Implementing Secure Remote Firmware Updates

May 2011

Embedded Systems Conference Silicon Valley 2011
ESC-202

Loren K. Shade
Allegro Software Development Corporation
1740 Massachusetts Avenue
Boxborough, MA 01719
+1 (978) 264-6600
loren@allegrosoft.com

Abstract

The ability to securely update a deployed embedded system can provide many hidden benefits. The most important benefits are service and support, although some vendors have successfully used this capability to expand service offerings and collect valuable usage metrics and configuration dynamics. This presentation provides a framework for incorporating secure remote updates into your embedded designs by employing a standards based networking protocols model. Often thought of as an advanced networking capability, remote provisioning utilizing standards based communication protocols can be easily achieved.

Secure Remote Updates

Embedded systems have become an integral part of the connected world. The proliferation of network-enabled devices is astounding. Network connectivity once thought to only be applicable to desktop computers is now found within many consumer, commercial, industrial and mobile devices. OEM manufacturers have embraced network connectivity and fully integrated it into their products and more importantly found ways to re-vitalize their products and business models. As wired and wireless network enabled embedded systems become more pervasive at home, work and play; Internet connectivity combined with the capability to securely refresh a product have dramatically extended product lifecycles. As an example, game consoles that once had a lifecycle of 1 to 3 years are now deployed with the full intent of a 10+ year life cycle [1]. The same can be said for DVD/Blu-ray players, printers, radios, amplifiers, networking equipment and many more devices.

Although many embedded devices advertise the ability to perform remote updates, it is often the case that proprietary protocols are used to communicate, validate and install updates. This leaves network IT professionals with little assurance that the implemented update scheme is secure [2]. In critical networking environments such as military and defense, medical devices, ER rooms, clinical diagnostic devices or core enterprise networking equipment this can and often presents itself as a significant threat. After an embedded firmware update a device may continue to function properly, although with additional malice intent to monitor, capture and ultimately report specific network traffic to an outside entity. Or worse yet, contain a small segment of software laying dormant until called upon to contribute to a Distributed Denial of Service (DDOS) attack.

Security is often thought of as a technology, when in reality it is process. While cryptographic technology is readily available, security technology needs to be encapsulated in process to be effective [3]. The most sophisticated cryptography is useless if the keys are in the open. In order for remote updates to be effective and safe, security as a technology and a process must encompass the full lifecycle of the product (development, manufacture, distribution, deployment, remote provisioning and retirement).

Before Implementing Remote Updates

Before implementing remote update capabilities in your embedded design, there are a number of topics to consider. This article specifically targets the security and communications issues with remote firmware updates. To fully implement remote updates, there remain several topics that are outside the scope of this article. Rather than turn a blind eye to these issues, the following section highlights topics regarding hardware, software, operating system and operational considerations common to many embedded designs implementing remote updates.

Hardware Support

Many designs today already contain some type of flash. It may seem obvious, however it is important to fully understand the intricate details when working with reprogrammable flash in your design. Without proper planning, design and coordinated hardware and software implementation, performing a remote update can unfortunately render a product non-functional. Here are a few additional items to consider:

- Is there a distinct area of flash or separate boot ROM?
- When writing to flash is it possible to inadvertently erase the entire part?
- What happens when power is pulled in the middle of an update (not an if …. it will happen). How can the embedded device recover?
- Can the design support partial updates or must the entire flash be reprogrammed with a new update?

Operating System Support

There are volumes of articles and whole texts written about embedded operating systems. When implementing remote updates, from an embedded OS perspective, what becomes part of the update? Does the new image include device drivers, just a new or modified task, or must it include a whole new OS, boot code, etc. Linux and many other embedded OS's support dynamic linking and loading natively or with a little slight of hand. The ability to support dynamic linking and loading can potentially enable additional flexibility when implementing remote firmware updates. A few additional items to consider are:

- Must the updates be mapped to specific physical address spaces?
- When originally provisioned, embedded operating systems typically like to know the maximum number of OS related objects (tasks, queues, semaphores, etc.), does the update now violate these initial assumptions?
- Can or should the OS run while an update is being performed, for some embedded systems this may lead to safety related issues?

