The Foundations of Computer Security We Need Some

Donald I. Good
29 September 1986

I think the time has come for a full-scale redevelopment of the logical foundations of computer security. These thoughts have been forming in my mind over the last several years, and the discussion that ensued from John McLean's system Z example in the Computer Security Forum (Volume 5, Issue 14, June 22, 1986) prompts me to try to put some of them into writing.

My belief that this redevelopment is necessary comes from my own attempt to try to review and understand the mathematical foundations upon which much of our current thinking about security is based. Some of this review has formal proofs. Some of it has been simply to try to understand these foundations and explain them to others. I was quite surprised to find that many of these foundations are imprecise and, in some cases, quite inadequate even to define precisely such fundamental concepts as "secure system." There is a pressing need for this redevelopment because we are now beginning to build a computer security industry, and it is very important that it be built on solid foundations.

John McLean's Z system is an important case in point. It is widely accepted that a system is secure if and only if it produces a sequence of secure states. It is not surprising that this view is widely accepted. This is the property that was proved about the Bell and La Padula state transition rules, and it is the view that is embodied in the TCSEC.

John's Z system clearly demonstrates the inadequacy of this widely-accepted definition of a secure system. The Z system satisfies this definition. Given an initial secure state, it produces only secure states (according to the traditional, Bell and La Padula definition of "secure state"). But system Z obviously is not secure because it downgrades everything in sight so that any subject can have access to any object.

Then, in the discussion that follows John's example, the Z system is dismissed as a "stupid" and "bizarre" system. Such is the price of asking fundamental questions about sacred cows! Here I must rise fully in support of John McLean. What his example exhibits is not a stupid, bizarre system, but the inadequacy of a widely-accepted definition of "secure system." The state sequence definition of a secure system accepts system Z as secure, and it obviously is not! The problem is not the system, it is the acceptance criteria. That was John's key point, and the thrust of his question was "What else do we need for an adequate definition of 'secure system'?"

This is precisely the kind of question that needs to be asked about all of our traditional foundations for computer security. I applaud and encourage the work that is going on at NRL, SRI, Honeywell, Odyssey (and probably other places that I don't know about) that is stimulating reconsideration of some of these fundamental issues. John McLean's Z system raises one of the most basic ones. What does it mean for a system to be secure? It is sometimes surprising and rather alarming how inadequate our answers to that very fundamental question really are.

But this is by no means the only question that needs to be reconsidered. There are many others. If a secure system is required to follow the Bell and La Padula transition rules, what is the definition of "secure system" that is embodied in those rules? We need to know that definition so that if we modify those rules (to adapt them to other systems), then we will know what we have to prove about the modified rules to know that they are secure. In formal proof terms, what is the invariant property that a set of rules needs to satisfy in order to be secure? Is there such an invariant? In less formal terms, what are the criteria that determine if a set of rules is secure? Proving that a set of transition rules provide a sequence of secure states is a useful thing to do, but as McLean's Z system demonstrates, that alone does not ensure that the rules are secure in any realistic sense.

The list of questions does not stop here. It goes on and on. What does it mean to enforce an access matrix? Putting an access matrix into a system and proving that the system keeps it in proper form doesn't help much if the matrix isn't enforced. How do we include the enforcement of the matrix in our definition of a secure system? This is another key element that is missing from the traditional "state sequence" definition of "secure system."

While we're asking questions, let's get right to the most sacred cow of all, the reference monitor. I have never seen a rigorous proof that a reference monitor provides a secure system. Even such simple words as "mediates every access to every object" can become surprisingly fuzzy and open to a wide variety of interpretations. Certainly, the reference monitor concept is useful and captures some aspects of security. But, what is the fundamental notion of secure system that is embodied in a reference monitor? Let's see if we can state that definition and prove that a system (even a toy one) with a reference monitor satisfies it. Let's study what the logical consequences of that definition are. This surely would be very enlightening.

Are reference monitors an essential requirement for security, as the TCSEC imply? Is it possible to build a secure system without a reference monitor? Of course it is. Our research group has built and proved two secure systems, the Encrypted Packet Interface and the Message Flow Modulator. Both of these examples have full "code proofs." Neither of them contains a reference monitor. In both of these examples, there are many references to security sensitive objects that are not mediated. Also, neither of these examples is based on a state sequence definition of security. I'll be happy to match the security of these systems against the security of any reference-monitor based system.

