Java's Evolving

Security Model : Beyond the Sandbox for

Better Assurance or a Murkier Brew?

SANS

Global Incident and Analysis Center

GIAC Security Essential Certification Practical

Matthew J. Herholtz

March 2001

Matthew J. Herholtz
GSEC Practical **March 2001**

Generally, Internet compressed software development schedules have outstripped the need for rigorous security design and analysis. This is a common trend in consumer software, and one that is destined to flame-out under the rigorous security demands of e-commerce.

Felten, McGraw

Overview

Sun Microsystem's Java is a portable proprietary single programming language that manages distributed network security challenges of mobile code or executable content downloaded from the web. Java's undisputed popularity and universal acceptance, without formal assurance verification of the languages security model, has drawn extensive and justifiable scrutiny from security experts. Many Java security vulnerabilities have been discovered, well-publicized and subsequent fixes deployed. Numerous recommended articles, to include SANS practicals, discuss these and other complimentary aspects of Java security. This paper will not discuss specific Java exploits in detail. Instead, it introduces and applies security policy and security model principles to Java's evolving security model to better understand it, and to discern confidence in Java's policy strategies for the future of secure distributed computing.

Security Policy and Security Models

First and foremost - - a security policy IS NOT the same as a system's security model. Confusion between security policies and models sometimes results as both have multiple meanings. Although sometimes subtle, the distinction between policy and models is crucial for critical information security systems like electronic commerce systems that require a high degree of trust and assurance of quality design and operational reliability.

Generally, a security policy is expressed in words and emulates real world security challenges as a set of laws, rules, roles and procedures defining how an organization allocates resources to protect assets and achieve business security objectives. Security models on the other hand, tend to be either mathematical (formal methods), graphical (latticed information flows) or tabular (access matrix) for ease of system policy conceptual visualization, understanding, implementation and verification. [7, 103]

Security policy normally has three fundamental types, each with a different purpose: organizational, issue and system specific. An *organization's security policy* is management's security strategy for protecting assets and attaining an enterprise's business security objectives. It outlines specific employee roles, responsibilities, rules and procedures for compliance. An *issue or technology specific* policy outlines similar responsibilities, details and procedures dealing with a specific technology, system or behavior challenge like e-mail, physical security, software, Internet, or LAN. Lastly, a *system policy* must address internal computer system or application's technical mechanisms (e.g. *automated system or application security policy*) that controls the security access policy for subjects and objects within a specific computer system.

Security policies vary from organization, business, environments, technical application and network systems.

A renowned expert and author in the security field, Edward Amoroso, defines the "*security policy* of a *computer system* [system specific] as the conditions [or set of rules] which subject accesses to objects are mediated by the system reference monitor functionality … with appropriate mechanisms [for policy enforcement]." [1, p.91] Successful mechanisms, normally an underlying part of the computer environment, enable effective implementation and enforcement of security policy requirements. [1, p. 92] Further, the Department of Defense (DoD) 'Orange Book' Trusted Computer Systems Evaluation Criteria (TCSEC) 5200.28 defines [system specific] security policy as: "Given identified subjects and objects, there must be a set of rules that are used by the system to determine whether a given subject can be permitted to gain access to a specific object." [8, p.112]

Computer security *models* restate the policy in more *formal or mathematical terms* to: (1) provide framework for understanding concepts; (2) provide an unambiguous, formal representation of a general security policy; (3) express the policy enforced by a specific computing system. [7 p. 110]

Security models may be formal or informal. Formal models are based upon logic and formal mathematics, information or complexity theory. As conceived, a computer system's foundation of mathematics readily lends itself to mathematical accuracy by a logical theorem or mathematical proof. A common formal model, Bell LaPadula (BLP), features latticed information flow, strict labeling and mandatory access control (MAC) emphasizing confidentiality (common in military models) for secrecy. In commercial and medical systems, Biba and Clark -Wilson integrity oriented models are common and focus on integrity levels of information flow for transaction accuracy and separation of duty. [7, pp.112-145] Informal models can be precise narratives of system operations supplemented with concept facilitating diagrams, graphs or tables like an access matrix. Discretionary access control (DAC) common in academic research (UNIX, V/MLS informal documentation) is typically modeled by a simple access control matrix (Figure A) Mobile code technologies are relatively new, and evolving. Java and Active X trust models implement restrictive sandboxes, protection domains and/or digital signatures. Whether formal or informal, models are a means to abstract essentials from the typically ambiguous policy narrative to precisely represent and better understand the policy and evaluate assurance of the system's security.

Author Amoroso acutely summarizes the important distinction between system *specific security policy* and *models*. Computer **security policy** is a set of requirements for a *specific system*; a **security model** is a *restricted representation* of a class or types of systems that *abstracts unneeded details to highlight a specific property* or set of behaviors. Models then, ideally, and in theory, are useful to design and better understand a *specific system* policy and provide a general framework for understanding a type of system's implementation of [security] mechanisms employed to enforce a defined policy. [1, p.92] In short, real life verification of a model's mechanisms to accurately mediate an automated security policy ultimately determines the success of an information system's 'state of security' and validates its design.

Assurance, Verification and Validation

Computer security assurance is the degree of confidence one has that the security measures, mechanisms both technical and operational work as intended to protect the system and information processes. [4, p. 89] Additionally, assurance of a security policy or model establishes unambiguous grounds for confidence that the model meets or accurately portrays system security objectives. [18] Assurance, is confidence in proper or correct system design and the capability of a technology to execute as specified. Primary assurance methods are testing, verification, and validation.

Verification includes documentation of actual penetrations, or attempts to penetrate an on-line system in support or in contradiction of assumptions developed during system review analysis. Verification determines if system implementation is consistent with system requirements specification." [7, p. 238] It implies correctness and confidence through formal testing and verification activities.

Validation includes verification, and also includes 'rigorous' methods for convincing people of the correctness of a program or a security model. Validation is "… concerned with predicting how well a system and its security policy will meet real world needs and expectations, and is partly always subjective"[7, p. 238] Typical validation techniques include white and black-box testing, formal technical reviews, design and code reviews, penetration tests, quality and disciplined mature coding and engineering development processes as outlined in ISO 9001. (See Annex VIII)

The most popular, economical and typically only means for security system model verification and validation includes module and system testing. However, it is crucial to understand that this only demonstrates the existence of suspected flaws, not the absence of flaws. Testing is based upon observable effects, but does not ensure any degree of completeness. Further, testing is time and resource intensive, complex and subsequently conveniently curtailed in commercial arenas. [6, pp. 308, 312] Post fielding operational system 'penetration testing' sometimes called 'ethical hacking' best done by an objective third party, advocates enhancing system assurance and discerning a relative pulse setting of system security by 'breaking into your web site'. More rigorous security assurance methods of design and evaluations include: TCSEC Orange book (Trusted Computing System Evaluation Criteria – U.S.), ITSEC (European Information Technology Security Evaluation Criteria) and Common Criteria respectively.