Operational Considerations

Embedded systems are found in many diverse environments. If enabling remote updates, operational considerations can often be a high priority. Typically, reprogramming in-circuit flash requires more power than standard operation. A wireless sensor network that harvests energy from the environment will likely have different operational considerations or operational power budget than an enterprise network switch or router. One specific study looks at how a series of progressive updates can be implemented to conserve energy in power sensitive applications [4].

A Remote Update Framework

The following presents a generic and flexible framework for implementing secure remote firmware updates. A high level architecture and set of requirements are presented first followed with specific implementation guidelines.

Architecture

Figure 1 illustrates a generic high-level architecture for implementing secure remote firmware updates.

Figure 1 - High Level Remote Update Architecture

There are three (3) main elements that comprise the remote update architecture.

1. ***Embedded Device*** – this is the unit to receive the update.
2. ***Communications Path*** – The *Embedded Device* must have a method for receiving an update. This is dependent on the overall design. In some cases this is an Ethernet connection, while in others it could be a backplane or a combination of a USB memory stick and sneaker-net to deliver the update. For this presentation we have assumed there is an Ethernet connection to a network.

3. ***Firmware Repository*** – this entity maintains a firmware update library for all devices, version, configurations, etc. For purposes of this presentation, we have assumed the *Firmware Repository* is on the same or a reachable network from the embedded device.

Requirements

The following summarizes a set of generic requirements for implementing secure remote firmware updates. Specific application implementations (e.g. Consumer Electronics, medical instrumentation, navigation equipment, banking ATMs, etc.) will likely have additional requirements.

- ***Standards***, the end solution should leverage available communication and security standards.
- ***Authenticate*** a downloaded image is from a trusted source.
- ***Validate*** a downloaded image is complete.
- ***Versatile***, the end solution must support a wide range of deployed network topologies.
- ***Scalable***, the end solution must be able to support potentially hundreds or thousands of devices, versions, product lines, etc. all requesting upgrades at one time.
- If the embedded device fails to authenticate or validate a downloaded image, an appropriate error is set and overall operation is not effected.
- Only original software must be accepted by the embedded system – specifically software must not be successfully downloaded to the embedded system that alters its defined behavior.
- Only authenticated parties may alter data (i.e. parameters) stored in the embedded system.

Authentication and Validation using Standards Based Security

The world of networking is often referred to as an alphabet soup of technology. Protocol implementations and networking standards are referenced by three and four letter acronyms and supplemented with additional numerical designations. Digital signatures are no different. The following discusses how standards based digital signature technology can be used for authentication and validation.

Digital signatures provide a method for an embedded device to authenticate that it has downloaded an image from a known entity and validate the entire image made the transfer. In our case the *Firmware Repository* stores digitally signed binary images. Data that is digitally signed by a sender can not be altered by a third party while in transit without being detected by the intended receiver. Furthermore, the receiver can uniquely identify and authenticate that the sender created the digital image for private consumption. Moreover, the sender can not deny he is the originator of the digital signature.

Asymmetric cryptographic algorithms such as Elliptical Curve Cryptography (ECC) [5] and Rivest Shamir Adelmen (RSA) [5], form the underlying foundation for this capability. In both cases a pair of digital keys, one public (*pk*) and one private (*sk*) are utilized.

A digital signature is calculated as shown in the following diagram. A public key is freely available to anyone and can be transmitted in the clear without comprising security. The private key (*sk*) must be kept secure and is only known to the *Trusted Authority*. In many cases this is a Secure PC that is not connected to a network. Additionally, there are typically well-established procedures for access and use of the Secure PC to generate a digital signature. A message that is digitally signed using the secure private key (*sk*) can be authenticated and validated only with the matching public key (*pk*). Additionally, if a message has been authenticated and validated, the message is said to have come from a know identity and the entire message has been received.

