Essay 11

Penetration Testing

Clark Weissman

The TCB shall be found resistant to penetration.

— Department of Defense, "Trusted Computer System Evaluation Criteria," DoD 5200.28-STD, December 1985 (The Orange Book).

Near flawless penetration testing is a requirement for high-rated secure systems — those rated above B1 based on the Trusted Computer System Evaluation Criteria (TCSEC) and its Trusted Network and Database Interpretations (TNI and TDI). Unlike security functional testing, which demonstrates correct behavior of the product's advertised security controls, penetration testing is a form of stress testing which exposes weaknesses — that is, flaws — in the *trusted computing base* (TCB). This essay describes the Flaw Hypothesis Methodology (FHM), the earliest comprehensive and widely used method for conducting penetrations testing. It reviews motivation for penetration testing and penetration test planning, which establishes the goals, ground rules, and resources available for testing. The TCSEC defines "flaw" as "an error of commission, omission, or oversight in a system that allows protection mechanisms to be bypassed." This essay amplifies the definition of a flaw as a demonstrated unspecified capability that can be exploited to violate security policy. The essay provides an overview of FHM and its analogy to a heuristic-based strategy game.

The 10 most productive ways to generate hypothetical flaws are described as part of the method, as are ways to confirm them. A review of the results and representative generic flaws discovered over the past 20 years is presented. The essay concludes with the assessment that FHM is applicable to the European ITSEC and with speculations about future methods of penetration analysis using formal methods, that is, mathematically

specified design, theorems, and proofs of correctness of the design. One possible development could be a rigorous extension of FHM to be integrated into the development process. This approach has the potential of uncovering problems early in the design, enabling iterative redesign.

A security threat exists when there are the opportunity, motivation, and technical means to attack: the when, why, and how. FHM deals only with the "how" dimension of threats. It is a requirement for high-rated secure systems (for example, TCSEC ratings above B1) that penetration testing be completed without discovery of security flaws in the evaluated product, as part of a product or system evaluation [DOD85, NCSC88b, NCSC92]. Unlike security functional testing, which demonstrates correct behavior of the product's advertised security controls, penetration testing is a form of stress testing, which exposes weaknesses or flaws in the *trusted computing base* (TCB). It has been cynically noted that security functional testing demonstrates the security controls for the "good guys," while penetration testing demonstrates the security controls for the "bad guys." Also, unlike security functional testing by the product vendor, penetration testing is the responsibility of the product evaluators. However, product vendors would be ill advised to ignore their own penetration testing as part of the design, test, and preparation for a high-rated security product evaluation, for such vendors will surely be surprised by unanticipated debilitating vulnerabilities long after the development phase, when repairs are impractical.

Of all the security assurance methods — including layered design, proof of correctness, and software engineering environments (SEE) — only penetration testing is holistic in its flaw assessment. It finds flaws in all the TCB evidence: policy, specification, architecture, assumptions, initial conditions, implementation, software, hardware, human interfaces, configuration control, operation, product distribution, and documentation. It is a valued assurance assessment tool.

This essay is in 10 parts and describes a comprehensive method for conducting penetration analysis, of which penetration testing is but one aspect. The parts include background motivation, test planning, testing, and the analysis of the test results. The essay is largely based on the author's Flaw Hypothesis Methodology (FHM), the earliest and most widely used approach [WEIS73].

The "Background" section reviews the reference monitor concept of policy, mechanism, and assurance that forms the basis of the TCB. Penetration testing, a pseudo-enemy attack, is one method of evaluating the security strength of the reference monitor TCB. The section "Develop a penetration test plan" establishes the ground rules, limits, and scope of the testing. The test team identifies what is the "object" being tested and when the testing is complete. The section advances

the idea that the tests seek to confirm security claims of the vendor and to support the evaluation class rating. Testing is not a challenge or invitation to the test team to "crack" the system and steal something of value.

The section "Flaw hypothesis methodology (FHM) overview" provides a technical foundation for penetration testing. It establishes the idea that a security flaw is an unspecified capability that can be exploited to violate security policy — for example, to penetrate the protection controls. Finding and assessing flaws is a four-phase process of flaw generation, flaw confirmation, flaw generalization, and flaw elimination. These phases are covered in separate sections. The section "FHM experience" provides examples of results of application of the method to dozens of systems. Some costs of penetration testing are also discussed. The final section, "Penetration analysis for the 1990s and beyond," examines the ITSEC and where FHM can be applied, and work in formal methods and promising future approaches to flaw detection.

Background

Per the TCSEC, a TCB is the amalgam of hardware, software, facilities, procedures, and human actions that collectively provide the security enforcement mechanism of the *reference monitor* [ANDE72]. A reference monitor mediates every access to sensitive programs and data (security *objects*) by users and their programs (security *subjects*). It is the security policy mechanism equivalent of abstract data-type managers in strongly typed programming languages such as Algol, Modula, and Ada. The reference monitor software is placed in its own execution domain, the privileged supervisor state of the hardware, to provide tamper resistance to untrusted code. The reference monitor software, often called the *security kernel*, is small and simple enough in its architecture to enable it to be evaluated for correctness with assurance that only the authorized security policy is enforced and never bypassed. The strength of this triad of policy, mechanism, and assurance of the reference monitor is the basis for the evaluation of the TCB. Penetration testing is but one method for assessing the strength of a TCB.

Traditional methods of testing and repair are poor strategies for securing TCBs. Such strategies lead to games biased in favor of the hacker, who possesses modern power tools to attack aging computer systems. The "hack-and-patch" approach to assure secure systems is a losing method because the hacker need find only one flaw, whereas the vendor must find and fix all the flaws [SCHE79]. Furthermore, there are many flaw opportunities with little risk of detection or punishment for the interloper. So why bother with penetration testing by FHM? Because FHM penetration testing is not hack-and-patch, but a comprehensive, holistic method to test the complete, integrated, operational

TCB — hardware, software, and people. It is an empirical design review and a credible bridge between abstract design (theory) and concrete implementation and operation (practice). It is a peer review of all the TCB assurance evidence. It is one of many methods for satisfying assurance requirements. It works. It finds flaws. But we must not overstate its value. Penetration testing cannot prove or even demonstrate that a system is flawless. It can place a reasonable bound on the knowledge and work factor required for a penetrator to succeed. With that information, together with countermeasures, we can restrict the penetrator's access freedom below this bound, and therefore have a degree of assurance to operate the system securely in a specific threat environment [CSC85].

Develop a penetration test plan

Establishing the test ground rules is a particularly important part of penetration analysis. The rules are captured in the penetration test plan, which defines the test objective, the product configuration, the test environment, test resources, and schedule. It is important that penetration testing use ethical evaluators who are nonantagonistic toward the vendor to encourage cooperation, to protect proprietary information and vendor investment, and ultimately to yield an improved security product. Test results and flaws discovered during penetration testing must be kept strictly proprietary and not be made public by the test team.

