Reducing the Attack Surface

**SOMETIMES YOU
CAN GIVE THE
MONKEY A LESS
DANGEROUS CLUB**

who is
KV?

click for video

Dear KV,

My group is working on a piece of software that has several debugging features. The code will be part of a motor control system when we're done. One of the features we've added is a small CLI (command-line interpreter) that we can use to change the parameters that control the motor and to see what effect these changes have on power consumption, heat, and other values tracked by our software. We first added the CLI just for our small group, but now we have found that both the QA and factory teams have come to depend on having it in place to do testing (QA) and preshipping checks in the factory.

As you might imagine, the ability to change the parameters of the motor once it's shipped could lead to problems such as overheating, as well as a catastrophic failure of the motor. Even though our product isn't meant to be some sort of IoT (Internet of things) device, we do have a network connection available on our higher-end products so that the performance and wear on our motors can be measured over a network in the field.

I've told the QA and factory teams that there is no way we should leave this code in our shipping product because of the risks that the code would pose if an attacker could access it. They say the code is now too important to the product and have asked us to secure access to it in some way. Networked access to the device is provided only over

a TLS (Transport Layer Security) link, and management now thinks we ought to provide a secure shell link to the CLI as well. Personally, I would rather just rip out all this code and pretend it never existed. Is there a middle path that will make the system secure but allow the QA and factory teams to have what they are now demanding?

Dear CLI,

See earlier editions of KV to find my comments on prototypes, because they're relevant here (e.g., Beautiful Code Exists, If You Know Where to Look; <http://queue.acm.org/detail.cfm?id=1454458>). The problem is that once you give a monkey a club, he's going to bash in your brains with it if you try to take it away from him. The CLI you and your team have created is a nasty-looking club, and I would hate to get whacked with it.

The best way to reduce the attack surface of a piece of software is to remove any unnecessary code. Since you now have two teams demanding that you leave the code in, it's probably time to think about making two different versions of your binary. The application sounds like it's an embedded system, so I'll guess that it's written in C and take it from there.

The traditional way to include or exclude code features in C is via the prolific use of the `#define/#ifdef/#endif` preprocessor macros and abuse of makefiles. The first thing to do is to split the CLI functions into two sets: readers and writers. The readers are all the functions that return values from the system, such as motor speed and temperature. The writers are all the functions that allow someone to modify the system's parameters. The CLI

**CLI_WRITER
should be
defined only
via the build
system and
never as a define
in the code.**

itself, including all the command-line editing and history functions, is its own piece of code. Each module is kept under an `#if/#endif` pair such as this:

```
if defined(CLI_WRITER)
/* XXX Dangerous Code, do not ship! */
endif
```

CLI_WRITER should be defined only via the build system and never as a define in the code. You are liable to forget that you defined the value during some of your own testing or debugging, and commit your fixed code with the value defined.

With the code thus segmented, you now define two versions of your binary: TEST and SHIP. The TEST version has all the code, including the readers, the writers, and the CLI itself. The TEST version can also have any and all debug functions that the QA and factory teams want to have prior to shipping.

The SHIP version of the code has none of the debug features and only the reader module for the CLI. I'd say it goes without saying that the CLI must not have a `system()`-like function that allows the execution of arbitrary code. I would love to believe that could go without saying, but, guess what, I said it because I've seen too many systems with a "secure" CLI that contains a `system()` function.

If at all possible, you should link all of your binaries statically, without using dynamic libraries or KLDs (kernel-loadable modules). Allowing for dynamically loadable code has two downsides. The first downside is that some monkey can come along later and re-add your writer functions to the system. The second downside is that you lose your

protection against someone accidentally leaving in a call to a writer function when they should not. In a statically linked binary, all symbol references must be resolved during the linking phase. If someone leaves a stray call to a writer function somewhere in the code, this error will be flagged at link time, a final binary will not be produced, and you will not be able to ship a polluted binary accidentally.

In each of the reader, writer, and CLI modules you should place a specially named symbol that will remain in the final binary. Pick obvious names such as `cli_reader_mod`, `cli_writer_mod`, and `cli_mod`. Before any binary is shipped, either placed into a device at the factory or put up on the company's software-update server, a release script must be run to make sure that the `cli_writer_mod` symbol is not present in the shipping binary. The release script could look for a known function in the writer module, but programmers often like to change the names of functions, so adding a special symbol is easier and it is unlikely to change. For double extra bonus points, you can also have a version in each module to make debugging in the field somewhat easier. Do not add the version to the `cli_foo_mod` symbols. Those symbol names are inviolate and should remain with the modules for their entire usable lifetime.

I mentioned the build system as well. With the code now split into separate modules, you can easily make a build target for TEST and SHIP binaries. It's the build system that will define things such as `CLI_WRITER` at build time to add the module to the TEST binary. Your CI (continuous integration) system (you are using a CI system, right?!) can now pop out binaries of both types and even run the release script that tests for the presence of the correct modules in each release.

When you can't take the club away, sometimes you can give the monkey a less dangerous club. Putting the dangerous debug code under `#ifdef` protection, splitting the code into its own modules, and modifying the build and release system to help you make sure you don't ship the

wrong thing are just some of the ways to shrink the monkey's club.

KV

Related articles

➡ Porting with Autotools

Using tools such as Automake and Autoconf with preexisting code bases can be a major hassle.

Kode Vicious

<http://queue.acm.org/detail.cfm?id=1952748>

➡ Playing for Keeps

Will security threats bring an end to general-purpose computing?

Daniel E. Geer, Verdasys

<http://queue.acm.org/detail.cfm?id=1180193>

➡ Security Problem Solved?

Solutions to many of our security problems already exist, so why are we still so vulnerable?

John Viega, Secure Software

<http://queue.acm.org/detail.cfm?id=1071728>

Kode Vicious, known to mere mortals as George V. Neville-Neil, works on networking and operating-system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code [OK, maybe not that last one]. He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix

Association, and IEEE. Neville-Neil is the co-author with Marshall Kirk McKusick and Robert N. M. Watson of The Design and Implementation of the FreeBSD Operating System (second edition). He is an avid bicyclist and traveler who currently lives in New York City.

Copyright © 2017 held by owner/author. Publication rights licensed to ACM.