View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Calhoun, Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive

Faculty and Researcher Publications
Faculty and Researcher Publications

2011

Lessons Learned from Building a High-Assurance Crypto Gateway

Levin, Timothy E.

http://hdl.handle.net/10945/35022

Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

Lessons Learned from Building a High-Assurance Crypto Gateway

The construction of a complex secure system composed from both secure and insecure components presents a variety of challenges to the designer. The example system described here highlights some lessons learned from first-hand experience in attempting such a task.

Clark Weissman
Independent Consultant

Timothy E. Levin
Naval Postgraduate School

In surveying a dozen developing security products in the mid-2000s, it appeared that they all satisfied the basic security requirement for handling national security information (National Security Telecommunications and Information Systems Security Committee [NSTISSC] Policy 11), which is that they had been "evaluated and validated" in accordance with National Information Assurance Partnership (NIAP; www.niap-ccevs.org/cc-scheme) or other security criteria. However, we found that additional untrusted code was needed to integrate these products with their deployment environments, resulting in uncertified end systems in each case. This broad deficiency presented a challenge: to build a complete system that met the highest requirements of both Policy 11 and the NIAP scheme. In our combined 75 years of experience in the security R&D community, we've witnessed almost every new approach to building secure systems that has seen the light of day. The most meritorious were cumulative efforts that extended the science of computer security, ultimately providing a broad menu of technologies with which to address the challenge. Thus armed, a new project emerged with considerable enthusiasm and optimism, forming the background for the several lessons we describe here.

The goal of the Encryption-box Security System (eSS) project was to produce a trusted hardware foundation (see the "Trusted Components" sidebar) that could be applied to any IP-based network of host computers.[1] In an eSS, the hosts are called *arbitrary application processors* (AAPs). An eSS adds a trusted network security controller (NSC) that defines a security policy over network communications, and trusted encryption gateways (see G in Figure 1) that control network access by each AAP and NSC and encrypt the related session traffic. The NSC policy establishes each AAP's security level and determines which pairs of AAPs are allowed to communicate, consistent with the security levels. The NSC provides the encryption gateways of each such pair with a unique session key. The design of the encryption gateways is simplified by treating AAPs and NSCs equivalently, as hosts. In fact, the key management and encryption actions of the gateways don't require any special input from the hosts. An encryption gateway simply renders all outgoing cleartext (red) messages into encrypted (black) network messages and renders all incoming (black) messages into cleartext (red) messages, based on the session keys. The NSC, AAPs, and encryption gateways can also detect security faults and generate messages for an audit server, if one is present. The eSS design supports the inclusion of a third-party audit server, which is simply another AAP.

The rest of this article uses the lessons learned in building eSS to shed light on several high-trust security engineering challenges found in the past 50 years, presented here in terms of secure architecture, secure implementation, and trustworthy development issues.

Secure Architecture

Most modern large-scale systems employ a complex organization of distributed components, described

 1540-7993/11/$26.00 © 2011 IEEE

Trusted Components

A component of a network security architecture may trust, or depend on, another component to perform certain functions. The trustworthiness of the first component is, then, limited by the trustworthiness of the other. A trusted component must be verified as being worthy of that trust with respect to some criterion for trustworthiness that, itself, has been vetted to reflect the concerns of the information stakeholders. The trust can range from incidental to security critical. The greater the trust in a component, the more rigorous its trustworthiness must be verified. For example, in an MLS system, a component that reads and writes a wide range of information sensitivity levels must be verified to the highest degree to perform only the intended data movements, since the system itself depends on the component to maintain the security policy. For the Common Criteria, the highest degree is EAL7, which corresponds generally to a TCSEC classification of A1.