While we're redeveloping our logical foundations and asking fundamental questions, let's clean up our terminology and the presentation of our basic computer security concepts. What is the purpose of all this "model" terminology? We don't need models of security. What we need are clear, precise definitions of the basic elements of security. We need those definitions to state the security requirements for the systems we need to build. The extensive use of modeling terminology in computer security doesn't illuminate the real issues. It obscures them and confuses people. I have wasted countless hours helping fully competent newcomers to the area see through this confusing terminology.

Have no doubt, modeling can be useful. System performance modeling, for example, has a long and productive history. But we need to keep in mind what models are and, more importantly, what they are not. They are replicas, approximations of the real thing they are modeling. Sometimes, they are very gross approximations. Sometimes they are much more accurate. Think of the plastic model car you might buy in a hobby shop versus a pre-production model that GM might build

versus the final product that you drive off the showroom floor.

Models can provide useful insight into reality. (I have spent several years of my professional life helping some of the best petroleum engineers in the country build mathematical models of oil and gas flow.) Models can be informative. Proofs about models can be informative. But, would you want to draw a conclusion of far-reaching consequences about the real thing from a proof about its model? No. Why? Because we don't know if the model is a sufficiently accurate approximation of the real thing. In computer security, it is the real system that is the issue, not the model. Are we really concerned about someone penetrating a model? Not really. John McLean's Z system shows how to do that for the model upon which probably every TCSEC certification to date is based. I haven't noticed any great panic. Are we concerned about someone penetrating the actual running systems that have been certified? Absolutely!

Let's put models in their appropriate place, and get on with the real issues of computer security. What are the real issues? What is our basic goal? It is to build secure systems. What we need to do that is fundamentally very simple:

1. We need some clear, precise definitions of the security concepts needed to express what is required of a secure system.
2. We need some ways of building systems to those requirements.

The fundamental process we are dealing with is system engineering. To paraphrase H. O. Lubbes, security is just a special interest group in this larger process.

The first thing we need in this process is the ability to state computer security requirements clearly and precisely. These requirements are the criteria by which we judge whether a system is "secure." To state them properly, we need clear, precise definitions of the basic concepts that pertain to system security. Ideally, what we need is a virtual library of these definitions that we can draw from to state system security requirements. By using these definitions, we should be able to compose a system requirement sufficiently clear so that a competent professional can study it for a reasonably short amount of time and, say, "Oh, yes, I agree. If you build that particular system to that particular requirement, it's secure enough for that particular purpose." Obviously, we need this concept base for expressing requirements before we can move on to the second step of the system engineering process, building systems to meet those requirements. Our current concept base is inadequate for this second step of the system engineering process, building systems to meet those requirements. Our current concept base is inadequate for this purpose, and it is for this reason that I believe we need to revisit, review and redevelop thoroughly the logical foundations of computer security.

Some of our foundations surely will remain intact. Some will need to be extended. Others will need to be discarded, and new ones will need to be developed. Last year in his after-dinner talk at the NBS seminar, Bob Courtney reminded us of the virtues of audit trails. For some systems, maybe they are a sufficient deterrent to potential violators of computer security. Knowing that you probably will get caught is powerful preventive medicine. Maybe for some systems, having an audit trail on each output port of the system is quite enough security. For these systems, we need precise definitions of the basic audit trail concepts so that they can be stated as system requirements. And guess what? You probably don't need a reference monitor or a security model or 50 pages of mathematical squiggles or even a subject or an object to express that requirement!

I don't mean to imply by any of this that we should trash all of our current foundations. Many of them were good, sound research at the time they were done, and they are still useful. But, many of these foundations are not well enough developed to support a vigorous and growing computer security industry, and there certainly are still new concepts, probably ones we haven't even thought of yet, that need to be developed.

Somewhere along the line we began to act as if the computer security problem were solved, and all that remained was to put these solutions into practice. Enter the NCSC with its heavy emphasis on certification and a large number of bright, young college graduates who were supposed to do that work. But they were asked to do that without, in my opinion, an adequate foundation for them, or anyone else, to know what they really had to do. The NCSC was an honest attempt to put what we knew at the time into practice on the well-founded assumption that some computer security was better than none. But, many of us seem to forget that a key part of the initial charter of the NCSC was research. How much research in fundamental computer security has been sponsored by the NCSC since its inception? Very little. Some of us have learned how to log onto Dockmaster, and that has been educational. But somehow that doesn't seem to get at the basic problems of computer security.

