

A Framework for Considering Security in Embedded Systems

Eric Uner

The need for security in many embedded systems is not always readily apparent, and too many embedded systems designers are paying too little attention to the subject. I hope to convince you of that in this article, and offer a simple framework for getting started addressing security in your designs, even when you don't think you need to.

I'm sure you already know that embedded systems form the dominant basis of computing today. I like to think of them like insects. Most people feel that the human animal dominates the Earth, when in fact it is the ubiquitous insect. There are nearly 200 million of them for every one of us, and our very survival as a species depends on them. Aside from the ones that are pests, however, insects go about much of their life unnoticed. In the same way, though the average person remains largely unaware of the existence of embedded systems, we all rely completely on these usually small, deceptively simple little insectoid devices to run our most critical services. But all this is changing, isn't it?

First and foremost, the average person is becoming more aware of our work. The "device formerly known as a cellphone" is my favorite example. Already phones are blurring the line between what we normally consider a single-purpose embedded system and a fully capable general-purpose computing platform. These devices have the potential to be used as an electronic wallet, including as a security token for access to other devices and services. My current research, as a matter of fact, at Motorola Labs is centered around increasing the trust level of these and other devices to combat "meaking", or mobile device hacking (mobile phreaking).

News from the Front

Just as serious are the implications of attacks on embedded controllers in industrial environments. A company not far from where I live manufactures small electrical components. On the manufacturing floor they use an industrial controller module, which interfaces to a Windows PC. I looked up the suggested plant floor network diagram from the vendor of these modules, and summarized a key part of it in Figure 1.

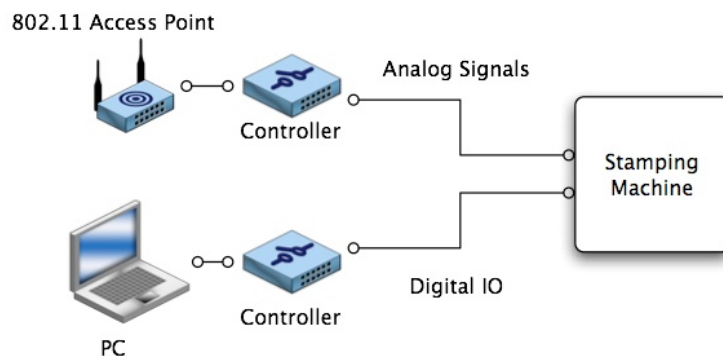


Figure 1. Networked Controller

Does anyone see the problem here? I threw my old laptop in my VW and drove on by the factory. I found three wireless networks in the vicinity. Not having done a complete penetration test or security review of any kind, I certainly can't be sure that there is any connection from the wireless networks to the controllers. Nor can I necessarily equate the presence of a wireless network with being vulnerable. It is possible, however, that the factory floor equipment is susceptible to a direct attack through the local wireless network.

There are, in fact, cases of hackers using the IT infrastructure to ultimately effect embedded controllers. For example, there's the disgruntled employee in Australia who released 250 million tons of raw sewage by attacking a wastewater control system. I don't think I need to go into the more alarming scenarios that could involve chemical and power plants, or other environments where industrial sabotage could be dangerous or disastrous.

All this increased connectivity is a major source of security weaknesses. Embedded systems used to be blind ants, but these days the ants are required to communicate with the beetles, and do it all inside a bee hive. My earlier insect analogy successfully over-extended, I am referring to the fact that today's systems are more inter-connected. Whereas a sensor that used to have a physical readout on it was sufficient ten years ago, today's sensors may be part of an ad-hoc 802.11 wireless network, or they may report their readings back to a central management console over TCP/IP. All these new lines of external communication represent new attack vectors for hackers. The more ways in, the more exploitable your device may be.

So what am I saying? That in addition to knowing about rate monotonic analysis and priority inversion, now we need to be IT experts as well? Actually, that wouldn't help in most cases. Security issues often stem from system-level design issues, and have nothing at all to do with whether or not someone configured a firewall appropriately, or whether or not the device was designed to withstand attacks from the Internet.

