Security fundamentals for embedded software

David Kalinsky - March 24, 2012

Click for more content from ESD April 2012

I was preparing for a trip to the Eastern European city where my parents had lived as children. I had never been there. I googled the name of the city, and was quickly led to a story that was surprising and chilling: A high school student there had modified a TV remote control so that it could control the city's tram system--thus converting the urban railways into his own giant model train set. While switching tracks using his infrared gadget, this kid caused trams to derail. Twelve people were injured in one derailment.[1]

Recently, new terms like *Stuxnet* and *Duqu* have entered our lexicon. Embedded systems including those that do supervisory control and data acquisition (SCADA) are under relentless security attacks.

Many embedded software developers feel that embedded systems security should be handled at the systems-engineering level or by the hardware that surrounds their software. And indeed many things can be done at those levels, including:

â€¢ Secure network communication protocols.
â€¢ Firewalls.
â€¢ Data encryption.
â€¢ Authentication of data sources.
â€¢ Hardware-assisted control-flow monitoring.

David Kalinsky

David Kalinsky is a teacher of intensive short courses on embedded systems and software development for professional engineers. One of his popular courses is "Introduction to Software Security for Embedded." His courses are presented regularly in open-class format at technical training providers in international locations such as Munich, Singapore, Stockholm, and Tel-Aviv, as well as in his "home market" of the USA. See www.kalinskyassociates.com.

esc SEE ME SPEAK! March 26 - 29 San Jose, CA

But these traditional techniques aren't enough, as was frighteningly described at last year's DesignCon East 2011 talk "Strong Encryption and Correct Design are Not Enough: Protecting Your Secure System from Side Channel Attacks." The speaker outlined how power consumption measurements, electromagnetic leaks, acoustic emissions, and timing measurements can give attackers information they can use to attack your embedded device.

Clearly then, system-level and hardware defenses are not enough. Most security attacks are known to exploit vulnerabilities within application software. Vulnerabilities are introduced into our embedded systems during software design and development. Since system-level and hardware defenses against security attacks are far from perfect, we need to build a third line of defense by dealing with vulnerabilities in our application software.

While our software line of defense will surely be less than perfect, we need to work on that line of defense with the immediate

ESC classes:

Dave Kalinsky will teach these classes at ESC/DESIGN West in March, 2012:

ESC-101: Software Design for Multicore Systemsâ€”2012 EDITION

8:00 AM to 12:00 PM, March 26

ESC-402: Is Static Code Analysis Ready for Real-time?

3:15 PMâ€"4:15 PM, March 29

objective of reducing the size of the "attack windows" that exist in our software. The very first step in doing this is to try to think like an attacker. Ask how an attacker could exploit your system and your software in order to penetrate it. You might call this a *threat analysis*. Use the results to describe what your software should not do. You might call those *abuse cases*. Use them to plan how to make your software better resist, tolerate or recover from attacks.

Don't forget that our attackers have a big advantage when it comes to embedded systems: Most embedded software has severe execution time constraints, often a mixture of hard real-time and soft real-time tasks. This coaxes us to design application software that is "lean and mean," by reducing to a minimum intensive run-time limit checking and reasonableness checking (for example, invariant assertions) in order to meet timing requirements. Our attackers have no such execution time constraints: They are perfectly happy to spend perhaps weeks or months researching, preparing, and running their attacks--possibly trying the same attack millions of times in the hope that one of those times it might succeed, or possibly trying a different attack each day until one hits an open "attack window."

How can attackers attack via our own software?
Quite often embedded software developers dismiss the issue of embedded software security, saying: "Hey, our device will never connect to the Internet or to any other external communication link. So we're immune to attack." Unfortunately, this is naÃ¯ve and untrue. I'd like to present a counterexample:

Many embedded devices use analog-to-digital-converters (ADCs) for data acquisition. These ADCs may be sampled on a regular timed basis, and the data samples stored by application software in an array. Application software later processes the array of data. But an attacker could view this in a totally different way: "What if I fed the ADC with electrical signals that, when sampled, would be exactly the hexadecimal representation of executable code of a nasty program I could write?" In that way, the attacker could inject some of his software into your computer. No network or Internet needed.

Seems like a lot of work to build an "ADC Code Injector" device just for this purpose. But the attacker might not be just a high-school kid. He might be a big industrial espionage lab, or a large, well-funded team working at the national laboratory of a foreign government.

