White Paper

Building Secure Software with Java

By Dr. Kelvin Nilsen,
CTO, Atego

May 26, 2009

Building Secure Software with Java

This white paper discusses the applicability and desirability of Java as a programming language for use in secure systems. Java is much more secure than C and C++ because the byte-code verifier prohibits unsafe type coercions and address arithmetic, garbage collection prevents dangling pointers, run-time checks prohibit null pointer dereferencing and out-of-bound array subscripts, and the security manager restricts application behavior according to the credentials of an application provider. Java completely eliminates many of the most common mechanisms exploited in common virus and Trojan Horse attacks. The ease with which existing Java software capabilities can be ported and repurposed makes Java an ideal language for creating customized secure functionality by assembling and integrating independently developed off-the-shelf Java software capabilities.

Introduction

Secure software has relevance in a variety of application domains, including banking, electronic commerce, homeland defense, and military systems. The primary objectives of securing software are to:

1. Hide confidential information from individuals who are not supposed to see this information.
2. Assure that transmitted information is received completely and accurately by intended recipients.
3. Authenticate senders of information, so that recipients are confident that a received message originated from the individual who is identified as the sender of the information.
4. Assure that any automation activities performed as a result of "secure communication" are based on accurate and trustworthy information.

With rising threats of terrorism and other forms of asymmetric warfare, an emphasis on the security of software systems has increased dramatically. The common desire is to take existing software capabilities and strengthen the security in order to deliver those capabilities in environments that represent significant risk if the security is compromised. Architects responsible for designing such systems must carefully weigh the benefits, costs, and risks associated with various design tradeoffs.

There exists, for example, considerable tension between:

1. The end user's desire for ease of use and unfettered information access on the one hand, and the security enforcer's desire to monitor and restrict information flow in order to minimize the risk of intrusions;
2. The software developer's desire to leverage commercial off-the-shelf (COTS) software components and middleware, and the security enforcer's desire to subject all such software to the scrutiny of a security audit; and
3. The purchasing agent's desire to schedule incremental payments against incremental demonstration of functionality, and the software development team's need to invest heavily in security provisions while the software is being created, long before the complete security audit can be completed.

Java offers significant benefits in comparison with C and C++ for implementation of secure software systems.

- First, the Java language has inherently fewer risks than C and C++ because it prohibits unsafe type coercions, address arithmetic, dangling pointers, null-pointer dereferencing, array subscript out-of-bounds errors, and stack overflow.
- Second, when existing software capabilities need to be ported and integrated within a secure computing environment, Java software integrators are five to ten times more productive than developers using the C or C++ languages.
- Third, the use of static analysis tools to assist with the data-flow analysis required for security audits is much easier with the Java language than with C and C++ because there are far fewer opportunities for aliasing in Java. Aliases exist whenever multiple variables refer to the same memory location.

This paper provides an overview of critical considerations relevant to development and certification of secure software systems in the Java language.

Security Requirements for Complex Software Systems

Modern software systems deal with thousands of distinct constituents, each representing a distinct privacy realm. Often, the account information of one constituent must be hidden from all other constituents. Note that the security realms associated with individual users are not necessarily hiearchical. Many security realms may exist at the same level, though they must be maintained as independent from each other. Certain administrators have access to more information than a typical "user". However, even administrative privileges are normally restricted, and some administrators have more access privileges than others.

The principles of building secure software are similar whether the list of constituents represents customers of Amazon.com, different branches of the U. S. military, or different allies who have combined forces in a joint NATO exercise.

In general, security policies must prevent the flow of private information belonging to one constituent to other constituents. They must also prevent information from masquerading as if it had a different origin or a different credibility than it rightly deserves. For example, the top-secret realm of an unmanned aircraft's mission planning software may take responsibility for targeting and firing certain on-board weapons. The same mission planning software may also include unclassified information, such as publicly available regional weather reports or even road maps. Since the unclassified information may not have been treated with the same level of care that is associated with top-secret information, it is important to distinguish within the mission planning software that the unclassified information is less trustworthy than the top-secret information. In particular, an enemy might have found it far easier to corrupt the weather report than to have forged a special directive from the commander in chief.