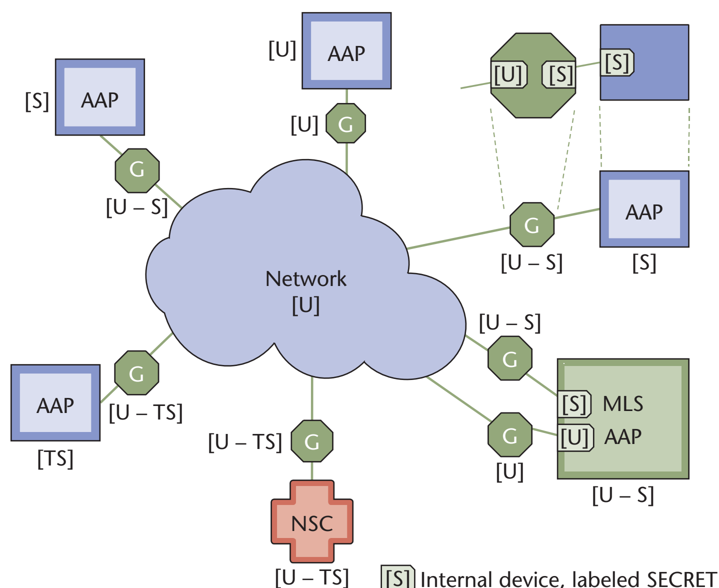

Figure 1. Encryption-box Security System (eSS). Each arbitrary application processor (AAP) operates at a single security level, in this case, [S]. The encryption gateway [G] operates at the security levels of both its attached AAP and the network [U].

variously as a network of networks, a system of systems, and so on. Each stand-alone node can also be comprised of individual functional components such as modules, layers, and programs. Key concepts for security analysis of complex distributed systems include the security perimeter, the allocation of policies to specific components, and the security policy domain. Each trusted computing base (TCB) has a security perimeter; one or more TCBs (or subnetworks) with common policies can comprise a security policy domain. A policy domain helps enforce a set of security policies including information flow (noninterference), mandatory access control (MAC), and discretionary access control (DAC), as well as supporting policies such as audit and login.

The enforcement of a given policy element within a policy domain can be allocated to a selected component. For example, the audit storage and protection policy can be centralized in one component, and the DAC policy might be allocated to another component. The eSS architecture allocates the MLS policy decisions to the NSC, and MLS policy enforcement to the encryption gateways, similar to John Rushby's trusted network interface unit.[2] Although the NSC could equally enforce any policy regarding access to host-to-host messages, one of the eSS's first-order requirements was the ability to support the US national confidentiality lattice (ordered security labels such as "top secret" [TS], "secret" [S], "confidential" [C], and "unclassified" [U], which can be combined with the power set of nonordered categories as a cross product),[3] a form of MLS.

eSS MLS Design Decisions

The designation of particular security labels to various system components is a critical design decision for an MLS network security architecture (see Figure 1). The simplest decision is the labeling of the AAP hosts within the architecture because of the design decision that each AAP would have a static label.

The choice of level for encrypted information plays a key role. In this project, the developers made two design decisions to keep the system simple. The first was that all black data and components would have the same level; to do otherwise—for example, to label each encrypted datum at its native data level—would result in a multilevel network and the need for trusted routers and other intermediate components. The second decision was that the level of black data and network components would be U, meaning that the network itself could be protected as unclassified. In other words, it would be acceptable to the security policy if any component on the network could read any (encrypted) data on the network, and unclassified components could write to the network (this could support, for example, allowing U elements to use the network for unencrypted traffic). To choose a label other than U for black components would have required protecting the network from access by unclassified components. A similar analysis holds when you consider an integrity policy: if all black data were labeled with the lowest integrity level, low-integrity components could both read from and write to the network.

An encryption gateway separates an AAP's red data and actions from the black network. Internally, the encryption gateway is divided into a black and a red

side. The red side manages the AAP traffic and encrypts cleartext data and places it in black-side message buffers. The black side then employs the IPsec protocol stack to process the message for output to the network. Thus, the encryption gateway is allocated a security level (such as [U – S]) that spans that of its AAP (for example, [S]) and that of the network ([U]), resulting in a multilevel device (see "MLS AAP" in Figure 1).

The typical AAP is assigned a single sensitivity level and bears no responsibility for supporting the eSS MLS policy. The eSS can support multilevel AAP components, which would be somewhat more complex. For example, an AAP host might be assigned a security level range from U to TS and is then trusted to keep separate the individual levels within that range, U, C, S, and TS. To support separation, a different encryption gateway is used for each separate security level that's configured for the AAP, with a single-level internal device to interface with each gateway. This approach avoids the need for multilevel internal devices.

There are relatively few choices of accredited, high-assurance MLS platforms for hosting an MLS AAP, including Integrity and GEMSOS (http://NIAP-CCEVS.org/CC-Scheme/VPL/). However, designers should be wary about attempting to build their own trusted host component: it's possible, but only at a considerable cost in labor and time (years).