Establish testing goal. There can be many goals for penetration testing, including security assurance [DOD85], system design research [KARG74], and systems training [HEBB80, WILK81]. For this essay, penetration testing will focus only on the goal of generating sufficient evidence of flawlessness to help obtain product certification to operate at a B2, B3, or A1 security assurance level.

The ground rules for the analysis define successful completion. The analysis is successfully concluded when

1. a defined number of flaws are found,
2. a set level of penetration time has transpired,
3. a dummy target object is accessed by unauthorized means,
4. the security policy is violated sufficiently and bypassed, or
5. the money and resources are exhausted.

Most often the last criterion ends the penetration test, after a defined level of effort is expended. For some systems, multiple independent penetration teams are used to provide different perspectives and increased confidence in the flawlessness of the product if few flaws are found. As a holistic assurance technology, penetration testing is best

used to explore the broad capabilities of the object system for flaws rather than to create a gaming situation between the vendor and the penetration team of trying to acquire an identified protected object by unauthorized means. Dummy target acquisition penetration goals waste effort by forcing the test team to prepare and debug a break-in, rather than focusing their energy on proven methods of finding flaws.

The reference monitor defines a "perimeter" between itself and untrusted user application code, and between different user processes. Flaws within the application code have no impact on the TCB and are of little interest to security or penetration testing. Flaws internal to the security kernel are of interest to security, but they cannot be exploited unless the security perimeter is breached. In international travel, it requires passports and visas to control border crossings. In banking, turning account balances into cash is a form of boundary crossing called "conversion." Control is imposed at the interface. Therefore, much of penetration testing focuses on the design, implementation, and operational integrity of the security perimeter, the control of the boundary crossings of this critical security interface.

Define the object system to be tested. FHM can be applied to most any system whose developers are interested in TCSEC evaluation. However, C1, C2, or B1 candidate systems are intended for benign environments, protected by physical, personnel, procedural, and facility security. Systems in these benign evaluation classes are not designed to resist hostile attack and penetration. Such attacks are always likely to uncover flaws. The TCSEC wisely does not require penetration analysis for these systems. FDM is most valuable for testing security resistance to attack of candidate systems for evaluation classes B2, B3, or A1, systems designed to operate in hostile environments.

A system intended for TCSEC evaluation at B2 or higher is delivered with a collection of material and documentation that supports the security claim, including a security policy model, a descriptive top level specification (DTLS), a formal top level specification (FTLS) for A1 evaluations, code correspondence matrices to the DTLS or FTLS, and security functional test results. All the source and object code, the design and test documentation, and the security evidence must be under configuration management control for B2, B3, or A1 evaluation classes. This controlled collection of security material will be referred to as the security "evidence," and defines the security system to be penetration tested. Not all the evidence will be complete if the penetration testing is performed by the vendor during the development phase. The evidence must be frozen and remain unmodified during the penetration testing period to avoid testing a moving target.

Experience has shown that probing for security flaws may require system halts and dumps by the penetration team. When tests succeed,

they yield unpredictable results — for example, uncontrolled file modification or deletion, or system crash — which disrupt normal operation. Therefore, penetration testing should be performed in a controlled laboratory environment on a stand-alone copy of the target system to assure noninterference with real users of the system.

When the object system is a network, the TCB is distributed in various components, the whole collection of which is called the network TCB (NTCB). As noted in the TNI, penetration testing must be applied to

1. the components of the NTCB, that is, the partitions of the NTCB; and
2. the whole integrated network NTCB [NCSC87a].

Therefore, the TNI Mandatory (M), Audit (A), Identification & Authentication (I), and Discretionary (D) M-A-I-D network components must be penetration tested individually and collectively — individually during the component evaluation, and collectively during the network evaluation.

In a similar manner, a trusted application, for example, a DBMS, must be penetration tested individually as a component and collectively with the operating system TCB on which it depends, according to the "evaluation by parts" criteria of the TDI [NCSC91].

Posture the penetrator. When an actual test is required to confirm a flaw, a host of test conditions must be established, which derive directly from the test objectives and the test environment defined in the plan. These conditions derive from the security threats of interest and the posture of the "simulated" antagonist adopted by the evaluators. Will it be an "inside job" or a "break-and-entry" hacker? These assumptions demand different conditions for the test team. The test conditions are described as "open-box" or "closed-box" testing, corresponding to whether the test team can place arbitrary code in the product (open box) or not (closed box). In the latter case, the team is restricted to externally stimulated functional testing. Open-box penetration testing is analogous to computer software unit (CSU) testing, where internal code is accessible, and closed-box penetration testing is analogous to computer software configuration item (CSCI) integration testing, where code modules are an integrated closed whole. The TNI testing guideline calls these "white-box" (internal) and "black-box" (functional) testing, respectively [NCSC88b].

In open-box testing we assume the penetrator can exploit internal flaws within the security kerne and work backward to find flaws in the security perimeter that may allow access to the internal flaws. In the case of a general-purpose system such as Unix, open-box testing is the most appropriate posture. For special-purpose systems such as network NTCB components, which prohibit user code (for example, where code is

in ROM), closed-box penetration testing by methods external to the product is analogous to electrical engineering "black-box" testing. In closed-box testing the penetrator is clearly seeking flaws in the security perimeter and exploiting flaws in the interface control document specifications (ICD). Open-box testing of the NTCB is still a test requirement to determine the vulnerability of the network to Trojan horse or viral attacks.

Fix penetration analysis resources. It was believed that finding flaws in OS VS2/R3 would be difficult [MCPH74]. However, another study claimed:

> The authors were able, using the SDC FHM, to discover over twenty such "exposures" in less than 10 man-hours, and have continued to generate "exposures" at the rate of one confirmed flaw hypothesis per hour per penetrator... Only the limitations of time available to the study governed the total number of flaws presented [GALI76].

Penetration analysis is an open-ended, labor-intensive methodology seeking flaws without limit. The testing must be bound in some manner, usually by limiting labor hours. Small teams of about four people are most productive. Interestingly, penetration testing is destructive testing. It is intense, detailed work that burns out team members if frequent rotation of the evaluators is not practiced. Experience shows the productivity of the test team falls off after about six months. Therefore, a penetration test by four people for no more than six months — 24 person-months — is optimal. The test team must include people knowledgeable in the target system, with security and penetration testing expertise. Much time is spent perusing the security evidence. However, there must be liberal access to the target system to prepare and run live tests. The team needs access to all the TCB creation tools — compilers, editors, configuration management system, word processors — and a database management system to inventory their database of potential flaws and to store their assessments of the flaws.

Flaw hypothesis methodology (FHM) overview

COMPUSEC's (computer security's) raison d'être is to automate many of the security functions traditionally enforced by fallible human oversight. In theory, a trusted system should perform as its security specifications define and do nothing more. In practice, most systems fail to perform as specified and/or do more than is specified. Penetration analysis is one method of discovering these discrepancies.

