Securing to Vulnerable Web Application (Java Based Application)

[1]**Preyas Y Soni,** [2]**Mr. Ravi K Sheth**

[1]***Student M.Tech(Cyber Security),*** [2]***Assistant Professor(IT)***

[1,2]*Department of Information Technology*

[1,2]*Raksha Shakti University, Ahmedabad, India.*

ABSTRACT

Today most of organizations either have some kind of web application security programs. However most of these programs are not getting expected results to organization, neither long lasting nor able to deliver value in continuous and efficient manner and also unable to enhance minded developers to build/design secure web applications. Creating codes are based on the idea that by injecting realistic vulnerabilities in a web application and attacking them automatically we can assess existing security mechanisms. We used these tools for set of experiments to demonstrate the feasibility and effectiveness of the proposed methodology. By the injection of vulnerabilities and attacks is an effective way to evaluate security mechanisms and tools. Also we can provide security mechanisms for web applications as a defensive way via tools and rules of coding for authorization, authentications to secure application. This paper presents a detailed literature description and analysis on Web Application Security, steps to mitigate the vulnerabilities via providing security tools and cryptography.

Keywords: *Authentication, Authorization, Vulnerabilities, Cryptography, access control, Byte code Verification, File Locations, SSL/TLS, Permissions.*

I. INTRODUCTION:

The Platform of Java Programming was designed with a strong emphasis on security. Java is a type safe and gives automatic garbage collection, enhancing the robustness of application coding[2].

The initial version of the Java platform was created safe environment for running potentially untrusted code, such as Java Applets downloaded from a public network (unsecure). As the platform has grown and wide range its deployment, the Java security architecture has correspondingly evolved into support an expanding set of services [1]. Today Java security architecture includes a large set of application programming interfaces (APIs), tools, and implementations of commonly used security algorithms, mechanisms, and protocols, codes. This provides developer a comprehensive security framework for writing applications, and also provides the user or administrator a set of tools to securely managing applications.

The Java security APIs spread a wide range of areas. Cryptographic and Public Key Infrastructure (PKI) interfaces provide the underlying basis for developing secure applications. Interfaces are performing as authentication and access controls enable applications to guard against unauthorized access to protected resources [4].The APIs are allow for multiple interoperable implementations of algorithms and other security services. Services are implemented in *Providers*, which are plugged into the Java platform via a standard interfaces that makes it easy for applications to obtain security services without having to know anything about their implementations. This allows to developers to focus on how to integrate security into their applications.

The platform of Java includes a number of providers that implement a core set of security services. It also allows for additional custom providers to be installed. This enables developers to extend the platform with new security mechanisms [2].

This paper gives a broad overview of security in the Java platform, from secure language features to the security APIs, tools, and built-in provider services, highlighting key packages and classes where applicable.

II. Java Language Security and Byte code Verification:

Java is designed to be type safe and easy to use. Java provides automatic memory management, garbage collections, and range checking on arrays.

In addition, the Java defines different types of access modifiers that can be assigned to Java classes, objects, members, methods and fields, enabling developers to restrict access to their class executions as appropriate. Java defines four distinct access levels: private, protected, public if unspecified package. The most open access specifier is public access is allowed to anyone user. The most restrictive modifier is private access mode which is not allowed outside the particular class in which the private member is defined [10]. The protected modifier allows to access any subclass or to other classes within the same packages. Package level access only allows access to those classes which are within the same package.

A compiler translates Java programming codes into a machine independent bytecode representation. A bytecode verifier is executing to ensure that only legitimate bytecodes are executed in the Java runtime. It checks that the bytecodes confirm to the Java Language Specification. The verifier also checks for memory management violations, stack overflow or underflow, and illegal data typecasts.

III. Basic Security Architecture:

The Java platform defines set of APIs expanding major security areas, including cryptography, public key infrastructure, and authentication, secure communication, and access control. By APIs programmers to easily integrate security into their application code. They were made by the following principles:

1. **Implementation independence:**
   Applications are independent to implement security themselves. They can request security services from Java platform. Security services are executed in providers, which are plugged into the Java platform by standard interface. An application may depend on multiple independent providers for security functionality [3].

2. **Implementation interoperability:**
   Providers are interoperable between applications. An application unbound to a specific provider and a provider unbound for specific application [3].