Building Secure Systems by Composition of Components

Modern software development consists largely of assembling existing off-the-shelf software components into complex systems offering the combined capabilities of each of the integrated software components. A difficulty with building secure software systems as a composition of independently developed off-the-shelf components is that the security attributes of most existing off-the-shelf software components are not readily available. Even if they were available, it would be necessary to independently ascertain that the security attributes are trustworthy.

Ideally, the same modular composition practices that make it possible to build large software systems by assembling independently developed components would allow the security attributes for a large composition of software components to be automatically derived by combining the security attributes associated with each of the assembled components.

Consider that a *basic software component's* security attributes consist of the following information:

1. A list of all paths by which information may flow into the component.
2. A list of all paths by which information may flow out of the component.
3. A characterization of internal state which may preserve data derived from information that previously flowed into the component.

A MILS (Multiple Independent Levels of Security) software component has the following additional security attributes:

4. For each input to the component, identify the security level associated with this input.
5. For each output from the component, identify the security level associated with this output.
6. For each internal state value, identify the security level associated with the state variable.

Note that abstraction may be necessary to represent the security levels associated with certain data flows. For example, a component that represents a customer data base may have an input channel dedicated to placement of a new order. The information supplied to that input channel should be treated as private to the particular customer. However, the same input channel is used to place orders of other customers. In this particular case, the security realm associated with the inputs to this channel are identified by at least one of the arguments (e.g. the “customer id”) passed into the channel.

Given an existing secure system, a new secure software can be integrated into this secure system provided that one of the following two conditions is satisfied:

1. If the new component is a “basic subsystem” (i.e. inputs and outputs make no distinction of security level), the new component may be integrated if all of its inputs and outputs are connected to the same security realm.
2. If the new component is a “MILS subsystem”, all of its inputs and outputs at each security level are connected to the same security realm.

Analyzing Software Components to Trace Data Flows

One difficulty with building a secure software system from the integration of off-the-shelf software components is that it may be difficult to determine all of the inputs, outputs, and state information associated with each integrated component. Reuse of off-the-shelf components is easiest if all of the inputs and outputs are connected to the same security level. In this configuration, it is not necessary to prove that there are no data flows within the component from certain inputs to certain outputs. However, even this determination requires a full accounting of all inputs and outputs. Either through malicious intent or accident, any given software component may present more data flows than are identified in the description of the component’s security model.

When identifying the potential input channels associated with a software component implemented in the Java language, consider each of the following possibilities:

1. Every non-private method and all arguments passed to the non-private method.
2. Every non-private static or instance field.
3. Every field of an external component whose value is fetched by this module.
4. Every result value returned from an “external” method that is invoked from within this module.
5. If this class is non-final, extensions of this class may introduce new methods or fields which introduce new input channels. Non-final methods might be overridden in an extension of this class, making methods more visible in subclasses than in the declared superclass.
6. Any invocations of I/O services to read data from an I/O device (i.e. a network socket or file or keyboard).
7. In Java, reflection APIs allow methods to be invoked and fields to be accessed using syntaxes that may not be understood by a static analysis tool that is searching to identify all data flows. It is advisable to seriously restrict the use of reflection APIs in a secure software system.
8. If the Java code includes native methods implemented in C or other legacy languages, it is very difficult to assure the validity of the data-flow analysis. C code integrated into a Java virtual machine does not necessarily honor the Java security model. The C code may fetch and modify private fields, may modify final (constant) fields, and may invoke methods (functions) that would normally be hidden. Additionally, the C security model is much weaker than Java’s. C code may exhibit, for example, unsafe type coercions between integers and different pointer types, dangling pointers to deleted objects, stack overflows, undetected array subscripting errors, and derefencing of pointers to non-existent objects. Each of these behaviors represents an opportunity for covert data flow.
9. The code itself includes manifest constants and encoded algorithms which also must be treated as “inputs” to the behavior of the computer.

To identify the potential output channels associated with a Java software component, consider the following:

1. Every result returned from a non-private method.
2. Every non-local method invoked from this module and any arguments passed to that method.
3. Every non-private static or instance field.
4. Every field of an external component whose value is overwritten by this module.
5. If this class is non-final, extensions of this class may introduce new methods or

fields which introduce new output channels. Non-final methods might be overridden in an extension of this class, making methods more visible in subclasses than in the declared superclass.