Evidence implication chain. For high-rated trusted systems, the trust evidence must show that theory and practice agree — that the "evidence implication chain" is correctly satisfied at each step. The implication chain of evidence shows:

1. the operation of the target system is compliant with hardware and software design, and human behavior;
2. those elements comply with the "correct" specifications;
3. the specifications satisfy the security policy exclusively; and
4. the policy meets operational requirements.

The TCSEC requires evidence that each step in the implication chain is correctly satisfied by various techniques, including human facility management and login procedures, audit trails, security officer oversight, automatic monitoring and alarms, code-to-specification correspondences, test results, peer review, mathematical proof, model-to-reality correspondence, and other methods. Penetration analysis is a disciplined method of examining weaknesses or failures in the implication chain.

Essentially, the method seeks counterarguments to the "truth" asserted by the evidence — that is, it seeks to establish the evidence is false or incomplete. A flaw is such a counterargument. A flaw is a demonstrated undocumented capability that can be exploited to violate some aspect of the security policy.

The emphasis of FHM is on finding these flaws. It is not on building demonstrations of their exploitation, though such examples may have merit in some cases. Exploitation demonstrations consume valuable resources that can better be applied to further flaw assessment of the implication chain.

The induction hypothesis. At the heart of the TCSEC is mathematical induction, sometimes called the *induction hypothesis*. It is the theoretical basis of TCSEC security and argues that

1. if the TCB starts operation in a secure state, and
2. the TCB changes state by execution from a closed set of transforms (that is, functions), and
3. each transform preserves defined security properties,
4. then, by mathematical induction, all states of the TCB are secure.

Finding flaws begins with finding weaknesses in implementation of this protection theory — policy, model, architecture, FTLS/DTLS, code, and operation. The evidence implication chain of the previous section forms the basis of the flaw search for violations of the induction hypothesis. As examples, false clearances and permissions void initial

conditions (rule 1), bogus code (for example, a Trojan horse or virus) violates the closed set (rule 2), and a large covert channe does not preserve the information containment security properties of functions (rule 3).

Stages of the Flaw Hypothesis Methodology (FHM). FHM consists of four stages:

1. *Flaw generation* develops an inventory of suspected flaws.
2. *Flaw confirmation* assesses each flaw hypothesis as true, false, or untested.
3. *Flaw generalization* analyzes the generality of the underlying security weakness represented by each confirmed flaw.
4. *Flaw elimination* recommends flaw repair or the use of external controls to manage risks associated with residual flaws.

These stages are shown in Figure 1.

Figure 1. FHM stages.

FHM can be likened to a computer strategy game. In artificial intelligence (AI) game software, there is a body of logic that generates "plausible moves" which obey the legal constraints of the game — for example, it ensures a chess pawn never moves backward. In like fashion, penetration testing needs for *flaw generation* a "plausible flaw generator." Flaw finding begins with the evidence implication chain, our experience of security failures in the reasoning chain in other systems, and their potential for existence in the target system. The security evidence for the target system is the principal source for generating new flaw hypotheses.

Continuing our AI game analogy, there is a body of heuristic rules the game uses to distinguish good plausible moves from poor ones. Likewise, in penetration testing *flaw confirmation*, there is human judgment — a "cerebral filter" — that evaluates and rates each prospective flaw in terms of its existence and how significantly it violates the security policy. Filtering flaws for confirmation involves desk checking of code, specifications, and documentation evidence, as well as live testing.

The *flaw generalization* stage of penetration testing gives an assessment of our results in progress, the game analogy of "winning" or improving game position. Flaw generalization assesses confirmed flaws, seeking reasons why they exist. For example, in the penetration testing of OS VS2 [LIND75], a simple coding error was traced to a library macro and multiple instantiations of the flaw in the code. Inductive reasoning on the cause of confirmed flaws can lead to new flaws, generators for still more weaknesses.

The *flaw elimination* stage considers results of the generalization stage and recommends ways to repair flaws. Implementation flaws are generally easier to repair than design flaws. Some flaws may not be practical to repair; slow covert timing channel flaws may be tolerable, for example. These flaws remain in the system as residual flaws and place the operational environment at risk. However, external countermeasures can be recommended to the approving authority for managing these risks, by lowering the TCSEC Risk Index [CSC85], for example.

Flaw generation

Flaw generation begins with a period of study of the evidence to provide a basis for common understanding of the object system. Early in the effort there is an intensive "attack-the-system" session of team brainstorming. Target system expertise must be represented in the attack sessions. Each aspect of the system design is reviewed in sufficient depth during the session for a reasonable model of the system and protection mechanisms to be understood and challenged. The vendor's

evaluation evidence is available for in-depth reference by the team. Flaws are hypothesized during these reviews. Critical security design considerations are the basis for the penetration team's probing of the target system's defenses. These design considerations become the "plausible move generators" of the flaw generation phase. The most productive "top 10" generators are the following:

1. Past experience with flaws in other similar systems.
2. Ambiguous, unclear architecture and design.
3. Circumvention/bypass of "omniscient" security controls.
4. Incomplete design of interfaces and implicit sharing.
5. Deviations from the protection policy and model.
6. Deviations from initial conditions and assumptions.
7. System anomalies and special precautions.
8. Operational practices, prohibitions, and spoofs.
9. Development environment, practices, and prohibitions.
10. Implementation errors.

Each candidate flaw is documented on a flaw hypothesis sheet (FHS), which contains a description of the flaw speculation. The total set of FHS becomes the flaw database that guides and documents the penetration analysis.

Past experience. The literature is filled with examples of successful penetration attacks on computer systems [ABBO76, ATTA76, BELA74, BISB78, BISH82, GALI75, GALI76, GARF91, KARG74, MCPH74, PARK75, SDC76]. There is also a body of penetration experience that is vendor proprietary or classified [BULL91, LIND76a, PHIL73, SDC75]. Although general access to this past experience is often restricted, such experience is among the best starting points for flaw generation.

Unclear design. The design must clearly define the security perimeter of the TCB. How are boundary crossings mediated? Where are the security attributes — permissions, classifications, IDs, labels, keys, and so on — obtained, stored, protected, accessed, and updated? What is the division of labor among hardware, software, and human elements of the TCB? And how are all the myriad other secure design issues described [GASS88]? If the secure design cannot be clearly described, it probably has holes. The team will rapidly arrive at consensus by their probing and uncover numerous flaws and areas for in-depth examination, particularly weaknesses in the evidence implication chain.

Circumvent control. What comes to mind is Atlas down on one knee holding up the world. The anthropomorphic view of TCB design gives the numerous protection control structures omniscience in their critical

Atlantean role of supporting the secure design. If such control can be circumvented, the security can be breached. The security architecture evidence must show noncircumvention or bypass of security controls. The attack sessions will rapidly identify these omniscient objects, be they password checkers, label checkers, I/O drivers, or memory maps. A method of determining their vulnerability to attack is to build a "dependency graph" of subordinate control objects on which the omniscient ones depend. Each node in the graph is examined to understand its protection structure and vulnerability to being circumvented, spoofed, disabled, lied to, or modified. If the security design is weak or flawed, control can be bypassed. The penetration testing of OS VS2/R3 [SDC76] gives a detailed example of the use of dependency graphs to determine the vulnerability of VS2 to unauthorized access to job data sets, virtual memory, and password protection control objects.

