

Embedded in Thin Slices

Internet of Things Security (Part 1)

Command Injection

```
if ($_.GET['defaults'] = 1)
{
    $reboot_needed = 1;
    $response = "/apps/cmdxmlin defaults";
}
```

In the first part of his new article series on IoT security, Bob examines command injection. Command injection is a type of attack that involves injecting and executing an unwanted shell command or operating system command into your IoT system.

COLUMNS

By
Bob Japenga

For more than 40 years I used old mercury switch thermostats in my homes. My argument was that they were reliable and did everything I needed. Then about 25 years ago, I upgraded to a digital 7-day programmable thermostat. It was great to be able to set different temperatures for night and day and for when we were away. But I noticed that the house was not as comfortable as it was with the old mercury switch thermostat (**Photo 1**). It would cycle from slightly cool to slightly warm during the coldest months of the year.

My best friend and business partner told me that there were two problems with the new thermostat. First it had only 1 degree of resolution. When the house reached 68°F, the heat would go off. When it reached 67°F the heat would come back on. A simple bang-bang controller. The second problem was that the control would often overshoot and the temperature would actually go to 69°F. My old mercury switch thermostat had a really neat feature called a heat anticipator that prevented this (see sidebar “How the Anticipator Works”). I never knew these simple looking devices had this feature.

As often happens in first- and second-generation electronics, my inexpensive digital thermostat didn’t work as well as the even cheaper electro-mechanical mercury switch. But I was so I cheap I kept it for 25 years. This month, I decided to enter the 21st century

and put in smart a Wi-Fi-enabled thermostat. Guess what? The heating (and cooling) in my home is now more stable than it was with my cheap and very old digital thermostat. And the anticipator algorithm is far more sophisticated than the little variable resistor in the mercury switch thermostat. But that’s not what we are talking about this month. What interested me about this new thermostat was the End User License Agreement (EULA) that I signed up to with this thermostat. Among other things, I agreed to the following:

Not to collect information from the thermostat using an automated software tool

Not to capture the data stream to or from the thermostat (we call this sniffing)

That got me thinking. What about the systems that we design? Would I want to prevent my users from doing this? This is exactly what hackers do maliciously. But many others are out trying to make a name for themselves and demonstrate holes in the security of devices. They show the world how clever they are and how “stupid” we are. But in the process, they do us a great service. They expose our faults. Typically, the good ones report the problem to us and we solve it before it is hacked maliciously.

Just recently, one of our designs was hacked into by a very smart researcher. He

first contacted our customer's customer support but did not get a satisfactory response. Customer support didn't know how to handle this. We, the designers, were never contacted. The result was that he wrote us up in the Industrial Control Systems Cyber Emergency Response Team (ICS-CERT) data base (**Figure 1**) for all the world to know about our "stupidity." Each vulnerability gets a score and our score was very high. "Very high" is not good—it means that the vulnerability is a real serious threat to life on the planet (not really). He did find the security hole by using automated software tools and sniffing our data stream to and from the unit. If our EULA would have had the words from the thermostat's EULA and he was honest, he would not have done this. Hacker's will not pay attention to EULAs, but serious researchers will.

WHAT HAPPENED

At the time of the attack, when new software was sent down from the host, it was signed (preventing someone from sending unauthorized software to our system) but it was not encrypted. This enabled any hacker to get access to the PHP code that handled the HTTP interface. With code in hand, anyone could find an obvious and gaping security hole where we were taking input from the HTTP interface and passing it to the Linux command line interpreter. We did not limit or bound the input in any way. This enabled the user to spoof the host and basically send whatever commands he wished to the Linux shell. He could create new user accounts. He could read our symmetric keys. Ouch!

This is a technique called "command injection" which is a subset of "code injection." How did this happen? We know better. We know what command injection is. We know the dangers of command injection. We know how to stop it and even exploit it. In my article "The Internet of Things (Part 5): IoT Security" (*Circuit Cellar* 307, February 2016) I discussed how we used extremely clever command injection to update some devices remotely that were destined for RMAs. How could this happen? Simple. Our systems are more complex that we are capable of perfectly protecting. But that doesn't mean we don't keep pushing for perfection. With that in mind, let's look a little more in depth at command injection.

