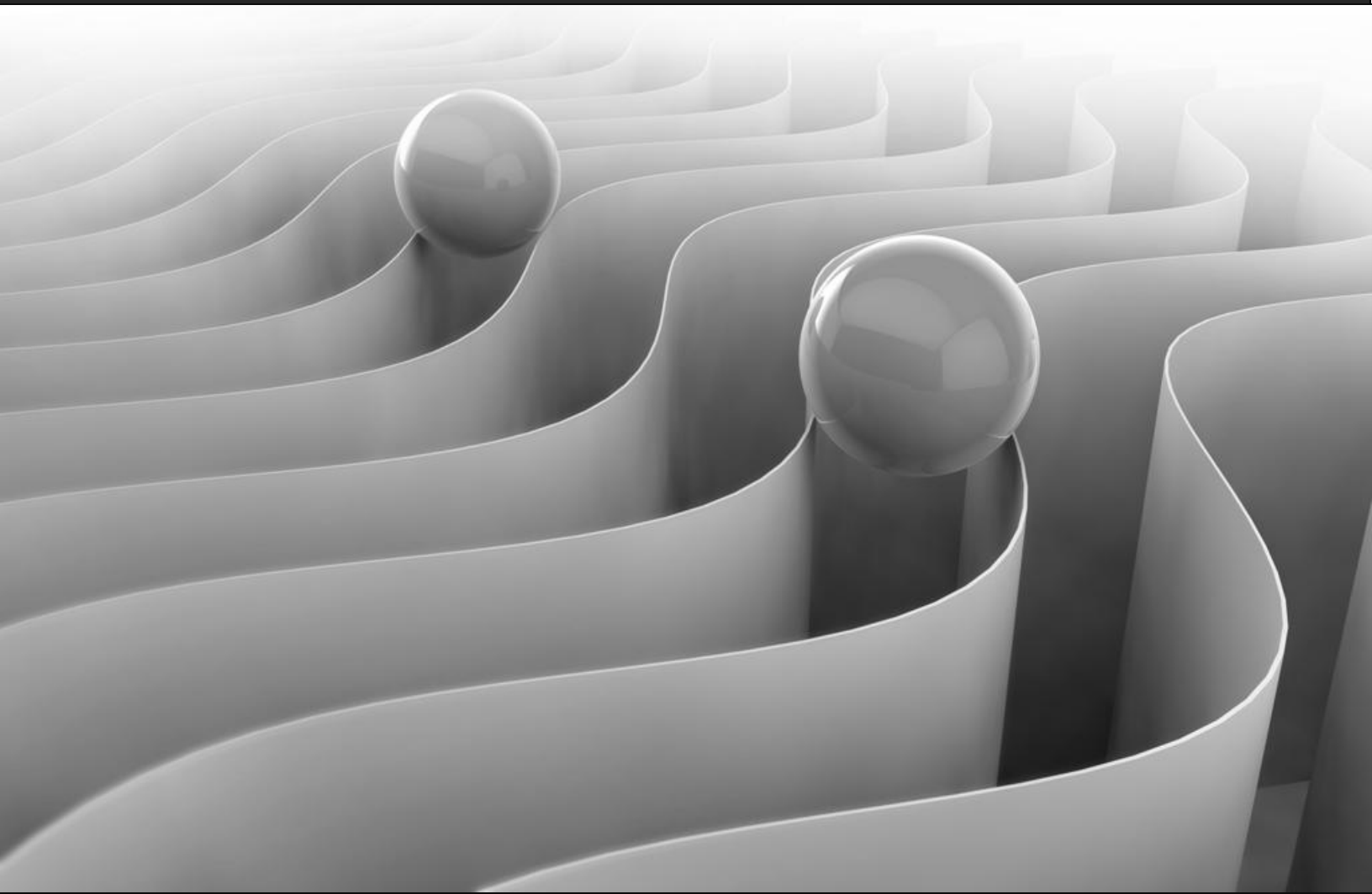
Real Time Logic

SECURING EDGE NODES

Internet of Things Concept Design



Envision Secure

INTRODUCTION

Security is the number one issue and complexity for billions of embedded Internet of Things (IoT) devices entering the market by 2020. Increased consumer awareness and market demand have encouraged manufacturers to produce secure products that are resilient to both known threats and new attack profiles that target device security.

