EMBEDDED IN THIN SLICES

The Internet of Things (Part 6)

Internet of Things Security

Bob presents another real example of a device with some security issues related to the Internet of Things. He identifies the security issues and provides some tips for embedded systems designers who want to want to secure their devices.

By Bob Japenga

In December of last year, a "glitch" was discovered in the state of Washington computer system that gave thousands of prisoners early release. Some people blamed out-of-date software. Those of us who design complex systems know that new software can let serious errors get past us. At my company, our motto is: "If it's not tested, it doesn't work." But the corollary to that motto is: "Even if it is tested, it might not work."

Last month we continued our article series on the Internet of Things by looking at actual security failures in a real system. We examined the faults in order to see what we could learn from them so as to not repeat their mistakes. This month we will look at a medical device that had a number of security issues.

Billy Rios is a well-known security expert. In May of 2014, he privately documented to Hospira, a pharmaceutical and medical instrument supplier, a large number of vulnerabilities in their LifeCare PCA Infusion System. In April of 2015, these deficiencies were publicly disclosed by another researcher.

INFUSION SYSTEM PROBLEMS

What are infusion systems? They are used in medical facilities to dispense medications, nutrients, or other fluids into a patient. They consist of an infusion pump and an embedded system to provide a user interface and the controls for its operation. They are supposed to be designed so that no single point of failure will harm the patient. The devices have the capability of delivering lethal doses of some medications.

What's the nature of the problem? The FDA began notifying users in May of 2015 that the LifeCare PCA had serious security issues. Since the units can be programmed remotely through a healthcare facility's Ethernet or wireless network, these security issues opened up significant risks for the users of these devices. To quote the FDA, "An unauthorized user with malicious intent could access the pump remotely and modify the dosage it delivers, which could lead to over- or under-infusion of critical therapies."

What were the security problems? An investigative arm of Homeland Security did further research on the device and identified the following security problems:

Stack Buffer Overflow: The device contained conditions which would allow a stack buffer overflow. This could be exploited to allow execution of an attacker's code on the device.

Improper Authentication: The device allowed unauthenticated remote access with root privileges over telnet

Hardcoded Passwords: The device allowed the same password to authenticate users for all devices

Insufficient Verification of Data Authenticity: The device did not protect against unauthenticated access that modified the operation of the device.

Wi-Fi keys were stored in plain text.

Private keys were stored on the device.

Vulnerable Third-Party Software: The device used a version of a software library that had known vulnerabilities

Externally Induced System Lock-Up: Under a denial of service attack, certain unspecified functionality required a manual reboot to regain operation

Some of these were covered in the last article so we will not cover them again in depth. But let's look at the others more closely.

STACK BUFFER OVERFLOW

In 1999, the common web server, Netscape, was vulnerable to a stack buffer overflow problem. Attackers were able to do malicious things by causing Netscape's stack to overflow when a large input was sent to the server. That was my first exposure to this kind of attack. Just what is it and how can we protect against it?

Data sent over the external network is written to a buffer inside the system. When the size in bytes of external input data is not checked or is incorrect before storing its contents, it's possible that the destination buffer can overflow. If that destination buffer is on the stack, the stack variable overwrites other parts of the stack. Normally, that causes the program to just stop working because the return address is corrupted. But a carefully planned attack overflows the stack in such a way that it creates a valid address in the return address ("Function Return Address" in **Figure 1**) with new code that it has included in the buffer ("Locally Declared Variables in Figure 1). In that way it can take control of the software with its own code. Generally, this requires access to the source code or a lot of trial and error attempts at reverse engineering the code to figure this out.

Let me see if a picture and some poorly written code can illustrate how this happens.

```
ssize_t Listener (int Socket)
{
  char DataInputBuffer[1000];
  ssize_t NumberOfBytes;
  NumberOfBytes=recv(Socket,DataInputBuffer,2000,MSG_WAIT_ALL);
  ProcessTheData(DataInputBuffer);
  return NumberOfBytes;
}
```

LISTING 1
An over-simplified, buggy "Listener" function that takes data from a TCP/IP socket, puts it in DataInputBuffer, and then processes it.

Figure 1 is a representative C stack. The code in **Listing 1** is an extremely over-simplified, buggy `Listener` function that takes data from a TCP/IP socket, puts it in `DataInputBuffer`, and then processes it. `DataInputBuffer` is an automatic variable that is stored on the stack among the locally declared variables (the green area in Figure 1).

The problem is that a typo set the buffer size to 1000 instead of 2000. Most records are less than 1000 so the problem is never discovered during testing. An attacker can send data to the socket that includes code that overwrites the return address ("Function Return Address" in Figure 1) with a pointer to his own code allowing him to do all kinds of mischief and harm.

What can you do to prevent this (other than write perfect code)? Here are three things presented in decreasing order of effectiveness. First, you should digitally bind all memory writes to the size of the buffer. The size parameter of a function like `recv` should be dependent upon the actual buffer size—not

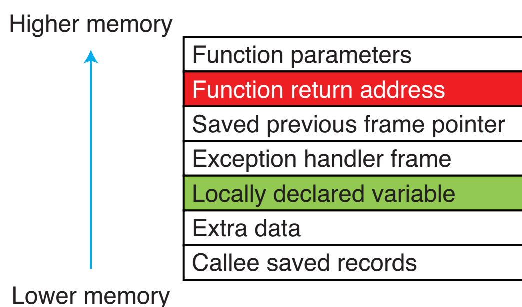

FIGURE 1
Typical C stack structure

COLUMNS

COLUMNS

ABOUT THE AUTHOR

Bob Japenga has been designing embedded systems since 1973. In 1988, along with his best friend, he started MicroTools, which specializes in creating a variety of real-time embedded systems. With a combined embedded systems experience base of more than 200 years, they love to tackle impossible problems together. Bob has been awarded 11 patents in many areas of embedded systems and motion control. You can reach him at rjapenga@microtoolsinc.com.

a magic number like 2000. Second, never use `strcpy` if you are writing in C or any transfer that doesn't bound the number of bytes. Third, you should avoid using stack variables for external input buffers. Using a nonstack buffer overflow to gain rogue control of the software is very difficult. In the LifeCare PCA, the input buffer was on the stack.