The eSS project created a formal security policy model in the Alloy language, along with a formal specification and operational code to match the model.[4] We crafted the architecture to minimize the number of trusted components—for example, only the AAP and the encryption gateway on the red side are exposed to classified information. For the encryption gateway device, we employed high-grade encryption algorithms (AES256 and HMAC SHA-1) that satisfied the US National Institute of Standards and Technology's Federal Information Processing Standards Publication 140-2 (csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf).

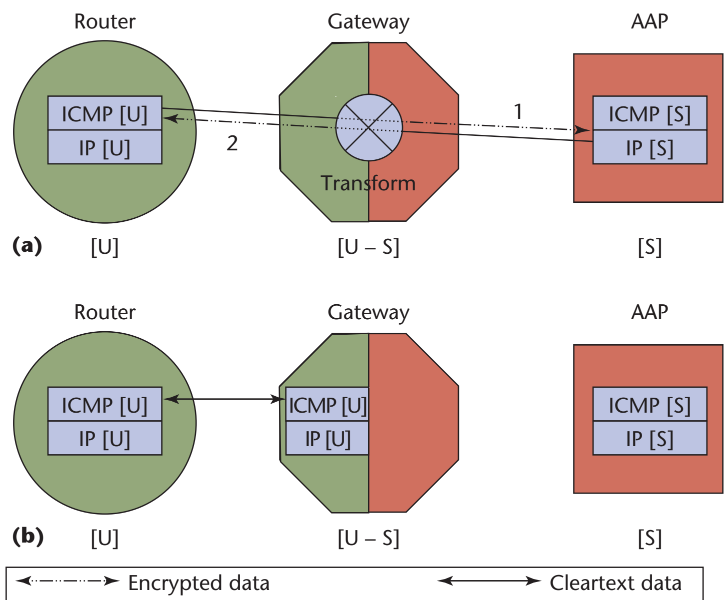

Figure 2. Internet Control Message Protocol (ICMP) query problem. In this example, incoming ICMP messages (shown as line 1 in (a)) cross from U to S, and require a decryption operation. Similarly, response messages (shown as line 2) cross from S to U, and require an encryption operation. (b) The alternative short-circuits the need for cross-domain communication and related cryptographic transformation by resolving ICMP messages in the U side of the gateway, simplifying the design and avoiding the policy violation of S to U communication.

Two-Way MLS Communication

One of the toughest challenges in designing a distributed MLS architecture is how to handle two-way protocols for transmission of information between components with heterogeneous security levels. Most communication protocols include data flow in both directions, if only to acknowledge message receipt. However, lattice-based security policies such as those described by the Bell-LaPadula (B&L) model allow only one direction of flow between components of different levels.[5] For example, data may flow from S to TS, but not from TS to S. The contradiction between the communication protocols' needs and the policy's restrictions is at the root of many design challenges.

Internet Control Message Protocol (ICMP). Our first product version was incomplete because we forgot to provide for the standard ICMP health messages—the pings—that travel between an endpoint such as the AAP and one of the network routers. In addition to the problem that the black-side router is unclassified and the red-side AAP generally is not, ICMP raised a major design issue of communication between encrypted and non-encrypted data domains.

An ICMP/IPsec stack resides on each router, encryption gateway, NSC, and AAP. For each communicating pair, the ICMP component on one end generates queries that are responded to by the IP stack on the other end of the pair. As Figure 2a shows, the problem is that the router components don't know the encryption keys with which to talk to the AAP (see line of communication marked "1")—likewise, the encryption gateway would encrypt an AAP response ("2").

This became a show stopper because the only solution appeared to be to allow high-to-low message flow, a significant violation of the eSS security policy.

To solve this, we established an ICMP/IPsec stack on both sides (red and black) of each encryption gateway. The encryption gateway's black-side IPsec stack intercepts and responds generically ("no op") to any ICMP query from a router to the associated AAP. Similarly, the encryption gateway's red-side IPsec stack intercepts and responds to a query from the classified AAP to a router (the solution is also simplified when the AAP is U—in which case, the encryption gateway passes ICMP messages to the AAP host for its red-side IPsec stack to respond to the query—or when ICMP communication goes between two AAP end system at the same security level). This eliminates the need for two-way communication across security levels and resolves the violation of security policy on the secure network.