Now, how could he get your processor to execute his program that he's injected? He might gamble that your software stores the ADC data array on a stack (perhaps using `alloca()` or `malloca()`). If his luck is good, he could cause an array overflow, possibly by toying with the hardware timer that controls the ADC data sampling. A typical normal stack layout is shown in **Figure 1**.

If the attacker succeeds in causing an array overflow, the stack could become corrupted, as shown in **Figure 2**. Note that "return address" was stored on the

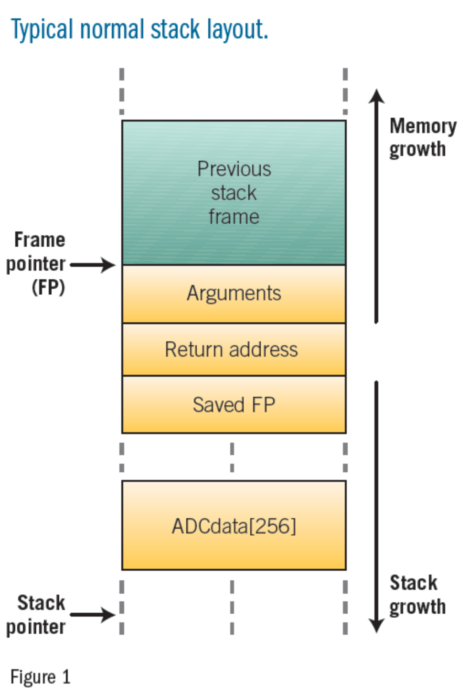

Figure 1: Typical normal stack layout.

Click on image to enlarge.

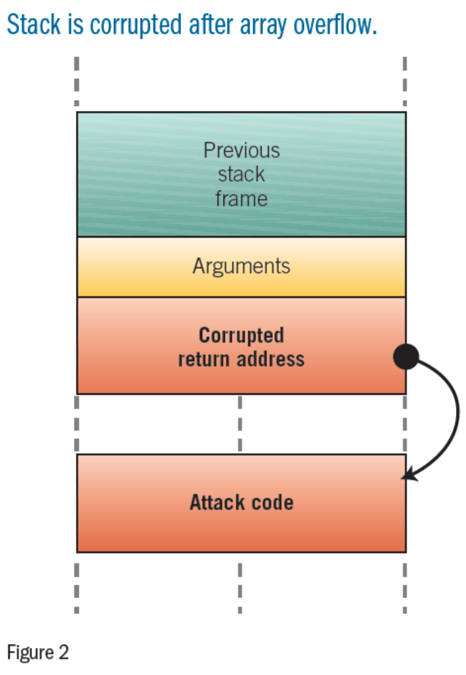

Figure 2: Stack is corrupted after array overflow.

Click on image to enlarge.

stack at a location beyond the end of the array.

If the attacker plans the corruption just right, the overflow will reach the location on the stack where the current return address was stored. This can be used to insert into this stack location a pointer to his own code. As a result, when "Return Address" is used by your code, control will pass to the attacker's code. Suddenly *his* code is executing on your processor, instead of your code.

This is called a *stack smashing* attack. Please note that it was done in this example without an Internet connection, and without a connection to any external communication line.

Of course, it could have been helpful for our attacker to have the source code for your embedded software--as a disgruntled ex-employee might. But I think a patient and resourceful attacker team could develop this kind of attack even without your source code.

Can you think of an easier way for an attacker to develop an attack on your current project? **Page 2, software remedies**

What's a software developer to do?
During embedded systems software design, you can enhance software security by keeping several fundamental ideas in mind:[2]

Mindframe #1: *Distrustful decomposition*
Separate the functionality of your software into *mutually untrusting chunks*, so as to shrink the attack windows into each chunk. (In embedded software, we sometimes call these chunks processes or subsystems or CSCIs.)

Design each chunk under the assumption that other software chunks with which it interacts have been attacked, and it is attacker software rather than normal application software that is running in those interacting chunks. Do not trust the results of interacting chunks. Do not expose your data to other chunks via shared memory. Use orderly inter-process communication mechanisms instead, like operating system message queues, sockets, or TIPC (Transparent Inter-process Communication). Check the content you receive.

As a result of mutually untrusting chunking, your entire system will not be given into the hands of an attacker if any one of its chunks has been compromised.

