All.Net Analyst Report and Newsletter

Welcome to our Analyst Report and Newsletter

So-called secure computing

I see these things online from time to time telling me something about ‘secure computing’ and I must admit I cannot stand to read most off them. The thing is, there is no such thing and there is not likely to be such a thing for the foreseeable future.

Here’s why…

I’m just going to start with a short list…

- The term ‘secure’ is not well defined.
- No logical system can be consistent and complete.
- Without a threat model there is no hope.
- I have some experience in breaking them.
- Code exists within a context.
- Hardware has a non-zero chance of bit (or bigger) errors.
- The physics of digital information.
- Majority rules (or some form of it).
- Covert channels (in all their forms).
- Inputs must make a difference.
- All possible sequences in all possible states.
- Running out of resources.

OK – that’s enough for now… not that I am done… I will add another theoretical issue… We don’t have a complete list of what we even mean by it. And we might never be able to have one…

Drill-down

If you have not given up by now, I am planning to go into a bit of detail on each of these. But I thought I would give you a chance to quit while you are ahead.

- **The term ‘secure’ is not well defined.**
  - Since we don’t have a common agreement on what the term means from a technical point of view, we cannot every show that we have met the spec (or failed to do so), so the hypothesis that some ‘code’ is ‘secure’ or not is not testable, which is to say, it is not a scientific claim. The inability to translate between the English term and a detailed specification means we cannot even really talk about it in sensible ways. But even if we took a pathetic, old, ridiculous, widely used non-definition like “CIA” (Confidentiality, Integrity, Availability) which misses some really important things, and each of which is not well defined enough to say something is

kept “Confidential by the code” (for example), there is almost no definition under which this is achievable today. But more on that later.

- **No logical system can be consistent and complete.**
  - Those darned mathematicians keep figuring out things that make it impossible to claim to do our job perfectly. So whatever you mean by ‘secure’ the implementation cannot be both complete (cover all possibilities) and consistent (without disagreeing with itself on some of them). That is somehow closely related to the problem of undecidability (for things like detecting bad code – whatever non-trivial definition of bad you may choose) that says you have infinite false positives, negatives, or both even though in a finite state machine (which most current computers actually usually are) it’s only too complex to solve (even with quantum computers by the way).
- **Without a threat model there is no hope.**
  - The need to constrain the problem space is fundamental in order to get anywhere close to a reasonable solution. For example, can the threat make the temperature in the devices executing the code that they start to melt? I know the code people out there will tell me that that’s not a problem with the ‘code’ but of course it is. The problem is the unstated assumptions of the claim of ‘secure’, which there are a potentially infinite number of. And of course in breaking systems, I start with the assumptions and violate them (see 50 ways to defeat any system).[1]
- **I have some experience in breaking them.**
  - A long time ago I wrote a ‘secure’ Web server. It was proven to meet its (incomplete but consistent) specifications by a graduate student using mathematical stuff. Of course he found 1 flaw in the program that, while it did not violate the ‘security’ specification, I did fix, but that’s not the point. In that time frame I was also doing code reviews of other peoples’ programs, and I don't think I ever got past the first few dozen lines of code before I found something that was imperfect. A common example is adding 2 integers and storing the result as an integer. Unless there are other constraints in place, this can produce a wrong answer (integrity problem), or with intercepted overflows, a program not completing (availability problem). Of course exception handling programs have the same problems as other code, but my real point is that most programmers have no idea of even the simplest things that can go wrong with their code.
- **Code exists within a context.**
  - If you are in an embedded system and your code implements a fully specified finite state machine correctly and is the only code running on the hardware, you may be able to avoid a lot of the context-related problems. But if you are in a time-sharing system with an operating environment not your own code present, you need to operate properly in all possible contexts of that environment in order to meet almost any requirement you likely can think of for ‘secure’. So when a race condition ends up messing up some assumption you made about the sequential nature of a series of system calls and their interactions, you best handle it...

