print | close

Common Embedded Vulnerabilities, Part 1: Code Injection

Electronic Design
Paul Anderson
Wed, 2015-05-06 14:20

Because many embedded systems have not historically been connected to networks, or since it was reasonable to expect that the devices would operate in a trusted environment, there's been relatively low risk of security vulnerabilities. However, as the Internet of Things era comes upon us, all of that has changed.

Embedded devices are now being built with a high level of networking in mind, and they will be deployed in environments where hostile and resourceful attackers are poised to take advantage of source-code-level security vulnerabilities. Developers of software for embedded systems would be well advised to understand the different kinds of security vulnerabilities, so that they can take measures to defend against hackers.

This article, which focuses on code-injection vulnerabilities, is the first in a series of articles that will discuss how an attacker can exploit defects in an application's source code.

What is Code Injection?

The term "code injection" means that a regular data input to a program can be crafted to contain code, and that the program can be tricked into executing that code. To an attacker, a code-injection defect is golden, because that opens the door to hijack an existing process and then execute whatever code is preferred with the same level of privileges as the original process.

Related

Q&A With GrammaTech

GrammaTech CodeSonar

Klocwork Source Code Analysis

In many embedded systems, processes need to run with the highest privileges available, so a successful code-injection attack can give the attacker complete control over the entire machine. Once in, it becomes possible to steal data, cause the device to malfunction, infect it with a worm, recruit it as a member of a botnet, or even render it permanently inoperable.

Many of the famous computer security incidents over the last few decades have been caused by code-injection vulnerabilities. For example, certain kinds of buffer overruns can be exploited for code injection. Similarly, SQL injection defects are also in that class. A good technical description of the category, along with some examples, can be found here.

As mentioned above, the key aspects of a code-injection vulnerability are the following:

• The program reads data from an input channel.

ıe program treats the data as code and interprets it.

In most cases, it’s unusual for a program to deliberately execute data as code (although this is a requirement of a shell or an interpreter with a read-eval-print interface). However, it’s very common for data to be used to construct an object that is intentionally executed.

For example, a naïve programmer who wishes to issue an SQL query might read a string from the user into a variable (say nameString) and then construct a query in a string as follows:

"SELECT * FROM Names WHERE Id = " + nameString

If the user enters a well-formed name, then all is well. However, a malicious user can easily exploit this by entering a string containing the SQL statement "x;DROP TABLE Users;", which executes the SQL query:

"SELECT * FROM Names WHERE Id = x; DROP TABLE Users;"

The net effect is that one of the tables gets deleted from the database. For an amusing take on this, see this xkcd.

In the above example, the “code” is an SQL query that got injected into the SQL interpreter. An embedded system is unlikely to contain an SQL interpreter (although some certainly do), but plenty of other examples of code-injection vulnerabilities are more likely to appear in embedded code. C programs, for instance, are prone to the format-string vulnerability.

Format-String Vulnerabilities

Almost all C programmers are familiar with the printf family of functions. Roughly speaking, these take a format string followed by a list of other arguments, and that format string is interpreted as a set of instructions for rendering the remaining arguments as strings.

The language for specifying the format is quite complicated and can be tricky. Most users are familiar with the approaches to writing the most commonly used format specifiers, such as those for strings, decimals, and floats (%s, %d, %f), but not many are aware that some other format-string directives can be badly abused.

Before I explain how the code-injection vulnerability can arise, let me point out the most common misuse of the printf function. Unfortunately, some programmers are in the habit of printing strings as follows:

printf(str);

Although most times this will have the desired effect, it’s still wrong because that first argument to printf will be interpreted as a format string. Therefore, if str contains any format specifiers, they will be interpreted as such.

For example, if str contains “%d”, it will interpret the next value in the argument list to printf as an integer and convert it to a string. In this case, there are no more arguments, but the implementation cannot know that. All it knows is that some number of arguments to the function was pushed on the stack. Because no mechanism in the C runtime exists to let it know there are no more arguments, printf will simply pick the next item that happens to be on the stack, interpret that as an integer, and print it. It’s easy to see that this can be used to print an arbitrary amount of information from the stack. If str contained “%d %d %d %d”, for example, then it would print the values of the next four words on the stack.

This is a code-injection security vulnerability in its own right, but one might be forgiven for concluding that the only potential damage is that it’s used to gain access to data on the stack. This can be bad if it contains sensitive data such as a password or a certificate key. However, it could also be a lot worse, because an attacker can write to arbitrary memory addresses.

