WHITEPAPER

Security Awareness

Grant Courville

Security Awareness

Background, where did you come from, where are you going?

The most secure system in the world is powered off, embedded in concrete, and mined with tamper-sensitive explosives. Unfortunately, it is also the most useless system in the world. In order for a system to be useful, it needs to interact with the environment and provide some level of functionality. Historically, embedded systems were mostly islands – that is, they provided their functionality in a stand-alone manner, and weren't connected, or had limited connectivity to other systems. With these kinds of systems, it was relatively easy to analyze their security – the number of ways of getting in and compromising the operation was extremely limited.

Systems today are more connected. The connections come in layers – at the hardware layer, there are serial connections (e.g., RS-232, RS-422, USB, Firewire, I2C, CAN), network connections (e.g., Ethernet), radio (e.g., Wi-Fi, Cellular, NFC, FM), and so on. At the next higher layer, systems can be connected via communications protocols built on top of the hardware layer (e.g., HTTP, SMTP, TCP/IP, UDP, MQTT). Higher and higher levels of protocols exist (e.g., DNS, BGP). A lot of these protocols and connections are incorporated into modern devices as a matter of course, without enough thought given to their security implications. For example, you may be building an embedded device, and include several USB ports. Why? Because you need to share media, and have input devices. A radio receiver may be connected to your infotainment system. Why? Because you want to receive and display RDS/RBDS information such as traffic congestion reports. An Ethernet port provides a host of TCP/IP based services, such as a web interface (HTTP/HTTPS) for maintenance, email alerts (SMTP), remote updates (SFTP/SSH) and so on. Your device may have Wi-Fi, Bluetooth, cellular, and/or NFC connectivity for the benefit / convenience of the end user. And the list goes on. Take a moment and think about all of the connectivity that surrounds you today. Then take another moment and think about all the connectivity that will surround you tomorrow.

What's the problem, and why should I care?

Security goes hand in hand with safety, but there's a fundamental distinction between "safe" and "secure.[1]" Often, you'll hear the question, "is your system safe?" in reference to security. In this whitepaper, we'll specifically reserve the use of the word "safe" to mean "operates correctly and according to specification," and use the word "secure" for dealing with compromise detection and mitigation. So, even though your system is "safe," it may not necessarily be "secure," and vice versa.

In thinking about security, the first thing to understand is motivation. Why would someone want to attack your system? The reasons could fill an entire whitepaper, but in general, they can be grouped into a few categories:

- **Monetary / gain** – the attacker can make money (directly or indirectly)

[1] Interestingly, in some languages the same word is used for both, e.g., in German it's "sicherheit," in Czech it's "bezpečnost."

- **Fame / notoriety** – the attacker can gain fame, notoriety, prestige, "1337 credz"[2], etc.
- **Control** – the attacker can gain control of your system in order to perpetrate a further breach or attack
- **Disruption** – the attacker may wish to control your system in order to disrupt something (e.g., a business, society in general) or protest some policy

The next thing to understand is the damage that can be done. Data breaches affect the global economy. Some examples are consumer credit card information theft (e.g., from high-profile targets such as Home Depot and Target), or account compromise (resulting in stealing money from accounts), website defacement, denial of service attacks, access to sensitive information, including industrial espionage, and in some cases even taking over systems and demanding ransom money to release control.

There's a general correlation between level of connectivity and the motivation for an attack – a system that's not well connected will, in general, be less of a target than a system that is well connected. Obviously, well-connected systems present more points of attack (also called the "attack surface" – how big of an "area" there is for the attacker to attack), but they may also be desirable targets because they may lead to other systems (for example, the system is a gateway to other, less well-connected systems; consider, for example, the Stuxnet[3] worm attack).

In assessing the security of your system, we need to talk about "attack vectors." These are different ways that an attacker can gain access to your system. Ubiquitous communications protocols are one attack vector – the attacker can compromise the software responsible for handling a communication protocol, and be able to get the software to do something that the designer didn't intend. Assuming that the attacker is "in your system," (that is, has some form of control) they may be able to compromise other aspects of your system using other attack vectors (for example, insecure permissions on files, programs with unnecessarily broad privileges).

Surely my device is secure?

So far, we have illustrated some advantages of decoupling the HMI from the back-end logic. Securing your system is a constantly moving target. What may be considered secure today may be compromised tomorrow. The bash shell was found to be vulnerable (shellshock[4]); the secure sockets layer (SSL) protocol implementation had major weaknesses (heartbleed[5]), and so on. Encryption protocols that are secure today (because they are thought to be computationally infeasible to attack) may become vulnerable tomorrow (due to higher speed computers, or weaknesses discovered in the protocol itself). And frankly, our discussion so far doesn't even include the greatest weakness in the security chain – humans! It's usually much easier to compromise the system by way of a human (bribery, extortion, social engineering, ignorance[6], revenge) than it is to crack it using technology.

