To Trust or Not to Trust, That is the Question!

By Tim Kremann

1). Introduction

Trust of software must be measured from the view of the developer, the evaluator, the user, and the defender/attacker. To meet the measuring stick of each of these we must balance conflicting goals of assurance, functionality, ease of use, cost, and time to delivery. Hence trust is no easy matter. However we believe that we can do better within these constraints.

Here we consider the solution space of trusting software to have three axes, namely: confidence, containment, and detection. Confidence requires some assurance about the behavior of software and the environment it runs in. Containment relies on the underlying system to scope the effects of trust violations. For example, one could use of separation kernels to enforce process boundaries and security. Detection allows the monitoring and reaction to the behavior of the software.

The challenge of determining the degree of each of these axes with respect to the application‘s security requirements. We plan to investigate ways to have these three aspects of trust work together to achieve our risk management goals.

2). What is Trust?

Defining trust requires a composition of assurance, correctness, security, integrity, and reliability. First lets state some definitions to use in defining trust.

We define **correctness** to be the property that *the software does work as specified when executed*.

We define **security** to be the property that *the software does only what it is supposed to and does not permit unauthorized actions or prevent authorized actions.*

Integrity can be defined as the property that *no undetected or unauthorized modification of the software has occurred*.

We define **reliability** to be the *probability that the software will run as required without failure for a given period of time.*

Assurance is *the amount of evidence that can be amassed that a property holds*. Techniques such as testing, disciplined software development, formal methods, type enforcement, monitoring and others can be used to build up this evidence for each of these desired properties. Now we are ready to define trust.

We define **trust** as *confidence in the software based on the available assurances that it will behave reliably and correctly while maintaining the integrity and security of itself and the system*.

Given this definition of trust, we consider how to determine the appropriate amount of trust needed to operate within the security constraints of the application. Using the best software engineering practices and processes available goes a long way towards achieving our goal, however pitfalls exist along the way. Furthermore, these techniques give us a sense of trust based on a snapshot in time and may not reflect the operational software and environment. In all cases, the difficult problem of complete trust in one’s software remains unachieved.

However, trust does not have to be all or nothing. Based on what the software does, one should determine how much trust to shoot for. Security and integrity critical software should be developed to be highly trusted. Those applications that do not affect the secure behavior of the system need not be developed to as

high a standard of trust. Keep in mind, though, one can not easily determine whether software has a security impact. The amount of trust desired should be based upon the application and the risk tolerance required. Methods for increasing trust will be discussed after we detail the difficulty of trusting software and software's vulnerability at all points in the software life cycle.

3). Why is Software Hard to Trust?

To trust software you need to know that it works, that it has not been changed, that its environment protects it from undue influences, and that it does not do anything other than what it was developed to do. For the following reasons the requisite knowledge for basing how much trust to place in a piece of software remains difficult to obtain:

- Too often software developers do not maintain a disciplined process. Errors can be introduced (by accident or intentionally) that might not be discovered until too late.
- Software, one of the most complex creations of mankind, defies the capabilities for analysis and testing existing today. Because it takes forever to exhaustively test even simple software routines, developers have to rely on less than ideal methods to determine if software behaves as specified. Of course this complexity creates a severe challenge for evaluators as well. No way exists to find and fix all intentional and unintentional errors within software.
- Software's greatest asset, its ease of change, both in development and in the field, also enables its greatest weakness. Since one can change software easily, opportunities exist all throughout the life cycle to replace crucial pieces of software with untrustworthy code.
- Software runs on necessarily complex software and hardware. For example, the NT operating system is millions of lines long with many known and unknown bugs waiting to be executed. If you can not trust the software you run on, then no matter how much trust you place in the software you are running, it can still be compromised through the underlying system.
- When combining two trusted software components together, determining the resultant trust level of the combination remains an area of research. As the world increasingly relies on dynamic environments, determining the trustworthiness of components constructed on the fly grows in importance. The distributed nature of the problem increases the complexity of the trust equation, as unexpected interactions of components become a nightmare.