3. **Algorithm extensibility:**

The Java platform has a number of built-in providers that implement a basic set of security services. However, some applications may depend on gathering standards not yet implemented, or on proprietary services. Java platform controls installation of custom providers that implement such services [3].

Security Providers

A java.security.Provider class is encapsulating the notion of a security provider in the Java platform. It specifies the provider's name and lists the security services which implements in it. Multiple providers may be configured at the exact same time, and are listed in order of preferences. When a security service is requested, the highest priority provider that implements that service has been selected[4].

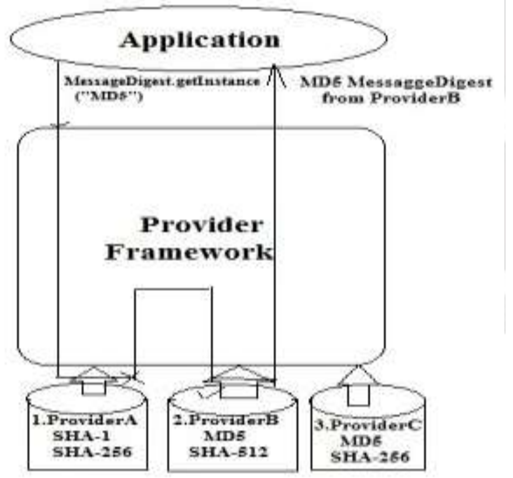
Applications depend on the getInstance method to obtain a security service from an underlying provider. For example, message digest creation is one type of service available from providers. An application request the getInstance method in java.security.MessageDigest class to obtain an implementation of a particular message digest algorithm, such as MD5 [4].

```
MessageDigest md1 = MessageDigest.getInstance("MD5");
```

This program may optionally select an implementation from a specific provider, by indicating the provider's name, as in the following:

```
MessageDigest md2 = MessageDigest.getInstance("MD5", "ProviderC");
```

The providers are ordered by preference from left to right (1-2-3). In Figure 1, an application requests an MD5 algorithm without specifying a provider name. The providers are searched in preference order and the implementation from the first provider supplying that particular algorithm, ProviderB which is returned. In Figure 2, the application requests the MD5 algorithm execution from a specific provider, ProviderC. At that time the implementation from that provider is returned, even though a provider with a higher preference order, ProviderB, also supplies an MD5 implementation [4].


Application
MessageDigest.getInstance ("MD5")
MD5 MessaggeDigest from ProviderB
Provider Framework
1.ProviderA SHA-1 SHA-256
2.ProviderB MD5 SHA-512
3.ProviderC MD5 SHA-256

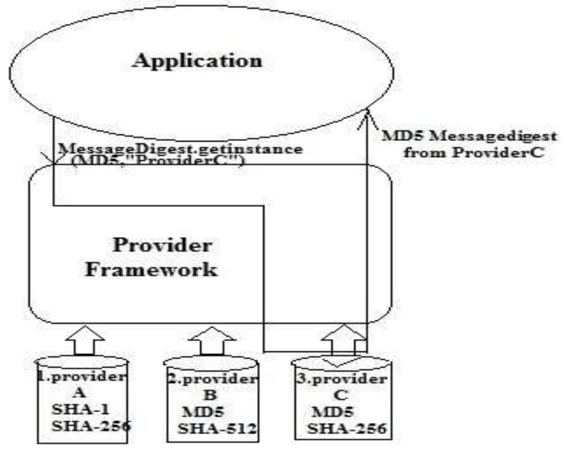

Application
MessageDigest.getinstance (MD5,"ProviderC")
MD5 Messagedigest from ProviderC
Provider Framework
1.provider A SHA-1 SHA-256
2.provider B MD5 SHA-512
3.provider C MD5 SHA-256

Figure1-Provider Searching **Figure2**- Specific Provider Requested

The Java executions includes a number of pre-configured default providers that execute a basic set of security services that can be used by applications from sun microsystems. Other vendors implementations of the Java platform may include different sets of providers that encapsulate vendor specific sets of security services. When this paper mentions built-in default providers, it is referencing those available in Sun's implementation [11].

The sections below on the various security areas (cryptography, authentication, etc.) each include descriptions of the relevant services supplied by the default providers.