[2] 1337 is "hacker speak" for "elite", the numbers represent letters ("LEET").

[3] https://en.wikipedia.org/wiki/Stuxnet

[4] https://en.wikipedia.org/wiki/Shellshock_%28software_bug%29

[5] http://heartbleed.com/

[6] For example, having a trivial password, like "password1," or using the same password on multiple systems, or having an easily-guessed recovery question, and so on.

So, your device is probably not “secure” – at least not out-of-the-box. And the more valuable and easy to exploit your device is (whether directly or indirectly), the more likely it is to be attacked.

An excellent place for getting more information (probably more than you ever wanted to know) about attacks is the Blackhat[7] and Defcon[8] sites. Both are associated with yearly conferences, the most well-known being in Las Vegas in the summer; the two conferences are back-to-back and have attendance in the tens of thousands. This is a sobering thought – these are merely the hackers that can afford a plane ticket and hotel, and/or have the desire to travel.

A typical attack cycle exposed at the conference goes like this. A hacker discovers a vulnerability (for example, a Wi-Fi controlled colour-changing light bulb is susceptible to compromise). The company is informed, and either ignores the report (because they don’t care, don’t understand, or just don’t know what to do about it) or downplays the severity (for example, in the light bulb case, the response might be, “So what? You can change the colour of my light from across the street. LOL! A minor annoyance at best.”) The hacker presents at a conference, and sheds light on the actual severity (“Yes, I can control the colour of your light from across the road. I can also get your home’s Wi-Fi password and get into all of your other devices.”). Mass panic ensues.

In the example above, the device itself was used as a gateway – remotely controlling the colour of the light in a room is fun for about 5-15 minutes (depending on your maturity level), but won’t get the attacker anywhere. Using the device to gain access to all of the devices in your home (or office, or factory, etc.), however, is a major compromise.

So the question is, what can you do about it?

So far, we’ve introduced the terms “attack vector” and “attack surface,” we need to look at one more – the “attack tree.” An attack tree is a hierarchical model containing assets or goals (shown as the root of the tree) and enumerates the means to get there (the leaves). Security expert Bruce Schneier explained it well with his example of gaining access to the contents of a safe[9]. The root of the tree (the goal) is access to the safe – but there are multiple ways of getting there:

- Pick the lock,
- Learn the combination,
- Cut open the safe,
- Install it improperly to begin with.

[7] http://www.blackhat.com
[8] http://www.defcon.org
[9] https://www.schneier.com/paper-attacktrees-ddj-ft.html

Each way of getting to the goal has different costs associated with it – whether the costs are money, time, moral considerations (legal vs illegal), is immaterial. The point of the attack tree is to evaluate the most likely means that the attacker will use to get at the goal. Defining the cost in terms of money, if the contents of the safe are worth only $10K, then spending $50K to get at it doesn't represent a viable threat. On the other hand, if all it takes is to buy someone dinner ($200) and socially engineer them to reveal the combination, then the cost is well worth it (net profit $9, 800, plus you got a nice dinner and interesting conversation – "priceless" as the commercials say).

A full discourse on attack trees is well beyond the scope of this whitepaper (the referenced Schneier article is well written, and well worth reading).

Is this going to hurt?

Now that we've seen security from the perspective of the attacker, let's turn around and look at it from the perspective of the defender – that is, the system designer. The first thing to know is that security needs to be designed in (baked-in) from the beginning – it can't just be "sprinkled on" right before final shipment. In this regard, security is much like reliability – it too needs to be designed into the system, rather than just added at the end.

The QNX security model, like most security models in the software industry, can be thought of as analogous to an onion. There are layers of security, just like there are layers to an onion. Generally, no one layer provides sufficient security against an attacker; but together, they may provide enough of a hurdle to discourage all but the most determined ones[10]. Can you make your system entirely attack proof? No. A dedicated, well-funded attacker, given sufficient time and resources, can attack any system. As we saw in the attack tree discussion, however, the question is, is it worth it? By using an onion layer approach, you can increase the cost of the attack past the point where it's worthwhile. To put that into perspective – is a foreign government going to risk an international incident in order to crack your computer in order to read your private email? Probably not, unless they can use that information for greater leverage, such as getting approval on a large purchase contract.