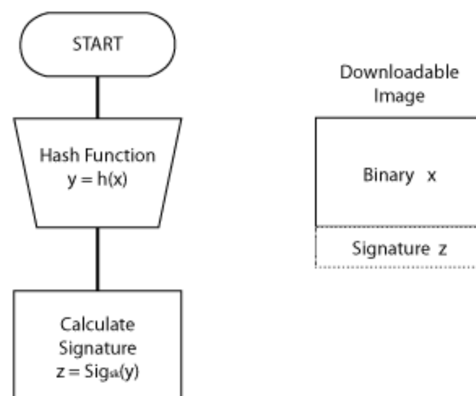

Figure 2 - Generating a Digital Signature

The hash function ($y = h(x)$) is one of a number Secure Hash Algorithms (SHA) published by the National Institute of Standards and Technology [6]. The resulting value of y is then used to generate a unique digital signature z using the secure private key ($z = Sig_{sk}(y)$) [5].

For this presentation, the *Trusted Authority* maintains the secure private key (*sk*) while the public key (*pk*) is installed into the embedded device during manufacture. The overall embedded development process now includes the generation of a digital signature as shown below.

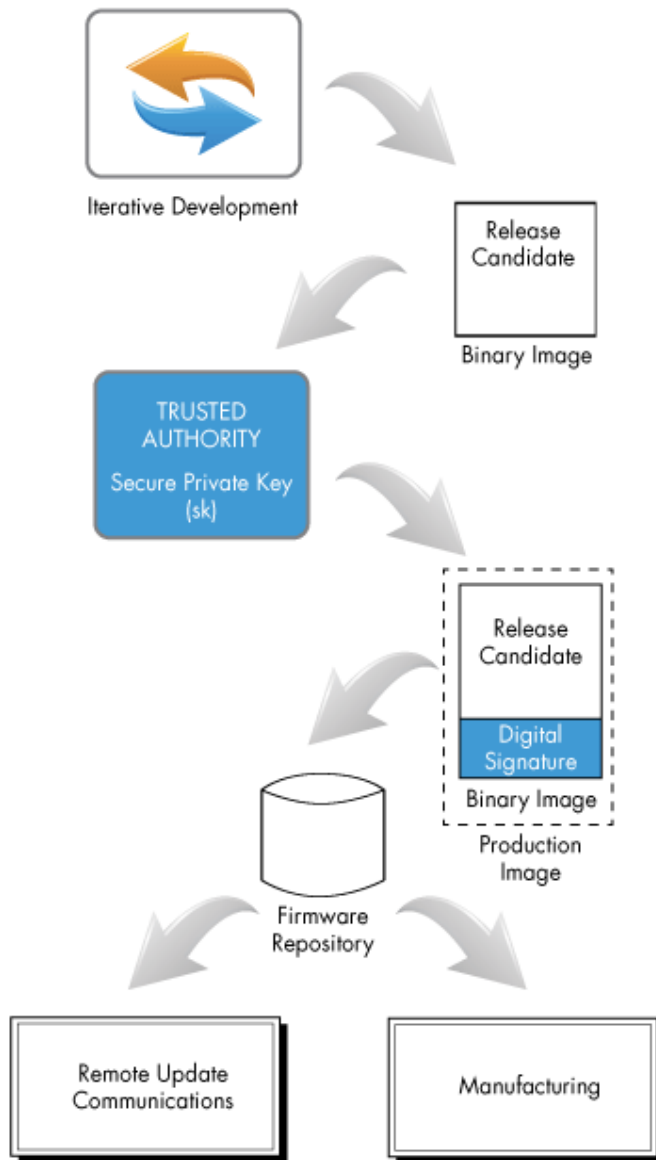

Figure 3 - Development Model supporting Digital Signatures

The iterative process of developing and debugging embedded software eventually yields a final *Release Candidate* that is a binary image. As part of established security processes in place, the binary image is turned over to a *Trusted Authority* to create a unique digital signature based on the secure private key (*sk*). The digital signature is then appended to the binary image. The *Firmware Repository* now stores the Production Image with the digital signature, typically in some type of database. Likewise the *Trusted Authority* public key (*pk*) is installed in the embedded device during manufacture or stored within the program code.