A. Cryptography

Cryptography of Java architecture is a framework for accessing and developing cryptographic functionality for the Java platform. It includes APIs for a verity of cryptographic services, including:

- Message digest algorithms
- Digital signature algorithms
- Symmetric stream encryption
- Password-based encryption (PBE)
- Elliptic Curve Cryptography (ECC)
- Key agreement algorithms
- (Pseudo-)random number generators

For historical (export control) reasons, the cryptography APIs are organized into two distinct packages. Package java.security has classes that are *not* subject to export controls (like Signature and MessageDigest). Package javax.crypto contains classes that are subject to export controls (like Cipher and KeyAgreement) [5].

The cryptographic interfaces are provider based and allowing for multiple and interoperable cryptography implementations. Some providers may implement cryptographic operations in software, others may perform the operations on a hardware token (for example, on a smartcard device or on a hardware cryptographic accelerator)[8].

The Java platform includes built-in providers for the most commonly used like cryptographic algorithms, including the RSA and DSA signature algorithms, DES, AES, and ARCFOUR encryption algorithms, the MD5 and SHA-1 message digest algorithms. These default providers implement cryptographic algorithms in Java code [8].

B. Public Key Infrastructure

Public Key Infrastructure (PKI) is used for a framework which enables data in secure manner by public key cryptography. It allows identities (like people, organizations, industries etc.) to be bound to digital certificates and provides a means of verifying the certificate authentication. PKI includes keys, certificates, public key encryption and trusted Certification Authorities who generate and digitally sign certificates [4]. The Java platform includes API and provider supporting to X.509 digital certificates and certificate revocation lists (CRLs)[3]. The classes of PKI are located in java.security and java.security.cert packages.

I. Key and Certificate Storage

Java platform provides for long term exact storage of cryptographic keys and certificates. Class java.security.KeyStore represents Key store, a secure repository of cryptographic keys and trusted certificates (to be used during certification path validation), and the class java.security.cert.CertStore represents a *certificate storage*, a publically vast repository of unrelated and untrusted certificates [2].

KeyStore and CertStore executions are distinguished by types. The Java includes the standard *PKCS11* and *PKCS12* keys store types (there executions are protest with the corresponding PKCS specifications from RSA Security), as well as a proprietary file based key store type called *JKS* (Java Key Store) [3]. "*cacerts"* is a special built in JKS key store in Java which includes a various certificates for well-known, trusted CAs. The keytool documentation lists the certificates included in *cacerts.*

The Java platform also support an LDAP certificate store type as well as an in memory Collection certificate store type (accessing certificates managed in object java.util.Collection)[12].

II. PKI Tools

There are two tools for working with keys, certificates and key stores:

keytool can create and manage key stores. It can

- Create public or private key pairs
- import and export X.509 v1, v2, and v3 certificates stored as files
- Create self signed certificates
- Issue certificate (PKCS#10) requests to be sent to CAs
- Import certificate replies (gotten from CAs sent certificate requests)
- Designate public key certificates as trusted

The **jarsigner** tool can used for sign JAR files, or to verify signatures on signed JAR files. The Java Archive (JAR) file format can invokes a bundling of multiple files into a single file. JAR file includes the class files and resources associated with applets and applications. When you want to digitally sign code, we first use keytool to generate/import appropriate keys and certificates into our key store, then use the **jar** tool to place the code in a JAR file. The jarsigner tool used a key store to find any keys and certificates needed to sign in JAR file[14].

C. Authentication

Authentication means process of finding the identity of a user. In Java runtime environment, it is the process of identifying the user of an executing Java program.

The Java platform includes APIs that enable an application to perform user authentication from pluggable login modules. Application call in the class LoginContext (in package javax.security.auth.login) goes to a configuration. Configuration defines which login module (an execution of interface javax.security.auth.spi.LoginModule) is used to perform the actual authentication.

When applications connected to the standard LoginContext API, they can remain independent from the underlying plug-in modules. New and updated modules can be plugged in for an application without having to modify application itself.