This is a key point – often, a product's security analysis will consider all possible attacks, and fail to take into account motivation, skill level and resources required, and what's actually being protected. Sometimes, this may look like such an overwhelming list that the problem will be seen as too big to solve. On the other hand, sometimes the exact opposite it true – in the email example above, the security analysis might fail to take into account the consequences of the breach, and may thus downplay the security risk.

So, the first step is a proper attack tree – what are you trying to protect, from whom, and what are the costs? Once you have this, you are in a position to prioritize your asset protection, and come up with a design.

[10] Hostile governments are usually cited as attackers capable of compromising almost any system – they operate outside of the law and have access to vast amounts of money and talent. Look at the attack tree, and see where the weakest link is – in the best-protected system, it will usually be the human component.

What can I do?

And speaking of design, a good place to start is with the QNX operating system. QNX uses industry standard security models (e.g., POSIX file permissions, access control lists, encryption, sandboxing, etc.), industry standard analogues (the “abilities” functionality is often compared to FreeBSD’s Capsicum security feature set), as well as QNX-specific enhancements (features tailored to a distributed trust model). The foundational advantage of the QNX operating system though, is its architecture – a true microkernel. Contrast this with a monolithic operating system (where a lot of the functionality is in one address space; the kernel’s) or “pseudo-microkernel” architectures (where drivers are still in the kernel).

Having key software components isolated within their own address spaces helps on a number of fronts. First of all, the components are testable[11] in isolation – there are well-defined interfaces, and only those interfaces need to be protected from attack. Because the component doesn’t share its address space with other components, its attack surface is by definition smaller. That’s because when two (or more) components share their address spaces, any exploits from one component may be used to affect the operation of the other component. When many components share their address spaces, the potential for compromise increases dramatically – each component potentially provides an attack vector into every other component. Secondly, by decoupling the components from each other, the principle of least privilege[12] can be applied. The principle of least privilege basically means that a component should be given only the privileges it needs to perform its job, and only for the period that it needs them. This is hard to do if multiple components all share the same address space – the entire aggregate gets the privileges of the most-privileged component. This is an attacker’s delight – compromise the weakest component, and get the privileges of the most-privileged one! Finally, component decoupling allows effective sandboxing; only the interface presented by the component is public, everything else can be moved to a secure location (whether it’s just another process, or another virtual machine in a hypervisor-managed system, or even a separate, trusted computing module).

Permissions (including access control lists, or ACLs) and privilege are another place where you, the system designer, have a lot of control over the system. Use the standard POSIX permissions to ensure that all assets (files, directories, devices, shared memory) have the appropriate (read: “minimum”) permission set required. ACLs can be used for finer-grained control as required. For example, don’t open up the ability for everyone (“world”) to read, write, and execute arbitrary files. All too often, the assumption in security is “the attacker will never get in” – and the system is designed with that in mind. It’s like a house that has a number of rooms in it, each containing something valuable. All the doors inside the house are wide open (no locks) because the assumption is that the only way the attacker will get in is through the front door. And sure enough, the front door has all kinds of state-of-the-art protection (locks, alarms, surveillance, and so on). Unfortunately, the 2nd story bedroom window is wide open, and there’s a ladder in the garage. Once the attacker is in the house, they can freely go from room to room, and collect all the assets. If, instead, each room had a lock on its door (and the doors were actually locked), this would greatly increase the

[11] Penetration testing (or “pentest”) as opposed to software QA testing.
[12] https://en.wikipedia.org/wiki/Principle_of_least_privilege

amount of time the attacker required to go into each room. It may even have increased the time required to the point where it was infeasible for the attacker to get into all the rooms they wanted. Same thing with your system – you should be designing based on the attacker being in the system, and having root access.

That said, there's no reason to run everything as root (as too many embedded systems tend to do).
Use the POSIX user and group IDs to your advantage; set up users with access to specific resources, and use groups where you need to. This is the idea of locking all the rooms in the house. QNX's abilities model comes in to enhance this. In the standard POSIX model, if you are the root user, you can do anything. This dates back in its present form to the inception of the UNIX operating system (and even before that, in various other embodiments). This is also the reason why, in most attack trees, the overall objective can be boiled down to "obtain root access" – also known as "rooting the device." Because the root user can do anything, most attackers are content with gaining root access as the ultimate goal – from that point, they can do whatever they want. This also explains the popularity of so-called "root kits" – collections of utilities that allow an attacker to obtain root on a given system. QNX allows the creation of "rootless" systems – by using the QNX abilities model, systems can be constructed in which there aren't any processes running as root.