Research, development, design and deployment of various successful and other less so successful security systems have proven that articulating a system's security policy(s) and model(s) dramatically enhances a system's development and promotes mechanism verification for security policy enforcement. An accurate model is indispensable in providing system proof of principle without likely expensive iterations of system prototyping, and provides the benchmark for modifications. This is very pertinent in the case of Java's ever-evolving security model.

Although policies and models vary with environment, the scope and sensitivity of information being protected, authors Ford, Summers and Amoroso generally define essential threat deterring networked computer system security services to include authentication, confidentiality, authorization, availability, and integrity. [1,5,7] (Annex II) Assurance that these essential services can be reasonably delivered is suspect unless the security model's mechanisms implementing the system's underlying policy can be *verified*. In lieu of formal methods for assurance verification, an evaluation based upon sound, time-tested subjective informal security principles may be employed. These principles include least privilege, simplicity of mechanism, complete mediation, open design, least common mechanism, ease of use, and tamper resistance. [11, pp.1-2] (See Annex I)

Network or distributed applications security can be viewed as "an instantiation of a computer security problem for a specific type of application of threat vulnerability, attack assessment, model, policy definition and safeguard and countermeasure implementation". [1, p.318] This, unfortunately, seems overly simplistic given computer network complexity, their innumerable possible configurations, scope, and applications in equally innumerable possible environs. Applying our earlier definition of security policy as required behaviors for a *specific* system to distributed applications like Java that 'ports' itself to multiple browsers, and platforms tends to be extremely problematic – namely networked distributed applications *inherent lack specifics* in scope and definition. Therefore, some latitude in assumptions to scope this paper's context is taken. Never the less, security models and principles are invaluable in defining and evaluating security requirements and can immeasurably enhance designer and user confidence in verifying and validating a system's security and assurance.

With the necessary security policy and model essentials covered, the rest of this paper will focus on Java's ever evolving security model and its assurance.

Java First Generation (JFG) Security Policy and Model (Java 1.0, Java Development Kit JDK 1.0.x, JDK 1.1.x)

Java is a powerful object oriented programming language similar to C++. Sun Microsystems designed and implemented Java in 1995 for distributed networked computing with the goals of hardware portability, operating system (OS) independence and extensibility for *secure* web based applications. Embedded into a client user's web browser (e.g. Netscape, HotJava or Microsoft Internet Explorer), Java operates near seamlessly without user interaction, incorporating it's own environment or Java virtual machine (JVM) to execute downloaded code called 'applets' from remote Internet servers on the user's hardware platform. The security implications to the unwitting user executing remote code of unknown origin and intent, which could surreptitiously contain a malicious 'Trojan horse', are obvious. Java's proactive but informal design approach to such challenges exhibited several flaws, but was not the typical "penetrate and patch" security development paradigm. Even still, Java's security model can best be described as evolutionary.

Central to Java First Generation (JFG) security policy and model is the principle of containment of processes to a *Sandbox* concept. The Java sandbox entrusts local code (OS, Java application, and web browser) with full access to vital system resources, but restricts execution and interaction of suspect bytecode, and remotely downloaded applets. Essentially, the default Java sandbox includes three components: bytecode verifier, ClassLoader, and SecurityManager. In JFG, all three work together in the Java security model to enforce an all or nothing, trusted or untrusted security model. All three components are critical to the security of the model, and if one is compromised, like links in a chain, Java's security is broken. [11]

The original Java sandbox model was conceptually sound, and intuitive to understand. Client requests to servers for programs (applets or executable content), downloaded from the Internet are assumed untrusted, while other sensitive Java applications, local platform code and resident system (operating systems OS) components are trusted. Java limits security breaches via segregating untrusted downloaded executable content or 'bytecode' into a *sandbox* segregated from system and resident Java application resources (objects). Further, untrusted applets are assigned their own memory name-space, limiting interaction with other programs, data and system classes. Mixed source (server and host) bytecodes are verified, compiled, then run in a system or browser's native Java runtime environment. Attempts to access objects outside this sandbox by ClassLoader objects is checked by the Java SecurityManager (see Annexes III, IV, V, VI, and VII). This distinction among objects and methods restricts access and safeguards essential system resources (e.g. memory). Although functional, simplistic, and portable, Java's originally model and policy were poorly documented and demonstrated fundamental weaknesses of security design principles addressed later.

JDK 1.1 enhanced fundamental security features with basic protection domains, access control, authorization and selected method delegation via the security policy manager [13]. It facilitated digitally signed applets for authentication and integrity. Digitally signed (X.509) and verified downloaded applets are considered trusted and run with local code and Java applications (like Java 1.0). Unsigned applets still remain in the sandbox. Signed applets are bundled with signatures and necessary object files into a Java Archive (JAR) formatted file for performance enhanced network transmissions. Lastly, the resident Java SecurityManager enabled savvy programmers to somewhat tailor security levels restricting access permissions of 21 restricted risky operations (ANNEX II). JDK 1.1's black and white security policy, like Java 1.0 views code as either *trusted* or *not trusted* at all. However, digital signature verification of downloaded applets from remote web or Internet based servers enabled *them* to be trusted. The many JFG versions of JDK (1.0.x, 1.1.x, 1.2) generally conform to the following four layered functional model. In summary, because much of Java security and safety depends upon language mechanisms, many refer to this model as language-based security protection.

Java Security Model Environment has Four System Layers. (Refer to Annexes III, IV, V, VI, VII) [From 16]

- Layer 1 – Java Programming Language (ensures semantics/syntax, memory access protections, strong typing (safety) e.g. no forged pointers, buffer overflow, memory leakage. Segregates name-spaces (memory) of local and network obtained resources.)

- Layer 2- Java Virtual Machine (JVM) normally resident on a Client's Web browser- (ensures typed memory access, **byte-code verifier***, memory garbage cleansing/reallocation).

- Layer 3- Libraries/**Class Loader***(s). Three types with 2 basic functions: Internal, Class Loader Objects (applet, RMI, Secure) Roll your own classes (access to files, and network resources- implements network classes/objects, methods/functions, disallows unauthorized access, maps names to class objects, invokes security for necessary classes) 1) Instantiates bytecode as classes. 2) Manages namespace [3]

- Layer 4 – **Security Manager*** /Runtime environment (defines & implements security policy, centralizes access control). Configurable portion of model, client or platform independent (discretionary access control). Layer 4 must be specifically SA configured to invoke security resident of bottom three layers. Layer 4 allows tailoring of the Java Security Model to specific security policies.