If we do not perform this research to really understand and solve the computer security problem, we run a very grave risk indeed. The risk is that the real, live human beings who use the systems being certified today without adequate foundations will put far too much trust in them just because they have a certification stamp. There are real dangers in trusting a technology beyond its limits. Used within its limits, the Titanic could have performed many useful services with its advanced, new technology. But, if we want to sell tickets for a head-to-head confrontation with an iceberg, maybe we'd better think some more.

I have two purposes in writing this note. The first is to make a case that there is a real problem here that needs attention. Before we can solve the problem, we must realize that it exists. The second is to point out that we now have an important, new technology that can help us in our search for a solution.

It is interesting to try to anticipate the various reactions that this note will evoke. I hope that at least some who read this will hear what I have said so far as just good, common-sense system engineering. That certainly is my intent. Others will have observed that essentially what I have done in the preceding paragraphs is apply to computer security the knowledge we have learned over the past 20 years about proving things about computing systems. What I am asking for is the development of some sound, clear, precise, useful theories within the problem domain of computer security. And yes, by that I do mean to say that we do not now have them. But, we need those theories to put the foundation of our current computer security house in order, and they are absolutely essential if we are even to think about going "beyond A1."

The new technology that we have to help us rebuild the theoretical foundations of computer security is rigorous proofs about computing systems. Traditionally, this technology has been, and continues to be, called "formal verification." However, I hesitate to use the term because it is so widely misunderstood in the computer security community. This is probably as good a time as any to try to clear up some of the confusion.

There is wide-spread belief that "formal verification" means "flow analysis" and the kind of thing that is required for an A1 certification. Well, "flow analysis" and traditional "formal verification"

are about as much alike as plastic apples and organic oranges. There are some gross similarities, but they serve very different purposes. Both are ways of proving "something", but the "something" that is proved is very much different. Flow analysis, as normally used in computer security applications, proves some very crude information flow properties about the SPECIFICATION of a system. Formal verification proves that the real, running SYSTEM meets some stated requirement. The requirement need not have anything to do with information flow.

Traditional formal verification has not been developed specifically for computer security. Many of the leaders of the field - Dijkstra, Hoare, Mills, and others - have never participated in computer security research. Formal verification refers to the use of rigorous mathematical reasoning and logic in the system engineering process to produce systems that are proved to meet their requirements. The work that we have done at Texas has been effective in applying these methods to computer security requirements.

In the early 1970s, formal verification was not powerful enough to be of any real use to people like Dave Bell and Len La Padula and others who were wrestling with the foundations of computer security. There were no

any real use to people like Dave Bell and Len La Padula and others who were wrestling with the foundations of computer security. There were no effective ways of proving things about complex, real systems (like Multics) or even about simple, real ones, for that matter. So, it was appropriate to gain insight into computer security by proving things about models of systems (and hence the introduction of modeling terminology into our vocabulary).

Even then, though, it was recognized that if formal verification could be developed, it would provide a powerful and rigorous way of proving things about the real systems that needed to be built. See, for example, Ted Linden's article, "Operating System Structures to Support Security and Reliable Software" in Computing Surveys (December 1976). Today, formal verification technology is well enough developed so that we can at least begin using it on some real systems. If we can prove that the real, running system is secure, then we don't have to bother proving something about a system model and then face the inevitable question of whether the model really does adequately model the real system. Thus, from proofs about the system itself, we get a much higher level of assurance (well beyond the "A1" level), and they provide a way of developing rigorous theories of computer security. These theories are the solid foundations the industry needs to have.

To explain how this works, I need to explain just a little about the process of proving that a system meets a requirement. The basic ideas can be illustrated by a simple system S with a single input x and output y. First, we express the requirement for S by defining a relation $R(x,y)$ on its inputs and outputs. This relation is the criterion that distinguishes those input/output pairs that satisfy the requirement from those that don't. Next, from the semantics of the language in which system S is expressed, we derive from S a logical formula $F(S,x,y)$, that accurately describes the run-time behavior of S. This formula is not a "specification" of S in the sense of an FTLS. It is a formula that describes the procedural, step-by-step, run-time execution of the system. To prove that every execution of the system meets its requirement, "all" we have to do is prove that $F(S,x,y) \rightarrow R(x,y)$ for all x and y. Completing this proof is roughly comparable to exhaustive testing of system S against the particular requirement R. Of course, the actual process is more complex than what I have explained (and we have spent almost 20 years of research learning how to derive these

logical formulas for a reasonably useful set of system building constructs). But this is the basic idea, and it has now been applied successfully to both software and hardware systems.