Incomplete interface design. Interfaces are rife with flaw potential. Where two different elements of the architecture interface, there is a potential for incomplete design. This is often the case because human work assignments seldom give anyone responsibility for designing the interface. Although modern methodologies for system design stress interface control documents (ICD), these tend to be for interfaces among like elements — for example, hardware-hardware interfaces and software-software protocols. The discipline for specifying interfaces among unlike elements is less well established. Hardware-software, software-human, human-hardware, hardware-peripheral, and operating system-application interfaces can have incomplete case analyses. For example, the user-operator interface to the TCB must deal with all the combinations of human commands and data values to avoid operator spoofing by an unauthorized user request. Operating procedures may be hardware configuration dependent. For example, booting the system from the standard drive may change if the configuration of the standard drive is changed. All the various error states of these interfaces may not have been considered.

Implicit sharing is now a classical source of incomplete design flaws. Sharing flaws usually manifest themselves as flaws in shared memory or shared variables between the TCB and the user processes during parameter passing, state variables context storage, setting status variables, reading and writing semaphores, accessing buffers, controlling peripheral devices, and global system data access — for example, clock, date, and public announcements. Careful design of these interfaces is required to remove system data from user memory.

Policy and model deviations. For B2 and higher evaluation classes, the security evidence includes a formal security policy and a model of how the target system meets the policy. Subjects and objects are de-

fined. The rules of access are specified. For lower evaluation classes, the policy and model are less well stated and, in the early years of penetration testing, required the penetration team to construct or define the policy and model during the attack sessions. However, penetration testing is not required for these classes today.

Consider the adequacy of the policy and the model for the target system. Is the model complete? Is the policy correct? Are there policies for mandatory and discretionary access control (MAC and DAC), identification and authentication (I&A), audit, trusted path, and communications security? Examine the security architecture and the TCB design to see if there are deviations from the stated policy or model. For example, are there user-visible objects that are not defined in the model, such as buffers and queues? Omniscient control objects, as described in the earlier section "Circumvent control," should certainly be represented. Are there deviations in the implementation of the policy and model? This consideration receives greater emphasis during flaw confirmation; however, there may be reasons to generate implementation flaws at this time.

Initial conditions. Assumptions abound in secure system design but are not documented well, except in evaluation class A1, where formal specifications require entry and exit assertions to condition the state-machine transforms. For all other evaluation classes the assumptions and initial conditions are often buried in thick design documentation, if they are documented at all. If these assumptions can be made invalid by the user, or if in the implementation reality the initial conditions are different from the design assumptions, the policy and model may be invalid and design flaws should exist. The induction hypothesis presented earlier begins with "starts operation in a secure state." Initial conditions determine the starting secure state. If the actual initial conditions are other than as assumed in the design, attacks will succeed.

The whole range of security profiles and administrative security data on user IDs, clearances, passwords, and permissions (MAC and DAC) defines the "current access" and "access matrix" of the Bell-LaPadula policy model [BELL76]. These data are initial conditions. Their correct initialization is a testable hypothesis. Other assumptions and initial conditions need to be established and tested by penetration analysis, including the computer hardware configuration, software configuration, facility operating mode (periods processing, compartmented, system high, MLS), operator roles, user I&A parameters, subject/object sensitivity labels, system security range, DAC permissions, audit formats, system readiness status, and more.

System anomalies. Every system is different. Differences that may have security ramifications are of particular interest. The IBM Program

Status Word (PSW) implements status codes for testing by conditional instructions, unlike the Univac 1100, which has direct conditional branching instructions. The IBM approach allows conditional instructions to behave as nonconditional instructions if the programmer avoids checking the PSW [SDC76]. That is an anomaly. The Burroughs B5000-7000 computer series compiler software has privilege to set hardware tag bits that define "capabilities," many of which are security sensitive, such as write permission. The Master Control Program (MCP) checks the tag bit for permission validity. User code does not have this privilege. Code imports can circumvent such checks [WILK81]. That is an anomaly. The IBM 370 I/O channel programs are user programs that can access real memory via the "Virtual = Real" command without a hardware memory protect fault [BELA74]. That's an anomaly. Nearly every software product has clearly stated limits and prohibitions on use of its features, but few define what occurs if a prohibition is ignored. What happens when an identifier greater than eight characters is used? Is the identifier truncated from the left, right, or middle, or is it just ignored? Anomalous behavior may not be security-preserving functionality per the induction hypothesis theory. This behavior can be exploited.

Operational practices. The complete system comes together during operation, when many flaws reveal themselves. Of particular interest are the man-machine relationship, the configuration assumptions, and error recovery. A well-designed TCB will have the system boot process progress in stages of increasing operating system capability. Each stage will check itself to ensure it begins and ends in a secure state. If there is need for human intervention to load security parameters, the human must be identified, authenticated, and authorized for known actions. The penetrator must see if the boot process progresses correctly. For example, how is the security officer/administrator authenticated? If via passwords, how did they get loaded or built into the initial boot load? Where does the operator obtain the master boot load? From a tape or disk library? Is the physical media protected from unauthorized access, product substitution, or label switching? If the security officer loads or enters on-line permissions to initialize the security parameters of the system, how does the security officer authenticate the data? If users send operator requests to mount tapes or disks, print files, or execute a myriad of other security-sensitive actions, how does the TCB protect the operator from spoofs to take unauthorized action? If the system crashes, does the system reboot follow a process similar to the initial "cold" boot? If there is a "warm" boot mode — a shortcut boot that salvages part of the system state — does the security officer have a role in the boot to ensure the system begins in a secure state? How is the assurance determined?

A common initialization flaw occurs when the system is shipped from the vendor with the "training wheels" still on [STOL89]. This class of flaw has been known to include training files that provide ID-password authorizations to users so they may train for jobs as security officers, system administrators, database controllers, and system operators. These files were not removed by good system operational practice per the Trusted Facility Manual (TFM) and can be used by unauthorized parties to circumvent security controls.

The development environment. Flaws may be introduced by bad practices in the security kernel/TCB development environment. A simple example is the conditional compilation, which generates special code for debugging. If the released code is not recompiled to remove the debugging "hooks," the operational system code violates the closed set (rule 2) and the secure transform (rule 3) of the induction hypothesis, similar to a trap-door bypass of security controls.

Large untrusted reuse and runtime libraries are properties of many programming environments. The TCB may be built using code from the library, which finds its way into operational use. All kinds of security flaws may obtain from such environments. If the libraries are not security sensitive, they can be searched for flaws that are exploitable in the operational TCB. If the penetration team can substitute its own code in the libraries, even more sophisticated flaws can be created. Runtime linkers and loaders have similar properties of appending unevaluated code to the trusted object code being loaded to enable code-operating system communication. If access to such tools is unprotected, similar code-substitution attacks are possible.