Device side Edge Node security is inherently dependent on the internal hardware and software building blocks available to the application. If any of the core components of the device are not part of a secure system architecture they are set to fail and invite malicious activity by system entry, and unauthorized access.

This whitepaper explores the precautionary steps necessary to assist manufacturers to secure connected offerings at the device level.

Recommended Precursor: Certificate Management for Embedded Systems

TABLE OF CONTENTS

Firmware vs. Embedded HLOS

Use of a HLOS (High-Level Operating System) within an embedded application may be born out of necessity, whether based on reasons of legacy, familiarity, or requirement. From a security standpoint they also increase the risk element for a device, given the native ability to allow dynamic modification of the system architecture at runtime. One of the key benefits for using a HLOS is the ability to easily add or modify programs. This is consequently a double edged sword when considering security because it is also the same vehicle used to open an array of attack vectors and vulnerabilities to the system. Any attack which gains access to the HLOS gains the ability to add and-or start a system process that is capable of controlling the hardware or installing a backdoor.

The September 2014 Shellshock exploit is a global-scale example where a CGI scripting manipulation was used to execute arbitrary commands via the widely used Unix Bash Shell, thereby rendering many HLOS architectures exposed to full system access.

Firmware by contrast is based on a fixed monolithic system structure of software that may enable control, monitoring, and data manipulation for a given system architecture, and is traditionally stored in non-volatile memory such as ROM, or Flash memory. The combination of Application Code, RTOS (Real Time Operating System), and Middleware utilities are assembled into a binary component, that constitute a low-level program control for the device.

Solutions designed in firmware are provided with inherent access limitation to attack vulnerabilities, because the complexity involved in a complete firmware replacement is much greater than adding a process to a HLOS.

Developers have access to many connectivity options with capacity to off-load traditional device centric architectures (popularized through mid-2000), that would otherwise implement a HLOS structure. Client-Server IoT Protocols now provide SSL|TLS secure transports for server-side offloading of the device. The benefit not only provides a safer environment but can also reduce BOM costs related to hardware processing, memory, and power consumption and is trending as today's most practical and affordable alternative for connected Edge Node products.

Microcontrollers vs. CPU Based Systems

Systems that include a CPU with external memory such as [ROM | Flash] and-or for random access memory, make it possible for an attacker to eavesdrop on the Computer BUS and copy the Flash Memory content, thus enabling an attacker to inspect the system code.

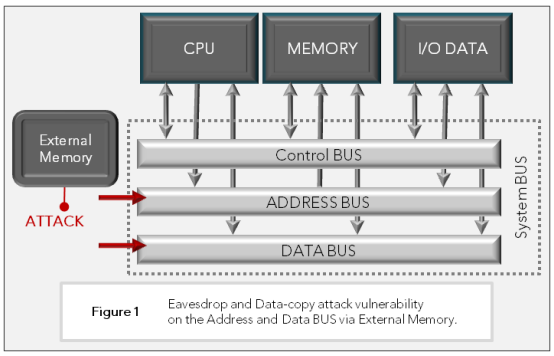


Figure 1 Eavesdrop and Data-copy attack vulnerability on the Address and Data BUS via External Memory.

Microcontroller based systems, where no external memory is present, can be made virtually tamper proof since there is no accessible entry to the firmware stored inside the microcontroller.

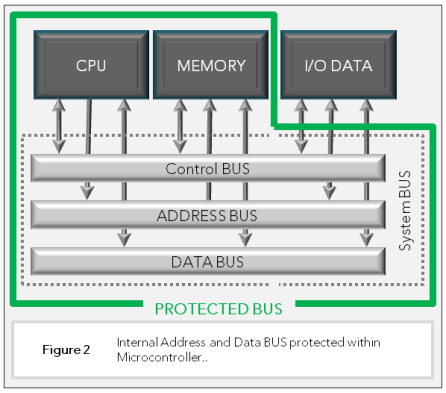


Figure 2 Internal Address and Data BUS protected within Microcontroller..

