Minimum requirements for testing source code

While testing is itself a complex issue, before an test can be realistically worth being performed, a clear understanding of what the software (in it's environment) does is required. For this reason, a strong specification capability, hardware description capability, and sound change control scheme must be in place as the precursor to effective testing. However, this is presently far beyond the realistic situation for almost all software in today's World.

It has long been held and never refuted to my knowledge, that beyond a few tens of lines of code, even the most skilled people are unable to detect intentional subversion of systems by software at the source code level. For this reason, the notion of human or even human assisted code inspection as a defense against malicious code or alteration of code would seem quite limited.

Software testing cannot be complete for any but the most trivial of situations, coverage cannot be defined or calculated in most cases, and of course malicious actors knowing the specifics will put forth relatively little effort to defeat the testing scheme no matter what it is. Proof of correctness, even if done, does not guarantee minimality, and is never adequate to cover the complexities of transient failures in the environment. Methods like checking all paths and branches through programs and all of their edge cases is useful for detecting many sorts of faults, but we don't even have a good fault model for intentional subversion of systems via software. Indeed the coverage of such testing in the context of a multiprocess environment is negligible because of the vast number of different possible sequences of operations within the overall environment. Most such software testing methodologies also ignore timing issue, interruption issue, changes in available memory, caching behavior, inter-process communications, resource consumption, and other forms of interference, as well as the effects on information leakage, effects of transient memory failures, and on and on. Similarly, we don't have hardware fault models that deal with the full spectrum of malicious subversions. And we are not likely to have such in the foreseeable future, among other reasons because we don't really have an underlying theory of 'security' or even its many myriad component parts.

While all of these are important and worthwhile methods and issues with those methods, none of the techniques are themselves adequate to the need today, and indeed never have been effectively used at large scale. Testing of source code is also infeasible in the sense of testing as it is historically understood, because source code does not actually execute. As such, all 'testing' is hypothetical relative to the realities of operating in an actual environment. Indeed the transformation from source into executable and installation into operating environments is also a cause of failures, as is library interaction, changes in the underlying operating environment, name space issues, and a wide range of hardware-related mechanisms that cause interference or leakage.

Conceptually, code review, static and dynamic analysis, and composition methods are not testing at all. They are purely analytical in nature, simulating what might happen in a test. Penetration testing is a testing methodology based on the premise that threat actors take select paths through systems that include exercising software in unintended ways. But large classes of penetration methods actually available need not exploit any "flaw" in the mechanisms. Computer viruses, as an example, need not exploit any flaw in software at all, but rather, are an inherent side effect of sharing, transitivity of information flow, and Turing capability. As such, no software source code testing methodology will get at the underlying causes that result in the undesired effects at the system level. That is because the undesired effects are inherent in the desired effects. The problem is not in our stars, it is in ourselves.

If we take all of the 'strong' methods that are inherently or practically limited off the table, we are left with the weaker methods that, while they tend to detect some classes of things, also tend to miss large unknown classes of things. These are, in effect, in two major categories: random and systematic testing within a space associated with defined fault models. Each has known, or at least calculable coverage within different classes of faults, and the coverage of each tends to be extremely limited, typically on the order of the number of test vectors (or sequences) divided by the total number of execution paths through the system (which grows exponentially with the number of instructions executed). As we execute more instructions in more parallel processing devices… you get the idea.

The minimal testing requirements then have to be recognized as not meeting any useful theoretical quality requirement. They are quite literally less likely to solve the long-term software security problem than random shotgun blasts in the dark at the sky are likely to hit falling meteors and stop them from hitting the ground without breaking up first. Nevertheless they do find things, which can be taken as an indicator of how bad we are at writing software rather than the quality of the testing methodologies.

As a minimum approach to testing, one method is spewing random input sequences at programs using the normal input paths and seeing what they do. This monkey at the keyboards approach may seem the height of foolishness, but in fact, these sorts of tests reveal faults all the time. However, they do not tend to show faults in source code directly, as they operate in the execution environment rather than the source environment. The random sequences produce unexpected output (or failures) and, if repeatable, can often be traced in execution to a known fault or set of faults associated with a known fault type that should have been handled in design and implementation but was not. Such things as input buffer overruns, edge cases, don't care states not determined, and occasionally race conditions or get/set problems are encountered and fixes attempted, often resulting in the introduction of other faults not yet known or detected, and typically not resulting in a systematic change to eliminate similar faults in the future. The problem of detection is another critical issue here, because not all faults produce failures, and not all faults that produce failures produce detectable failures under any particular detection approach. But not producing a failure or the detection of a failure does not mean there was not a fault or failure. It may be a failure in detection.

Nevertheless, as a minimum, these sorts of tests run over a defined space to produce reasonably random coverage and over a minimal number of vectors is reasonably inexpensive. As long as detections are produced, the software is not good enough to pass even these tests, and more work is needed. It is a low bar indeed, but apparently not low enough for most modern software and systems.

A next step up would be something like load testing. Most current implementations do not address load testing in even a minimal way. In load testing, increasing numbers of inputs per unit time are pushed at system in an attempt to hit resource limits and cause failures associated with those limits. For example, in a networked environment, TCP ports are limited, and as they are exhausted, lots of things fail, often including internal TCP connections used for things like database access, and so forth. When these fail, internal operations operate incorrectly, conditionals produce wrong answers, and the cascade of errors produce behaviors often not within the scope of designer anticipation. Such things may also be simulated in internal testing by, for example limiting total available resources and testing beyond the availability. The idea here is to intentionally induce error conditions and verify that the software operates properly under such conditions. While the number of sequences of such conditions is still large, there are typically a finite number of finite resources (memory, CPU time, operating environment resources types, etc.) that can be identified for limitation, and these can typically be controlled to test to the failure point. At the failure point (and beyond), some resource-related faults will be exercised and consequences will result. This should readily demonstrate whether the program notices the condition at all, and if so, how it responds, assuming we put some sort of instrumentation in place to observe effects.

While there are many other similar sorts of things to do, rather than try to list more of them, it would seem reasonable to address the instrumentation problem that has now been identified for random and load testing. With the lack of clarity around the theoretical bounds of fault types, the problem of detection of faults and failures would seem apparent. We do not know what to instrument or how to instrument it to detect faults and resulting failures, because we don't have good models of what those faults and failures are or may be in the future. If we don't know what to detect, it's hard to figure out how to detect it. And if we don't know how to detect it, or even what a detection might look like, we could see a detection and not recognize it as such. This is aggravated by the problem that we don't really have a good definition for security or even a good list of the objectives we may be seeking. How many tests do we have to detect loss of integrity, availability, use control, accountability, transparency, and custody? Does our software design even consider them? Or are we merely testing minutia?[1]

1 Fred Cohen – CEO – fc@all.net – please consider me as a speaker at the workshop