ARTICLE:

FIRMWARE FORENSICS: BEST PRACTICES IN EMBEDDED SOFTWARE SOURCE CODE DISCOVERY

By **Michael Barr**

Software has become ubiquitous, embedded, as it is, into the fabric of our lives in literally billions of new (non-computer) products per year, from microwave ovens to electronic throttle controls. When products controlled by software are in the subject of litigation, whether for infringement of intellectual property rights or product liability, it is imperative to analyze the embedded software (also known as firmware) properly and thoroughly. This article enumerates five best practices for embedded software source code discovery and the rationale for each.

In February 2011, the U.S. government's National Highway Traffic Safety Administration and a team from NASA's Engineering and Safety Center published reports of their joint investigation into the causes of unintended acceleration in Toyota vehicles. While NHTSA led the overall effort and examined recall records, accident reports, and complaint statistics, the more technically focused team from NASA performed reviews of the electronics and embedded software at the heart of Toyota's "electronic throttle control subsystem" (ETCS).[1]

These reports are very interesting in what they have to say about the quality of Toyota's firmware and NASA's review of the same. However, of greater significance is what they are not able to say about unintended acceleration. It appears that NASA did not follow a number of best practices for reviewing embedded software source code that might have identified useful evidence. In brief, NASA failed to find a firmware cause of unintended acceleration – but their review also fails to rule out firmware causes entirely.

This article describes a set of five recommended practices for firmware source code review that are based on the experiences of the author as an embedded software developer and as an expert witness. Each of the recommendations will consider what more could have been done to determine whether Toyota's ETCS firmware played a role in any of the examples of unintended acceleration. The five recommended practices are: (1) ask for the list of faults[2]; (2) insist on an executable program; (3) reproduce the software build tools setup; (4) request the version control repository; and (5) remember that the hardware is also significant and should not be overlooked. The relative value and importance of the individual practices will vary by the type of litigation.

Ask for the list of faults

Depending on the facts at issue, litigation involving embedded software will require an expert review of the source code. The source code should be requested early in the process of discovery. Owners of source code tend to strenuously resist such applications, but procedures limiting access to the source code to only certain named and pre-approved experts and only under physical security (often a non-networked computer with no printer or removable storage in a locked room) tend to be agreed upon or ordered by a judge.

Software development organizations commonly keep additional records that may prove more important or useful than a mere copy of the source code. Any reasonably thorough software team will maintain a fault list (also known as a defect database) describing most or all of the problems observed in the software,

1 *Redacted public versions of the official reports from each agency, together with a number of related documents, can be found at http://www.nhtsa.gov/UA.*

2 *In the industry, faults are commonly called 'bugs'.*

together with the current status of that fault (for instance, "fixed in v2.2" or "still under investigation"). The list of faults fixed and known – or the company's lack of such a list – is germane to issues of software quality. Thus the fault list should be routinely requested and supplied in discovery.[3]

Very nearly every piece of software ever written has defects, both known and unknown. Thus the fault list provides helpful guidance to a reviewer of the source code. Often, for example, faults cluster in specific source files in need of major rework. To ignore the company's own records of known faults, as the NASA reviewers apparently did, is to examine a constitution without considering the historical reasons for the adoption of each section and amendment. Indeed, a simple search of the text in Toyota's fault list for the terms "stuck" and "fuel valve" might yet provide some useful information about unintended acceleration.

Insist on an executable program

In software parlance, the "executable" program is the binary version of the program that is actually executed in the product. The machine-readable executable is constructed from a set of human-readable source code files using software build tools such as compilers and linkers. It is important to recognize that one set of source code files may be capable of producing multiple executables, based on tool configuration and options.

Though not human-readable, an executable program may provide valuable information to an expert reviewer. For example, one common technique is to extract the human-readable "strings" within the executable. The strings in an executable program include information such as on-screen messages to the user (e.g., "Press the '?' button for help."). In the experience of the author, in a copyright infringement case, several strings in the defendant's executable helpfully contained a phrase similar to "Copyright Plaintiff"!