In my article "The Internet of Things (Part 8): Security for Web-Enabled Devices" (*Circuit Cellar* 313, August 2016) I talked about code injection specifically with SQL which provided a simple mechanization for injecting unwanted code into the system. Our failure made it even easier to inject commands. Command Injection is a subset of Code Injection. In the Common



INDUSTRIAL CONTROL SYSTEMS CYBER EMERGENCY RESPONSE TEAM

Weakness Enumeration Database referenced in my article "The Internet of Things (Part 7): DoS Attacks with a Twist" (*Circuit Cellar* 311, June 2016) command injection is identified as: "CWE-77: Improper Neutralization of Special Elements used in a Command." Basically, a command injection vulnerability means that external to your IoT device, a shell command or operating system command can be executed that is not intended.

COMMAND INJECTION EXAMPLE

Our security breach didn't happen over Internet traffic. It happened through use of an installer accessible configuration screen from an Ethernet port. When the configuration screen saved the settings, it created a URL with all of the settings as parameters. These were sent to an XML parser. These parameters were visible to someone sniffing the local communication which could then be modified to include any Linux command to be executed. The code snippet is shown in **Listing 1**.

cmdxmlin is an app that we wrote that parses xml input. Notice that we didn't call the PHP shell_exec directly. A sophisticated hacker could still simulate the web browser and append his own commands by utilizing a feature of the shell command called sequential execution. Or the hacker could take advantage of many of the flexible features of the bash shell to perform all sorts of nefarious attacks.

```
if ($_GET['submit'])// when saving config
{
    $keys = array_keys($_POST); // get all the parameters
    $config_string = '';
    foreach ($keys as $key)
    { // Fill config_string with the parameters on the URL
        $config_string .= "\"$key=$_POST[$key]\" ";
        $response = `/apps/cmdxmlin set ${config_string}`;
    }
}
```

LISTING 1

Code snippet that captures configuration settings.

FIGURE 1

The ICS-CERT coordinates control systems-related security incidents and information sharing with Federal, State, and local agencies and organizations, the intelligence community, and private sector constituents, including vendors, owners and operators and international and private sector CERTs.

For detailed article references and additional resources go to: www.circuitcellar.com/article-materials

When I first introduced the topic of IoT security in this magazine, I described it as defending a castle where there are many layers of protection including the moat; the bridge, the entrance gate, the wall and even the boiling oil. I recommended that you use as many layers of protection as possible to mitigate risks. Our security gap could have been closed or mitigated through a number of layers as listed here:

Encrypting all program updates: Without being able to read the code, many forms of code and command injection security breaches can be avoided or at least be physically impossible to find by trial and error. A lot of web services code is interpreted rather than compiled. That means the source code is sent down during an update. Sending your code down unencrypted is just asking for trouble.

Don't provide root access to the web interface: When we released our first IoT device, one of our customer's customers asked us if our applications ran as root. They did because we didn't know better. They refused to buy the product until we changed all of our application code to run as non-root. What would have been easy at the start of the design was difficult after the design was complete. Without root access, the hacker could not have done much damage even with the command injection security hole.

Limit write access to all critical files: This is similar to limiting root access. A careful look at your permissions to your files is in order. The web server app should not be able to write any file without some sort of authentication. Ideally the web server itself

should be limited, but most are not. If your web server doesn't support this, then some sort of watchdog that monitors certain critical files would be helpful.

Bound all input: Input from the web should be limited to be valid only within the narrow range of what you expect. We must recognize our inherent blind spots in this area. It is hard for us to think of all of the possible ranges of input. I love the first example from the classic "The Art of Software Testing" by Glenford Myers called a "Self-Assessment Test." In it, he asks you to create a set of test cases for the following simple program:

The program reads three integer values from an input dialog. The three values represent the lengths of the sides of a triangle. The program displays a message that states whether the triangle is scalene, isosceles or equilateral.

Put down this article and try this now. Write out every distinct test case you can think of. You will have to buy the book to see how many you got right. Or email me your test cases and I will grade it. At the end of the article are three of the fourteen that I missed completely. This self-assessment test demonstrates how difficult it is to plan for every possible input.