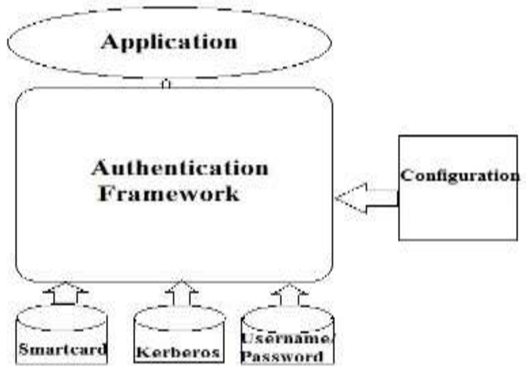


Figure3 – Authentication login modules plugging into the authentication framework

It is important that although login modules are pluggable components that can be configured into the Java platform, they are not plugged in via security providers. In above diagram, login modules are administered by their own unique configuration.

D. Secure Communication

Data travels across a network can be accessed by someone who is not the intended recipient. When the data like private data such as passwords and credit card numbers steps must be taken to make the data intelligible to authorized users. It is also important that you are sending the data to the appropriate user and that the data has been unmodified either intentionally or unintentionally during transport [3]. Cryptography forms basis required for secure communication. The Java platform also provides API support and provider executions for a number of standard secure communication protocols [2].

I. SSL/TLS

The Java platform has APIs and an implementation of the SSL and TLS protocols that performs functionality for data encryption, message integrity, server authentication and optional client authentication. Applications can use SSL/TLS for the secure passage of data between two users over any application protocol such as HTTP of TCP/IP.

Class javax.net.ssl.SSLSocket represents a network socket that includes SSL/TLS support normal stream socket (java.net.Socket). Some applications might want to use alternate data transport abstractions (New-I/O), Class javax.net.ssl.SSLEngine is available to produce and consume SSL/TLS packets.

The Java platform also contains APIs that support the concept of pluggable (provider based) key managers and trust managers. A *key manager* is encompass by class javax.net.ssl.KeyManager and manages the keys used to perform authentication. A *trust manager* is involved by the TrustManager class and takes decisions about who to trust based on certificates in the key store it manages [1].

II. SASL

Simple Authentication and Security Layer (SASL) is an Internet standard describes a protocol for authentication and optional establishment of a security layer between client and server applications. SASL defines how authenticated data is to be exchanged but doesn't specify the contents of that data. It is a framework in which specific authentication mechanisms specify the contents and semantics of the authentication data.

SASL API means classes and interfaces for applications that use SASL mechanisms. It defines to be mechanism neutral an application and it uses API need not be hardwired into using any particular SASL mechanism. Client and server applications are supported by API. Class javax.security.sasl.Sasl is used to create SaslClient and SaslServer objects [4].

SASL mechanism executions are supplied in provider packages. Each provider can support one or more SASL mechanisms and is registered and invoked via the standard provider architecture.

Java supports built in provider executes following SASL mechanisms like:

- DIGEST-MD5, CRAM-MD5, EXTERNAL and PLAIN client mechanisms
- DIGEST-MD5,CRAM-MD5, server mechanisms

E. Access Control

Architecture of Access Control in the Java protects access to sensitive resources (like local files) or sensitive application code (like methods in a class). Security manager mediates all access control decisions, represented by the java.lang.SecurityManager class. Java applets and Java Web Start applications are automatically run when Security Manager installed. Local applications executed which are by default *not* run with a SecurityManager installed. In order to run local applications with a Security Manager, either the application itself must programmatically set one via the setSecurityManager method (in the java.lang.System class), or java must be start with a java.security.manager argument in the command line [10].

I. Permissions

When class loader is stored Java code a into the Java runtime, the class loader automatically associates the following information with that code:

- Where the code was loaded from
- Who signed the code (if anyone)
- Default permissions granted to the code

Data is associated with the code regardless of whether the code is downloaded over an untrusted network (an applet) or got from the file system (a local application). The location from which the code was loaded described by a URL code signer and default permissions are represented by 'java.security.Permission' objects [8].

The default permissions permitted to downloaded code include the ability to make network connections back to the host from which it originated. The default permissions automatically granted to code loaded from the local file system include ability to read files from the directories.

User's identity executing the code is not available at class loading time. It must be application code to authenticate the end user. When the user has been authenticated, Application can dynamically associate that user with executing code by invoking the doAs method in the class javax.security.auth.Subject [5].

II. Access Control Enforcement

Java runtime keeps track of the sequence of Java calls that are made as a program executes. When protected resource access is requested, the entire call stack by default is executed to determine whether the requested access is granted.