So how does this proof process relate to building a theory of computer security? The construction of a theory begins with the definition of the requirements relation *R*. *R* is composed of certain functions whose definitions we create. For example, we might define

```
R(x,y) == [y = if dominates(secret, label(x))
               then mail(x)
               else sound_alarm(x)]
```

These new functions -- "dominates", "label", "mail", "sound_alarm", etc. -- are defined in terms of other functions whose definitions we also create. These new functions are defined in terms of other new functions and so on until finally we bottom out on primitive functions whose properties we understand. Thus, the process of defining the requirements relation *R* essentially is one of building new functions from known primitive ones. These new functions are elements of a theory.

There's another very important kind of element of a theory, theorems about the functions we have defined. The theorems serve two very important purposes. First, they provide a useful check on whether we have defined our functions as we intended. This is not a foolproof check. But usually, as we are defining our functions, we can postulate various theorems that we believe should follow from them IF we have defined them properly. Thus, stating and proving these theorems gives us some confidence that we have defined the "right" functions to state our requirements.

The real purpose, though, of the theorems in the theory is to prove the formula $F(S,x,y) \rightarrow R(x,y)$ that was derived from the system. (This is the one that states that the real system meets its requirements.) Asking if this formula has a proof is equivalent to asking if this formula is a theorem in the theory. Thus, deriving the formula $F(S,x,y) \rightarrow R(x,y)$ for the actual system creates a candidate for a theorem in the theory. If that candidate is a theorem, the system is proved, and it's celebration time. If it is not a theorem, then either the system or the theory or both need to be adjusted.

Hence, the process of building proved systems helps focus the search for useful theories. Only by using a theory to prove real systems can we validate its practical utility. The postulation and proof of theorems to increase our confidence that we have the right functions defined in the requirements relation is important and useful. But, those theorems often are not the ones we need to prove the formula $F(S,x,y) \rightarrow R(x,y)$. I have great sympathy for any of my colleagues who have labored mightily to prove things about an FTLS of a system under the expectation that what they have proved about the FTLS will be useful in proving something about the system. Be prepared for some rude surprises.

One of the things that is essential to proving that a system meets a requirement is abstraction. Abstraction provides a way of suppressing details that are not relevant to the situation at hand. This is our means of coping with complexity. Abstraction doesn't reduce the total number of details that have to be dealt with; but, it provides a way of organizing those details so that we only have to think about a few of them at a time. Without abstraction, we couldn't do tractable proofs. To keep them tractable, we need abstraction both in our theories and in the systems we build.

We need abstraction in our theories. When we are considering the formula $F(S,x,y) \rightarrow R(x,y)$, we

don't want to have to prove it by expanding through 117 layers of function definition and prove the resulting huge formula from some primitive axioms. That is hopelessly complex and tedious because we have "expanded away" all the useful abstractions. We want to prove $F(S,x,y) \rightarrow R(x,y)$ from a couple of well-chosen theorems about "dominates", "label", "mail", etc. These theorems, in turn, are proved from theorems about some of the lower levels functions, etc. These intermediate theorems are abstractions that are important to the tractability of the proof. They are an important part of the theory.

The use of abstraction in the system building process also is important to the proof. If the system we are trying to prove something about consists of 5000 lines of code, we don't construct the formula $F(S,x,y) \rightarrow R(x,y)$ by "expanding the definition" of all 5000 lines. That, too, would be hopelessly complex and tedious. We use a high-level abstraction of the system, and prove something about the abstraction. Then we refine that abstraction to include a few more details and do proofs about that. What we end up with is not one large proof but a lot of little ones. Whether we do this top-down or bottom-up or whatever isn't what's important. What is important is that the abstractions are done in such a way so that when we prove some property about the abstraction, then that property is true of the real, running system. (This is in contrast to proving properties about system models. These properties may or may not hold for the real system depending on the accuracy of the model.)