Covert channels. A "covert channel" is a means of signaling over a mechanism that was not intended for communication, which violates the security policy. Illicit access to an explicitly exported data object is considered a design flaw, whereas covert channels are based on the modulation of internal values or metadata by a high-sensitivity source AAP, which is visible to a low-sensitivity sink AAP. A complete security design results in complete virtualization of shared resources in which there are not any extraneous internal attributes that could be used to covertly signal from high to low. When complete virtualization is not possible, the goal is to narrow covert channels to acceptable levels. According to historical guidance,[6,7] storage channels above 100 bps must be closed; covert storage channels that are less than 10 bps are acceptable; covert storage channels of 10 to 100 bps should be audited for misuse; and covert timing channels, which are very difficult to close without crippling system performance, must be audited.

Shared, exported resources. In eSS, the only shared, exported resources are the messages flowing through the network, which comprise the data objects of the access control policy. The first order concern of the design was to ensure that the access control over these objects was complete and correct. The next concern was whether message attributes could be modulated to create a covert channel. An easy modulation mechanism is the message length, which can be visible to unclassified network components such as routers. eSS closes that channel by ensuring messages are of a standard length. A timing channel results if the source AAP can generate messages at known intervals, which might be visible to network components as well as to end systems in the form of traffic congestion. eSS lowers the bandwidth available by generating random messages between AAP crypto channels: this isn't a perfect solution, but the resulting covert channel bandwidth is acceptable per the guideline.[7]

Brave new flows. eSS effectively creates a network of single-level subnets, much like virtual private networks (VPNs) that can't interact with each other. However, the eSS policy, a variant of the B&L security policy,[5] permits information to flow upward, from lower to higher classification levels. This led us to wonder whether a constrained form of the two-way protocol could enable a more efficient, yet secure, connection between AAPs at different levels: a multilevel network. We studied the feasibility of a multilevel network in which a canned downward acknowledgement (ACK) message could be allowed in response to an originating upward TCP message—assuming the overall stream of downward messages is throttled to a minimum bandwidth. Although downward message flow violates B&L, innovative handling of two-way network traffic has received serious consideration among certifying agencies, such as the US National Security Agency (NSA), when the write-downs are shown to be tightly limited (to 10 bps or less). This sort of ACK policy has many applications to secure networks, including that of a trusted guard to manage the flow of current-event news to a classified environment.[8] It appears that in the future, the trade-off between practical increased access to information versus possible classified data leakage might permit an ACK write-down channel in tightly controlled security situations, such as those that are audited or throttled by real-time feedback mechanisms—for example, the Global Information Grid program[9] characterizes this change in policy as a shift from "need to know" to "need to share."

Secure Implementation

A secure implementation of any trusted architecture component must satisfy a rigorous certification process and consider issues such as incomplete design and use of breakpoints (see "Common Criteria Evaluation and Validation Scheme (CCEVS)" sidebar).

Incomplete Design

Certification of a product is an exhausting process, with increasing rigor required for products that separate a broader range of security levels. The most critical aspect of certification is to ensure that the submitted documentation reflects a complete design. A system design specification often shows what the system is supposed to do in an ideal environment, but in practice, the environment is seldom ideal. A complete design specification must therefore describe secure behavior when the system or environment doesn't conform to nominal expectations: the burden

of complete specification is to ensure that the system continues to operate securely over the full range of inputs and failure modes.

Our project experience has shown that roughly 10 percent of most system specifications describe ideal performance, whereas 90 percent describe performance under non-ideal circumstances. The encryption gateway establishes the initial cryptographic keys at both AAPs of a cryptographic connection in a series of six or so exchange messages before the network becomes operational—that is, the keys are initially established in an ideal environment. However, if a cryptographic period expires, the encryption gateway must be rekeyed, which takes many hundreds of trusted messages to avoid threats from the operational environment, such as attempts to steal rekeying information. Key theft isn't a threat in the initial key exchange, in the sense that no users are yet operating—that is, the environment is nearly ideal.

> Lesson 1. The commercial Internet is the classical example of an incomplete design that enables hostile attacks. It was originally designed as a benign environment with cooperating users rather than the hostile users with which it operates today. Because Internet protocols are incomplete with respect to their design for response to hostile action, the Internet is a target for attacks ranging from defacement by attention-seeking kids to clandestine coordinated attacks by teams of government-sponsored, well-funded attackers. The entire situation is exacerbated when maintenance of key components follows the "penetrate and patch" strategy often associated with office productivity software.