Security sensitive code in Java and in applications protects access to resources via code like the following:

```
  SecurityManager sm1=System.getSecurityManager();
   if (sm1 != null) { sm1.checkPermission(perm); }
```

Here perm is the Permission object that corresponds to the requested access. if an attempt is made to read the file */tmp/abc*, permission may be constructed as follows:

```
   Permission perm = new java.io.FilePermission("/tmp/abc", "read");
```

The default execution of SecurityManager entrusts its decision to the java.security. AccessController executions. Policy finds whether the requested access is granted by the permissions configured by the administrator. If access is not granted, AccessController throws java.lang.SecurityException [8].

In Figure 4 Here there are initially two elements call stack ClassA, ClassB. ClassA interrupts a method in ClassB, which then attempts to execute the file */tmp/abc* by creating an instance of java.io.FileInputStream. FileInputStream constructor creates a FilePermission, perm and then passes perm to the SecurityManager's checkPermission method. Only the permissions to ClassA and ClassB need to be checked because all system code including FileInputStream, SecurityManager and AccessController automatically receives all permissions [3]. Here ClassA and ClassB have different code characteristics they come from different locations and have different signers. The AccessController only permits access to the requested file if the Policy indicates that both classes have been granted the required FilePermission [2].

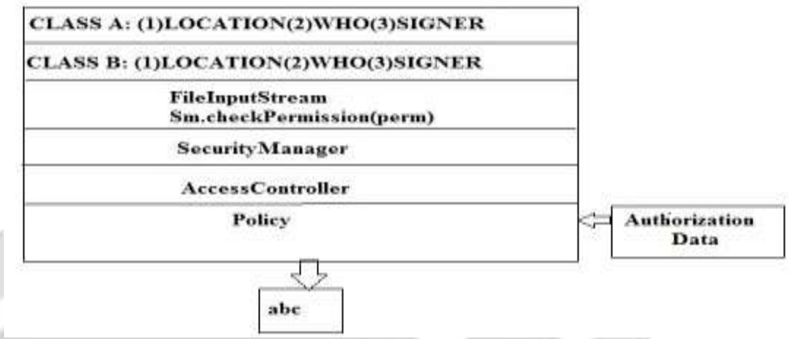


Figure4- Access Control Enforcement

E. Policy

As mentioned earlier, a limited set of default permissions are granted by class loaders. Administrators can flexibly manage additional code permissions via a security policy. Java encapsulates the notion of a security policy in the class 'java.security.Policy'. Only one Policy object can installed into the Java runtime at any given time. The basic responsibility of the Policy object is to find whether access to a protected resource is granted to code (characterized by where it was loaded from, who signed it, and who is executing it)[6].

The Java includes a default Policy implementation that reads its authorization data from one or more ASCII files configured in the security property files. These policy files contain the exact sets of permissions granted to code loaded from exact locations, signed by particular entities, and executing as particular users. The policy entries in each file must conform to a documented proprietary syntax form, and may be composed via a simple text editor or the graphical **policytool** utility [4].

IV. LITRETURE REVIEW:

Code source authenticity is provided through the use of JAR files and special permissions. This functionality is implemented in the class loader as sealed JAR files, signed JAR files, and protected packages. Java 2 provides a special mechanism called "sealing" within JAR files to ensure that only one JAR file supplies classes for a specific package. JAR files are "sealed" by adding a text attribute to the manifest file. This attribute can be applied to all packages in the JAR file, or only to specific packages. When a package is "sealed", all classes that claim membership to that package must come from a single JAR file [5]. The class loader enforces special restrictions on classes and packages when these come from signed JAR files. If a class comes from a JAR file that is signed, and the digital signature across the class file is verified, then the class file is given a new security attribute in its CodeBase. This security attribute is the certificate of the signer of the JAR file [4].

Since the certificate is added to the CodeSource and the CodeSource is used to assign permissions to classes through the ProtectionDomain the presence of a certificate can be used to assign special permissions to a class or set of classes. When the first class for a particular package is loaded, the class loader associates any certificates assigned to that class to that package as well. Then, when additional classes for that package are loaded, they must also have the same certificates associated with them. This enforces equal integrity protections across all classes within a package and prevents some types of malicious code insertion attacks. The class loader provides special protection to packages defined in the java.security file under the defineClassInPackage and accessClassInPackage properties. These properties have list of package names which require a special permission in order for the package to be used. For all packages listed in the accessClassInPackage property, class loaders must have a special permission to access classes in that package. In the default installation of the java.security file, the "sun" package is listed in the property list for accessClassInPackage [8].