* Essential components of Java's security kernel, Security Manager as the reference monitor.

Policy IAW with the above four-layer model is primarily implemented by the abstract classes ClassLoader and SecurityManager. During runtime, the ClassLoader loads and labels remote network and system native objects and methods whose accesses are controlled by the SecurityManager. The SecurityManager controls access of public, private, and protected classes, methods and variables verifying legitimate ClassLoader origin. Depending upon the environment's specifics policy, normally less stringent than Java's default policy, a security exception is initiated for an unauthorized object access (Annexes V, VI, VII) requiring a local configured policy access decision. [16]

Java Development Kit (JDK) JDK ver 1.2 or Java 2.0 ^{TM} Platform

The Java 2^{TM} $^{(}$JDK ver 1.2) platform significantly improves upon the inflexibility of *the original black and white sandbox model* (recall in original sandbox model all objects are either trusted, or not-trusted) with the abstraction of protection domains, access controller classes and use of digital signature encryption to verify applets (refer to Annex VII). In Java 2, the sandbox concept of security, once a model fundamental, is reduced to default. Fundamentally, Java 2 employs a very complex trust model implemented by an even more complex memory stack inspection process. Java 2 employs various levels of security and different mechanisms including multiple restrictive domains (like many of *the* old sandbox) and digital signatures for authenticating remote applets. Java's ClassLoader manages distinct protective domains or memory namespaces for applets to execute. Conceptually, one JVM could have several running ClassLoaders that invoke many 'on-demand' virtual protective domains for instances of methods and classes. Protection domain classes eliminate the notion of 'trusted local Java application code', bolstering the overall model's security. Principals (entities with authority) are assigned permissions and likewise accountability. Each domain with specific rights protects or encapsulates resource classes of the same granted permissions. Applets or applications segregated by permissions execute in restricted domains or namespace unique to their CodeSource. This CodeSource consists of its code base (where it comes from-e.g. java.net.URL) and the set of authorizing certificates verified by public-private key pairs. To reduce risk, classes of like permissions, but from different code sources are 'hidden' in separate namespaces or domains and theoretically, cannot collaborate for an attack. Digitally signed verified applets may be fully, partially or not trusted depending upon Security Manager policy configuration. Domains (e.g. system protection) are furthered defined by AccessControllers sub-classed by the SecurityManager to establish each environment's unique specific security policy. Since local administrators can tailor an environment's SecurityManager for a specific access policy, or choose an API default policy, Java 2.0's security policy can best be described as a form of *discretionary access control* with various permissions and levels of trust essentially granted to application and system protection domains. SecurityManager invoked AccessControllers classes refines Java 2.0 's access control granularity to further mediate access to critical system, and privileged code. This insures permission contexts between various class instances and system states are consistent and safe. This complex implementation is done by a sequential memory stack inspection of principals; associated methods and assigned domains; discussions essentially beyond this paper's scope. As before, Java's ClassLoaders insure all applets, resident files, applications, crucial system files (in system domain), and invoked classes are subjected to rigorous type checking, protection domain authorization and access control verification of the JVM and SecurityManager. Although a relatively complex security model, JDK 2.0 does provide a friendly GUI interface – SDK to facilitate security administration for configurable 'fine grained' authentication and authorization based specific security policy.

Java 2's protection domains and memory stack inspections are applauded as an improved design measure to *mitigate* future programming security risks. However, implementation is complex when considering multi-thread computations operate and have access through multi-domain (e.g. system, application) transitions. With such memory transitions and application definable or 'roll your own' ClassLoaders, and local SecurityManagers, it seems crystal clear why Sun Microsystems left the complex specification for application domains conveniently to programmer's application developers. Security analysts argue that application definable ClassLoader and SecurityManager objects, which perform crucial tasks of byte-code instantiation and namespace memory management, is a poor model fundamental which excessively decentralizes control of security and increases security risks. [3] It essentially violates a key design principle that security reference monitors must be tamperproof. [8, 11]

Java Access Control Model

Java's security model is based upon discretionary access control (DAC) with varying degrees of permissions or trust granted to principles restricted to protection domains. (Annex VI) DAC essentially means that an object's owner or creator completely determines access [and to some degree, security policy] authorizations. Meaning, "…a certain amount of access control is left to the discretion of the object's owner, or anyone else [e.g. Client Administrator] who is authorized to control the object's access." [5, p. 290] The SecurityManager and AccessController subclass are the fundamental security mechanisms for access control, similar to a security reference monitor, which can be modeled via an access matrix. In Java 2.0, the SecurityManager, which is an invoked class itself, mediates various methods or privilege (Annex V) requests the SecurityManager and AccessController must appropriately authorize.

Figure A: Access Control Model

Object/Classes (o)

SUBJECTS (S)

	Admin	SecMgr	User D_1	ClassLdr	System Class D_1	App - Class D_2	UsrFile	SysFile
Admin	o,r,w,x	r,w,x					r	o,r,w,x
SecMgr	o,r,w,x	o,r,w,x		x		o,r,w,x		r,x
User D_1	o,r	x	o,r,w,x	x	o,x		o,r,w,x	
ClassLdr	x	x	o,x	o,r,w,x	w, x	w, x	x	
System PD_1			o,r,w,x	x	o,r,w,x			
App PD_2			o,r,w,x	x		o,r,w,x	w,r,x	

For this informal model we define:
Subjects: {Administrator, SecMgr, User, ClassLoader, ProtDomain$_{1,}$ App ProtDomain$_2$}
Objects: {Administrator, SecMgr, User, ClassLoader, System Class (of Domain$_n$), Application system class domain $_2$, UsrFile, SysFile }

Privileges: {o - owner, r - read, w - write, x - execute}

Assume: 1. All classes mediated (access controlled) by SecurityManager (default)

2. Class accesses during Java runtime environment w/ SecurityManager invoked.

The access matrix is intuitive and demonstrates that subjects creating objects, generally either possess all possible privileges for that object or class, or are restricted to privileges in accordance with DAC security policy enforced by the SecurityManager class (Annex V). For example, system UserDomain1 (D1), is an 'ordinary user' of basic access privilege can {r,w,x} (read, write, execute) its own files but not critical system files. The System Administrator defines the security policy and therefore has full privileges to the SecurityManager. The ClassLoader must verify and execute files that are IAW SecurityManager policy accept critical system files protected by the system protective domain. Likewise, user of protection domain1 has full privileges to invoked class objects of domain 1, but not of a subsequent classes in domain2. Although the SecurityManager can read and execute strongest protected system files, only the SA has full system access privileges. In summary, DAC is the right to execute programs, and methods on specified objects according to an owner (user) or administrator configures the SecurityManager policy. For further safety and mediation, only one SecurityManager is invoked per JVM session.