: format specifier that makes this possible is "%n". Normally, the corresponding argument is a pointer to an :ger. As the format string is being interpreted to build up the result string, when the %n is seen, the number of bytes written so far is placed in the memory location indicated by this pointer. For example, after the printf below has completed, the value in i will be 4:

```
printf("1234%n", &i);
```

Remember that if there are fewer actual arguments to the function than format specifiers, printf will just interpret whatever is on the stack as the arguments. So, if the attacker can control the format string, he can write essentially arbitrary values to stack locations. The stack is where local variables are located, thus making it possible to change their values. If some of those variables are pointers, then this gives the attacker a platform to reach other non-stack addresses in memory.

The really juicy targets are those that give the attacker control over the execution of the program. If one of those local variables is a function pointer, then subsequent calls through that pointer can be to code of the attacker's choice. Or, the attack can overwrite the address of the instruction to where control will be transferred upon return of the function.

I have made some assumptions about stack layout, and for space reasons I can't go into the full details of exactly how these attacks are crafted, but there is plenty of information available for those who wish to know more. The CWE entry is a good starting point.

Avoiding Code Injection

The best way to avoid code injection is through design. It's best if you can use a language where such vulnerabilities can never show up, because your code is then immune by construction. Or design your code to prohibit interfaces that may lead to these kinds of issues.

Unfortunately, in embedded systems, these choices aren't always feasible. Even though C is a highly hazardous language that's riddled with vulnerabilities such as these, it remains the language of choice for many organizations. Given that, developers should be aware of other methods of avoidance.

There are two golden rules for preventing code-injection vulnerabilities:

• Don't interpret data as code if you can avoid it.
• If you can't avoid it, make sure you validate that the data is well formed before using it.

To avoid the format-string vulnerability, the first of these rules is most appropriate. You can write the code as follows:

```
printf("%s", str);
```

This way, the contents of str are treated only as data. This is a no-brainer as long as you can easily find all of the places that should be changed, which can be tricky for large programs, and especially so if you're using libraries of third-party code.

To avoid the SQL injection example from above, you can inspect nameString to make sure it's a single word with no whitespace or semicolons.