6. Any invocations of I/O services to write data to an I/O device (i.e. a network socket or file or GUI display).
7. In Java, reflection APIs allow methods to be invoked and fields to be accessed using syntaxes that may not be understood by a static analysis tool that is searching to identify all data flows. It is advisable to seriously restrict the use of reflection APIs in a secure software system.
8. As with the analysis of inputs, native code written in C or C++ makes it much more difficult to confidently identify all output channels associated with a particular software module.

The above lists characterize all of the potential inputs and outputs for the subsystem component. In any given context, a thorough analysis of how this component interacts with other components may reveal that many of the potential channels are not exercised. This may be enough to prove that the subsystem is secure when integrated within a particular context. However, because modern software systems are continually evolving, it is best to treat each potential input or output as if information were flowing through that channel. In general, restricting the number of potential inputs and outputs associated with each reusable software component will improve the ease with which this software component can be integrated within a larger secure system and will in general reduce the costs of maintenance associated with this component.

Besides identifying all inputs and outputs, a security audit must identify all paths by which information flows from inputs to outputs. This is especially important for MILS subsystems that take responsibility for assuring that certain inputs are not connected to certain outputs.

ANALYSIS OF INFORMATION FLOWS WITHIN A SECURE MODULE

Data-flow analysis is a well understood analysis technique that is commonly used to enable certain programming language compiler optimizations. The same analysis technique provides valuable information to assist with security audits of software modules.

Consider the following representative Java code:

```
[1]   class MultiLevels {
[2]         int field1, field2;
[3]
```

```
[4]    void topSecretMethod(int a) {
[5]        field1 = a;
[6]    }
[7]
[8]    void unclassifiedMethod(int b) {
[9]        field2 = b;
[10]   }
[11]
[12]   void riskyUnclassifiedMethod(int c) {
[13]       field1 = c;
[14]   }
[15]
[16]   int getTopSecretInfo() {
[17]       return field1;
[18]   }
[19] }
```

Standard data-flow analyses reveal that field1 is modified by both topSecretMethod() (at line 5) and riskyUnclassifiedMethod() (at line 13). Assuming that these two methods represent top-secret and unclassified inputs respectively, the data-flow analysis concludes that field1 must be treated as potentially holding either top-secret or unclassified data. In most contexts, the analysis will conservatively assume that field1 holds top-secret information.

Note, however, that the value of field1 is provided as the result returned from getTopSecretInfo() (at line 17). Assume the result returned from this method represents a secure top-secret output channel. Depending on the security policies that govern this software system, allowing indiscriminate data flow from an unclassified realm to the top-secret realm may violate the intended security isolation. A problem arises if, for example, this allows arbitrary unclassified data sources to introduce unreliable or even intentionally misleading information into the secure realm of top-secret decision making. Thus, a data-flow analysis of the above program would likely identify the flows of information through field1 as a security flaw.

Standard data-flow analysis traces the flow of bits from one variable to another. However, it does not recognize indirect flows of information as a result of conditional control.

Consider, for example, the following method:

```
[20]   void anotherRiskyUnclassifiedMethod(int c) {
[21]         if (field2 > 10)
[22]             field1++;
[23]    }
[24] }
```

In this case, normal data-flow analysis does not detect any flow of information from field2 to field1. However, the value of field1 may be affected by the number of times this method is invoked, and by the value held in field2 each time this method is invoked. Thus, a security policy that prohibits flow of information from less secure domains to more secure domains must monitor indirect data flows such as the above in addition to the standard data-flow analysis.

Best Practices in Secure Java Development

Java is inherently much more secure than C and C++. However, simply choosing to use the Java language will not guarantee that a large system of software is secure. Developing secure software involves rigor and discipline that begins long before code development, starting with requirements capture, which flows directly into architecture and design.

Secure software development with the Java language typically adheres to one or more of the following practices:

1. Significant restrictions on the use of Java's reflection services.
2. Significant restrictions on dynamic loading of classes.
3. Significant restrictions on the use of native code within a Java application.
4. Selective restrictions on standard and third-party libraries to reduce the costs and effort required to complete a full security audit.
5. Careful scrutiny of the standard libraries and third-party middleware that are allowed in the application to understand the security characteristics of that software.
6. Virtual and physical isolation between software modules so as to reduce and/or filter the information that flows between modules.
7. Enforcement of special guidelines for all newly developed code associated with the project, including:

   a. the use of declarations to restrict the visibility of particular methods and fields;
   b. the use of annotations or other meta-data to describe the security levels associated with each input and output and of all state information affiliated with each software module;
   c. the use of static analysis tools and/or managed code reviews to assure that there are no covert channels that allow information to cross security boundaries without an appropriate "guard" to filter the information.