Intuitively, then, a subject request to access a system critical file by an object outside of the protected system security domain would be mediated false by the SecurityManager and 'throw a security exception'. Admittedly, Java's flexible DAC policy of varying trust levels is complex and difficult to specifically model accurately even informally. Perhaps this is why know one, including Sun has yet embraced the public challenge to formally prove the security of Java's specific policy and model.

Java employs many mechanisms at each of its security models four functional layers. However, the critical question remains – does this 'protective environment' clearly articulate a sound and *specific* model with verifiable mechanisms for high assurance? Some experts agree, "Without a formal high level defined security model, the policy of JDK1.0.x boils down to low-level detail checking" [3, pp. 2-13]. Indeed, without an initial more concrete specific statement what the JFG security policy *was*, *is* and *plans to be*, given the above layered ever-evolving model, Java's security will likely continue to 'morph' in the dynamic environ of remote computing and distributed applications. Perhaps this *was* the ambiguous model that Sun envisioned from inception. Even Li Gong, Java's Security guru, refers to "Java's *evolving* sandbox and security model". [13, p.2]

General Java Security Model Limitations

Although Java purports a four-layer model, the perceived depth or bastions (rings) of protection are misleading. Unlike the 'defense in depth' security approach of the MULTICs (MULIT-plexed Information and Computing Service) with concentric protection rings to be penetrated, in Java an attacker needs only breach one layer or component, like the byte code verifier, for a significant system security violation. In fact, it is erroneous to assume that these layers do indeed mutually and functionally support each other for perceived graduated levels of assurance as in the system ring design of MULTICs. "…The parts [of the security model] are more like links in chain: If any of the three (verifier, ClassLoader, SecurityManager) parts breaks, the entire system breaks." [3] Or alternately stated, defensive aspects do not overlap for mutual support. For example, if an invalid bytecode pattern is missed by the verifier, no other level of the JVM will be able to catch this oversight." [11, p. 37] In fairness however, Sun's admittedly conservative initial goal for Java's earliest versions, was "to enable browsers to run untrusted code in an entrusted [trusted] environment." [13, p.10]

Although JFG is a recognized pioneer for distributed computing and remote applications execution, both Sun Microsystems (HotJava) and Netscape Navigator distinct browser application versions exhibited several security failings that were incrementally found, subsequently rectified and resulted in some 21 security class check methods (See Annex VII). There are undoubtedly more to be exploited. In fact, on average, a flaw is exploited every four months in Java. [3] For *secure information systems*, standardization, design methods and quality control and assurance is of grave concern. Java's SecurityManagers are unique to browser vendors and only as capable of enforcing sound security as the vendor designs them. As a failsafe mechanism, browsers start only one SRM per virtual machine installed at any given time. [3, p 2-8] Disconcertingly, the means of implementing memory stack inspection for protection domains and digital signatures for identity verification is inconsistent and not interoperable between three primary browser vendors.

- Language and Bytecode Flaws. "The Java language has neither formal semantics, nor formal description of its type system. Java lacks a formal description of its type system, yet the basic security of Java relies on the soundness of its type system." "Type verification cannot be proven. Object-oriented type system's security a current research topic; it seems unwise for the system's security to rely on such a mechanism without a strong theoretical foundation." [11, p.7, 8] These flaws have led to several improvements in Java's security model definition and more robust implementation. Third party evaluators stipulated "These are all issues that can be addressed with good design practices and code reviews." [11, p.9]

- Strength of mechanism. Access control is provided by a browser's runtime invocation of the SecurityManager class, which is not adequately protected by Java language strong typing. Strength of this access mechanism is suspect given above. Additionally, the SecurityManager was not always invoked, not tamperproof and cannot be (due to Java poor language semantics) be verified. Sun has introduced improvements via the AccessControl class in Java 2.0 TM using stack inspection techniques for protected security domains mediated by an AccessController sub-class.

- Simplicity of mechanism. Suspect. Dean et al. demonstrated that verified bytecode could invoke classes not possible in compiled native Java code. Bytecode can possibly traverse the security model in at least three ways. The additional code combining the JVM and system Java applications seems over cumbersome and not 'simple'. Many ask, "Why even have the bytecode" – "and why not transmit encrypted source code over the network?" Java 2's protection domains whose methods stacks may be traversed by multiple threads of various levels of trust and invocation (browser application, system, other domain namespace) is certainly not 'simple'.

- Complete Mediation. This principle is violated in Java. Critical runtime environment initial design flaws in mediating authorizations enabled untrusted applets access to system native methods. Independence and mediation is a design goal of Java but still problematic with foreign code sources executed in systems of multiple security levels in networked environs given issues of delegation, association, inheritance and polymorphism.

- Auditing. No *default* audit trail exists in Java to reconstruct and analyze a breach. [11 p.9]. When and security violation occurs or a JVM exception is thrown, precluding a security breach, no record or logging is *automatically* done. Consequently, the lack of audit data inhibits detailed post mortem analysis for forensics and evidence gathering. Developers however, may provide an auditing security manager. At a minimum, a reliable audit trail must record local file access reads and writes and network accesses. Sun promises to rectify this. Given Java's evolving scale, security APIs enabling pruning events will be a management necessity. Common threats include masquerade (misuse), information disclosure, and integrity violations. Rectifying access violations mandate a refined audit system be resident in Java's security architecture. Consequently, Java's model design suffers from a lack of defense in depth. There are no redundant or backup security mechanisms. The integrity of critical security components: verifier, ClassLoader and SecurityManager must remain intact or the entire security model is defeated.

- Verification. Verification of both policy and subsequent Java model mechanisms is suspect given inadequate formal definition and specification of same. Java's runtime system is some 28,000 lines of code long and is continuously growing, making it difficult to verify. [11] Lack of verification, implies 'buggy code' inevitable leading to vulnerabilities. Finally, given three separate web browser application implementations, there is no centralized authority ensuring Java security code is bug free. [3, pp. 2-13-2] Another related, but secondary issue is of Microsoft's Windows 2000 apparent 'melding' of operating system and browser functionality to a similar 'look and feel', to further blur verification of the security functions of these distinct entities.

- Ease of Use. Java seamlessly executes downloaded active content without a user to worry. Or, maybe not? User interaction is all but eliminated, but the complexity of Java's multiple trust model is of concern. Administration is simplified via a JDK 2.0 GUI for defining security policies of finer grained access control and enforcement. Configuration and management of these various specific 'finer grained' security policies, particularly at the enterprise level, is certainly not 'easy'

- Tamper Resistance. Many documented attacks attest to Java's lack of resistance to tampering. Sun admitted known Java Security flaws are documented on their web-site and annotated in several references attest to Java's, various related browser and OS environs affinity to tampering. Additionally, components of Java's reference monitor may be modified for to enforce policies for various applications and environs. Such modifications provide flexibility in the model, but may inadvertently introduce flaws and vulnerabilities. However, to be fair, the majority of Java security shortcomings occurred in the laboratory 'with only a few manifesting in the wild'.