This prevents classes loaded by the system class loader from using classes considered private by Sun and residing in a package beginning with the name "sun." The defineClassInPackage property is intended to work in a similar fashion for the permission to define classes. This mechanism of package protection was deemed insufficient to protect the "java.*" packages, presumably because Sun Microsystems wanted to ensure the base packages of Java could not be supplanted by merely changing a text-based configuration file. Additions to the "java.*" packages are protected using explicit code within the base class loader [6].

V. Conclusion:

In this paper we showed how a general class of security errors in Java applications can be formulated as instances of the general tainted object propagation problem, which involves finding all sink objects derivable from source objects via a set of given derivation rules. We developed a precise and scalable analysis for this problem based on a precise context sensitive pointer alias analysis and showed extensions to the handling of strings and containers to improve the precision. Our approach tracks all vulnerabilities matching the specification within the statically analyzed code. Here errors may be missed if the user-provided specification is incomplete.

Support for flexible security policies, encryption and other more advanced security features are also being added. Any organization which is considering adding Java applications or Java enabled software to its network should carefully consider how Java will affect their current security policies. While no set of security policies can ever eliminate all risk from a networked environment, understanding how Java's security model works and what sorts of attacks might be committed against it, keeping current with new developments by both attackers and other security officers, and evaluating Java in light of the organization's overall security policy can reduce risks to an acceptable level.

VI. REFERENCES:

[1] The ACM RISKS Forum. Moderated by P.G. Neumann. Online at (https://www.risks.org).

[2] L. Gong, "New Security Architectural Directions for Java (Extended Abstract)". In Proceedings of IEEE COMPCON, San Jose, California, February, 1997, pp.97--102.

[3] L. Gong. Java Security: Present and Near Future. IEEE Micro, 17(3):14--19, May/June 1997.

[4] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers., "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2". In Proceedings of the USENIX Symposium on

Internet Technologies and Systems, Monterey, California, December, 1997, pp.103--112. [5] L. Gong. Inside Java 2 Platform Security, First Edition, Addison Wesley, 1999.

[6] L. Gong, G. Ellison, and M. Dageforde. Inside Java 2 Platform Security, Second Edition, Addison Wesley, 2003.

[7] L. Gong, Inside Java 2 Platform Security: Architecture, API Design, and Implementation, Addison-Wesley, 1999.

[8] L. Gong, Java 2 Platform Security Architecture, Oct 1998, Copyright 1997-1999, Sun Microsystems, (http://java.sun.com/j2se/1.3/docs/guide/security/spec/securit y-specTOC.fm.html)

[9] M. Pistoia, et. al, Java 2 Network Security, 2nd Edition, IBM, 1999.

[10] G. McCluskey, "Adding Security Features to Applications". JDC Tech Tips, July 14, 1999, Copyright 1995-2000, Sun Microsystems,(http://developer.java.sun.com/developer/JDCTechTips/).

[11] Sun Microsystems, "Controlling Package Access with Sealed JAR Files", JDC Tech Tips, Jan 30, 2001, Copyright 2001, Sun Microsystems,(http://developer.java.sun.com/developer/JDCTechTips/).

[12] Sun Microsystems, "Controlling Package Access with Security Permissions". JDC Tech Tips, Jan 30, 2001, Copyright 2001, Sun Microsystems,(http://developer.java.sun.com/developer/JDCTechTips/).

[13] Sun Microsystems, The Java Virtual Machine Specification. http://java.sun.com/docs/books/vmspec. [8] Sun Microsystems, "Relating Class Loaders to the CLASSPATH", JDC Tech Tips, Oct 31, 2000, Copyright 2000, Sun Microsystems. (http://developer.java.sun.com/developer/JDCTechTips/).

[14] MageLang Institute, "Fundamentals of Java Security", Copyright, 1998, MageLang Institute, (http://developer.java.sun.com/developer/onlineTraining/Security/).

[15] R. Srinivas. "Java Security Evolution and Concepts, Part2", Java World, July 2000, Copyright, 2000 Itworld.com Inc.