Communication Architecture

There are many potential communications scenarios where you might encounter embedded systems. Figure 4 is one example of a possible scenario.

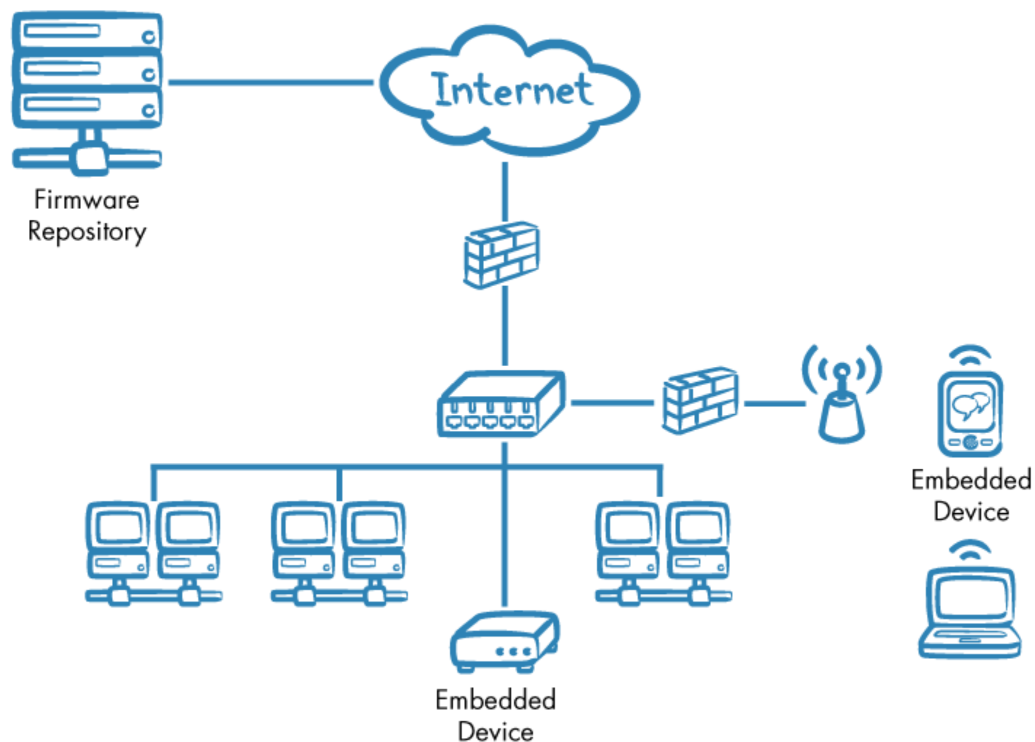

Figure 4 - Sample Communications Architecture

The selection of a communications protocol that can handle a broad range of deployed environments is essential. It must be able to work across long distances as well as in the lab next to a development machine. HTTP is a very versatile and effective transport protocol that is already used in most deployed networks. An HTTP client can be used to reach out to a well know address and query the contents of a *Firmware Repository*. From a security perspective, a user-initiated update via the HTTP client is preferable rather than having the embedded device always "online" to receive a push style update.

Additionally, many devices today employ an embedded HTTP server for device management and configuration (audio amplifiers, printers, routers, etc.). Although an embedded HTTP server is not necessary in this framework, the ability to utilize HTML/XHTML and a remote browser greatly enhances interaction with the embedded device during a user-initiated update.

Further, XML can be utilized in conjunction with HTTP for transferring data in cross platform environments. This enables tremendous flexibility in formatting data transfer messages between the *Firmware Repository* and the *Embedded Device*. With XML there are no worries about endian byte order, binary word lengths, and inability to look at the contents of a file. Later, we will see how important this becomes for organizing *Firmware Repository* content.

The following diagram illustrates an example embedded software architecture for implementing Secure Remote Updates.