8. Use of Multiple Independent Levels of Security (MILS) partitioned kernels to enforce the isolation of independent software modules. This is discussed further in the next section.

Partnering with a virtual machine vendor who specializes in supporting mission-critical secure deployments can simplify these activities. Such a vendor will be familiar with the use of Java with secure MILS kernels and may already have a port available to support your MILS kernel of choice. Also, this vendor may be able to supply a secured configuration of their virtual machine to reduce the effort and costs associated with certification of security. Such a configuration may lack support for dynamic class loading and certain reflection capabilities, and may seriously restrict the use of native code, networking, and file subsystem libraries. Atego has many years of commercial experience supporting secure deployments of mission-critical Java software systems.

Using Java with MILS Partitioning

In the days preceding computerized information processing, information security was protected by physical boundaries. As computers assumed increasingly important roles in the processing of secure information, initial approaches to maintaining security have relied on physical isolation of computing hardware to assure that information does not flow between different security enclaves. Even today, certain defense applications require that users maintain multiple independent computers in their work areas, with each computer connected to a different secure network. One keyboard and monitor might represent a top-secret realm. Another might represent a classified realm. A third might represent unclassified information. As long as there is no information flow between these distinct security realms, the risk that higher security levels would be compromised by the behavior of less-secure software systems is very low.

In recent years, as computers have become more capable while at the same time requiring less space and less power, computers have migrated out of headquarters and into the trenches. At the same time, computers have assumed increasingly critical roles in gathering information and automating responses. Today, many unmanned sea, land,

and air vehicles operate autonomously, entirely under the control of sophisticated software programs.

With increases in computing capacity, more embedding of computers in war-fighting systems, and more automation of war-fighting activities, the use of physical isolation to prevent information flow between security enclaves is less practical. This is part of the motivation for introduction of MILS partitioned operating systems.

A MILS operating system uses trusted software to establish virtual partitions that behave similar to the physical partitioning of previous generations of systems. A typical configuration is illustrated in Figure 1. In this configuration, the trusted operating system maintains three distinct security realms, USAF top secret, NATO classified, and unclassified. Components within each realm are only allowed to communicate with other components residing within the same realm. The MILS operating system kernel provides virtual partitioning that achieves the same isolation of information that in previous systems was provided by physical isolation.