Never treat data from the web as a command to the shell: Our engineer assigned to this described his fix as following:

We closed the hole by writing the "save configuration" data to a temporary file, rather than passing them to the command line interpreter. The 'cmdxmlin' program

How the Anticipator Works

Before computers, electro-mechanical temperature controls used to have some pretty cool features. In today's era of very cheap processing, we sometimes miss these kinds of solutions to the problems we face. Mercury switch thermostats have a coil that expands and contracts based on the temperature and rotates the mercury switch which turns on or off the furnace. In addition, there is a resistor in the thermostat through which the current flows that drives the relay that controls the furnace. When the furnace is running, current flows through the resistor. As the resistor warms up, because it is close to the thermostat coil, it warms up the coil (above the ambient temperature). That uncoils the spring and tilts the mercury switch to shut off the furnace before the ambient temperature reaches the setpoint thus preventing overshoot. The resistor is adjustable and if not set right can early or late shut off.



PHOTO 1

Mercury switch thermostats had an electro-mechanical feature called a heat anticipator that worked very efficiently.

that used to take the configuration data as command line parameters now reads the configuration data from /tmp/set_args.txt. This way the configuration items are treated as data and not evaluated by the command line interpreter.


Study every use of calls to the command line interpreter (shell): Be very careful with the passthrough exec system in PHP, the subprocess or os.system in Python, the system in Ruby...and you get the idea.

Push hard to close vulnerabilities: We knew about this security hole in 2012. It was exposed in 2015 and embarrassed the customer in 2016. The customer closed the vulnerability as soon as it was reported. We didn't push hard enough to get this changed as soon as we knew about it. We as developers need to be more insistent about closing security holes.

An open EULA: We have come full circle from our introduction. I would challenge our users to find our security holes. Let them use automated tools to find our holes. Let them sniff our data. Then the EULA would ask them to report it to us before reporting it to a government watchdog agency. As I said earlier, the systems are more complex than we are capable of protecting. We need others help—especially those that want to

make a name for themselves at our expense. It is really to our benefit.

CONCLUSION

When you handle things in thin slices, there is so much to say and so little space to say it. IoT security is absolutely vital. We need to get this as right as possible. And we need each other's help. Software terrorism has just begun. In my opinion we'll only see it increase. Next time, more lessons learned in IoT security. 

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. MicroTools has 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.

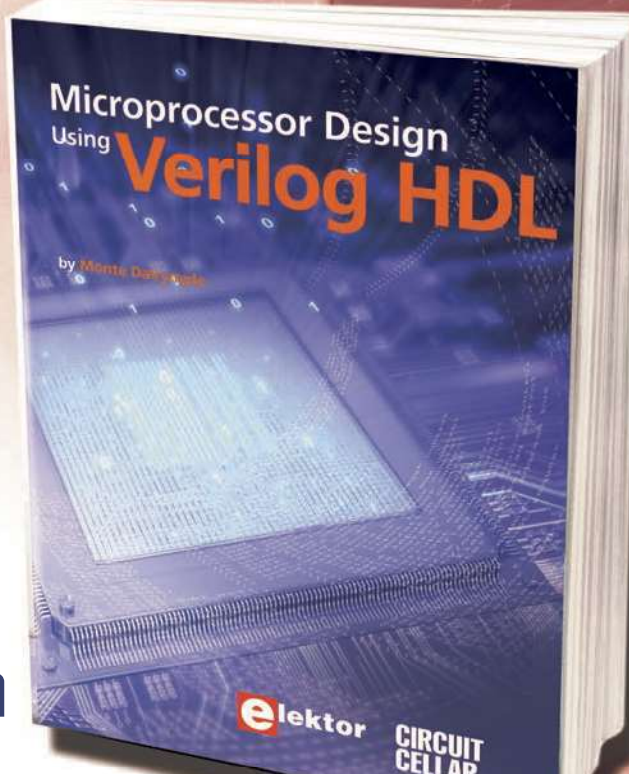


With the right tools designing a microprocessor can be easy.

Okay, maybe not easy, but certainly less complicated. Monte Dalrymple has taken his years of experience designing embedded architecture and microprocessors and compiled his knowledge into one comprehensive guide to processor design in the real world.

Monte demonstrates how Verilog hardware description language (HDL) enables you to depict, simulate, and synthesize an electronic design so you can reduce your workload and increase productivity.

cc-webshop.com



Verilog HDL