It doesn't even always have to do with software. Where I live, atop nearly every intersection with a traffic light there is a little sensor to detect transmissions from "Mobile Infrared Transmitters." MIRTs are used by emergency vehicles to preempt the normal traffic actuated lights and sensors in an attempt to remove any possible traffic backups. "Hobbyists" have created their own MIRTs, which is not only illegal (as far as I can tell) but obviously could create major traffic messes in congested areas. The bottom line is that the sensors have no way to tell a "valid" MIRT in an ambulance from a bogus transmitter in some random person's car.

Of course, there are plenty of cases of "consumer level" hacking against individual devices as well. A major camera vendor once made two versions of their product with similar hardware, but the less expensive version lacked some of the features of the more expensive model. Hackers realized they could "update" the firmware on the cheaper model, effectively getting capabilities they did not pay for.

There are some indications that the problem is getting worse. A report from the United States General Accounting Office (GAO) suggests an "escalation of the risks of cyber attacks against control systems." This is supported by other reports from the British Columbia Institute of Technology, as well as the general increase in information security vulnerabilities reported by CERT and other IT-centric entities.

This actually makes complete sense, and understanding why will help you avoid some security-related pitfalls. It all breaks down into the fact that bigger, more complex systems have more security weaknesses, and that designers and customers of embedded systems have been making some bad assumptions. Let's cover the embedded bloating problem first, then move on to the assumptions part.

The Software Exploitability Equation

It may help to see how different factors affect the “exploitability” of your device by showing it mathematically. So far, I have told you that your device can be more vulnerable if it has to interface with more devices. This can be expressed by showing remote exploitability E_r as growing with the number of interfaces φ , as in Equation 1. Note that ρ is always greater than zero, because physical access always counts as one way in. At some point, someone (albeit maybe only a trusted someone) will have physical access to your device.

$$E_r \approx \rho\varphi, \rho > 0$$

Equation 1

But there are many more factors left to consider. Today's embedded systems have more processing power and are more crammed with features and functions than ever before. There is a lot of data that supports the premise that with more code and more complexity comes more errors. Studies show the average number of latent defects in every thousand lines of code ranges from 3 to 45. Let's go with an average number of 7.5, and put that into the formula in Equation 2, which shows the approximate number of firmware-related exploits E_c in your system. We'll use n for the lines of code in a module, and M for the number of modules. To practice good computer science, we'll also add a factor for average complexity V for each module g .

We're doing all this because there is good data to support that bugs are a significant, though not the largest, source of security weaknesses. So let's add a constant, average probability of a bug becoming an exploitable vulnerability, $P(e)$ into Equation 2.

$$E_c \approx \left(\sum_{g=0}^{g=M} \frac{7.5}{1000} n_g V_g \right) P(e)$$

Equation 2

If we add a weight to each exploitability factor, p for the exploitability through remote connections, and q for the security weaknesses in the code, we get the abridged form of Eric's Software Exploitability Equation in Equation 3. Note that p is much greater than q because inter-connections are typically a greater source for potential exploits than are firmware bugs. This is logically true because with fewer interfaces, there are fewer vectors from which to launch an attack. It is not a product relationship, though, because not all weaknesses are exploitable through all vectors.

$$E = pE_r + qE_c, p \gg q$$

Equation 3

As I said, this is the abridged form, and I left out several factors. Equation 3 illustrates, albeit in an intentionally oversimplified way, that the number of security weaknesses in your implementation increases with the connectivity to, lines of code in, and complexity of your device. Remember, though, that I'm showing you a relationship here between exploitability and its many factors, I am not giving you a metric to judge how much attention you should pay to security. The equation does rightfully imply that a smaller, simpler, isolated system designed to do a particular task should have fewer potential software vulnerabilities than its larger, more connected cousin designed for the identical function. Remember also that I'm still talking about potential *software* vulnerabilities.

Bad Assumptions

The exploitability of the software is only part of the problem. Bad assumptions are more to blame for security problems than are software security weaknesses. I have noticed a pattern of common assumptions, and I am convinced that over half of all our security problems would disappear overnight if we could all change our thinking a bit in just three ways:

Assumption 1) Developers think that embedded systems are inherently more secure.

Nothing could be further than the truth, but I can see why developers feel this way. No one has their source code, the firmware may not even have a commercial RTOS, and even developers on the same team may not know that one picked Flash memory address 0xsomething to store the secret key, and the other decided to put the software version number at 0xsomething-else. In the security space, we call this "security by obscurity," and it never, *never* works. The truth is that hackers have access to the same debugging tools that you do. They can read symbol tables, step through the assembly, etc. Consider also that many attacks don't even need code-level knowledge.