Mindframe #2: *Privilege separation*
Keep to a minimum the part of your code that executes with special privilege.
Think about it for a moment: If an attacker succeeds in breaking into software that's running at a high level of privilege, immediately your attacker will be operating at a high level of privilege too. That'll give him an extra-wide open "attack window" into your system.

So let's avoid running application software in kernel mode, or master mode, or supervisor mode, or whatever your particular CPU architecture may call it. Leave that mode for operating system use only. Run your application software strictly in user mode. This will enlist your CPU hardware in efforts to limit your software's attack window.

Mindframe #3: *Clear sensitive information*
Clear every reusable resource when freeing it.

Think about it: After you've released the resource, be it a RAM buffer or a software-hardware interface data register, the next user of the very same resource might be an attacker. Embedded system attackers enjoy "phishing" these resources, just as much as Internet attackers enjoy phishing. Wouldn't they be overjoyed to read whatever data you had been working on in the buffer, or to read the data you've just given to hardware for output!

Most resource release services in embedded environments simply mark the newly freed resource as "available." They leave the old information contained in the resource potentially visible to new users, trusting the new user to over-write the old content rather than reading it. This is done since it's much faster than explicitly nulling out the resource.

So when an application is done with a resource, it's up to the application to prepare for releasing the resource by first zeroing out each and every:

- Heap buffer, memory pool buffer, memory partition segment.
- Statically-allocated memory buffer.
- Released stack area.
- Memory cache.
- File in a file system.
- Hardware interface data register, status register, control register. **Page 3, coding techniques**

What can be done during coding?
During embedded systems programming, developers can augment software security by avoiding a number of common software security vulnerabilities.

Some of us would say these are bugs. But I'd like to call them *vulnerabilities* here, to emphasize that some tiny software "defect" that might be too minor even to be called a bug--might be just what an attacker is looking for in order to mount his attack on your embedded system. Small vulnerabilities can open the window to huge attacks.

Vulnerability #1: *Buffer overflow*
Far and away, the most widespread security vulnerability in C-language coding is buffer overflow. It could be as simple as writing into element number 256 of a 256-element array.

Compilers don't always identify out-of-bounds buffer access as a software defect. Yet buffer overflow can lead to more serious consequences, such as stack smashing that was discussed earlier, code injection, or even *arc injection*--by which an attacker changes the control flow of your program by modifying the return address on stack. In arc injection, an attacker doesn't even have to inject any code, and he can jump to an arbitrary function in existing code, or bypass validity checks or assertions.

Here's an example of a buffer overflow attack: An embedded device is required to measure the temperature of water in a swimming pool and to display a histogram showing the percentage of time that the water is at various temperatures. The software developer creates an array of 100 positive integers, each element corresponding to one degree Celsius. Element 0 for 0°C. Element 1 for 1°C, etc. Each time the temperature sensor makes a water temperature measurement, the corresponding element of the array is incremented by 1.

Remember, this is a swimming pool to be used by humans. So the programmer feels safe and secure in designing his temperature array with lots and lots of room beyond the range of water temperature values that a human body can tolerate.

Until one day, an attacker pulls the temperature sensor out of the water and heats it up using a cigarette lighter. As soon as the sensor measures a value greater than 100°C, the histogram update software corrupts an address in memory beyond the end of the temperature array. If there's data there, the attacker will have corrupted the data. If there's machine code there, the attacker will have corrupted the executable software. In either case, this is a damaging attack. Please note (once again) that it was done without an Internet connection, and without a connection to any external communication line. Just a cigarette lighter.

How can we avoid buffer overflows? This vulnerability is so widespread (and so widely sought-after by attackers), that a multipronged approach is best: prevent, detect, and recover. Prevent buffer overflows by careful input validation: check that a temperature sensor is reporting a value within bounds. In our swimming pool example, explicitly check that it's not reporting a temperature that corresponds to ice or to superheated vapor (> 100°C).

Prevent buffer overflows also by avoiding dangerous library functions (like `gets()`) and exercising extra care with others (like `memcpy()`).

Detect buffer overflows by using the idea of "paint": Extend the buffer slightly at both ends. Fill the extension areas with unusual content I call "paint"; for example, a trap instruction in your processor's machine language. Then check the paint repeatedly at run time. If the paint has been over-written, you've detected a buffer overflow.

Vulnerability #2: *Pointer shenanigans*
If an attacker can modify a data pointer, then the attacker can point to wherever he likes and write whatever he likes. If an attacker can over-write a function pointer, the attacker is well on his way to executing his code on your processor.