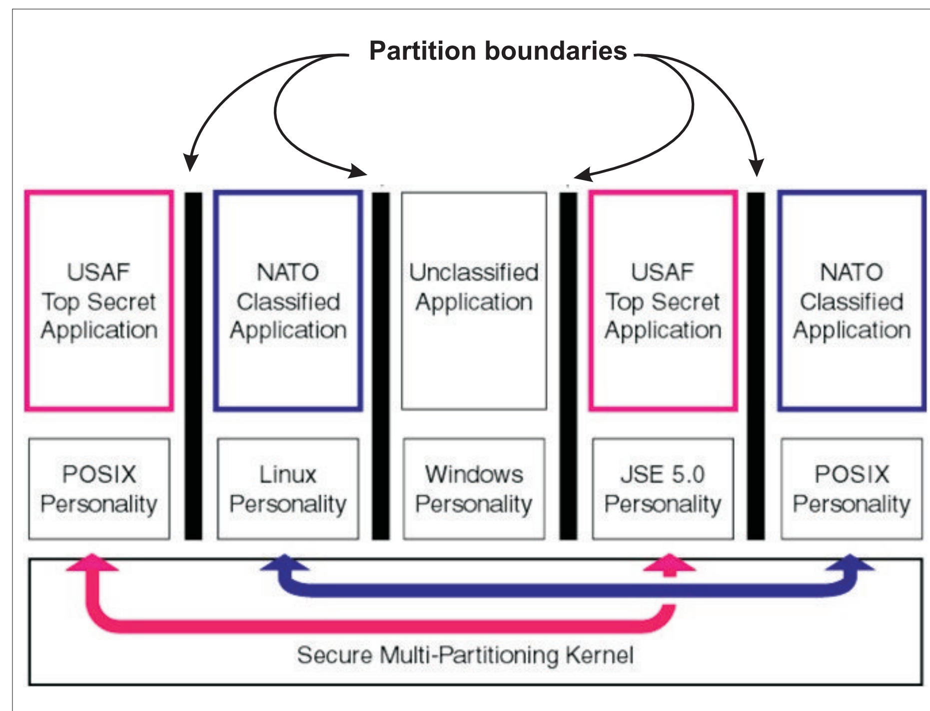

Figure 1. A typical configuration of MILS partitioned operating system

A benefit of using MILS kernels to isolate individual security realms is that it is not necessary to trace the information flows within the application software running in any particular partition. This is because all of the information contained within a particular

partition is treated as having the same security level.

Note that use of a MILS operating system does not obviate all security concerns. In particular, it is still necessary to assure that the software does not have Easter Eggs or other misbehavior built in. Unauthorized users may not be able to communicate with the software secured within a MILS partition, but the software, once resident within a top-secret partition, could still do considerable damage if its algorithms are not implemented correctly. Also, as with security protocols enforced by physical isolation, it is still necessary to assure that human operators do not compromise security by manually copying information between distinct realms. A user who has access to both a top-secret display screen and an unclassified keyboard could manually copy data from the top-secret to unclassified domains. Similar unauthorized data flow might occur if, for example, a human user moves a flash disk betweeen top-secret and unclassified USB ports.

EXPLOITING PORTABILITY AND SCALABILITY BENEFITS OF JAVA

In order to assure isolation of security realms, existing MILS operating system kernels severely restrict the services provided by the operating system. Though different vendors may differ slightly in exactly which restrictions are implemented, it is common for a MILS kernel to restrict behavior in the following ways:

- There may be no file system support or only very limited file-system access.
- There may be no networking layer, or severe limitations on the use of networking APIs.
- There may be limitations on the ability to send asynchronous signals to the code running within a particular partition.
- There may be limitations on the memory model, prohibiting, for example, the use of virtual memory that might page to non-volatile disk-based storage. The system may also prohibit application code from making dynamic changes to the status bits of particular virtual memory pages (e.g. changing a page from read/write to executable). This may prevent certain common virtual machine behaviors, such as dynamic class loading and dynamic adaptive JIT compilation.
- There may be no graphic user interface system or severe restrictions on how the graphic user interface is used. Certainly, the partitioning kernel would want to prevent users from cutting and pasting information between security realms.

Given all of the restrictions described above, it is usually not possible to directly move existing off-the-shelf software capabilities into a MILS partition, even if the MILS partition presents a personality that sounds like a traditional non-MILS personality. For example, a Linux personality may be available within a MILS partition. But this Linux personality may be missing certain capabilities normally present in Linux.

This means that most software installed into a MILS environment must be ported and possibly modified in order to run within a MILS partition. Developing custom software applications by assembling independently developed software components into working systems is an area where Java really shines. Typical developers find that they are five to ten times more productive than C and C++ developers in these sorts of activities.

LEVERAGING JAVA SECURITY IN IMPLEMENTATION OF MILS SOFTWARE

When application code is deployed within a MILS partition, much of the security is supplied by the underlying operating system. This reduces the amount of extra scrutiny to which the application software must be subjected. However, this does not completely eliminate the need for application security. In particular, many security policies require that the “code” that implements a MILS application be secured to the same level of security as the enclosing MILS partition. That is because any misbehavior of code running at that particular security level has the potential of interjecting malicious or erroneous behavior into that particular security level. Mitigating this risk constitutes an important reason for conducting a security audit.