- Ambiguity. See Language and Bytecode Flaws above). Lacking an initial formal security policy and model specification, Java's security model is certainly ambiguous to some degree. Lack of defined system policy requirements directly violates the TCSEC Orange Book fundamental Criteria of: "There must be an explicit, and well-defined security policy enforced by the system. However, Java's evolutionary 'beta test in the market' approach to 'secure' mobile code design has rooted it strongly in the industry for years to come. The lack of a Java's formal specification is worrisome in light of its requirements for and current employment in e-commerce enabled and highly sensitive applications.

To their credit, Sun improved Java's assurance by welcoming third party 'open design' evaluations that surfaced these and other security shortcomings. Problems uncovered by the Blackwatch group and by Princeton University professors, Dean, Felten and Wallach, did much to educate the community on the scope of the distributed object security challenge and bolster Sun's consumer image in their positive attitude and response to rectify flaws. Many flaws were immediately fixed attesting to Java's apparent evolution type development paradigm. However, what's even more troublesome then the specific uncovered flaws, are those most assuredly, left to be discovered? This unfortunately will be the case given the absence of formal analysis, testing and design rigor in Java's development methodology felt necessary by the community to field a *secure* information system.

Conclusions

Java's security model is far from formally defined, verified and validated. At best, its system level discretionary security model is in constant evolution (See Annex VII). This evolutionary type model, is suspect in traditional terms of assessing security model assurance. [3] Java 2's recent conceptual security model goes well beyond the sandbox employing protection domains digitally signed mobile code and a reference monitor. Protection domains and access control policies are mediated by the byte-code verifier, ClassLoader, SecurityManager and AccessController; a decentralized security kernel of sorts for class instantiation and method object method invocation. Principal model and policy concerns are issues of decentralized control of security functionality, lack of defense in depth and poor assurance verification.

'Trusted' access control necessary for sensitive e-commerce (electronic commerce) and publicly networked health and insurance information applications, are the most challenging and imperative issue for maintaining security in Java. Java is widely accepted and doing well at the client server level, but experts question how well it will scale with strong security beyond the enterprise. The fundamental security issue for Java, although many orders of magnitude tougher as it migrates more fully beyond the enterprise to fully distributed systems, is what components and operations are trusted, and, even more importantly, what mechanisms can adequately protect them?

From an assurance perspective, Sun's inability to formally define or completely verify Java's security model is worrisome. As Java technology is deployed to increasingly more sensitive systems, one must ask the difficult question of greater system trust. Despite extensive beta field-testing for attempted 'kluged verification', it is one thing to determine or state that a system 'should do nothing more', but is quite another perspective to be certain that the system does only what was *designed* to do- - AND nothing more. It is a well-known software development and design principle, that software does not actually wear out in the traditional hardware sense, it deteriorates after subsequent modifications; required departures from the original software requirements specification. [17, p10, 11]

Few can debate the desirableness of a configurable local security policy whose functionality continues to evolve. However, without a definitive original Java security model or policy specification, one must wonder what the specified point of departure is or was and will continue to be for Java's evolving and perhaps murkier security model. In the words of Dean et al. "The absence of a well-defined, formal security policy prevents the verification of a implementation…. For a higher assurance system. Without a formal basis, statements about a system's security cannot be definitive. [11, p11]. Although written in 1996, this point remains very relevant given the complexity of distributed applications, and as the challenge for assurance in Java or other mobile code application increases.

The age-old problematic issue is one of implementing a robust yet verifiable model and policy without excessive penalties in performance, operational functionality, security services and cost. Sun's approach with Java, was to accept *marginal risk*, by fielding an operational 'beta version' product to attain market share which ultimately was tested and verified by the consumer, post production and delivery. Without question, the fielding of Java's mobile code programming language enhanced distributed computing security and accelerated downloading of and remote code for local execution within web applications. Java's security model development methodology and continuing security mechanism migration is best described as 'evolutionary'. Unfortunately, this evolutionary paradigm does not lend itself to the rigor and resources for *formal* security policy specification and model verification necessary for high assurance. As enabling electronic technologies and infrastructures, like public key infrastructure (PKI), evolve to quench the dynamic demands of the Internet (e.g. e-commerce), so to will security functionality requirements and the models necessary to implement them.

"Java applets with bad intentions – exploit scripts – are the equivalent of every System Administrator's nightmare"

Garfinkel & Spafford, 1996

ANNEX I. Principles of Security Mechanisms

Lecture Notes on Trusted Information Systems: Principles for Security Mechanisms copyright (c) 1998 John McDermott, Computer Science Professor, James Madison University

These seven principles apply to the design and comparison of security mechanisms. The first seven, slightly reworded here, are due to Saltzer and Schroeder, as part of the MULTICs project. The final principle is generally accepted. Mechanisms are good to the extent that they follow these principles. A mechanism that fails to follow one of these principles may be flawed or impractical.

1. Least privilege - security mechanisms should give system components the least set of security privileges they need to fulfill their current responsibilities. For example, an electronic mail server should not be designed to require full administrator access privileges for every object in a system, but only those privileges needed to access the incoming mail and place it in the correct mailboxes. This principle is consistently violated in mass-market software, without good reason. Failure to preserve least privilege is the basis for many attacks.

2. Simplicity of mechanism - security mechanisms should be simple. Simplicity is needed for two reasons: 1) to make it possible to implement the mechanism correctly, that is with no flaws; and 2) to make it possible to independently verify that the mechanism works as claimed. Simplicity of both kinds is required: 1) the necessary design patterns, object classes, data structures, or algorithms must not be subtle or obscure, and 2) the sheer number of components, classes, lines of code, etc. should be small. This latter aspect is necessary to avoid introduction of flaws on a statistical basis, i.e. if our best software engineering practice results in one flaw per 10,000 lines of source code, then a 20,000 line security mechanism can be expected to have two flaws in it, even after we have done all of the planned engineering.

3. Complete mediation - it should not be possible to bypass a security mechanism and get to a resource that the mechanism is supposed to protect. In the client-server sense, a malicious client must not be able to request an unauthorized service via an alternate request, or be able to access the resource while the security mechanism is not activated. This principle is also frequently violated in mass-market software, because the implementers are not familiar with it.

4. Open design - the effectiveness of the mechanism should not depend on the ignorance of the attackers. The mechanism should be effective against an attacker who has access to all user documentation, design documents, and source code. This is similar to the principle used for cryptographic systems. Failure to follow this principle is often referred to as "security through obscurity."