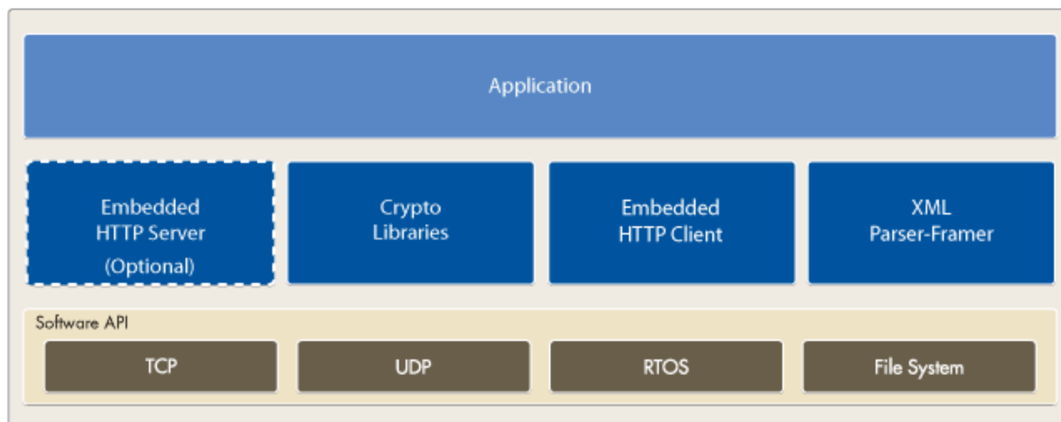

Figure 5 - Example Embedded Software Architecture

From the bottom up, the RTOS, TCP/IP protocol stack and file system are accessible via a software abstraction API. Next, the optional Embedded HTTP Server, Embedded HTTP Client, Crypto Libraries and XML Parser-Framer offer an array of services to all application processes or tasks. Depending on the overall design, the implementation of remote updates could be confined to a single task or require coordination across the entire embedded system.

Communication and Update Logic

Communication models for embedded systems come in all shapes and sizes. For this example, our goal dictates using a simplified, yet fully capable architecture to securely transport remote firmware updates to a deployed *Embedded Device*. The underlying support logic comprises two main components: *Trusted Download* logic, and *Update* logic.

The *Trusted Download* logic (See Figure 6) is referenced multiple times in our example architecture and can be thought of as a subroutine or supporting function for the larger *Update* logic.

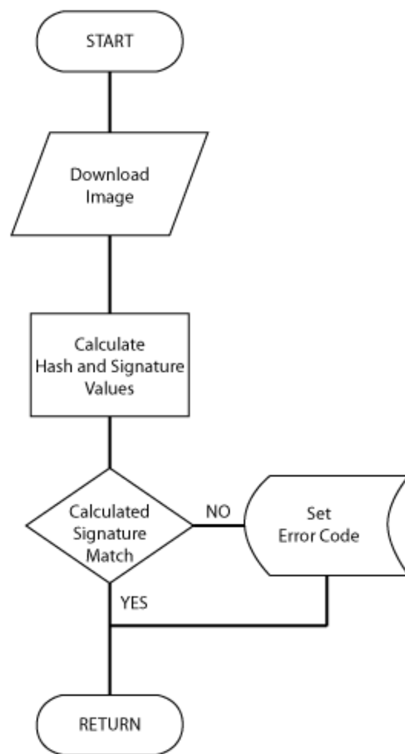

Figure 6 - Trusted Download Logic

The logic requires a single input (the URL of the file to download) and returns two items, an *Error Code* if necessary and the downloaded image. An *Error Code* indicates the downloaded image did not authenticate and validate using the public key (*pk*).

The overall *Update* logic is shown in Figure 7 below. Simply stated, if a *Trusted Download* returns an error indicating either an authentication or validation problem, the *Update* is canceled.