Memory intensive software solutions that carry large footprints such as OpenSSL and HLOS (Linux) by example are not well suited for small microcontroller based devices without adding large amounts of external memory. Furthermore, when used in these types of minimalistic environments they may yield poor results given the available system resources in processing power consumption, desired boot, and secure authentication timings.

Microcontrollers and JTAG

The 'Go To' interface for microcontroller device electronics is the JTAG interface for development, debugging, testing and uploading firmware during the manufacturing process. The standard allows for access to flash memory so that the device can receive field upgrades and additional services. While incredibly useful, the JTAG interface also leaves a gaping security hole that can be easily be exploited by hackers. Open access to flash memory, proprietary algorithms and other sensitive areas, enables the ability to extract keys, codes, data and processes without physical detection. For this reason it is important to disable the JTAG interface during the manufacturing process. How to disable the JTAG depends on the type of microcontroller used as various manufacturers may provide different options or recommended methods for disabling the JTAG. In most circumstances, particularly on the low-end of microcontrollers, burning the port fuse is the most practical solution. Depending on the sophistication of the chip simply blowing the fuse may not be the only or best method available. High-end chipsets may enable JTAG port configuration options to include setting modes of operation, while other solutions that tend to be less secure recommend disabling JTAG by using registers, or setting the port value to a particular pin. It is recommended to select a microcontroller where the JTAG fuse can be physically destroyed and that you research the available options for any select microcontroller to prevent physical JTAG hacking attempts.

Additional Attack Vectors

Microcontroller based systems, where the JTAG is physically destroyed is certainly one of the more secure ways to render a device nearly tamper proof from local access, however all external ports used by the device remain open to attack. Ports such as USB, Ethernet, and wireless access such as Bluetooth are all vulnerable to local and remote exploits. Attacks may probe for various software bugs such as buffer overflows by sending specially crafted packets to the device via the open ports creating the ability to inject software to take control of the device. To avoid these types of circumstances it is important to include secure automated firmware upgrade logic as part of the originating design. If and when, vulnerabilities are exposed, a patch may be applied via the secure firmware upgrade logic; thereby providing a course of activation and deployment to all affected devices.

An Edge Mode that resides within a private network and is designed to act as a network client connecting out (via router), to an online cloud server is much more secure than a device that functions as a server itself. Protocols such as SMQ, MQTT, and HTTP client, are more secure than a dual role client/server protocol such as CoAP, given the requirement that the device also acts as a server. Device participation in a client role only eliminates direct probing since there is no arbitrary data listening mechanism available to exploit. Client based attacks are therefore limited to "man in the middle" schemes which occur during connection, which are rendered impossible if communications are protected by TLS and whereby the establishment of the TLS connection is based on trusted certificates.

Automated Software Upgrades

It's critical that IoT devices are able to be maintained and updated automatically since firmware may contain vulnerabilities such as buffer overflows that must be patched. The firmware update process can be automated by allowing the device to connect with a cloud server to receive replacement firmware. The following figure illustrates the upgrade process for any single device, but any number of devices may be updated in the same manner.

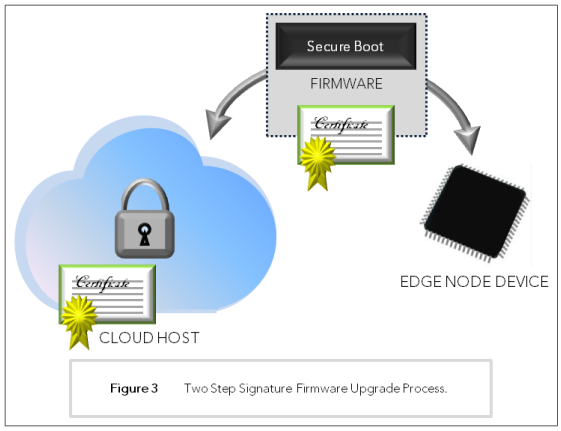


Figure 3 Two Step Signature Firmware Upgrade Process.