5. Least common mechanism - shared instances of security mechanisms should be minimized. For example, a single access control list should not be used to control access to two different files. Each file should have its own ACL.

6. Ease of use - it must not be substantially more difficult for a user to apply a security mechanism correctly than it is to leave resources unprotected. This does not mean that the user should spend no time considering security, but that security should not consume more than a small percentage of the users time and computing resources. In systems with very high levels of security, this percentage should be no more than 10, in most systems the percentage should be less than 5. If this principle is not followed, then users will turn the security off, or not use the system.

7. Tamper resistance - security mechanisms should resist attempts to damage, deactivate, or delete them.

ANNEX II. Essential Security Services of Information Systems (Summers, Ford, Amoroso) [1,4,7]

Authentication – Confirmation of user, principle or entity identity or source of origin. Reasonable confidence that all parties are 'authentic' (e.g. connected servers are whom they purport to be.) (Ford p.13.)

Confidentiality – ensures reasonable data non-disclosure to unauthorized persons. (privacy or secrecy)

Availability, Authorization or Legitimate use – ensures legitimate users or principles are not unduly denied access to and are limited to permitted uses of information and resources.

Integrity- ensures data consistency or accuracy; inhibits unauthorized creation, alteration, or destruction of data. Instills user confidence to prevent in data accuracy.

Auditing – A trusted means to record communications transactions to discern tact of malicious acts, and enable recovery and from breaches and remedy recurrence of same. "Audit information must be selectively kept and protected so that actions affecting security can be traced to the responsible party." (TSEC Orange Book)

Non-repudiation – as applied to messages. Ensures the inability of a sender to deny having sent a message and the inability of a recipient from having received it.

ANNEX IIIa. ‘Untrusted’ Methods and Checks Mediated by Java’s (JFG) Security Manager Class

The below assumed untrusted methods may be mediated by an appropriately configured Java Security Policy/Security Manager for versions up to an including Java (JFG) ver 1.0.2.

Method	**Method Check**	**Checks program authorized to:**
CreateClassLoader()	check CreateClassLoader()	Create a class loader
CreateSecurityManager	check CreateSecurityMgr()	Create Security Manager
Access()	check Access()	Modify a thread or thread group
Exit()	checkExit()	Exit the virtual machine
Execute()	checkExecute()	Execute specified system command
Link()	checkLink()	Link to specified dynamic library
Read()	checkRead()	Read the specified file
Write()	checkWrite()	Write the specified file
Delete()	checkDelete()	Delete the specified file
Rename()	check Rename()	Rename specified file
Connect()	checkConnect()	Connect specified host
Listen()	checkListen()	Listen to a specified port
Accept()	checkAccept()	Accept incoming network connection
LoadLibrary()	checkLoadLibrary()	Load dynamic libraries on client system
ListDirectory()	checkListDirectory()	List contents of a directory
PropertiesAccess()	checkPropertyAccess()	Access specified property
PropertyAccess()	checkPropertiesAccess()	Access all systems properties
DefineProperty()	checkDefineProperty()	Define specified system property(s)
TopLevelWindow()	checkTopLevelWindow()	Create a top level window (untrusted banner)
PackageAccess()	checkPackageAccess()	Access specified package
DefinePackage()	checkPackageDefinition()	Define a class in the specified package.

ANNEX IIIb. Permissions for Java 2’s AccessController Class

AllPermission - The java.security.AllPermission is a permission that implies all other permissions.
AWTPermission - A java.awt.AWTPermission is for AWT permissions.
FilePermission - A java.io.FilePermission represents access to a file or directory.
NetPermission - A java.net.NetPermission is for various network permissions.
PropertyPermission - A java.util.PropertyPermission is for property permissions.
ReflectPermission - A java.lang.reflect.ReflectPermission is for reflective operations. A is a *named permission* and has no actions.
RuntimePermission - A java.lang.RuntimePermission contains a name (also referred to as a "target name") but no actions list.
SecurityPermission - A SecurityPermission contains a name (also referred to as a "target name") but no actions list;
SerializablePermission - A java.io.SerializablePermission is for serializable permissions.
SocketPermission - A java.net.SocketPermission represents access to a network via sockets. A SocketPermission consists of a host specification and a set of "actions" specifying ways to connect to that host.

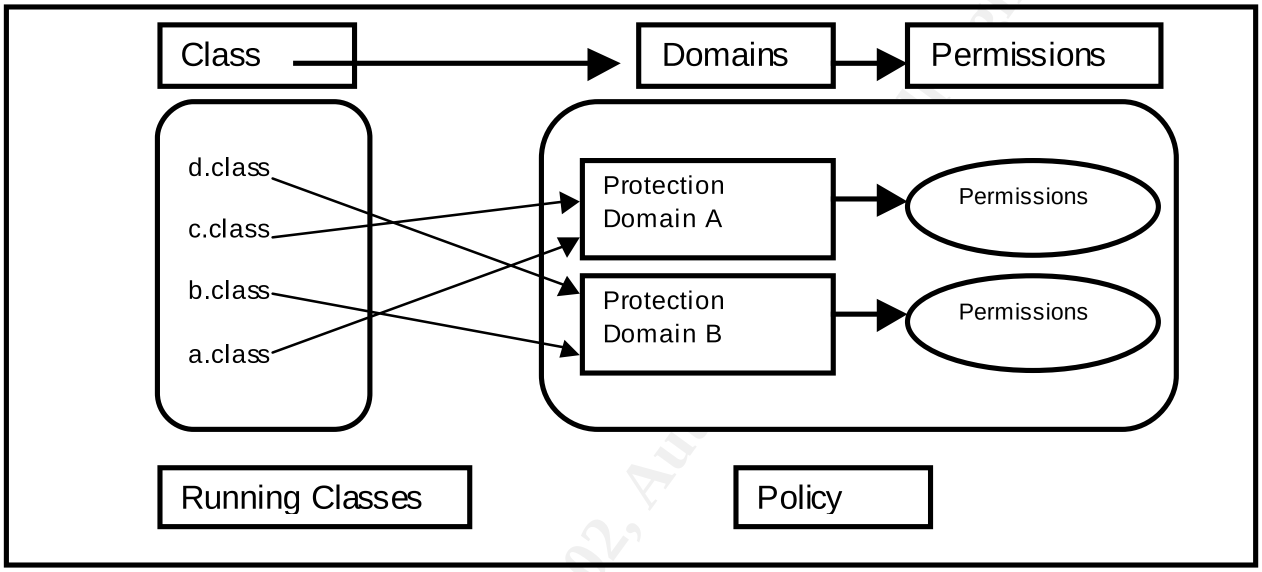
ANNEX IV. Essential Security Services and Mechanisms of Java (2.0)

The following table summarizes essential distributed computing security service challenges as currently addressed by Java 2.0.