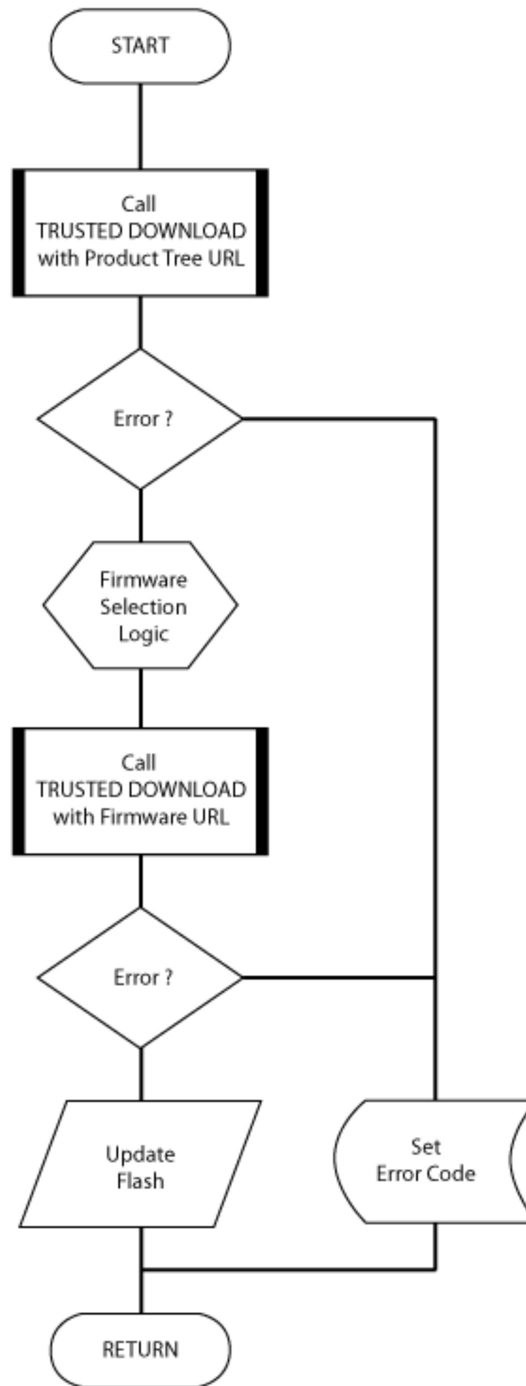

Figure 7 - Update Logic

Organizing the Firmware Repository

With the proliferation of connected embedded devices, it is easy to see that a firmware repository could rapidly become confusing while supporting multiple products with different embedded hardware configurations, software revisions and functionality differentiations. For our example, the following is an XML representation of what might be available for multiple products with multiple production firmware releases. The *Embedded Device* would initially download this XML file to query available firmware releases related to the product.

```
<?xml version="1.0"?>
<Revisions>
 <Product>
  <Name>RDMC 101</Name>
  <Major>1</Major>
  <Minor>50</Minor>
  <Beta>34</Beta>
  <Path>/files/RDMCv150b34.bin</Path>
  <Description>Beta 34 for RDMC 101 v1.5</Description>
 </Product>
 <Product>
  <Name>RDMC 101</Name>
  <Major>1</Major>
  <Minor>50</Minor>
  <Beta>35</Beta>
  <Path>/files/RDMCv150b35.bin</Path>
  <Description>Beta 35 for RDMC 101 v1.5</Description>
 </Product>
<Product>
  <Name>RDMS 202D</Name>
  <Major>1</Major>
  <Minor>02</Minor>
  <Path>/files/rdms.102.mud</Path>
  <Description>Release 1.02</Description>
 </Product>
 <Product>
  <Name>RDMS 202D</Name>
  <Major>1</Major>
  <Minor>03</Minor>
  <Path>/files/rdms.103.mud</Path>
  <Description>Release 1.03</Description>
 </Product>
 <Product>
  <Name>RPLAY 303</Name>
  <Major>1</Major>
  <Minor>00</Minor>
  <Beta>10</Beta>
  <Path>/files/rplay.100b10</Path>
  <Description>Beta 10 for RPLAY 303</Description>
 </Product>
</Revisions>
```

The example above shows multiple products with multiple production firmware images, even *Beta* releases. The XML file also identifies the URLs the *Embedded Device* will use to retrieve a specific production or *Beta* firmware image.