It may also be possible to reverse engineer or disassemble an executable file into a more human-readable form. Disassembly could be important in cases of alleged patent infringement, for example, where what looks like an infringement of a method claim in the source code might be unused code or not actually part of the executable in the product as used by customers.

Sometimes it is easy to extract the executable directly from the product for expert examination – in which case the expert should engage in this step. For instance, software running on Microsoft Windows consists of an executable file with the extension .EXE, which is easily extracted. However, the executable programs in most embedded systems are difficult, at best, to extract.[4] Extraction of Toyota's ETCS firmware might not be physically possible. Thus the legal team should insist on production of the executable(s) actually used by the relevant customers.

Reproduce the software build tools setup

The dichotomy between source code and executable code, and the inability of even most software experts to make much sense of binary code can create problems in the factual landscape of litigation. For example, presume that the source code produced by Toyota was inadvertently incomplete in that it was missing two or three source code files. Even an expert reviewer looking at the source code might not know about the absent files. For example, if the fault the expert is looking for is related to fuel valve control and the code related to that subject does not reference the missing files, the reviewer may not notice their absence. No expert can spot a fault in a missing file.

Fortunately, there is a reliable way for an expert to confirm that she has been provided with all of the source code. The objective is simply stated: reproduce the software build tools setup and compile the produced source code. To do this it is necessary to have a copy of the development team's detailed build settings, such as make files, preprocessor defines, and linker control files. If the build process completes and produces an executable, it is certain the other party has provided a complete copy of the source code.[5]

Furthermore, if the executable as built matches the executable as produced (actually, ideally, the executable as extracted from the product) bit by binary bit, it is certain that the other party has provided a true and correct version of the source code. Unfortunately, trying to prove this part may take longer than just completing a build; the build could fail to produce the desired proof for a variety of reasons. The details here get complicated: to get exactly the same output executable, it is necessary to use all of the following: precisely the same version of the compiler, linker, and each other

3 It is also recommended that a request be made for copies of software design documents, coding standards, build logs and associated tool outputs, testing logs, and other artifacts of the embedded software design and development process.

4 Note that if it is possible for the expert to extract an executable from one or more exemplars of the product, an automated comparison should always be made between the installed and produced binary files. It is not certain what might be found, and any difference could have important implications for the facts underlying the case.

5 Further additional technical details include the need to start with a "clean" set of files that contains no object files or libraries, and it may be necessary to obtain third-party header files or libraries.

build tool as the original developers; precisely the same configuration of each of those tools; and precisely the same set of build instructions. Even a slight variation in just one of these details will generally produce an executable that does not match the other binary image at all – just as the wrong version of the source code would.

Request the version control repository

Embedded software source code is never created in an instant. All software is developed one layer at a time over a period of months or years in the same way that a bridge and the attached roadways exist in numerous interim configurations during their construction. The version control repository for a software program is like a series of time-lapse photographs tracking the day-by-day changes in the construction of the bridge. But there is one considerable difference: it is possible to go back to one of those source code snapshots and rebuild the executable of that particular version. This becomes critically important when multiple software versions will be issued over a number of years. In the automotive industry, for example, it must be possible to give one customer a fault fix for his v2.1 firmware while also working on the new v3.0 firmware to be released the following model year.

Consider, for the sake of discussion, that the executable version of Toyota's ETCS v2.1 firmware that was installed in the factory in one million cars around the world had an undiscovered fault that could result in unintended acceleration under certain rare operating conditions. Now further suppose that this fault was (perhaps unintentionally) eliminated in the v2.2 source code, from which a subsequent executable was created and installed at the factory into millions more cars with the same model names – and also as an upgrade into some of the original one million cars as they visited dealers for scheduled maintenance. In this scenario, an examination of the v2.2 source code proves nothing about the safety of the hundreds of thousands of cars still with v2.1.