Breakpoints

Another classical incomplete design for security is the popular "breakpoint" function in many debugging tools. This function lets a programmer designate target instructions in an instrumented program where the debugger will stop the program, report on specified system and program conditions, and then continue executing the instrumented program. A sequence of breakpoints can provide great insight as to how the program is behaving and is often used to help debug a program before it's tested and released.

In a typical OS, a program executes in one of two modes: kernel or user. Kernel mode allows the program to execute privileged instructions that affect system security. To access all the required system status information, the debugger runs in kernel mode—thus, it's a trusted process. Debugging tools are written to carefully control the user and instrumented program so that kernel mode isn't abused. Specifically, the debugger must be designed to ensure that untrusted programs are only executed in user mode. A key flaw occurs during the breakpoint function if an instruction is executed in kernel mode.

A malicious programmer can exploit the breakpoint flaw as follows (see Figure 3). The programmer defines the breakpoint in terms of a malicious transition instruction and a benign report sequence (which, here, returns the status of three program variables). When the program arrives at the breakpoint, the debugger executes the report sequence followed by the transition instruction. However, the transition instruction changes the processor to the kernel mode, which permits subsequent instructions to execute unconstrained and perform malicious operations. Under normal conditions, a program couldn't manipulate the processor mode because it requires a privileged instruction; however, breakpoint operates in privileged mode, from where the flaw is exploitable. Penetration testing unveiled this design flaw.

> Lesson 2. Variants of the breakpoint function allow this attack because of incomplete design of the trusted code that controls program execution. The countermeasure is that trusted programs shouldn't allow user programs to operate in kernel mode. This problem also points to the lack of assurance in compilers and other development tools. So, another take on this lesson would be that untrustworthy develop-

Common Criteria Evaluation and Validation Scheme (CCEVS)

There are three players in a high-assurance Common Criteria evaluation: the vendor prepares the product technical material; the Common Criteria Testing Laboratory (CCTL) performs the product testing and evaluation per the CCEVS; and NSA evaluators confirm the tests meet the CCEVS via a series of CCTL/CCEVS structured interactions called validation oversight reviews (VORs). The product vendor defines a clear, logical boundary of the target of evaluation (TOE) and a security functional requirement (SFR) of the security target (ST) to be certified. There must be an NSA-acceptable security policy for the ST, for example, Bell-LaPadula or a suitable cryptographic policy. The SFR must correspond to the ST features, advertising literature, and ST user and administration guides. The NSA reviews all test plans, procedures, and results, including penetration tests and senior valuator evidence review. A final VOR concludes the CCEVS evaluation with a fail or pass grade. A pass grade results in a certificate and a Validated Product Listing (VPL) for the TOE. Furthermore, many deployment environments require a separate certification and accreditation process, indicating that the IT products involved (which may have separate Common Criteria certifications) are used correctly and are suitable for the documented risks.

```
Mode          Name                     Comment
User1.        Instruction              # normal instruction
User2.        Instruction              # normal instruction
User3.        Breakpoint               # pseudo instruction, jump to (3)
Privileged4.  Set kernel privilege on  # Transition instr: set mode to
                                         privileged
Privileged5.  Privileged instr         # malicious action
Privileged6.  Privileged instr         # malicious action
User7.        Set kernel privilege off # set mode to user
User8.        Instruction              # resume normal operation
9.            Instruction              # normal instruction

(3) In privileged mode do:
      I. Execute Report Sequence
         a) Write status var1   # Report on system state
         b) Write status var2   # Report on system state
         c) Write status var3   # Report on system
      II. Execute instruction 4
    end
```

Figure 3. Breakpoint exploitation. This pseudo code illustrates how breakpoint can be misused to transition the system into kernel mode, as in instruction 4, thus bypassing the system's policy enforcement mechanism.

ment tools, such as a debugger, shouldn't be used on shared systems.

Trustworthy Development

Highly trustworthy development is the single most difficult technical issue that separates secure product development from general product engineering. So what is "trustworthiness?" Essentially, it's the confidence that a product performs as specified and does nothing else, satisfying its advertised claim even in a hostile environment. This confidence must be earned for secure systems, where we consider the system trustworthy according to analyses, documentation, tests, and correctness proofs validated by an independent agency.