An attack on a Digital Rights management (DRM) system that stores the number of times a song was played in flash memory, for example, may be to simply copy the whole flash bank, and keep overwriting the entire flash image from the stored copy before playing the song again. This resets the system back to a previous state, effectively getting around any play counters or sequence numbers.

For another example, let's look at the memory dump in Figure 2 from an embedded system that manages several user logins. Perhaps I got this memory from attaching a debugger, or perhaps it showed up as padding in a network packet.

```
0x75 0x6e 0x65 0x02    u n e .
0x73 0x6E 0x61 0x69    s n a i
0x6C 0x69 0x6C 0x03    l i l .
0x7A 0x65 0x70 0x70    z e p p
```

Figure 2. Sample Memory Contents

Now, you have no idea what RTOS or even what processor this thing is running. But you know my last name, "Uner." Closer inspection shows the first 32-bit word contains part of my name as ASCII text. It could be a login. What follows is ASCII as well. If I were a hacker, I'd try "snail" for my password. Looking at the pattern, I may also try a user "lil"+something with password "zepp"+something, as they look to have a different user level or some other distinction appearing in the least significant byte. No source code inspection here, or known vulnerability - just simple hacker technique.

Assumption 2) Users assume that embedded systems are more secure.

If you bought a new JTAG wriggler or some other debugging device that had an ethernet port on it, would you go and tell your IT department about it? Probably not. But you may not realize that this little bugger is running embedded Linux, and is now a new target for hackers on your network. Too often do the users of our products, including ourselves, assume that something without a monitor and a keyboard is somehow secure. Most developers would blame the users for any incidents, such as when search engines revealed the images coming from embedded Linux cameras because the cameras responded to fixed URL's.

This is the user's fault for not taking the proper precautions, right? Not if the developer did not explain to the user any assumptions about the environment the device was supposed to operate in (e.g. a closed network). And remember that "explaining" in this case does not mean putting it in the manual that the user will leave in the box with the packing the device came in. It means requiring acknowledgment of this during set up, or putting features inside the device that verify it is operating within an expected environment. A networked controller may, for example, require the user to establish only a set of non-routable IP addresses or MAC addresses that the device will respond to. This data can be spoofed, but the extra configuration step helps convey intent to the user that the controller is not meant to be put on the Internet. We're going to revisit this topic later.

A savvy reader just noticed something wrong. I just suggested adding a feature, right after I got done showing you an equation where more features can lead to more exploitability. Validation code does increase the first factor, $(7.5/1,000)V_{gN_g}$ part, but it also reduces the probability of the bug becoming a vulnerability $P(e)$ and can also minimize the attack vectors rv . So the net effect will be less exploitability.

Assumption 3) Lastly, we often make incorrect assumptions about the security posture requirements.

Security people live by a mantra that a system is secure if the resources it requires to hack a device are more valuable than what the device is trying to protect. In the security space, we call what is worth protecting the "assets." There are temporal characteristics of assets as well as simple value. You could say that a device is secure if the data it is protecting is useless by the time the device has been cracked - like getting a note that I'm on my way to the airport after my plane had touched down at my destination. Makes sense. You wouldn't buy a \$100,000 titanium safe to keep your spare change in (but if you would, call me about an investment opportunity).

The trouble is that this breaks apart as many times as it works. In many cases, you have no idea what the value of the data or operation you need to protect is. A robot arm developer can't always know ahead of time how expensive an attack that causes the device to malfunction is, because it may be deployed in a

factory line where downtime costs \$10,000 an hour or \$10,000 a minute. And who could put a price on a human life when injury or death occurs because of a hacker? I won't dwell on this one, because the assumption does make sense in calculating resources to break cryptographic algorithms when you know what data you're protecting, and indeed in any case where you know the value of your assets a priori.

New Mantras

Before I get into the framework that I suggest using to address all these issues, I need to give you two important facts that are absolutely critical from here on out:

- 1) Security is not all about encryption. It's also about policy, procedure, and implementation. Case in point, encryption based on a secret key is only as good as the policy that controls access to the key.
- 2) Secure code alone does not a secure system make. You must consider security at each phase of the process, from requirements to design to testing, and even support.