With this information intact, the Embedded Device utilizes some type of logic to select the appropriate firmware to attempt to download. In many cases the *Embedded Device* has either a built in display or On-Screen Device (OSD) to guide a user through the process. However, others do not incorporate displays and utilize an embedded HTTP server for device and configuration management. For products with limited displays (printers, route, power supplies, telephones, etc.) the embedded HTTP server offers an extremely rich user interface and enhances the user experience. Figure 8 shows the user interface on an embedded device that has utilized an embedded HTTP Server and Client in addition to XML to successfully implement remote updates.

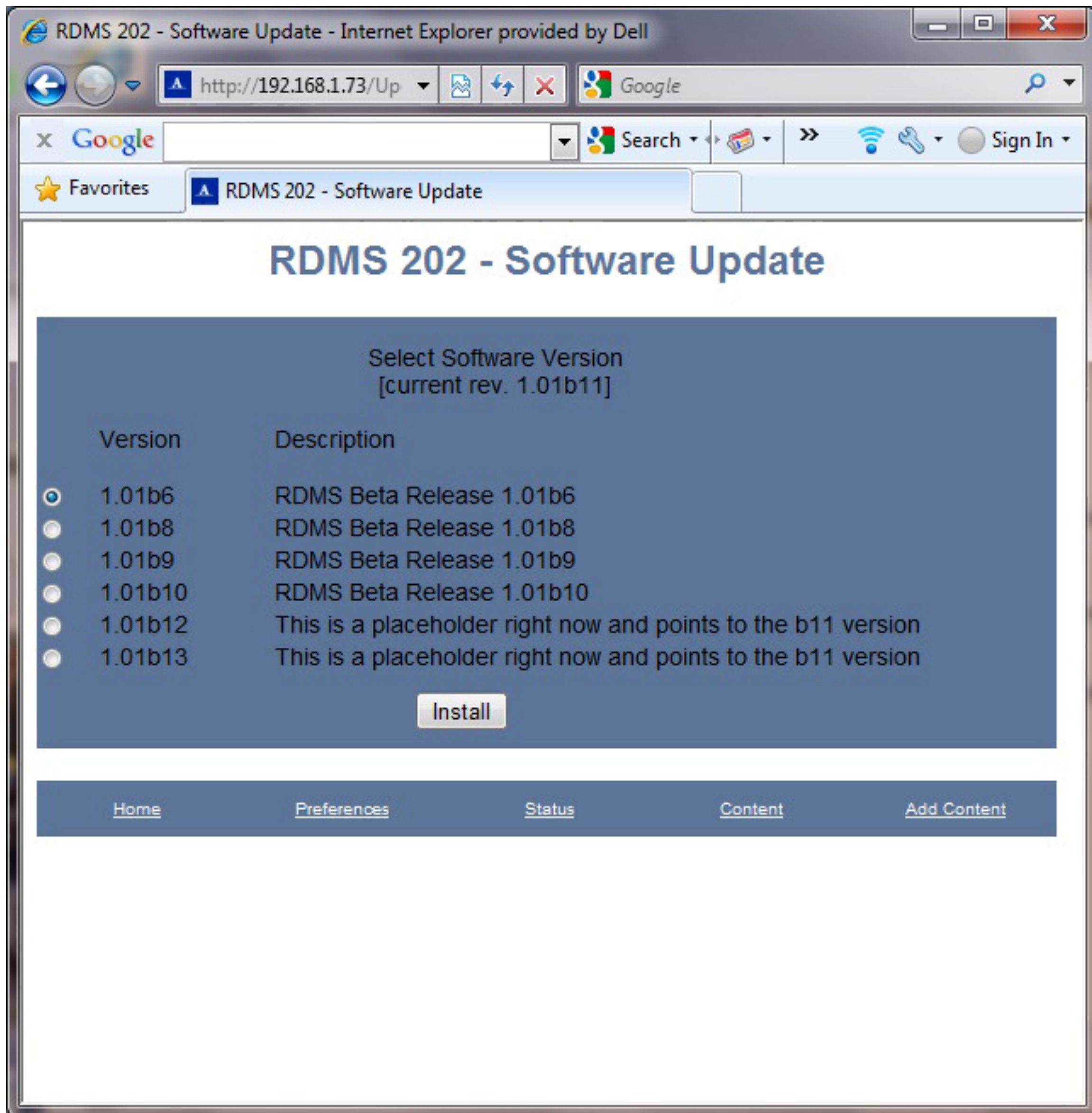

Figure 8 - Example User Interface utilizing an embedded HTTP server, embedded HTTP client, and XML to implement remote updates

Summary

The ability to securely update a deployed embedded device is an increasingly popular and important feature. Updates are often distributed to enhance security, upgrade functionality, fix bugs, add compliance and many more. Manufactures and consumers alike have come to fully understand and embrace the capability. Some have gone as far to fully integrate the capability into their business models by turning on and off product capabilities. However, remotely updating a device's firmware also comes at a risk. Unfortunately, if potential rewards are high enough, some may target the embedded device with malice intent. While not foolproof, the remote update architecture based on secure standard protocols enables a baseline ability to authenticate and validate downloaded firmware images.

Security is often thought of as a technology, when in reality it is process. While cryptographic technology is readily available, security technology must be encapsulated in process to be effective. To this end, for remote updates to be effective and safe, both security processes and technology must encompass the full lifecycle of the product.

Bibliography

[1] **Microsoft Expects 10 Years Lifecycle for Xbox 360 : Microsoft Xbox 360 to Have 10 Years Lifecycle,** Anton Shilov, Xbit Laboratories, June 2009, http://www.xbitlabs.com/news/multimedia/display/20090603230547_Microsoft_Expects_10_Years_Lifecycle_for_Xbox_360.html