A second reason to perform a security audit of application software running within a MILS partition is to allow that application to connect to more than one independent security level. All secure systems face the need to allow restricted sharing of information between distinct security levels. In traditional systems which use physical isolation to

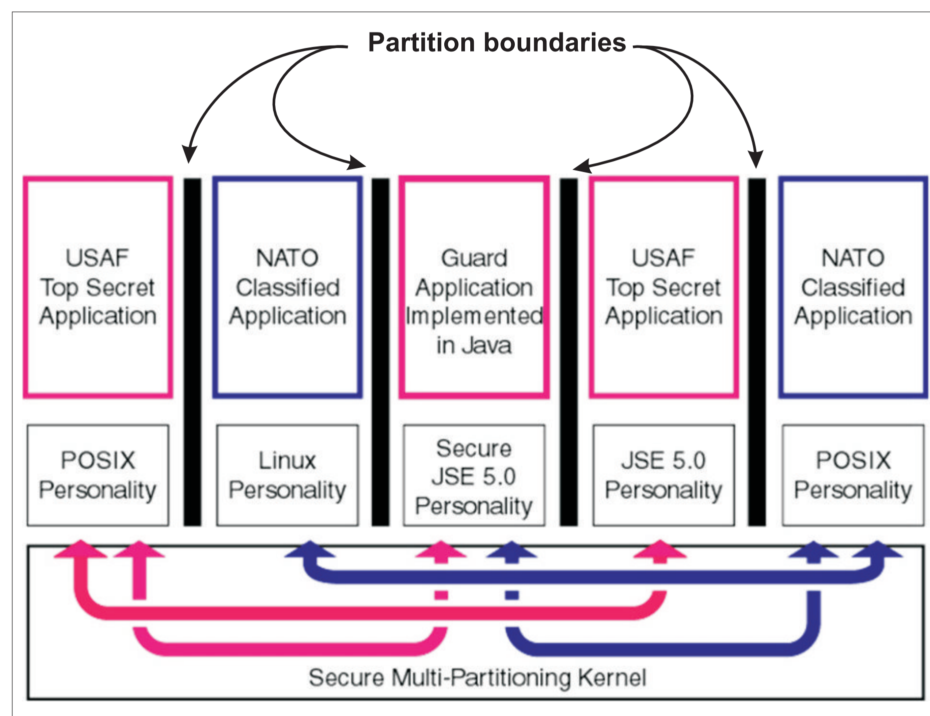

Figure 2. Multiple security levels within a MILS partition

separate security levels, human operators were responsible for monitoring and approving each such information flow. However, placing humans in this loop introduces latency to the information flow and increases the likelihood of security breaches due to inattention or fatigue or even malicious intent of the human operator.

A strength of the MILS architecture is that the selective filtering of information flow between distinct security levels can be largely automated. This is illustrated in Figure 2. In this example application, there is an occasional need to share information between the USAF Top Secret and NATO Classified security realms. A guard application takes responsibility for monitoring and filtering all information that flows between these two domains. Note that the guard application must be very secure. It's implementation, which must enforce the security policies of both NATO Classified and USAF Top Secret domains, must satisfy security auditors from both organizations. Implementing this guard software in Java rather than lower level languages like C or C++ reduces the effort required to develop the software and certify its security.

Summary

Using Java to build secure software systems has numerous advantages.

1. The software can be be assembled of existing off-the-shelf software components combined with small amounts of custom-written software to glue the component together. The cost savings are tremendous.
2. The Java programming environment has many security features built in. This means it is far less expensive to perform a security audit.
3. Due to its high-level programming features, Java systems are much more maintainable than systems implemented in lower level languages. This means it is much cheaper to add incremental new functionality as the system's requirements evolve, or to port the system to new hardware and operating system platforms as the underlyng platform requirements evolve.

MILS partitioned kernels extend the modularity benefits of Java to support building of large secure systems by assembling independent applications. Integrating secure components as independent applications secured within distinct MILS partitions enables very flexible and powerful reconfiguration options with minimal further investment in additional security audits for each new configuration.

To obtain more information, please contact Atego at **www.atego.com** or call one of our sales offices

North America
Phone: (888) 91-ATEGO
Fax: (858) 824-0212
E-mail: info@atego.com

France
Phone: +33 (0) 1 4146-1999
Fax: +33 (0) 1 4146-1990
E-mail: info@atego.com

United Kingdom
Phone: +44 (0) 1491 415000
Fax: +44 (0) 1491 575033
E-mail: info@atego.com

Germany
Phone: +49 7243 5318-0
Fax: +49 7243 5318-78
E-mail: info@atego.com

© 2010 Atego. All rights reserved. Atego™ is a trademark of Atego. PERC® is a registered trademarks or service mark of Atego. Java™ and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the US and other countries. All other company and product names are the trademarks of their respective companies.