The device connects to a secure server via a TLS session, where the server's certificate is validated during the handshake process. A secure connection is established only if the device trusts the server's certificate. Once the initial handshake process is completed the firmware is then downloaded and the firmware's signature is verified on the device side. A double security mechanism is required in order to properly secure the complete upgrade process.

Double Security Mechanism

The device connects to a secure cloud server and initiates a secure TLS handshake. In this process the cloud server is deemed trusted if 1) The server's certificate is trusted by the device, and 2) If the server's name (the domain name the device used to connect) matches the name found in the server's certificate. The secure download begins only after the device is able to confirm the above, thus preventing man in the middle attacks.

After downloading the new firmware, the signature attached to the firmware is verified by the device. The firmware is deemed trusted if the firmware signature is valid. This extra security check is needed since an attacker may have compromised (hacked) the online server and installed an alternate firmware payload. An attacker will not be able to sign the firmware with a trusted signature, thus the extra security measure implemented in the device will detect (reveal) that the online server has been compromised. If the firmware's signature does not match the expected signature the device will abort the loading process.

Digital signatures use asymmetric cryptography. Asymmetric key algorithms use two different keys in pairs-- a combination of a private key and a public key. The private key is known only to the computer signing the firmware, while the public key is stored in the initial and subsequent firmware or in the device's secure boot. There are two types of asymmetric cryptography in use by device authentication: 1) RSA and 2) Elliptic Curve Cryptography (ECC), where ECC is a preferred asymmetric cryptography for resource constrained devices due to its much smaller key sizes for the same level of security. A 224 bit ECC key is equally as strong as a 2048 bit RSA key.

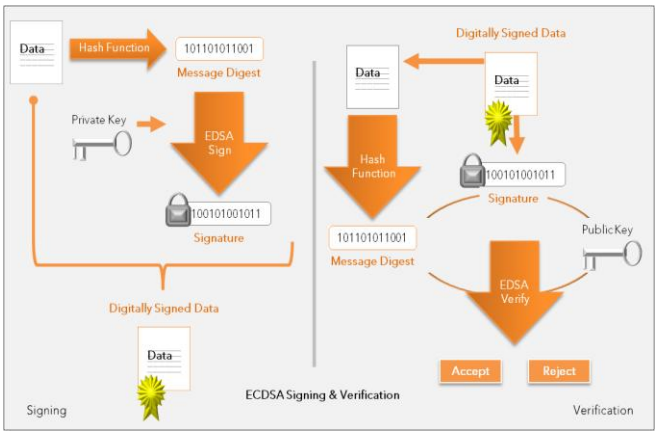
The firmware cannot be directly signed using the private key since the size of the firmware is too big to be used by asymmetric cryptography.

Limitations:

- The size of the signed data cannot be larger than the asymmetric key size. For example, a 2048 bit RSA key cannot encrypt more than 2037 bits of data and a 256 bit Elliptic Curve Cryptography (ECC) key cannot sign more than 256 bits of data.
- The larger the key size, the slower the asymmetric cryptographic operations will be, so it is not practical to use a key large enough to sign firmware.

For this reason, the firmware is not directly signed by using asymmetric cryptography, but instead the firmware's fingerprint is signed. The fingerprint is calculated by using a hashing algorithm such as SHA256. Older algorithms such as MD5 and SHA1 are no longer considered secure because of the relatively large probability of creating a hash collision.

The following figure illustrates the complete signing and verification process, where the private computer signing the firmware is shown in the left pane and the firmware verification process performed in the device is shown in the right pane.



The private computer with the private key calculates a hash on the firmware and signs the hash. The signature (signed hash) is attached to the firmware and the signed firmware is then uploaded to the cloud server responsible for firmware upgrades. When the device downloads the firmware from the secure cloud server, a hash is computed as the firmware data trickles in. The device then verifies that the signature of the computed hash matches the received signature (the signature attached to the firmware). The firmware is trusted if the two signatures match.