Design Model

The Common Criteria Evaluation and Validation Scheme (CCEVS) certification process (http://NIAP-CCEVS.org) ensures that the design documentation (called the security target, or ST) is complete. Yet, completeness in a complex system is daunting. Where does the designer start? Sequential design refinement is one of the primary models for secure product development. For security products, the design is built in several conceptual stages:

- Stage 1 is the core set of functions to carry out the system's principal purpose. The goal here is to provide correct functionality.
- Stage 2 adds the procedures that result in a secure initial state.
- Stage 3 adds processing to check all input parameters for correct range values and to respond appropriately when values are out of range (an error condition).
- Stage 4 adds the defensive response for violations in stages 2 and 3, for example, alarms, audit messages, locking out the user from any further function, and clearing all data generated to date for the user. It disables or repairs any malfunctioning modules, serving as an error trap state that essentially returns the ST to a secure initial state.

The completion of these stages results in an operational system capable of "steady state" processing. All security policy rules are imposed at all times and all classified boundary crossings are properly checked. For high-assurance systems, the correctness and completeness of processing is further assured through formal verification, discussed next.

Lesson 3. The Common Criteria provides a very detailed set of functional requirements for an ST. When used correctly, it's an effective tool to check your design, become knowledgeable about the structured development of requirements, and gain insight to what you'll face during the CCEVS process.

Formal Specifications

The design model serves as a framework upon which to build a set of formal specifications, resulting in a two-level formal system: an abstract formal security policy model and a more concrete formal top-level specification. The formal model and specifications are written in a precise mathematical (formal) language that allows a rigorous and consistent system description; several language options are available.[10]

The formal specification language consists of well-defined syntax and semantics, and a set of language-processing tools that manage the specifications and related text. Ideally, the tools can automatically generate correctness theorems about the specification (that is, the theorems are "conjectures" until successfully proved, at which point they're theories). The latter is a significant capability because otherwise a thorough manual inspection is required to ensure that the hand-crafted security theorems are precise and correct—that the right thing is ultimately proved.

Today, program code is manually generated from formal specifications. A one-to-one formal or informal mapping is required between the code and the formal specification, an arduous process that might be lessened in the future via automatic translation or execution of the formal specifications.[11]

Lesson 4. The code-to-spec mapping compares the formal top-level specification versus the relevant code, line by line, side by side in the listing and

with necessary comments to explain correspondence quirks of either language. A mapping that shows an absence of code for a component of the formal specification might indicate missing security functions. The other alternative—code without corresponding specification—might indicate unwanted extra code, a Trojan horse, or a trap door.

Finally, a theorem prover associated with the specification language processes the formal model and related specifications to discharge the proof obligations. Even a relatively simple security target, such as the eSS, can require hundreds of pages of formal specifications, and the proofs can be a long, tedious process. The security policy model's proof shows that it's consistent with its stated correctness criteria (for example, the system only allows secure access to resources). The proof of the formal top-level specification provides a formal demonstration that it's consistent with the formal model.[12]

Theorem generation and proof can uncover subtle problems with the system design or with its specification when theorems can't be proved, perhaps because the specification is inconsistent or specified boundary conditions aren't broad enough with respect to possible values. For example, in an earlier program for the US Department of Defense,[7] a proposed design change would have provided operational flexibility, such that the B&L policy could be suspended in emergency conditions—innocuously, it was thought—and restored after the emergency passed.

The proposed policy change satisfied the operational imperative, but it introduced a subtle inconsistency that made it impossible to prove the security theorems correct, thus halting progress of the formal proofs and the project itself. This problem became a *cause celebre* until a resolution could be formulated. The B&L policy defines restrictions regarding both security levels (MAC) and individuals (DAC). To resolve the problem, we modified the design change to apply to DAC only. Thus, we were able to enforce the MAC rules at all times and found that relaxation of the DAC rules was sufficient for operational needs during the emergency condition. This solution required additional effort to redefine the emergency security policy formally, but it resulted in consistent theorems that we proved to be true—plus, it delighted the customer.

Lesson 5. Flaws found by formal methods usually aren't as dramatic as code flaws found during testing. An initial-conditions theorem might not prove because the conditions are too constrained to permit proof. Alternatively, they might be too weak—underconstrained—and allow any state to satisfy the specifications. Finally, the functional specifications for transitions to new states might not correspond to the formally specified initial or terminal states.

Security Testing

Each correspondence step—from model to policy, from specification to model, and from code to specification—brings further evidence of the product's trustworthiness. However, the security code isn't the functioning product: there must also be a binding between the source code and the system in execution. This is the domain of security testing.