A classic way to attack an operational system is to attack its development environment, plant bogus code in the source files, and wait for the normal software update maintenance procedures to install your unauthorized code into the operational system object code. If the development and operational system are the same, then the penetration team must mount an attack on the development environment first, particularly the system configuration files. Flaws found there relate directly to the operational system, the source files of which are then accessible and modifiable by the penetrator without authorization. Substitute configuration files give the penetrator a high-probability attack and essential control of the TCB.

Implementation errors. In any system built by humans, there will be errors of omission and commission. This is not a promising class of flaws to explore, as there is no logic to them. Many are just typos. Implementation errors that can be analyzed are those of the IF-THEN-ELSE conditional form. Often a programmer fails to design or implement all the conditional cases. Incomplete case analysis may occur if the code logic

assumes some of the predicates are performed earlier. Most often, implementation flaws are just coding errors.

Other areas for investigation are macros and other code generators. If the original macro is coded incorrectly, the error code will be propagated in many different parts of the system. Similarly, if data declarations are incorrect, they will affect different parts of the code. Incorrect code sequences should be traced back to automatic source code generators to see if the code error appears in multiple parts of the TCB.

Sometimes there are errors in the development tools that generate bad code. Few configuration management tools provide a trusted code pedigree or history of all the editor, compiler, and linker tools that touch the code. Therefore, an error in these tools, which becomes known late in the development cycle and is fixed, may persist in some earlier generated modules that are not regenerated. The penetration team may find it fruitful to interview the development team for such cases.

Flaw hypothesis sheets. FHM candidate flaws are documented on flaw hypothesis sheets (FHS), one page (record) per flaw. An FHS is intended to be concise and easy to use throughout the penetration testing. The FHS contains seven fields:

1. a flaw name for identification,
2. a brief description of the flaw vulnerability speculation,
3. a localization reference of the flaw to a part of the system or module,
4. an assessment of the probability of the flaw being confirmed,
5. an estimate of the damage impact of the flaw on the protection of the system if confirmed,
6. an estimate of the effort/work factor needed to confirm a flaw, and
7. a description of the attack and result.

Probabilities for fields 4, 5, and 6 are measured on a scale of high (H), medium (M), or low (L). The combined assessment of HH, HM, HL, MH, ..., LL yields an overall scale of nine for ranking FHS. The ranking is valuable in allocating resources during the flaw confirmation phase of penetration analysis. The FHS documents an estimate of the work required to demonstrate a flaw. High work-factor flaws — for example, cracking encryption codes — are given lower priority, even if the flaw is ranked HH. An FHS has a section for describing the results obtained from the flaw confirmation phase. The total set of FHS becomes the flaw database that guides and documents the penetration analysis. The FHS can be mechanized with tool support — for example, a word processor or a DBMS — to permit flexible sorts and searches based on key fields of an FHS database of hundreds of records.

Flaw confirmation

Conducting the actual penetration test is part of the testing procedure developed in the plan. The bulk of the testing should be by "Gedanken" experiments, thought experiments, that confirm hypothesized flaws in the product by examination of the product's documentation and code, that is, the security evidence. There are three steps to the flaw confirmation stage: flaw prioritization and assignment, desk checking, and live testing.

Flaw prioritization and assignment. The flaw hypothesis sheets represent a comprehensive inventory of potential flaws. Sorted by the probability of existence, payoff (damage impact, if confirmed), work factor to confirm, and area of the system design, they provide a ranking of potential flaws for each design area from high probability/high payoff (HH) to low probability/low payoff (LL). Usually, only high and medium ranks are studied. Team members divide the rank lists among themselves based on expertise in the different system design areas. They move out as individuals on their lists. At daily team meetings, they share one another's progress and findings. Management may reallocate staff and FHS to balance the work load. Often confirmed flaws raise the priority of other FHS or provide the analysts with insight to generate new FHS.

Desk checking. The penetrator analyst studies the FHS and the TCB evidence. Code, models, code correspondence maps, or dependency graphs are examined to see if the flaw exists. The analyst must be flexible in considering alternatives, but concentrate on what exists in the actual code and other evidence. Analysts use code walk-throughs, prior test results, their own insights, and conversations with other team members to reach conclusions about the likelihood of the flaw's existence.

Results are documented on the FHS. Confirmed flaws are flagged in the database for later examination. An analyst spends a few days, at most, on each flaw. The desk checking continues for weeks, and possibly a few months, yielding an FHS productivity rate of 10 to 20 FHS per person-month. The work is tedious and detailed, and requires destructive thinking. Occasionally an FHS is of sufficient complexity and interest to warrant a live test, but the investment in the testing process will lower productivity.

Live testing. Test case design, coding, and execution are expensive, so live testing is not the preferred FHS evaluation method. However, testing is often the fastest way to confirm complex or time-dependent flaws. In penetration testing, live tests are similar to computer software configuration item (CSCI) functional tests with the FHS acting as the

(dis)functional specification for the test. There is little unique about these tests, except they may be destructive of the system. Avoid running them on the operational system since they can have unpredictable results. Also, the testing is to confirm the flaw, not to exploit it. Test code should be a narrowly focused (by the FHS) quick one-shot routine. This will be easier to write if there is a rich library of debug and diagnostic routines.

Flaw generalization

When the team assembles for the daily meeting, the confirmed flaws of the day are briefed and examined. Each team member considers the possibility that the flaw might exist in his area, and whether the test technique and code can be used on his FHS. Often a confirmed flaw has only medium payoff value but can be used with other confirmed flaws to yield a high payoff. This stringing of flaws together is called "beading" and has led to many unusual high-payoff penetrations.

Deductive thinking confirms a flaw. Inductive thinking takes a specific flaw to a more general class of flaws. The team examines the basic technology upon which each confirmed flaw is based to see if the flaw is a member of a larger class of flaws. By this generalization of the flaw, one may find other instances of the weakness or gain new insight on countermeasures. Inductive thinking proceeds simultaneously with deductive thinking of new instances of the flaw, so that the flaw becomes a new flaw hypothesis generator. Some classic flaws were discovered by this induction — for example, parameter passing by reference [LIND75, SDC76], piecewise decomposition of passwords [TANE87], puns in I/O channel programs [ATTA76, PHIL73], and time-of-check-to-time-of-use (TOCTTOU) windows [LIND75]. These are described in the upcoming section "FHM experience."

Flaw elimination

Experts have argued the futility of penetrate-and-patch and hack-and-patch methods of improving the trust of a TCB for substantial reasons that reduce to the traditional position that you must design security, quality, performance, and so on, into the system and not add it on [SCHE79]. However, most human progress is made in incremental forward steps. Products improve with new releases and new versions that fix flaws by patching, workarounds, and redesign.