The firmware signing process is typically performed on a host computer such as Windows or Linux when creating a new release or distribution. Any cryptographic library may be used during this process. The following illustration demonstrates a C program running on Windows and is making use of RayCrypto, (SharkSSL SSL TLS Crypto Engine).

Private keys may be loaded dynamically from a file system or embedded directly in the C program by converting the key from PEM format to the SharkSSL using the command line tool **SharkSslParseKey**.

Private Key Embedded in the C program

```
/* C data for 'private key' produced by SharkSslParseKey */
const U8 sharkSslPrivECKey256[104] =
{
   0x30, 0x82, 0x00, 0x00, 0x02, 0x20, 0x17, 0x20,
   0xF6, 0x9C, 0x88, 0x3D, 0x9C, 0x32, 0x41, 0xDE,
   0x25, 0x34, 0xD9, 0x23, 0xDD, 0x71, 0x0E, 0xF2,
   0x37, 0x83, 0xAE, 0xC6, 0xD5, 0xBF, 0xEA, 0x61,
   0xC9, 0x81, 0xE5, 0x0F, 0x84, 0x83, 0xFA, 0x3F,
   0x11, 0x23, 0xE8, 0xEA, 0x05, 0x14, 0x28, 0xE0,
   0x37, 0xAC, 0xF3, 0x0E, 0x35, 0x58, 0x58, 0xA8,
   0x16, 0x0B, 0x62, 0xEB, 0xEE, 0xC9, 0x06, 0x2F,
   0x2A, 0xCF, 0x83, 0x1F, 0x93, 0x3C, 0xDF, 0x84,
   0x04, 0xBF, 0x4E, 0x80, 0x5D, 0xA8, 0x0D, 0x15,
   0xD7, 0x02, 0xAD, 0x8E, 0xD0, 0xFA, 0xE2, 0xB5,
   0x69, 0x8C, 0x92, 0x00, 0xF5, 0xB6, 0xA8, 0x1C,
   0x0D, 0x7F, 0x9E, 0x5C, 0x27, 0x93, 0x5C, 0x57
};
U8 dataChunk[256];
U8 sha256digest[SHARKSSL_SHA256_HASH_LEN];
U8 *signature;
U32 len;
U16 sl;
SharkSslSha256Ctx ctx; /* Hash engine */
SharkSslSha256Ctx_constructor(&ctx);
/* Read firmware in chunks from local file system
   and compute hash.
*/
while(readFirmwareChunkFromDisk(dataChunk,&len))
{
   SharkSslSha256Ctx_append(&ctx, dataChunk, len);
}
/* All firmware data chunks read: Finish the hash calculation */
SharkSslSha256Ctx_finish(&ctx, sha256digest);
/* Determine the maximum length for the signature
   in order to allocate memory for it.
*/
sl = sharkssl_ECDSA_siglen(sharkSslPrivECKey256);
signature = (U8*)malloc(sl);
/* Sign the hash using the private key */
if(sharkssl_ECDSA_sign_hash(
      (SharkSslECCKey) sharkSslPrivECKey256, /* ECC private key */
      signature, &sl, /* out variables: signature and it's length */
      sha256digest, /* Firmware fingerprint (hash) */
      SHARKSSL_SHA256_HASH_LEN) == SHARKSSL_ECDSA_OK)
{
   /* Success:
      Attach the signature to the end of the firmware.
      Variables to use: 'signature' and length 'sl'.
   */
}
free(signature);
```

The verification process must be performed by an asymmetric cryptographic library that is sufficiently small for memory constrained devices. The RayCrypto engine included in SharkSSL is a good choice since SharkSSL is the smallest embedded TLS stack. Also, the asymmetric engine in SharkSSL includes assembler optimized code for common architectures thus providing very fast verification even on slow devices.

See the SharkSSL documentation for more information on the cryptographic signature and verify functions used in the example.