Lesson 6. We divide testing into three parts: functional testing that shows the code's correct functionality, security testing that shows the omnipresent security policy enforcement even under hostile conditions, and penetration testing that attempts to show any security vulnerabilities in the code, such as incorrect or incomplete policy enforcement and other anomalies that are the sustenance of the code hacker. Some people have observed that functional testing shows the strength of the good guys, whereas penetration testing shows the strength of the bad guys.

Our penetration testing revealed that attackers could hack the encryption gateway, a case of incomplete design. The design called for establishing the initial crypto keys over the network before there were any external users. However, the absence of users couldn't be guaranteed, thus, malicious users might be able to log on surreptitiously and capture the keys. The solution was to add a removable cryptographic token (a key on a microSD memory card) for each AAP and NSC. This foiled any hacker attempts, as without a physical token a given host couldn't access the network.

Lesson 7. We can liken a secure system to a security utopia that operates perfectly when its assumptions are met and it's perfectly initiated. But like all utopias, the question arises as to how you get it started, securely. A classic attack strategy is to perform a legal operation that causes the utopia to restart, but this time with the hacker camped on the initiation sequence when the system's guard is down.

Commercial Development

Our experience is that industrial-strength software development using the Common Criteria requirements paradigm can be very high quality. As discussed, much of this process is based on rigorous use of model-based development, design reviews, state transition analyses, and other diagrammatical tools. Trusted software must also be very tightly

configuration-controlled using high-quality tools because of the large number of objects to be controlled compared to those found in typical nontrustworthy projects. Although usual commercial practices work well for myriad products in many contexts, it doesn't match the NIAP CCEVS process in eliminating coding errors, security flaws, and Trojan horses. Assurance against the latter in particular differentiates the discipline of high-assurance security from that of product safety. In the end, commercial practice can be very good—a less expensive compromise than some of the CCEVS methods—but we're hopeful that advances in development tools might enable commercial use of CCEVS methods in the future.

The proposed policy change satisfied the operational imperative, but it introduced a subtle inconsistency that made it impossible to prove the security theorems correct.

Overall, rigorous development methods work considerably better than ad hoc schemes, but we pay a price for that rigor. In system development's early days, when the system failed to work as desired, we said, "now that we understand the problem, we can write the programs again." The rigor provided by trustworthy development is like doing the job multiple times, concurrently. The system is instantiated at several different levels of abstraction simultaneously, and the developer's job is to ensure that these instantiations are consistent. Modeling, formal specifications, coding, correspondence analysis, testing, and operational guides provide the different system instantiations. For eSS, all of these tasks covered a seven-year development period.

We built various aspects of eSS as prototypes running on our internal network. These experiments led to many areas of incomplete and even wrong design, providing valuable lessons. In constructing the formal model and specification, we found errors in our design and incomplete states, mostly dealing with key distribution and attack vulnerabilities. Trusted initial key distribution was the biggest issue that led to a new design based on a physical ignition key for each encryption gateway. The Common Criteria was our "bible," always close by to keep us focused on proper design. Writing the code from the formal specification was relatively easy and straightforward, although it required some adjustment in the I/O code to interface with the limited rate initial production (LRIP) microprocessor we selected. We built six prototype encryption gateways and a closed network to run extensive system testing before selecting the LRIP processor. After that, we found few errors in the code; those that did occur were related to the anomalies of compilation tools.

> Lesson 8. A large class of the anomalies found in our code resulted from our choice to drop all the open source C++ library macros and write the needed macro code ourselves from scratch. This closed a possible Trojan horse or trapdoor insertion vulnerability, but increased code writing and testing.

We spent many weeks on penetration testing trying to break the system. The system performed so well, we used recorded unmanned aerial vehicle (UAV) video as the AAP data flow between simulated S and TS sites, and also used the video as part of the product demonstration to management. We were excited, with our live tests proving out the value of long years of work. The final steps for us were to complete the evaluation and attain a Common Criteria certificate for eSS.

We studied the NIAP CC Testing Laboratories (CCTL) list, made our choice, and contracted with one that had extensive network and encryption experience. We entered the first phase of the NIAP certification process and prepared the necessary information for the NSA validation oversight review (VOR). We completed all of the secure and trusted engineering and were on the verge of Common Criteria certification success.