The TCSEC requires that all known flaws be repaired. The evaluators can suggest to the vendor repair of simple implementation and coding errors, or recommend known generic design flaw countermeasures. After repair, the system must be reevaluated to confirm the flaws were fixed and to ensure no new flaws were introduced. Reevaluation is a com-

plete repetition of the penetration testing process. However, application of the Ratings and Maintenance Process (RAMP) [NCSC89a] to B2 and better evaluations may be a possible method to avoid total repetition. This speculation has been tried on the B2 evaluation of trusted Xenix ported to new hardware platforms [MCAU92]. It is impractical for the vendor to fix some flaws. These residual flaws will result in a lower class rating. However, the using agency can prepare a risk analysis that shows the DAA (Designated Approving Authority) alternative security measures to counter the residual flaws.

FHM experience

FHM has been a cost-effective method of security system assurance assessment for over twenty years. Unlike other assurance methods, which focus on narrower objectives (for example, formal correctness proofs of design or risk assessment costs of failures), FHM seeks security flaws in the overall operation of a system due to policy, specification, design, implementation, and/or operational errors. It is a complete systems analysis method that uncovers flaws introduced into the system at any stage of the product life cycle.

FHM management experience. Management models and work breakdown structures (WBS) of tasks, schedules, and labor loadings to perform typical penetration testing are available in the literature [RUB86, WEIS92a]. For weak systems in the TCSEC C1 to B1 classes, experience predicts a typical penetration team of four people operating for six months will generate about 1,000 FHS and assess about 400 of the highest priority. Some 50 to 100 of these will be confirmed flaws. That yields a productivity rate of one flaw for every one to two person-weeks of effort. Stronger systems in the TCSEC B2 to A1 classes, by definition, must be flawless. However, even these systems have flaws — far fewer, of course, because of the greater attention to secure system development. Such flaws are repaired, audited, or considered an acceptable risk. Higher flaw rates may signal a lesser evaluation class than B2 is warranted for the target system.

Vulnerability classes of flaws. Confirmed flaws are sorted into vulnerability classes:

1. flaw gives total control of the TCB/system (TC),
2. security policy violation (PV),
3. denial of service (DS),
4. installation dependent (IN), and
5. harmless (H).

These vulnerability classes are based on the degree of unauthorized control of the system permitted by the flaw — that is, damage extent. The greatest vulnerability is TCB capture; the machine is under total control of the interloper (TC — total control — flaws). Flaws that permit lesser control are unintentional, undocumented capabilities that violate the policy model, but do not allow total control (PV — policy violation — flaws). Denial of service flaws permit the penetrator to degrade individual and system performance, but do not violate the confidentiality security policy (DS — denial of service — flaws). Installation-dependent flaws are weaknesses in the TCB that obtain from local initialization of the system, such as a poor password algorithm (IN — installation-dependent — flaws). Last, there are flaws that are harmless in the sense that they violate policy in a minor way, or are implementation bugs that cause no obvious damage (H — harmless — flaws). These codes will categorize flaws presented in subsequent sections.

Penetration results: Example generic attack methods. This section presents a representative collection of attack methods found effective in finding flaws by the innovative and skilled penetration teams using FHM. An extensive taxonomy of flaws is in preparation by NRL (Naval Research Laboratory) [BULL91].

Weak identification/authentication. Passwords are the cheapest form of authentication. Weak passwords, however, are quite expensive, allowing the penetrator to impersonate key security administrators to gain total control of the TCB (TC flaw). In one system, a weak password protected the password file itself. The password was a short English word that took but a few hours of trial and error to crack. A modern form of this attack on the Unix password file is embodied in a program called Crack, which runs for days as a background task generating popular candidate passwords and trying them for confirmation [MUFF4a]. Since most passwords are initialized by system administration, this is an example of an operational flaw and an initial-condition (IN) flaw.

On the DEC PDP-10 and many contemporary machines, there are powerful string compare instructions used to compare stored versus entered passwords. These array instructions work like DO-WHILE loops until the character-by-character compare fails or the password string ends. It was discovered in the Tenex operating system that the instruction also failed when a page fault occurred. This turned a neat binary password predicate — yes/no — into a trinary decision condition — yes/no/maybe — that enabled piecewise construction of any password in a matter of seconds. In this case the flaw was a weak password checking routine that permitted the user to position the candidate password across page boundaries [TANE87]. This is an example of a hardware anomaly and an implicit memory sharing (TC) flaw.

On the IBM OS/VS2 R3 and similar vintage operating systems, files or data sets are protected by password. When the user is absent during batch processing, surrendered passwords for the file are placed in the Job Control Language (JCL) load deck and queue file for command to the batch process. The JCL queue is an unprotected database, which is readable by any user process, thus permitting theft of passwords. This is an example of a badly designed user-system interface, where system data is placed in the user's address space [MCPH74, SDC75, SDC76]. It is also an example of bad security policy, a policy violation (PV) flaw.

In most systems the user logs into a well-designed password authentication mechanism. However, the system never authenticates itself to the user. This lack of mutual authentication permits users to be spoofed into surrendering their passwords to a bogus login simulation program left running on a vacant public terminal. This spoof has been around forever and is still effective. It is an example of a poor user-system interface that yields a policy violation (PV) flaw. Mutual authentication is a necessity in the modern world of distributed computing, where numerous network servers handle files, mail, printing, management, routing, gateways to other networks, and specialized services for users on the net. Without it, such services will surely be spoofed, modified, and/or falsified. New smart card I&A systems provide mutual authentication by use of public key cryptography [KRAJ92].

Security perimeter infiltration. Untrusted code must be confined and permitted to call the TCB only in a prescribed manner for secure access to needed system services [LAMP73]. These boundary crossings of the security perimeter are often poorly designed and result in "infiltration" flaws. A classic example of this is the uncontrolled channel program of the IBM 370. Since channel programs are allowed to be self-modifying to permit scatter reads and writes and the user can turn off memory mapping (for example, Virtual = Real), it is possible to write into protected system memory and modify code and/or process management data. Attempts to eliminate these problems by static analysis of the channel programs in VM/370 failed to prevent clever "puns" in the code from continued exploitation [ATTA76, BELA74, PHIL73].

Another example of poor confinement is the Honeywell HIS 6000 GCOS suspend feature. It allows a user to freeze an interactive session for a lunch break or longer suspension, and resume later by thawing the program. The design flaw stores the frozen image in the user's file space, including all the sensitive system context needed to restart the code. It is a simple process for a user to edit the frozen image file and modify the context data such that the restarted program runs in system state with total control of the TCB (TC flaw). This is yet another example of an implied memory sharing flaw.

Among the most sophisticated penetrations is the legendary breakpoint attack of Linde and Phillips [ATTA76, PHIL73]. It is an excellent example of a beading attack. When a user plants a breakpoint in his user code, the system replaces the user code at the breakpoint with a branch instruction to the system. The system's breakpoint routine saves the user code in a system save area. Later, when the breakpoint is removed, the user code is restored. The breakpoint feature helps the penetrator plant code in the system itself — that is, the replaced user code — an example of a harmless (H) flaw. It was observed that another existing harmless flaw, a move string error exit, left the address of the system memory map, rather than the address of the string, upon error return. It was possible to induce the string flaw by reference to an unavailable memory page — that is, a page fault. A third harmless flaw allowed control to return to the caller while the string move was in progress in the called program.