Gaining access to the entire version control repository containing all of the past versions of a company's firmware source code through discovery may be out of the question. For example, a judge might only allow the plaintiff to choose one calendar date and to receive a snapshot of the defendant's source code from that specific date. If the plaintiff was lucky, it would find evidence of their proprietary code in that specific snapshot. But the observed absence of their proprietary code from that one specific snapshot does not prove an alleged theft did not happen earlier or later in time.

There are some problems with examination of an entire version control repository. It may be difficult to make sense of the repository's structure. Or, if the structure can be understood, it might take many times as long to perform a thorough review of the major and minor versions of the various source code files as it would to just review one snapshot in time. At first glance, many of those files would appear the same or similar in every version – but subtle differences could be important to making a case. To really be productive with that volume of code, it may by necessary to obtain a chronological schedule provided by a fault list or other production documents describing the source code at various points in time.

Remember that the hardware should not be overlooked

Embedded software is always written with the hardware platform in mind and should be reviewed in the same manner. For example, it is only possible to properly reverse engineer or disassemble an executable program once the specific microprocessor (e.g., Pentium, PowerPC, or ARM) is known. But knowing the processor is just the beginning, because the hardware and software are intertwined in complex ways in such embedded systems.

Only one or more features of the hardware are enabled or active when the hardware is in a particular configuration. For instance, consider an embedded system with a network interface, such as an ethernet jack that is only powered when a cable is mechanically inserted. Some or all of the software required to send and receive messages over this network may be not be executed until a cable is inserted. A proper analysis of the software needs to keep the interactions between hardware and software such as this in perspective. Ideally, the testing of the firmware should be undertaken on the hardware as configured in exemplars of the units at issue – so is it useful to ask for hardware at the discovery phase if you are not able to acquire exemplars in other ways. It is not clear from the redacted reports if NHTSA's testing of certain Toyota Camrys was done using the same firmware version on exactly the same hardware as the owners who

experienced unintended acceleration. Hardware interactions can be one of the most important considerations of all when analyzing embedded software.

Sometimes a fault is not visible in the software itself. Such a fault may result from a combination of hardware and software behaviours or multi-processor interactions. For example, one motor control system the author is familiar with had a dangerous race condition.[6] The fault was the result of an unforeseen mismatch between the hardware reaction time and the software reaction time around a sequence of commands to the motor.

Additional analysis

The review of embedded software can be complicated. This is partly because the hardware on each embedded system is unique. In addition, the system as a whole generally involves complex interactions between the hardware, software and user. An expert in embedded software should typically have a degree in electrical engineering, computer engineering, or computer science plus years of relevant experience designing embedded systems and programming in the relevant language(s).

The five best practices presented here are meant to establish the critical importance of making certain specific requests early in the legal discovery process. They are by no means the only types of analysis that should be performed on the source code. For example, in any case involving the quality or reliability of embedded software, the source code should be tested via static analysis tools. This and other types of technical analysis should be well understood by any expert witness or litigation consultant with the proper background.

In the case of unintended acceleration in relation to a number of vehicles manufactured by Toyota, it is suggested that consideration should be given to critically analyze more fully the government analysis that has been discussed in this article and elsewhere. It is anticipated that expert review in the class action litigation against Toyota in the U.S. will identify all of the causes and determine if embedded software played any role in the accidents that occurred. Though funds for analysis by NASA are understandably limited, it is suggested that transportation safety organizations, such as NHTSA, should establish rules that ensure that safety-related technical findings in litigation are not hidden behind the secrecy of a settlement agreement.

Michael Barr, BSEE, MSEE is a former lecturer at the University of Maryland and has written extensively on embedded systems design. Mr Barr has testified as an expert witness in US and Canadian cases covering patent infringement and validity, software quality, theft of copyrighted source code, and satellite TV piracy.

mbarr@netrino.com

http://michaelbarr.info

6 *Michael Barr, Firmware-Specific Bug #1: Race Condition, 11 February 2010, available at http://embeddedgurus.com/barr-code/2010/02/firmware-specific-bug-1-race-condition/.*