But what if the stack buffer overflow is in a library you are using? My best advice is to test for it. If a library function is handling your external input data, you should have test cases that broadly overrun the bounds of the buffers. Don't just test to pass; test to break.

IMPROPER AUTHENTICATION

Another area of deficiency that was discovered was that the LifeCare PCA left ports open without requiring any authentication. We addressed this in detail in the last article. Anyone on the Ethernet or wireless network in the medical facility could do just about anything with the device. This was the core problem with the Infusion System. Without this problem, the hardcoded passwords, the Wi-Fi keys stored in plain text and the private keys issue would not be as serious. I cannot emphasize this enough. This is your first line of defense. Never leave a port open to an unauthenticated access. Make the authentication as strong as possible. And don't give them root access.

Hardcoded Passwords: As was mentioned in the last article, keeping hardcoded passwords secret can be next to impossible. Avoid them at all costs. However, if you leave open unauthenticated ports, you are still vulnerable with algorithmically generated passwords. As we saw with the Jeep last time, the hacker with root access will find the algorithms you are using for generating the password.

Wi-Fi Keys Kept in Plain Text: As with hardcoded passwords, if the hacker can access your device, encrypting and hiding the Wi-Fi keys just slows them down. But it is a good second line of defense—just in case someone breaks in. It will slow them down and require more sophisticated tools.

Insufficient verification of data authenticity: On several systems we have designed, we maintain an MD5 checksum of every static file (files that don't change). This is very useful for detecting when flash file systems start to fail or wear out. This is a topic for another day. We had to upgrade one system's ECC algorithm as some chips were wearing out early. And we detected it by maintaining an MD5 checksum for each file.

But maintaining an independent verification of the data authenticity can also help prevent attackers who have already broken over the wall and are changing files. As the Jeep break-in from last time illustrated, this is only a secondary level of protection. If they already have root access to your system, they can change things (including your MD5 files). But this will prevent other kinds of attacks from altering the operation of your device.

DEFECTIVE LIBRARY

One of the risks facing all of us who design complex systems with loads of libraries is that the "other guy's" library may have security holes in it. Everyone was shocked when we discovered in 2014 that the main open-source library used for using secure sockets (OpenSSL) had a security flaw in it. This affected a number of systems we designed. The LifeCare PCA was using a version of a web server that apparently had security flaws.

How can you design around that? You can't. But we can do two things. Have at least one person in your organization who loves to

circuitcellar.com/ccmaterials

RESOURCES

ICS-CERT, "Hospira LifeCare PCA Infusion System Vulnerabilities (Update B)," US Department of Homeland Security (DHS), 2015, https://ics-cert.us-cert.gov/advisories/ICSA-15-125-01B.

M. Martinez, "3,200 Inmates Mistakenly Release Early in Washington State," CNN.com, 2015, www.cnn.com/2015/12/22/us/washington-state-inmates-early-release-sentencing-errors/.

US Center for Devices and Radiological Health, "Content of Premarket Submissions for Management of Cybersecurity in Medical Devices," 2014, www.fda.gov/downloads/medicaldevices/deviceregulationandguidance/guidancedocuments/ucm356190.pdf.

US Food and Drug Administration, "Vulnerabilities of Hospira LifeCare PCA3 and PCA5 Infusion Pump Systems: FDA Safety Communication," 2015, www.fda.gov/MedicalDevices/Safety/AlertsandNotices/ucm446809.htm.

keep up with these kinds of things. You need to be aware of all of the libraries you are using (no small task) and keep your ears and eyes open for any possible deficiencies that may be uncovered in their latest release.

The other thing you can do is to make sure that there is a way to update the software. This is your final line of defense. The enemy is over the wall. You have poured boiling oil down their backs and they just keep coming. It's time to move to another castle and load the updated version of the library into your system. For devices that are certified, this may require recertification. This is time consuming and costly. It may already be too late if your reputation was seriously harmed in the attack.

DENIAL OF SERVICE ATTACKS

Finally, the LifeCare PCA was susceptible to a Denial of Service (DoS) attack. Just what is that? The most straightforward DoS attacks happen when someone sends more data than the web server can handle. Our devices should not become crippled in their critical functionality with a DoS attack. We cannot prevent a DoS attack. Perhaps our Internet connection will go down during such an attack. But we cannot stop dispensing the drugs. We cannot stop performing our central tasks.

How do we do this? Certainly, we have to test what our device does under a DoS attack. But that is not enough. To do this requires planning at the design stage. We cannot discover this during final test. We need to design our critical tasks (e.g., dispensing drugs) assuming that the network interface is locked up and not available. We need to design our support infrastructure assuming that our network interface is going to get the maximum amount of traffic. And we need to guarantee that it cannot steal the necessary processing time or memory resources needed to accomplish our critical functionally. If necessary, we have to have ways to monitor the network and throttle it so that you can still perform your key functionality—no matter what!

SECURITY MATTERS

Security is a big issue in the embedded world. We are not doing a good job at it. We need to get better at it. Looking at how others got in trouble can help us see our own blind spots. Next time we will look at another embedded device's security issues—but as always—only in thin slices.