Vulnerability #3: *Dynamic memory allocation flaws*
It's so easy to write defective code for dynamic memory allocation, that the use of dynamic memory allocation is forbidden in many embedded aerospace and safety-critical systems. Of course, attackers are eager to search out these defects, as they also represent golden opportunities for them to violate the security of an embedded system.

Common flaws include double-freeing, referencing of freed memory, writing to freed memory, zero-length allocations, and buffer overflows (again).

A flaw that is particularly sensitive in embedded software, is neglecting to check the success or failure of a memory allocation request. Some memory allocators will return a zero instead of a pointer to a memory buffer, if they run out of available memory. If application software treats this zero as a pointer, it will then begin writing to what it thinks is a buffer starting at memory address zero.

Many an attacker would be happy to have your software do this. Attackers know that embedded systems tend to be tightly memory constrained. They will try to make a system run out of memory by doing whatever they can to force your memory allocator to allocate more memory than usual--perhaps by leaking memory, possibly leaking it into some code they've injected. They may also try to flood your data-acquisition system with higher than normal volumes of data or higher rates of data--in the hope that the avalanche of data will exhaust your memory capacity. And then … if your software asks for a buffer but neglects to check for allocation failure, it will begin writing a buffer at address zero--trampling upon whatever was there. For example, if your interrupt enable/disable flags happen to be at that address, this could turn off the connection between software and peripheral hardware interfaces. Essentially, this could dis-embed your embedded system.

Vulnerability #4: *Tainted data*
Data entering an embedded system from the outside world must not be trusted. Instead, it must be "sanitized" before use.

This is true for all kinds of data streams as well as even the simplest of integers. Attackers are on the lookout for extreme values that will produce abnormal effects. In particular they're looking for unexpected values, like situations where a digital microprocessor would give a different result from what a human would calculate using pencil and paper. For example, an integer 'i' happens to have the value 2,147,483,647. If I were to add 1 to this value in a back-of-the-envelope calculation, I'd get +2,147,483,648. But if my microprocessor were to execute i++, it would get -2,147,483,648 (a large negative number). It wouldn't take long for a clever attacker to leverage this kind of quirk into some kind of havoc in an embedded system.

A useful technique for data sanitization is called *white listing*. It involves describing all possible valid values for a given piece of data and then writing code that only accepts those values. All unexpected values are viewed as "tainted" and are not used. **Page 4, next steps**

More to explore
Clearly the concepts of software security for embedded systems are not limited to three design "mindframes" and four coding "vulnerabilities." Attackers are a creative bunch, always finding new ways to threaten the security of our software and systems. The story of software security is incessantly changing. One way to keep up is to visit a website called CWE--Common Weakness Enumeration (**http://cwe.mitre.org/**) that keeps a continually updated list of software weaknesses for security.[3] Most of them are as relevant for embedded software as for non-embedded software. I'd start at their "Top 25 Most Dangerous Software Errors" list, which is updated each year.

We've seen several ways that a determined attacker can undermine an embedded system--even one without an Internet connection, and without a connection to any external communication line. We've also seen that embedded software designers and programmers can contribute an additional layer of defense against malicious attacks, beyond what can be done at system-level and in hardware. The embedded software community needs to be alert to the special "mindframes" and "vulnerabilities" involved in this new challenge to our embedded systems.

David Kalinsky *is director of customer education at D. Kalinsky Associates--Technical Training, a provider of intensive short courses on embedded systems and software development for professional engineers. He is a popular lecturer and seminar leader on technologies for embedded software in North America, Europe, and Israel. In recent years, David has built high-tech training programs for a number of Silicon Valley companies, on various real-time operating systems and other aspects of software engineering for the development of real-time and embedded systems. Before that, he was involved in the design of many embedded medical and aerospace systems. David holds a Ph.D. in nuclear physics from Yale University. Contact him through www.kalinskyassociates.com.*

Endnotes

1. *Daily Telegraph* newspaper London UK, "Schoolboy hacks into city's tram system." The city: Lodz, Poland. The date: 11 Jan. 2008.
2. Dougherty, C., K. Sayre, R.C. Seacord, D. Svoboda, and K. Togashi. "Secure Design Patterns," CERT Program, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, Technical Report CMU/SEI-2009-TR-101, ESC-TR-2009-010.
3. CWE's "Common Weakness Enumeration," The MITRE Corporation, Bedford MA, cwe.mitre.org.