[2] **Understanding the Security Challenges of Cloud Computing**, an internet.com Security eBook, a division of QuinStreet, Inc., 2010, http://www.internet.com/ebook/SSOTarget/44781/Understanding-the-Security-Challenges-of-Cloud-Computing?&CCID=20137842203728741&QTR=ZZf201101101654340Za20137842Zg172Zw38Zm510Zc203728741Zs9272ZZ&CLK=2941101200713076 25

[3] **Secrets & Lies: Digital Security in a Networked World**, Bruce Schneier, Wiley Publishing, Inc., 2004, ISBN 0-471-45380-3, pages 1-11, 85-101, 135-150

[4] **Using Software Flashing to Secure Embedded Device Updates**, Andre Weimerskirch and Kai Schramm, EETimes, 4-13-2009, http://www.eetimes.com/design/embedded/4008264/Using-software-flashing-to-secure-embedded-device-updates

[5] **Federal Information Processing Standards (FIPS) Publication 186-3, Digital Signature Standard (DSS)**, National Institute of Standards and Technology, June 2009, http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf

[6] **Federal Information Processing Standards (FIPS) Publication 180-2, Secure Hash Standard**, National Institute of Standards and Technology, August 2002, http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

Further Reading and Related Work

Secure Code Update for Embedded Devices via Proofs of Secure Erasure, Daniele Perito, Gene Tsudik, INRIA Rhones-Alpes, France and University of California, Irvine, USA, 2010, http://eprint.iacr.org/2010/217.pdf

An Introduction to Intel® Active Management Technology, Intel Corporation, 2008, http://www.intel.com/intelpress/articles/An_Introduction_to_Intel_Active_Management_Technology.pdf

Web Services for Management (WS-Management) Specification, Distributed Management Task Force (DMTF), Document Number: DSP0226, 2008-02-12, Version: 1.0.0, http://www.dmtf.org/sites/default/files/standards/documents/DSP0226_1.0.0.pdf

Remote Progressive Update for Code-Paging Firmware in Flash-Based Networked Embedded Systems, Jinsik Kim, University of California, Irvine and Pai H. Chou, National Tsing Hua University, 2009, http://absynth-project.org/Papers/islped09.pdf