If I can convince you to do nothing else, print these out two items and post them on the bathroom doors (everyone eventually goes there). They are my security mantras. They are also fundamental for working within my suggested framework. Speaking of which...

The Framework

What to do about all the issues that I have brought up so far would take me much more than one article to cover, but here is a basic framework within which to start considering the security of your device:

- 1) Environment: Determine the assumptions, threats, and required policies for the environment you are designing the device to operate in.
- 2) Objectives: Determine your device's security objectives. Consider the data (assets) or operation it will protect and which threats from step 1 require countermeasures.
- 3) Requirements: Determine your functional security requirements.

Each component is used in determining the elements of the next, as shown in Figure 3. Working in this hierarchy will prevent unnecessary security requirements, which occur more often than you would think. For example, a device may have a requirement to encrypt all event messages. Many designers just toss encryption in as a substitute for actual security. If you define your environment to be a closed network, or perhaps if your device is running in a car with eight byte messages on a CAN bus, you may be placing unnecessary demands on your device. Unless you can trace a requirement back to something about your environment, you are just adding processing or data transmission overhead.

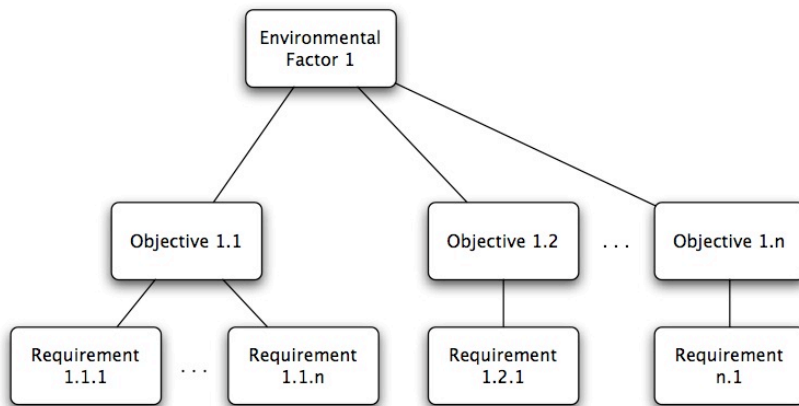


Figure 3. Framework Hierarchy

This framework is essentially a super-condensed form of the Common Criteria. The CC is an international effort to standardize a way to evaluate the security posture of any product, though it is most often applied to IT systems like firewalls or desktop computer software. It is not an evaluation per se, but rather a method for evaluating. I won't go into the details or the specific criteria language, but you can find out more at the CC Website at <http://www.commoncriteriaportal.org/>. It suffices to say that I find keeping even a small subset of the CC in mind, even if you do not intend to submit for an evaluation under the criteria, can result in a more secure, stable, and safer product. For those of you already working under security requirements, you can use this framework around more specific requirements such as FIPS, or environments such as SCADA or the use of a TPM.

A Worked Example

The best way to walk through what I suggest is by example. Lets imagine a fictitious robotic temperature sensor that must open a cooling valve on one of three vents. A basic diagram of the device appears in Figure 5.

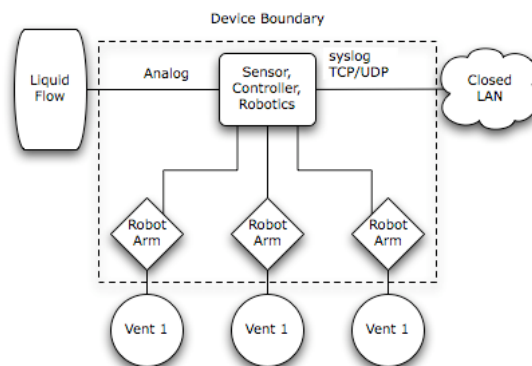


Figure 4. Robotic Sensor

For the purposes of this example, we'll use the general model of the development life cycle: requirements,

design, implementation, then testing. The need to apply the framework at certain cycles is sometimes very subtle, so I'll highlight some of those times as we go.