Unique Device ID

A unique device ID can be used as a method to strengthen overall security. A unique device ID must be protected from tampering and copying. If stored in the device's internal memory the JTAG fuse should be physically destroyed during manufacturing.

A device ID may be used by IoT protocols as a pre-registered ID, where cloud services allow connections from known keys. It may also be used by a hardened-- online firmware cloud server solution for mutual authentication. In [Figure 3], we illustrated how a device authenticates a server by using X.509 certificates. The server may also be designed to authenticate clients attempting to connect to the firmware upgrade service by verifying the device ID.

A device ID may be a number that is pre-registered in the server. Pre-registered IDs are also good for making sure that only known devices are able to connect. For example, a company leases out production to a manufacturer that produces more devices than reported and re-sells the products under a different brand. A cloud server accepting only known devices will make sure no third party clones can connect to the online service.

X.509 Certificate Management and ID

A very strong device ID and authentication mechanism is to install a unique X.509 certificate for each device. The cloud server solution can be configured to accept only TLS connections from clients with a trusted X.509 certificates. This is referred to as mutual TLS authentication, where the client authenticates the server and the server authenticates the client (device). Using a unique X.509 certificate for each device requires a Certificate Authority (CA) Manager to sign the certificates; with what is known as a (CA root certificate). See our Certificate Management Tool for an introduction to CA Management.

Secure Boot

Once the device is powered, the authenticity and integrity of the device software should be verified using an exchange of cryptographically generated digital signatures. The foundation of trust and validation process is very similar to how we use personal signatures added to legally binding documents to authorize and agree to everyday transactions. The device will verify a cryptographic generated digital signature that is signed by the originating authorized entity to the software image to ensure proper authorization occurs before any software is loaded on the device.

Secure Boot provides a full set of functions that will enable developers to verify firmware upgrades and/or loadable modules. Secure boot can greatly enhance the security of an embedded device by cryptographically verifying that any new software and firmware is authentically (produced by the manufacturer) and has not been unknowingly compromised or maliciously modified.

In general, there are two Secure Boot design methods for an embedded device by provisioning for use with a single or dual firmware upgrade. Each method holds relative pros and cons to the solution and the best selection for a particular application is mainly dependent on the available device memory and total firmware size.

Single Firmware Upgrade: Secure boot is firmware installed during the manufacturing process with software specifically designed for performing a software upgrade. The secure boot firmware contains at a minimum:

- Firmware Upgrade Logic
- Public Certificate
- TLS including Crypto Ciphers
- TCP/IP stack and related Drivers

Pro	Secure boot can be designed to require as little as 40Kb ROM / 15Kb RAM.
Con	Device is not able to operate during the firmware upgrade process.
	Failure will render the device inoperable until the secure boot is able to reconnect and load new firmware.

Dual Firmware Upgrade: The dual firmware upgrade requires the system to store two firmware versions on the device. Secure Boot is integrated into the firmware enabling the upgrade logic to use the TLS and TCP/IP stacks that reside in firmware. The linker produces two versions of the firmware that will execute in lower and upper memory regions respectively. The upgrade logic selects the correct firmware version to load from the online cloud server. The design requires that the initial startup code is able to be read by a persistent data region. It then uses decision criteria logic to know where to jump, based on a given region, i.e. (instruction pointer positioning).

Pro	Device is fully operational during the firmware upgrade process.
	Device will continue to use the most current available firmware in the event of a failure.
Con	Complete firmware payload can be no more than 50% of the available Flash Memory.

It is desirable for devices in field operation to receive hot patches and firmware updates. Manufactures need a method of safe device authentication, which does not consume bandwidth or impair functional aspects. The Secure Boot process offers an intelligent and elegant way to handle firmware upgrades to the Edge Node, while conserving bandwidth, and intermittent connectivity of an embedded device without compromising safety.

Contact: Real Time Logic for information and assistance with secure firmware upgrades and securing Edge Node devices in distributed network architectures:

Real Time Logic LLC
Dana Point, California, USA
Phone: 949-388-1314

General Information: info@realtimelogic.com
Website: https://realtimelogic.com