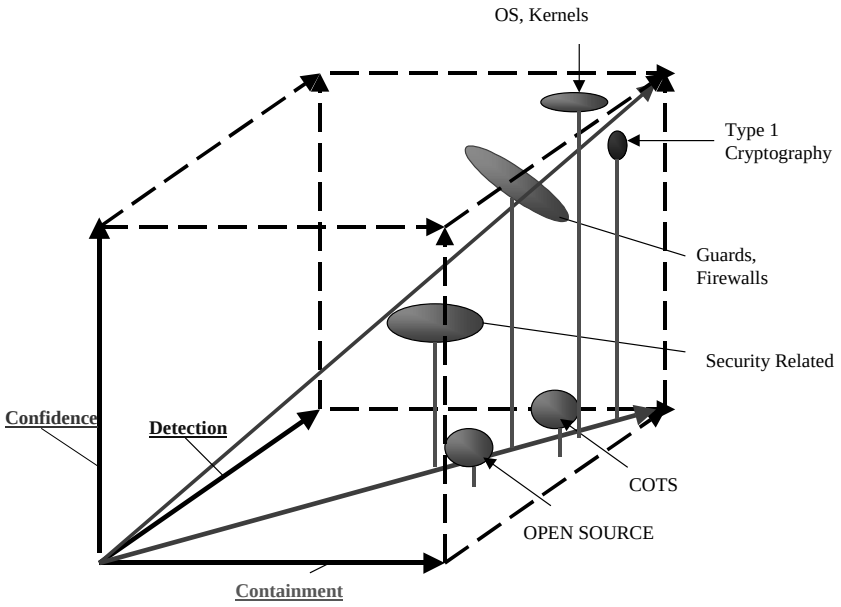
Note that how much trust we can place in software will be affected by unintentional faults and/or intentional faults, in other words, sabotage. Poorly designed and developed software will have more faults, affecting correctness, integrity, and reliability. In addition, anyone who does not protect and control software during construction, delivery, storage, and use cannot prevent sabotage, thus decreasing trust in a similar fashion. No matter which way the faults are introduced, by accident or by sabotage, it undermines our trust in the software. The impossibility of 100 percent trust in software has been touted as a reason to ignore trust altogether.

Although the task of building up trust in software seems impossible, there are people, process, and technology improvements available. Opportunities for increasing and safeguarding trust abound during the software life cycle.

4). Strategies for using Software of Varying Degrees of Trust

We want to enforce a defense-in-depth strategy to address trusting software. Three orthogonal components comprise this strategy. First, build high confidence into the security critical aspects of the system. Although this has been expensive to this point, tools with intelligence under the hood everyday appear on the market. Second, contain software to operate within certain enclaves. Examples of containment tools include separation kernels, secure operating systems, and AIM. Lastly, monitor, detect, and react to breeches of trust. We must build in the monitoring and fault detection circuitry that we had in hardware into software analogies.

Use the three complimentary trust techniques, high confidence software, containment, and detection, together to achieve the levels of trust required. The following diagram of a cube represents the amount of each applied to some operational software. The origin, the bottom left most point of the cube, represents software with low confidence, running with no containment and no monitoring. The upper right hand point, the end of the red arrow, represents high confidence software running with strongly enforced containment and with its operation monitored for violations of trust.



Notice that the lowest plane, the bottom of the cube represents software provided with little or no evidence of its trustworthiness. We want to move software that comes in at that level up higher in the space of the cube, representing additional confidence. This can be accomplished through the vendor, through better evaluation techniques, and through rigorous testing and analysis. We most certainly would want to expend the effort to do this for any security-related software being used by our customers.

An interesting discussion on what portions of this cube represent operationally acceptable software arises. Does it make sense to have high confidence software running in a low containment no detection environment? Without increasing the risk of a false sense of security, probably not. Can we run any software for which we have no evidence of its trustworthiness? With high containment for non security-related software, perhaps. The spheres here, representing differing types and origin of software are only guesses. We need to think about where software lies in the cube and determine a strategy that moves software to the safer region of the high confidence, high containment, and high detection.

5). Conclusions

The daunting task of building trust into software, of maintaining the level of trust while operating, and the capturing of and reacting to breeches of trust revolves around the commitment and resources we apply and enforce on the products we use. The emphasis on developing better tools, using and requiring better processes, informing users of the consequences of their actions, maintaining diligence in the application of defense-in-depth with containment and detection, begins our journey to trusting software in critical

situations. The techniques we have mentioned above should be the foundation of any security in depth strategy for using software in information assurance applications.

When one can build it right (high confidence), can limit its scope (containment), and can observe its behavior (detection), one can more easily satisfy the trust issues. When one cannot have the preferred amount of each of these, trust issues arise. Naturally one cannot build everything with high confidence, complete containment, and observe every behavior. Emphasis should be placed on efforts to determine what standards should be met for different information assurance applications. We can do better and should do so.

The foundations of the past should not be ignored. Much has been accomplished with respect to building techniques, policy, processes, training, etc. We need to move our past accomplishments into the model of the world today. Software bears down on us with tremendous force. There will always be a reliance of hardware at some point, even when burying the hardware under application, operating system, driver and kernel. When one can have the required levels of trust in all the software between the application and the trusted piece of hardware then we might reach the goal of trusting software.