1 http://all.net/Analyst/2021-03C.pdf

- **Hardware has a non-zero chance of bit (or bigger) errors.**
  - So in order for the ‘code’ to actually be secure, it has to handle hardware errors of at least some classes properly. Before you rush to say it’s impossible to handle them all, that’s part of my point. But certain types of hardware have certain types oif errors, and operate slightly differently, including having specific hardware problems that make some functions give wrong answers under specific conditions. For example a pattern sensitive error in 80x? Chips some years ago that caused some spreadsheets to get wrong answers. So secure best be for specific hardware configurations. Don’t tell me software implemented fault tolerance (SIFT) is not feasible – it is still working for Voyager...
- **The physics of digital information.**
  - Something about discontinuity and finiteness in time and space comes into play here. But let’s just take 1/3 for example. And then do the error expansion minimization at the proper level of granulation for the required accuracy and precision related to the problem your code is solving. If you haven’t done this for all problems your code can be used to solve, you are likely to get wrong answers… that darned integrity thing again.
- **Majority rules (or some form of it).**
  - If enough folks conspire against you, you lose. Whether it’s a voting think or an M of N thing or whatever, your code can only meet the requirements of ‘secure’ if it operates so that it cannot get ‘outvoted’ by other code in the operating environment. For example, they can conspire against your code to create thrashing so as to give it so little performance to meet timing requirements. Which brings us to a later point so I will not belabor it here.
- **Covert channels (in all their forms).**
  - Of course your ‘secure’ code is designed top operate in a double-rail hardware environment with identical performance characteristics for all possible input sequences. It’s not? How do you assure that your multiplier (in the specified hardware of course) doesn’t return results faster for a 0 than a 1 in any set of positions of the inputs? You know those timing things can exfiltrate bits… And how about power consumption. Your code is designed to assure that it consumes the same amount of power for each possible branch in the code for all input sequences – right? And it never allocates memory differently for different inputs and states – right? And there are no timing or volume differences in network traffic when you are sending content vs. not sending it? Of course not. So you must not be using a compiler or a high-level language, because none of them deal properly with these issues.
- **Inputs must make a difference.**
  - At some point, most code has to deal with inputs to produce results. If the inputs are not adequately precise and accurate, the answers will be wrong. The problem is that the users make mistakes, and do malicious things, and garbage-in garbage-out. The code countermeasure is to use redundancy on inputs that adequately restricts alteration or subversion to assure within the desired level of certainty, that

the inputs are consistent and correct. Of course the code has to be able to take on the requirements of implementation as parameters. So naturally, your code has parameters for all of these and related factors so that it runs at the proper levels of redundancy to assure the desired precision and accuracy at whatever level specified. Which of course means that it has to use unbounded arithmetic precision and numbers that have arbitrary length while at the same time having no allocation of memory or timing differences as identified above… Hmmm. There might be a problem here somewhere.

- **All possible sequences in all possible states.**
  - Of course to verify whatever the security properties are you either have to prove or test for all possible states that all input sequences with all timings produce the correct outputs, which also means you need to either formally describe the behaviors or have a way to test input vs. output and state to determine if they are in fact correct. And by the way, that includes all the inputs and outputs from and to all the other state machines that interact with your code.
- **Running out of resources.**
  - Of course if you write your program to not allocate or release any resources after it starts, and assuming it starts, and assuming all input and output are properly dedicated with adequate bandwidth always available with limited delay times for all possible acts of your code, and assuming you are properly scaled to assure that no input sequence can come faster than you can process it correctly, which you must have done to meet the fixed timing requirements above, you should be pretty good to go for this first level of review.

What to do?

I think we should stop bandying about the term ‘secure code’ unless we mean something about the code being locked in a physically secured safe or some such thing. The specific properties are what I think are the important things to think about and consider. But perhaps more importantly, I think we should stop trying to make such claims and admit that all code is not secure by any reasonable definition and move toward a better understanding and terminology for discussing the level of certainty associated with mechanisms performing as desired and specified… the term Surety comes to mind, but then that will become the new word for security and just as meaningless.

Conclusions

After you have corrected for all of the things identified above, tested it properly to assure that your theory meets practice over all identified operating environments, and so forth, you may be ready to start to look at the other various problems that I haven’t bothered to list here. Fear not, I will not run out of ways your code is insecure, at least not before you run out of time and money to fix all the problems already identified. And of course your analysis of the unsolvable issues will provide adequate evidence of limitation of possible ‘security’ failures to within defined bounds so that the user can properly judge the extent to which the not ‘secure’ things are acceptable to them. And I think we should stop thinking about perfection, as opposed to better than what was there before, in a specific and defined way, that we can measure and talk about reasonably. Is it secure? “No, there is no such thing. Security is something you do.”