The secret to tractable proofs about systems is abstraction, and even abstraction in the theory and abstraction in the system is not enough. Those abstractions also must find their way into the proofs. A proof that has to "see through" too many abstractions is almost certain to become too complex to manage. We use abstractions to suppress details that are not relevant to the situation at hand. That process is just as important in the proofs as it is in thinking about the system mechanisms or theoretical concepts. Our minds don't somehow expand their capacity to cope with detail when we start thinking about proofs (and it doesn't help much to drop all those details on a mechanical theorem prover either)! What is required to achieve tractable proofs normally is deep, rigorous, objective thought about how to organize the abstractions in our theories and in our systems so that we can reason (prove things) about both of them without getting lost in a maze of detail. Can there be any doubt that such an activity would be of great value to computer security?

Make no mistake. The kind of theory I have described so far is a specific theory for a specific requirement for a specific system. Typically, the first construction of such a theory is pretty much of a mess. But, as more thought is invested in it and our insight into the problem it is addressing deepens, we find better ways of organizing it. We begin to see ways to organize the functions and theorems differently so that some of them become reusable. This is real progress because we are beginning to understand our problem domain well enough to see some coherency in it rather than seeing it as a disorganized mass of complexity.

We pass another major threshold of progress when we understand the theories for a number of different systems well enough to begin to see functions and theorems that are common to several of them. At this point our specific theories are beginning to unify! If you look hard enough at computer security, you can begin to get some faint glimmers of this kind of unification starting to happen -- for example, the "non-interference" concepts. This kind of unification is theory building in the finest tradition of mathematics or physics.

I hope the preceding discussion gives some idea of how proving systems is interwoven with

building and proving theories. If our science were more mature, we would have more of these theories already built. Unfortunately, we don't. So, within the current state of the art, each new system proving effort, of necessity, has a high proportion of theory building. That is one of the primary reasons for the current high cost of proofs about systems. If a sufficiently reusable collection of these theories already existed, the effort required to prove a system would be greatly reduced. But let's not blame the proof methods for the immaturity of the science. The proof methods provide a way to advance the science. We need these theories, not to reduce the cost of proofs, but to understand the problems we are trying to solve and where to look for solutions. Once sufficiently reusable theories exist, the costs of proofs will be greatly reduced and we will be able to use system proof methods on a much more wide-spread basis to provide a much higher (than A1) level of assurance that our systems implement our theories.

How do we go about getting this process started? We begin by using system proof technology to build some experimental secure systems. But, before we mandate a "unified theory of computer security" that applies to all ADP systems, let's develop some specific theories for a few specific systems.

For each system, we begin by asking that most fundamental question, "What is a reasonable security requirement relation *R(x,y)* for this system?" It is precisely at this point where many of our current foundations begin to crumble, and we must begin afresh. As we search for an answer to this question we need to resist the temptation to think in terms of mechanisms. That comes later when we build and prove the system. What we need at this requirements definition stage are clear, precise definitions of concepts (that eventually will be implemented by mechanisms).

What kinds of systems should we focus on? "What is a secure operating system?" is a good question. But, let's not try to run before we can crawl. Let's not stop asking these larger questions, but let's also ask some smaller ones. "What is a secure message system or a secure network or a secure encryption box or a secure switch?" Because degenerate cases are usually enlightening, "What is a secure cable?" To be a bit facetious, what is an A1 cable? Does it have a reference monitor? Are all cables A1? How do those issues change if I put a microprocessor in the middle of that cable and change it from a dumb cable into one that is just a little bit smart, and then from that into one that is a whole lot smart? Maybe that smart cable even begins to look like an encryption box. What are the security theories that provide proofs that these various systems are secure? What are the similarities and differences in their requirements, their mechanisms, their proofs? From asking these basic questions, building some experimental systems, and proving that they meet their security requirements, a sound, precise theory of computer security will, of necessity, begin to evolve.

These experimental systems also will serve another very useful purpose. They will provide some "demonstrator" examples for the industry to follow in building true production quality systems. Actually, there is a pressing need for these demonstrator systems, whether they are proved or not. The fact that John Rushby has been in this country for over three years and hasn't yet been funded to build a demonstrator system for his separation kernel concept, which is one of the most promising new ideas to come along in computer security in a long time, is a national disgrace!

By now, I probably have said something to offend almost everyone in the computer security establishment. It is not my intention to be offensive, but sometimes it is necessary to speak loudly enough to be heard above the thundering herd. My message is very simple. Our current logical

foundations are inadequate to support a vigorous and growing computer security industry. We need to recognize that problem and solve it. If we do not, we invite serious consequences. Proving that some real systems are secure can help us start building the solid foundations we need. Let's get on with it!