Environment

Let's first consider the environment. Perfect security is nearly impossible, but an appropriate level of trust and security assurance for a given environment is completely achievable. When you work on a yellow, bumble-bee shaped calculator meant to be sold next to that impulse buy section next to the candy bars in the grocery store, you instinctively don't have the same level of concern about security as you do when thinking about our robot sensor. This instinctive, gut feel you have is because you are rightfully thinking about the environment that the two would be used in. So it all comes down to assumptions about the environment.

From the diagram in Figure 5, we can see that our device is connected to a TCP/IP network in order to send messages to a logging terminal. I think we can assume that the sensor will be used in some kind of industrial environment (as opposed to a kitchen appliance or in a car). That means it's going to be tough to determine cost associated with a security incident, because we don't know anything about where and exactly why the sensor is deployed. We can assume, however, that several people, some of which may be hackers, will have physical access to the device and the network, and that they will be motivated to hack it. If nothing else, our sensor could be used as a "zombie" to send spurious network traffic and disrupt other devices. We need to consider what impact the environment has on us, as well as impact what we have on our environment.

During the requirements phase, take the time to list all of these assumptions. At each phase, take some time to revisit the list and make sure you haven't made any new assumptions. For example, we assumed hostile hackers would have access to the network. If a developer working on the TCP/IP stack fails to think about Denial of Service (DoS) attacks, malformed packets, and the myriad of TCP/IP attacks which are documented all over the Internet, then the developer has made a bad assumption that the IT department would take care of all this. This is in contradiction with our original assumptions.

Speaking of developers, they need to remember these assumptions, and even include them as comments with the source code, at the module level. There is little arguing the virtues of software reuse, but reuse is also a major introduction point for security vulnerabilities. The developer for our sensor may grab some TCP stack or SNMP code that was perfect for it's original product, but has subtleties like requiring a memory manager to zeroize buffers at allocation time. Not doing so could cause sensitive data to leak out over the network, as in the earlier user login and password example. Software reuse in a secure environment is a tricky, delicate thing that merits more consideration than I can give it here. Please keep this in mind.

When we are all done with our list from this step in the framework, we should have all the following factors filled out with as much detail as possible:

- Intended usage - e.g. as part of an industrial assembly line, connected to a closed network
- Possible consequences of attack - e.g. line stops, explosive decompression and injury
- Access policies - e.g. no public access physically, password on device to be managed by IT

- Possible attackers - e.g. disgruntled employees, industrial hackers
- Threat vectors - e.g. hacking the firmware through debug port, network attack
- Assets that require protection – e.g. secret keys, operational data
- Motivations for attack - e.g. industrial espionage or sabotage

Objectives

The objectives are derived from the list we made in considering our environment (recall Figure 4). Although our list of objectives may be more specific and result in a longer list than the one from the previous step, each objective should trace back to something about our environment. Otherwise, why is it an objective? Go through all of the items from the "Environment" step and list the associated security objective. At this point, don't worry about exactly how you are going to accomplish any of this, just stay focused on what it is you need to accomplish. By way of example, we listed "hacking the firmware through the debug port" as a threat. So now it's decision time for our associated objective. We can decide either:

A) The organization operating the sensor is responsible for ensuring that no unauthorized persons have physical access to the sensor before, during, and after installation. Or -

B) The sensor will be designed in such a way as to minimize the possibility of unauthorized modification of the firmware.

"A" seems like kind of a cop-out, and if it feels like you're sweeping something under the rug, it's because you are. "B" will be a more difficult design, to be sure. But it is the only option that matches your security environment.

Another one we listed was "network attack". We need to break this down into more specific objectives such as "the sensor will provide separation between spurious network traffic and primary operation." This innocent looking little sentence says that no matter what someone does on the network, the sensor will keep opening and closing those vents. At first take this may seem like a standard hard real-time objective, but in fact what it means is that when an attacker is sending you malformed network packets, such as TCP packet asking your device to respond to itself ("Land" Attack), you won't let that affect operations. Your objective is to prevent an error in your environment (accidental or malicious) from becoming an error in your device's operations. Since we assumed that the closed network in our environment might have hostile devices on it, such as a PC infected with a virus, we have to be able to handle this kind of network traffic, even if it does not obey specifications for any protocols we are using.

Functional Requirements

Now we move into requirements mode. Just as objectives are derived from the environment, requirements are derived from the objectives. For each of the objectives from the previous step, list an associated requirement, but keep in mind the "environment" component as well.