> Lesson 9. The biggest lesson to be learned from this project was hard. Before we completed product certification, management cut off our funding. Their justification for this surprise was that there was no market for our product. A few months later, on 17 December 2009, the following article appeared in *The Wall Street Journal*: "Insurgents Hack U.S. Drones." The article noted the insurgents intercepted the cleartext video images from operational American drones using a $26 commercial software package called "SkyGrabber," made by Russian vendor Sky Software. The lack of encryption—equivalent to that of eSS—allowed this attack. The key lesson learned was to always make senior management part of the development team, so that their commitment to the project and to the market is clear.

While the result of the subject project was not taken to market, it nevertheless provided a foundation for confirmation and demonstration of the high-assurance development techniques we have discussed. As the need for high-assurance technology becomes

clearer, and subsequent secure products are developed, we hope that these lessons will help to light the way for those who follow. □

Acknowledgments

Steven Greenwald and Cynthia Irvine helped considerably in reviewing a preliminary version of this document. Brant Hashii built much of eSS.

References

1. C. Weissman, "MLS-PCA: A High Assurance Security Architecture for Future Avionics," *Proc. 19th Annual Computer Security Applications Conf.* (ACSAC), ACM Press, 2003, p. 2; www.acsac.org/2003/papers.
2. J. Rushby and B. Randell, "A Distributed Secure System," *Computer*, July 1983, pp. 55–67.
3. Controlled Access Program Coordination Office (CAPCO), "Authorized Classification and Control Markings Register," vol. 1, Director of Nat'l Intelligence (DNI) Special Security Center (SSC), May 2008.
4. B. Hashii, "Lessons Learned Using ALLOY to Formally Specify MLS-PCA Trusted Security Architecture," *Proc. ACM Workshop Formal Methods*, ACM Press, 2004, pp. 86–95.
5. D.E. Bell and L. LaPadula, "Secure Computer System: Unified Exposition and Multics Interpretation," tech. report ESD-TR-75-306, MITRE Corp., 1975.
6. US Dept. Defense, "Trusted Computer Systems Evaluation Criteria," (Orange Book) 5200.28-STD, US Nat'l Computer Security Center, Dec. 1985.
7. C. Weissman, "BLACKER: Security for the DDN: Examples of A1 Security Engineering Trades," *Proc. IEEE Symp. Security and Privacy*, IEEE CS Press, 1992, p. 286.
8. M.H. Kang and I.S. Moskowitz, "A Pump for Rapid, Reliable, Secure Communication," *Proc. 1st ACM Conf. Computer and Communications Security*, V. Ashby, ed., ACM Press, 1993, pp. 119–129.
9. P. Wolfowitz, "Global Information Grid (GIG) Overarching Policy," directive number 8100.1, US Dept. Defense, Sept. 2002.
10. A. van Lamsweerde, "Formal Specification: A Roadmap," *Proc. Conf. Future of Software Eng.* (ICSE 00), ACM Press, 2000, pp. 147–159.
11. M. Kaufmann, P. Manolios, and J. Moore, *Computer-Aided Reasoning: An Approach*, Kluwer Academic, 2000.
12. J. Rushby, "Formal Methods and Their Role in the Certification of Critical Systems," tech. report security level-95-1, SRI Int'l, Mar. 1995; www.csl.sri.com/papers/csl-95-1.

__Clark Weissman__ recently retired from Northrop Grumman Avionics and Systems as head of the Information Assurance/Multilevel Security (IA/MLS) group. He has more than 50 years of experience in secure systems research, development, and management and was principal investigator of the "e-Box Security System, eSS" program that evolved from a DARPA-funded R&D program. Weissman was the lead security developer of NSA's Class A1-certified BLACKER encryption appliqué for the Defense Data Net. He has a BS in aeronautical engineering from MIT. Contact him at ak096@lafn.org.

__Timothy Levin__ is a research associate professor at the Naval Postgraduate School. He has worked on the design, development, and formal verification of high-assurance computer systems for 25 years, during which he has had the relatively rare opportunities to develop a system security policy model that was rigorously evaluated by the NSA, formally verify the security properties of a commercially fielded OS, and participate as a principal author of a protection profile for highly robust separation kernels that has been accepted as a standard by the US government. Levin has a BA in computer science from the University of California, Santa Cruz, and is a member of the IEEE Computer Society and the ACM. Contact him at levin@nps.edu.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.