Security Service	**Java API/APL**
Authentication	Digital Signatures, Certificates (X.509 v3), MD5
Authorization	Security Manager Java Protected Domains
Confidentiality	SKIP, SSL Technologies Encryption
Containment	Sandbox Class Loader Java Protected Domains Byte Code Verifier
Availability	Fielded System, Customer Proven
Integrity	MD5, SHA
Auditing	Limited Security Manager Enhancements Pending
Trusted Components (Key Exchange)	Diffie-Hellman
Protocols	IIOP, SSL
Encryption (coms)	Triple DES Inherent in Protocols not connection persistent (JAR)

ANNEX V. Java 2.0 Security Mechanism – Protection Domains

Permissions are granted to protection domains, not directly to object or classes. The Java runtime environment maintains mapping from code to protection domains to permissions.



Grouping classes together to map to policy.
Classes map into what Sun calls protection domains which in turn map to permissions. Policy is defined in terms of protection domains. [3, p3-9]

ANNEX VI. Java Virtual Machine (a.) and Java Code Process Flow Components (b.

Figure a. The Java Virtual Machine - JVM

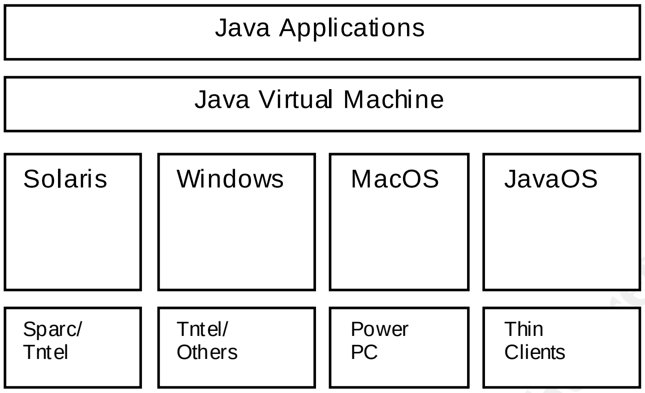
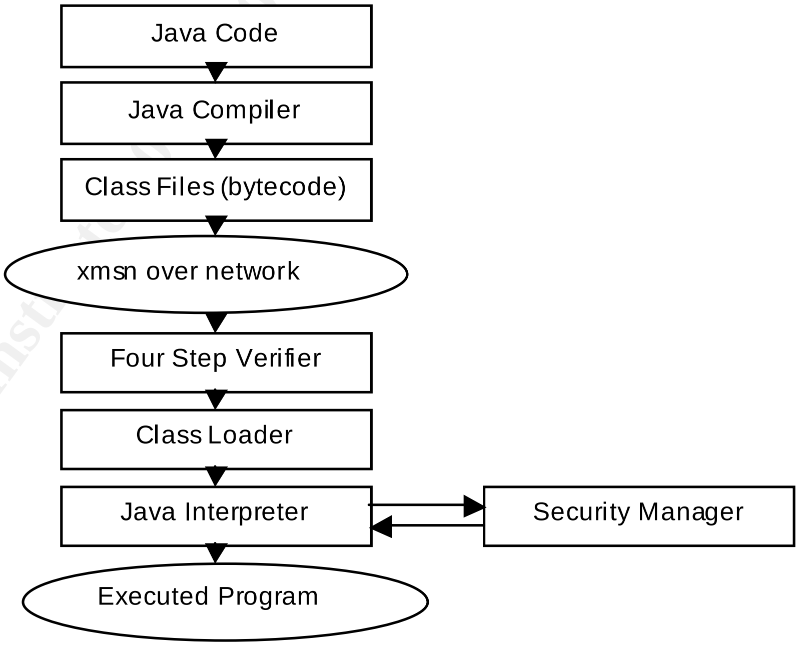


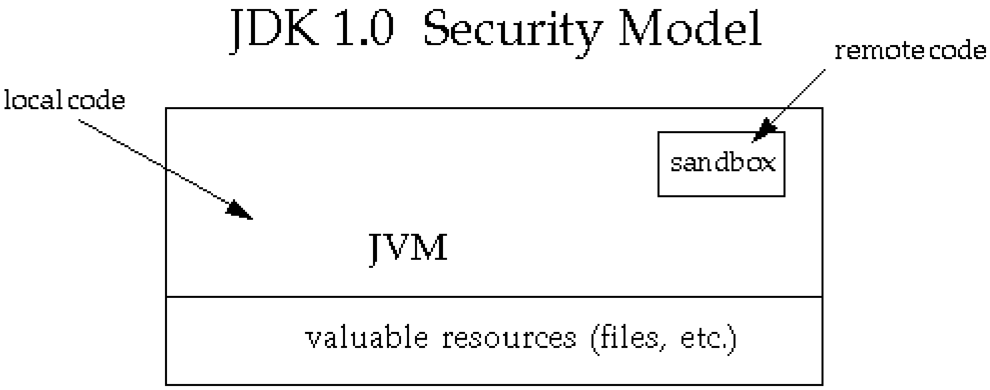
Figure b. Java (JFG) Code Process Flow Components



ANNEX VII The Evolution of the Java Security Model

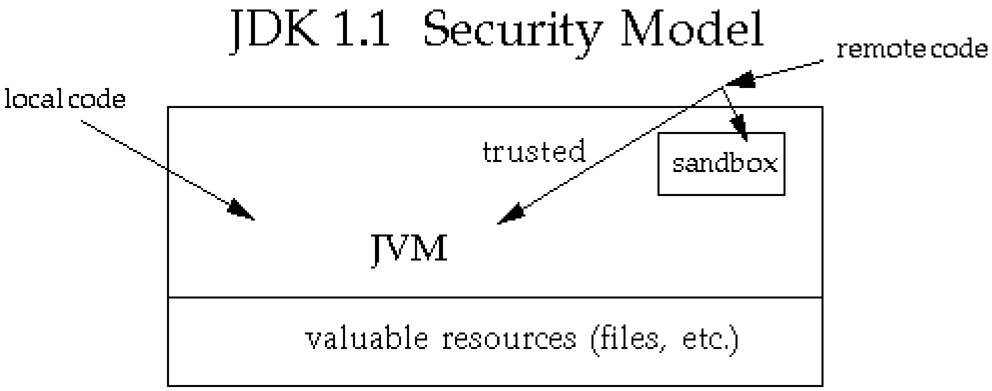
Graphics and narrative:
http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc1.html#18314

The Original Sandbox Model



The original security model provided by the Java platform is known as the sandbox model, which existed in order to provide a very restricted environment in which to run untrusted code obtained from the open network. The essence of the sandbox model is that local code is trusted to have full access to vital system resources (such as the file system) while downloaded remote code (an applet) is not trusted and can access only the limited resources provided inside the sandbox. This sandbox model is illustrated in the figure below. [Java Arch, p.26]