For example, one of our objectives for the sensor was firmware protection. Our environment included a network and hostile insiders. Our objective was to keep network-based hosts or insiders from altering our

firmware. If we had assumed a closed network where internal security procedures kept even inside attacks away from our device, a simple CRC may have been enough to ensure the device was running the intended code. Since we assumed a higher threat level, however, we need to look at trusted boot features available in processors from vendors including Freescale or Intel. These features help verify that the device boots up running the correct code. If other requirements, such as power consumption or physical constraints, preclude the use of these kinds of processors, we may need to revisit our assumptions. This is why the arrow between the security framework and the requirements in figure 6 goes both ways. If you cannot change your assumptions about the environment, however, the security requirement must take precedence over any other.

Besides the "nobility" of security requirements, another important difference between functional security requirements and traditional requirements is that, fundamentally, not all good top-level security requirements are fully testable. What? Non-testable requirements? Many functional security requirements are inverse requirements, and inverse requirements fundamentally are non-testable.

Consider our objective of being resilient to spurious network traffic. This could lead to the requirement that "the device must not be susceptible to the following attacks: (insert long list of known network attacks like Land, sequence number vulnerabilities, etc.)." Seems simple enough, and there are many tools we can test our implementation against (e.g. Nessus from <http://www.nessus.org>, which runs a battery of individual tests). But what about all the combinations of these attacks in different orders? A DoS attack may temporarily use up resources, for example, and cause the device to be susceptible to an attack where it was not before. If there are 27 known attacks, we have 27! test cases, or about 100,000,000,000,000,000,000,000 tests. Clearly, we cannot test them all.

One way to deal with this is to decompose the security requirements into as many testable elements as we can. For example, we can further decompose our TCP/IP attack requirement to incorporate a specific order to the Nessus tests. We still can't test against all combinations of attacks, and we can't test for weaknesses that we don't know about yet. To do so, we would have to generate all possible network traffic in all possible states of our device at all times, which, in an infinite universe, is an infinite set of test cases. We can, however, test for a "reasonable" subset of all possible attacks, where reasonable means achievable and appropriate to our environment.

Summary

Writing security requirements isn't easy, but following the three steps in the framework can help. If you do this right, you will find that it will impact every aspect of your process, including business processes (see Figure 6).

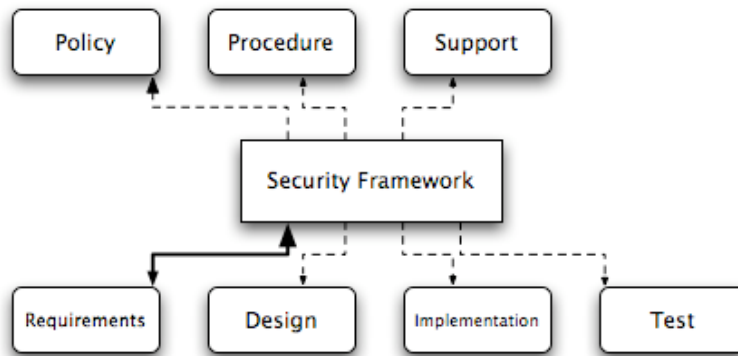


Figure 5. How It All Fits In

Remember to keep in mind some of the gotchas and subtleties I mentioned along the way, as this will save you iterations through your process. Consider your environment, but bear in mind attacks will try and alter aspects of your environment, so spend some considerable time finding all those variables that could affect your device. When you write your objectives, be sure to match every environmental threat with a countermeasure, and remember that countermeasures are not always firmware-based – some of them will be policy or procedure-based. If your objective includes protected access, and your requirement is for a password, be sure you design both policies to deal with users not protecting their passwords, as well as functional requirements that help enforce your policies (by the way, the BBC recently reported a survey where over 70% of people would reveal their computer password in exchange for a bar of chocolate). Lastly, try not to spend too much time on the philosophy of non-testable requirements. Simply do your best to come up with test cases that are suitable for your environment, and that will get you reasonable coverage.

I hope I've got you thinking about security for your device, and I would love to hear how it works out for you. Please send me your comments and your experiences at eric.uner@motorola.com.

This article appeared in three installments on EETimes.com and Embedded.com in September 2005.