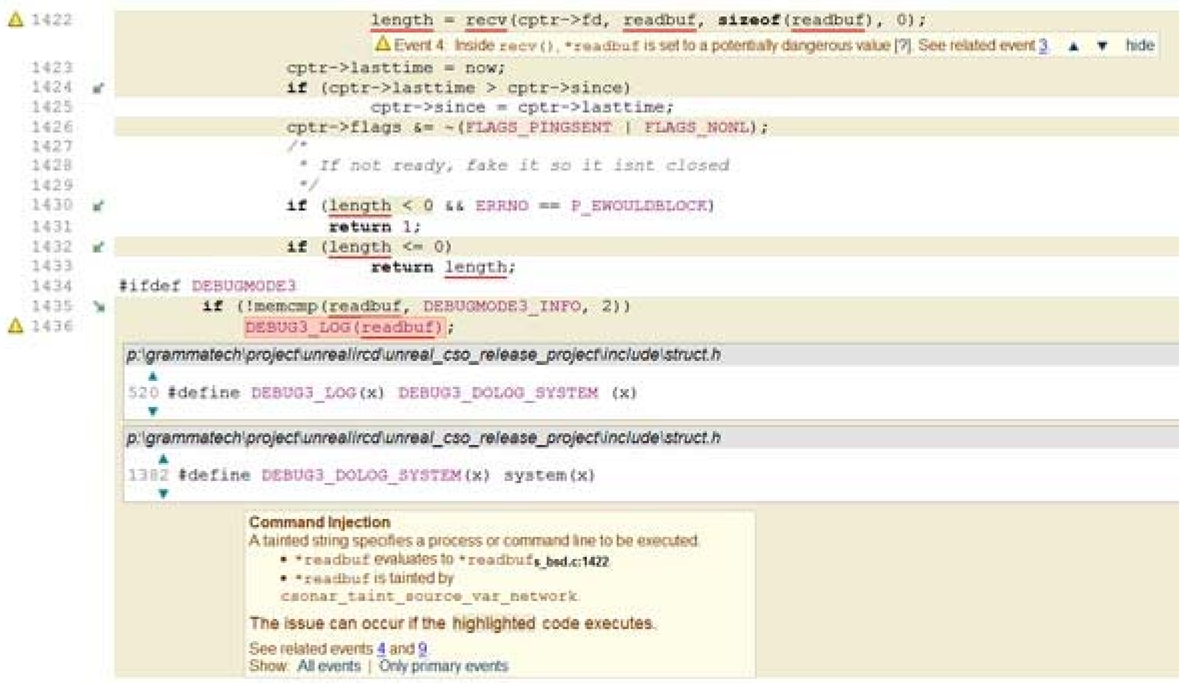
Dynamic Analysis

Testing for these kinds of vulnerabilities can be very difficult. Even tests that achieve very high code coverage can still fail to trigger these problems. Usually, test cases are constructed to validate functionality under normal umstances. When testing for security vulnerabilities, the tester must adopt the mindset of a determined and enious attacker. Techniques like fuzz testing can be useful for searching for particular kinds of code-injection defects, such as places where data is being used as format strings. However, that technique is typically too random to be highly reliable.

Static Analysis

Static analysis can be very effective at finding code-injection vulnerabilities. Note that early-generation static-analysis tools (such as lint and its immediate descendants) are weak at finding these because a whole-program, path-sensitive analysis is needed in order to be precise. The advanced static-analysis tools that have emerged in recent years are more effective. Vendors of these tools have accumulated a lot of experience about which interfaces are hazardous, and developed a knowledge base of what to look for, and how to do so effectively, without drowning the user in a sea of false positive results.

The key technique we use to do this is called “taint analysis,” or sometimes “hazardous information flow analysis.” These tools work by first identifying sources of potentially risky data, and by tracking how that information flows through the code to locations where it’s being used without having been validated. The best tools allow you to also visualize the flow.



```
                length = recv(cptr->fd, readbuf, sizeof(readbuf), 0);
                Event 4: Inside recv(), *readbuf is set to a potentially dangerous value [?]. See related event 3   hide
        cptr->lasttime = now;
        if (cptr->lasttime > cptr->since)
                cptr->since = cptr->lasttime;
        cptr->flags &= ~(FLAGS_PINGSENT | FLAGS_NONL);
        /*
         * If not ready, fake it so it isnt closed
         */
        if (length < 0 && ERRNO == P_EWOULDBLOCK)
            return 1;
        if (length <= 0)
                return length;
#ifdef DEBUGMODE3
        if (!memcmp(readbuf, DEBUGMODE3_INFO, 2))
            DEBUG3_LOG(readbuf);

p:\grammatech\project\unreal\ircd\unreal_cso_release_project\include\struct.h
#define DEBUG3_LOG(x) DEBUG3_DOLOG_SYSTEM (x)

p:\grammatech\project\unreal\ircd\unreal_cso_release_project\include\struct.h
#define DEBUG3_DOLOG_SYSTEM(x) system(x)
```

Command Injection
A tainted string specifies a process or command line to be executed.
- *readbuf evaluates to *readbufs_bsd.c:1422
- *readbuf is tainted by csonar_taint_source_var_network

The issue can occur if the highlighted code executes.

See related events 4 and 9
Show: All events | Only primary events

To illustrate its effectiveness, I offer the following example. In a program that implemented an Internet Relay Chat (IRC) server, a command injection vulnerability was found *(see the figure)*. This was in a location where unsanitized data from the network was being passed as input to the system() function. Shown is a screenshot from CodeSonar (the static-analysis tool I work on).

The call to system() in this case was buried two levels deep in the body of a macro whose name implied it was responsible for some benign logging of activity. Once the macros were expanded, however, it became clear that this was a backdoor that was maliciously and deliberately inserted and disguised as something harmless. When deployed, this allowed anyone who knew the code to run code on the host computer with the same privileges as the server.

Conclusion

le-injection vulnerabilities are extremely dangerous security issues because they can allow an attacker to rupt the program and sometimes even take complete control of the computer. Developers who care about making sure their embedded code is secure for use in a potentially hostile networked environment should try hard to eliminate these vulnerabilities early in the development cycle. They can be difficult to find using traditional testing techniques, so stringent code inspections and use of advanced static-analysis tools is highly recommended.

Source URL: http://electronicdesign.com/embedded/common-embedded-vulnerabilities-part-1-code-injection