The penetrators set up the bead attack by planting a breakpoint at a carefully prepared instruction on the same page as a string move command. They carefully selected a string that crossed a page boundary. They executed the string move, and upon regaining control, released the page containing the end of the long string. That caused a page fault when the string move crossed the page boundary, at which time the breakpoint was removed. In restoring the prebreakpoint user code, the system retrieved the saved user code but erroneously wrote the user code into protected system memory, specifically, the system page map. This unauthorized system modification was possible because a hardware design flaw in the page fault error return left the page address of the system memory map, not the page address of the original user's string. The attack had successfully modified the system map, placing user data in the system interrupt vector table. The attack gave arbitrary control of the TCB — another subtle flaw in implicit memory sharing (a TC flaw).

Incomplete checking. Imports and exports that cross the security perimeter per the TCSEC are either label checked or use implicit labels for the I/O channel used. Lots of flaws occur when labels are not used or are used inconsistently. Another attack exploits interoperability between systems that use different label semantics. The Defense Data Network (DDN) employs the standard IP datagram Revised Internet Protocol Security Option (RIPSO) security sensitivity label [RFC1038]. It differs from emerging commercial standards. Here is a situation ripe for a future security (IN) flaw.

Array-bounds overflow is a particularly nasty flaw and quite pervasive and difficult to counter. The flaw manifests itself in system operation, but its cause is usually traced to the development compiler's failure to generate code for dynamic array-bounds checking. When the array

bound is exceeded, the code or data parameters adjacent to the array are overwritten and modified. In one case the user-entered password was stored adjacent to the system-stored password, so the two strings (arrays) could be rapidly compared. However, there was no bounds checking. The user simply entered a maximum-size password twice so that it overflowed the user array into the system array, creating a password match [BISH82], a certain TC flaw.

Incomplete case analysis leads to flaws. Either the design specification has not considered all the conditions of an IF-THEN-ELSE form, or the programmer goofed. In either event, the penetrator creates the missing condition and forces the code to ignore the consequences, often creating an exploitable state, a PV flaw. The IBM PSW flaw mentioned in the earlier section "System anomalies" is an example.

The IBM 360 introduced the Program Status Word (PSW), which contained a status condition code for the machine instructions that have conditional execution modes. Many programmers ignore the PSW status code and assume the execution result of the instruction. This is poor coding practice but a surprisingly frequent occurrence [BULL91]. Often the programmer believes that prior checks filter the conditions before the instruction execution and that the data cannot cause the unanticipated condition, thus ignoring the condition code. The penetrator must find an opportunity to reset the parameters after the filter checks, but before the conditional code execution. time-of-check-to-time-of-use (TOCTTOU) attacks are exactly what's needed for penetration.

A TOCTTOU flaw is like a dangling participle grammatical flaw in English. The check code is distant from the using code, enabling intervening code to change the tested parameters and cause the use code to take incorrect, policy-violating actions (a PV flaw). The attack is a form of sleight of hand. The penetrator sets up a perfectly innocent and correct program, possibly an existing application program, and through multitasking or multiprocessing, has another program modify the parameters during the interval between check and use. The interval may be small, necessitating careful timing of the attack. The flaw is both an implicit memory sharing error and a process synchronization problem. The solution is not to place system parameters in user memory and/or prohibit interruptibility of "critical region" code [LIND75, MCPH74, PHIL73].

Read-before-write flaws are residue control flaws. Beginning with TCSEC C2 class systems, all reused objects must be cleaned before reuse. This is required as a countermeasure to the inadequate residue control in earlier systems. However, the flaw persists in modern dress. When disk files are deleted, only the name in the file catalog is erased. The data records are added to free storage for later allocation. To increase performance, these used records are cleared on reallocation (if at all), not on deallocation. That means the data records contain residue of possibly sensitive material. If the file memory is allocated and read

before data is written, the residue is accessible. A policy of write-before-read counters this flaw, but such a policy may not exist or may be poorly implemented. This flaw also appears with main memory allocation and garbage collection schemes. In one example, the relevant alphabetical password records were read into memory from disk for the login password compare. After the compare, the memory was left unchanged. Using other attacks, such as the array-bounds overflow described above, that residue memory could be read and passwords scavenged. By carefully stepping through the alphabet, the complete password file could be recreated (a TC flaw) [LIND76a].

Planting bogus code. The most virulent flaws are created by the penetrator inserting bogus code into the TCB (a TC flaw). Bogus code includes all forms of unauthorized software: Trojan horses, trap doors, viruses, bombs, and worms. Fundamentally, flaws that admit bogus code are flaws in the configuration control of the TCB. The flaw may occur any time throughout the life cycle of the TCB. When development tools are uncontrolled, bogus code can be imbedded in the tools and then into the TCB. Ken Thompson's ACM Turing Lecture aptly documented such an attack [THOM84]. But there are easier methods — for example, planting bogus code in the runtime package of the most popular compiler and/or editor. Recent attacks on the Internet were exploitations of poor configuration control. Known flaws in Unix were not fixed with the free vendor patches. The hacker used the flaws to obtain unauthorized access [SPAF89a].

Among the more colorful attacks against human frailty in controlling bogus code is the Santa Claus attack. It is a classic example of an unauthorized code import, achieved by spoofing the human operator of a secure system. A penetrator prepared an attractive program for the computer operator, who always ran with system privileges. The program printed a picture on the high-speed printer of Santa and his reindeer — the kind you always see posted at Christmas in and about the computer facility. However, there was a Trojan horse mixed in with Dancer, Prancer, and friends that modified the operating system to allow undetected access for the interloper. Before you belittle the computer operator's folly, consider your own use of "freeware" programs downloaded from your favorite bulletin board. There are as many user and operator spoofs as there are gullible people looking for "gain without pain." Eternal vigilance is the price of freedom from spoof attacks. Also, improved role-authorization controls can limit the damage propagation of such flaws from human foibles.

Penetration analysis for the 1990s and beyond

This essay concludes with speculations on future directions of penetration analysis. The assumption is that the battle between the TCB developer or operator and the hacker will continue unabated. Better legal protection for users will always trail technology, and the technology will improve for both antagonists. It will be continuous digital electronic warfare among security measures, hacker attacks, user countermeasures, and security counter-countermeasures: perpetual offense against defense.

The defense, developers of TCBs, are using better methods of designing and implementing trusted systems. Lessons are being learned from past penetrations, both tests and real attacks. The generic flaws are leading to better understanding of security policy for confidentiality, integrity, and service availability, and of the confinement of overt and covert channels when sharing common mechanisms in trusted systems. This understanding is being captured in improved machine architectures with segregated privilege domains or protection rings to reinforce security perimeters and boundary crossings. Improved hardware supports safe and rapid context switching and object virtualization. Cryptography is playing a larger role in confidentiality, integrity, and authentication controls. Computers are getting faster and cheaper so that security mechanisms will be hardware-rich and not limit performance in secure solutions. Software is improving in quality, tools, development environments, testing standards, and formalism.