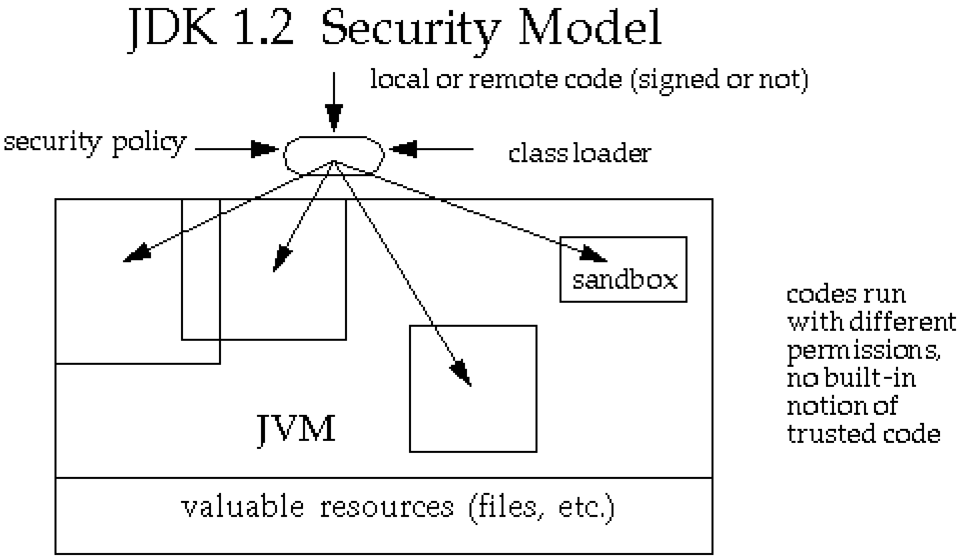
Java Development Kit 1.1



JDK 1.1 introduced the concept of a "signed applet", as illustrated by the figure below. In that release, a correctly digitally signed applet is treated as if it is trusted local code if the signature key is recognized as trusted by the end system that receives the applet. Signed applets, together with their signatures, are delivered in the JAR (Java Archive) format. In JDK 1.1, unsigned applets still run in the sandbox. [26]

ANNEX VII The Evolution of the Java Security Model (Continued)

Evolving the Sandbox Model , JDK 1.2 or Java 2


JDK 1.2 Security Model
local or remote code (signed or not)
security policy
class loader
sandbox
JVM
valuable resources (files, etc.)
codes run with different permissions, no built-in notion of trusted code

The new security architecture in JDK 1.2, illustrated in the figure below, is introduced primarily for the following purposes. Providing Fine-grained access control.

This capability existed in the JDK from the beginning, but to use it, the application writer had to do substantial programming (e.g., by subclassing and customizing the SecurityManager and ClassLoader classes). The HotJava browser 1.0 is such an application, as it allows the browser user to choose from a small number of different security levels. [26]

ANNEX VIII International Standards Organization (ISO) 9001 Standard for Quality Assurance in Software Design and Development activities.

20 ISO 9001 Quality Assurance Requirements

1. Management responsibility
2. Quality system
3. Contract review
4. Design review
5. Document and data control
6. Purchasing
7. Control of customer supplied product
8. Product identification and traceability
9. Process control
10. Inspection and Testing
11. Control of inspection, measuring and test equipment
12. Inspection and test status
13. Control of non-conforming product
14. Corrective and preventive action
15. Handling, storage, packaging, preservation, and delivery
16. Control of quality records
17. Internal audit control
18. Training
19. Servicing
20. Statistical techniques

Bibliography

Published Works:

1. Amoroso, Edward, Fundamentals of Computer Security Technology, Prentice Hall, New Jersey, 1993.

2. Baker, David and Wutka, Mark, Java Security in Depth, Chapter 36 of Special Edition Using Java 1.2, Weber, Joseph, editor, QUE Publishing, 1996.

3. Felten, Ed and McGraw, Gary, Securing Java, Getting Down to Business with Mobile Code, John Wiley and Sons, Inc, 1999.

4. Ford, Warwick, Computer Security: Principles, Standard Protocols and Techniques, Prentice Hall, New Jersey, 1994.

5. Guttman, Barbera, Roback, Edward A., NIST SP 800-12, An Introduction to Computer Security: The NIST Handbook, National Institute Standards and Technology (NIST), 1995.

6. Pfleeger, Charles P., Security in Computing, 2nd Edition, Prentice Hall, New Jersey, 1997.

7. Summers, Rita C., Computer Security: Threats and Safeguards, McGraw Hill, New York, 1997.

8. Department of Defense (DoD) Standard 5200.28 STD, Trusted Computing System Evaluation Criteria (Orange Book), 1985.

9. McDermott, John C., Principle of Mechanisms Lecture Notes, CS621 Trusted Systems, James Madison University, 1999.

Research Articles, White Papers and Pamphlets:

10. Curtis, David, Java, RMI, and CORBA.
http://www-rss.int-evry.fr/~defude/CORBA/wpjava.html

11. Dean, Felten, Edward W., Wallach, Dan S., Java Security: From HotJava to Netscape and Beyond, Dept of Computer Science, Princeton University, IEEE, May 1996. From: http://www.cs.princeton.edu/sip/pub/secure96.php3

12. Erdos, Marlena, Hartman, Bret, Mueller, Marianne, Security Reference Model for JDK 1.0.2 From: http://www.java.sun.com/security

Bibliography (continued)

13. Gong, Li, Java Security: Present and Near Future, an IEEE Micro article, May/June 1997

14. Gong, Li, Java Security Architecture specificatio , Sun Microsystems, Oct 1998. http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc1.html#18314

15. McLean, John, The Specification and Modeling of Computer Security, IEEE, January 1990.

16. Sterbenz, Andreas, An Evaluation of the Java Security Model, Graz University of Australia, IEEE, March 1996.

17. Pressman, Roger .S., Software Engineering: A Practioner's Approach, 4th Edition, McGraw-Hill, United States, 1997.

18. Common Criteria fro Information Technology Security Evaluation, Version 2.0, May 1998.

SANS Related GSEC Practicals from http://www.sans.org/*y2k/gsec.htm*

19. Wells, Marshall, signing a Secure Server Model, February 7, 2001. http://www.sans.org/infosecFAQ/securitybasics/model.htm

20. McQueen, Miles, Java Virtual Mahine Security and the Brown Orifice Attack, August 14, 2000.

21. Rothermel, David, The Zkey Exloit: The Capability of Malicious JavaScript Code, August 28, 2000.

22. Long, Jonnie, Make All Your Internet Traffic Play in the Sandbox, November 22, 2000.