Applicability of FHM to ITSEC. It has been questioned whether penetration testing has meaning for a Target of Evaluation (TOE) of the European Information Technology Security Evaluation Criteria (ITSEC), since "penetration testing" or its equivalent never appear in the criteria [ITSE90]. For most of the reasons expressed earlier in this essay, penetration testing will be required for ITSEC high-assurance evaluations. However, the ITSEC is different from the TCSEC and the TCSEC interpretations. The applicability of FHM to ITSEC for the 1990s is discussed here.

A TOE is either a security product or a system to be evaluated. Penetration testing of a TOE is comparable in scope to penetration testing in accordance with the TCSEC. However, a TOE's evaluation criteria consist of two-tuples (F*i*, E*j*): F*i* is one of 10 security functionality classes, and E*j* is one of seven independent evaluation levels. To match the TCSEC classes of FHM interest, we have the two-tuples (F4, E4), (F5, E5), and (F5, E6), which correspond to B2, B3, and A1, respectively. The first five functional classes of the ITSEC, F1 through F5, match the six functional classes of the TCSEC, C1 through A1, with F5 functionality the same for B3 and A1.

The seven ITSEC evaluation classes are applied to the TOE Development Process (DP), Development Environment (DE), and Operational Environment (OE), each of which is further divided as follows:

- DP: Requirements, architectural design, detailed design, implementation.
- DE: Configuration control, programming languages and compilers, developer security.
- OE: Delivery and configuration, setup and operation.

The ITSEC is so new there is no practical experience to draw from in looking at the applicability of FHM, so we look more closely at the text. Although penetration testing is not mentioned, testing is widely referenced, particularly for the evaluator to perform "his own tests to check the comprehensiveness and accuracy of the sponsor testing, and also to address any points of apparent inconsistency or error found in the results of the sponsor's testing" [ITSE90]. Testing for errors and vulnerabilities is required even at evaluation class E1, retesting of corrected flaws is required at E3, independent vulnerability analysis is needed at E4, and all these requirements are cumulative with higher evaluation classes. These test requirements are quite similar to those addressed by the FHM described in this essay.

Assurance of a TOE is divided in the ITSEC between correctness and effectiveness. Correctness is based on the seven evaluation classes E0 to E6. Effectiveness of a TOE involves a number of considerations: the suitability of the security functionality for the proposed environment, analogous to the TCSEC environment guidelines [CSC85]; whether the functionality yields a sound security architecture; ease of use of security functions; assessment of the security vulnerabilities during development and operation; and the strength of the security mechanisms to resist attack. All these items are "generators" in FHM (see the earlier section "Flaw generation").

The FHM depends on discovering failures in the evidence implication chain, starting with a security policy (see the earlier section "Evidence implication chain"). The application of FHM to a TOE would require a similar procedure. A TOE has a hierarchy of security policies: a System Security Policy (SSP), a System Electronic Information Security Policy (SEISP), and a Security Policy Model (SPM), corresponding to security objectives, detailed security enforcement mechanisms, and a semiformal policy model, respectively. These policies are tied to the functional classes and form the basis for the correctness criteria for testing. Together with the evaluation classes, an evidence implication chain is formed for a specific TOE, and FHM can be successfully applied.

In conclusion, FHM should be equally as applicable to ITSEC as to TCSEC/TNI/TDI evaluations. Under both sets of criteria, the most sig-

nificant open question is: How are evaluators to judge the security of a system composed of individually evaluated components? The composability controversy is beyond the scope of this essay. However, FHM is applicable to such composed systems and may be part of the controversy's resolution.

Formal methods of penetration testing. Formal methods use rigorous mathematical specifications of the TCB design (that is, FTLS), assumptions, initial conditions, and correctness criteria. Formal tools take these specifications and generate correctness theorems and proofs of the theorems and the design they portray. It is hoped that these formal methods can achieve similar success at the level of code proofs. A new form of penetration analysis is in progress with new TCB designs — rigorous formal models of the TCB. These models describe the TCB behavior as state machines, state-transition rules, security invariants, initial conditions, and theorems that must be proven. Penetration analysis is almost inseparable from the formal design process, producing conjectures of flaws with the model and trying to prove them as theorems. This is a rigorous extension of FHM. If successful, the correctness proof versus flaw conjecture proof becomes part of the design process and uncovers problems early in the design, enabling iterative redesign — unlike FHM, which often comes too late in the development cycle to permit more than hack-and-patch.

Recent work by Gupta and Gligor suggests a theory of penetration-resistant systems. They claim their method is "a systematic approach to penetration analysis" that "enables the verification of penetration-resistance properties, and is amenable to automation" [GUPTA91, GUPTA92]. They specify a formal set of design properties that characterize resistance to penetration in the same framework used to specify the security policy enforcement model — a set of design properties, a set of machine states, state invariants, and a set of rules for analysis of penetration vulnerability. Five penetration-resistance properties are described:

1. system isolation (tamperproofness),
2. system noncircumventability (no bypass),
3. consistency of system global variables and objects,
4. timing consistency of condition (validation) checks, and
5. elimination of undesirable system/user dependencies.

Gupta and Gligor contend system flaws "are caused by incorrect implementation of the penetration-resistance properties [that] can be identified in system (e.g., TCB) source code as patterns of incorrect/absent validation-check statements or integrated flows that violate the intended design or code specifications." They further illustrate how

the model can be used to implement automated tools for penetration analysis. They describe an Automated Penetration Analysis (APA) tool and their experiments with it on Secure Xenix source code. Early results from this work indicate that penetration resistance depends on many properties beyond the reference monitor, including the development and programming environment, which is characterized as the evidence implication chain earlier in this essay. Although limited only to software analysis of attacks on the TCB from untrusted user code and leaving significant other system avenues for attack, the work may pave the way for new approaches to building and testing trusted systems and tip the balance in favor of the "good guys."

Automated aids. Earlier, in the infancy of penetration testing, it was believed that flaws would fall into recognizable patterns, and tools could be built to seek out these generic patterns during penetration testing [ABBO76, CARL75, HOLL74, TRAT76]. Unfortunately, the large number of different processors, operating systems, and programming languages used to build TCBs, with their different syntax and semantics, made it difficult and impractical to apply such pattern-matching tools to penetration testing. It is costly to port or reimplement the tools for each new environment and different programming language. Also, the flaw patterns tended to be system specific.

As modern secure systems focus on a few operating system standards (for example, Unix and MS-DOS) and fewer programming languages (Ada and C), and more mature expert systems become available, future penetration testing tool initiatives to capture flaw patterns may be more successful. A few specific examples of this trend are beginning to appear: antivirus tools for the popular workstations and PCs; Unix security test tools [FARM90a, MUFF4a]; intrusion detection monitors [DENN86, SMAH88, BAUE88, DIAS91, LUNT92, WINK92]; and formal specification proof tools [KEMM86], flow analyzers [ECKM87, KRAM83], and symbolic execution tools [ECKM85, WHEE92].