5-6-2003

"Protecting Intellectual Property in a Software Development Environment"

Jeffrey Woldan
Messiah College

Honors Project: Securing an Open Source IDE
Final Project Report and Summary
Jeffrey Woldan
May 6, 2003

"Protecting Intellectual Property in a Software Development Environment"

Contents

General Discussion

In today's modern workplace, information theft has become an increasingly significant problem. The ease of portability and distribution of computer files has been a valuable tool for businesses, but has also opened the door for dishonest individuals to steal these files and use the information for their own purposes. Hackers are a threat, as more and more businesses go online, but the biggest threat of information theft may actually come from within a company itself. In September 2002, a study was released by ASIS International, Pricewaterhouse Coopers, and the U.S. Chamber of Commerce titled "Trends in Proprietary Information Loss."[1] It sited former employees and on-site contractors as two of the most significant risk factors in proprietary information and intellectual property loss. The study also found that the risk area for information theft most commonly sited by companies was research and development, which would include program code developed within a company. Thus software developers, among many other types of employees, could certainly be a security risk to a company.

It is becoming more important to protect a company's intellectual property from theft by those on the inside as well as the outside. So how does a company go about protecting from insider theft? Let's assume for the sake of argument that it is unreasonable to expect to be able to hire only trustworthy, honest people to work at a company. This is reasonable, as an untrustworthy person would probably not present themselves as such. In fact, some would argue that because computers make it so easy to share information, it could cause an otherwise honest person to take advantage of this to

[1] Trends in Proprietary Information Loss. Sept 2002. Asis International, Pricewaterhouse Coopers LLP, and US Chamber of Commerce. Available online at http://www.asisonline.org/pdf/spi2.pdf.

use company information for their own purposes. Whether statements such as these are valid could be debated for some time, but this is not the purpose of this paper. Instead, we will consider what practical modifications could be made to a workplace computing environment to prevent this type of insider information theft.

The idea behind modifying a workplace computing environment is to restrict an employee's access to files, so that they can access the files they need to do their job, but to prevent access to them in ways that make them insecure. This immediately establishes a conflict between convenience and security. For example, it may be convenient for an employee to take home company information on a floppy disk or CD-R, but this may also allows them to give this information to anyone they choose. The issue of information security is not cut-and-dry; instead, a company must make compromises between what insecurities are reasonable allowances for productivity, and what are excessive risks to the company's welfare. Thus, the issue can become quite complex. In this paper, however, we will simplify this problem by looking at how a particular program could be modified to create a more secure work environment. We will start by looking at the security problems posed by this program, and then consider various possible solutions to these problems. In examining these solutions, we will see the tension that often occurs between convenience to the user and the issue of securing intellectual property.

The Eclipse project (http://www.eclipse.org) is an open source project to create an IDE that can be a universal platform for essentially any type of development work. The program itself is a sort of framework upon which development tools can be implemented. These are created through plugins specifically designed for Eclipse. For example, included with Eclipse is a plugin that allows the development of Java code, with many if

not all of the features of commercial Java IDEs. There are many other plugins available, a significant number of which are created by third-party sources. Because program code is an important part of intellectual property, it made sense to focus this project on an IDE. Due to the open source, extensible nature of Eclipse, working with the Eclipse IDE seemed to make a lot of sense.

Eclipse is written almost entirely in Java. It is built on the SWT GUI toolkit, which was created by IBM. In fact, a large portion of the code base for Eclipse was created by IBM, and some of this code is used in their commercial IDE, Websphere. SWT, which stands for Standard Widget Toolkit, is a GUI toolkit like AWT or Swing, but independent from both of these. Like the rest of the Eclipse code base, SWT is open source. Eclipse also includes code from other open source projects, such as the Xerces XML parser, which is part of the Apache open source project. Since Java code runs on a virtual machine, the bulk of Eclipse is not its executable file. In fact, all the tiny executable does is start the virtual machine and load the required java libraries. These libraries are stored in the form of compressed .jar files. Contained in these files are compiled .class files, grouped in the .jar files by packages. Because of this, Eclipse is very modular and conducive to a plugin-dependent design.

Before considering the problem of securing the Eclipse IDE, let me admit that this project is more of a study in theory than an attempt to create a full-fledged solution for secure development. My goal in this project is simply to work towards a secure environment within the Eclipse IDE, without really dealing with security concerns external to Eclipse. Thus, no matter how secure the program is, as long as the files are stored in an insecure file system accessible outside Eclipse, it cannot be guaranteed that

they are truly secure. We will touch on the issue of secure files systems and operating systems, but it is not the focus of this paper. Still, in attempting to make Eclipse a secure development environment, we face the same issues that we would if we were working towards a more comprehensive solution, such as one designed on an operating system level. Thus it is still a valuable project, with considerable practical application. In this paper, I will discuss the security features necessary to make Eclipse a secure environment. I will discuss their importance, as well as their possible effects on user productivity.

First, let's consider ways that company files could be taken by an employee for their own use by using the development environment they work in. One way is by saving files to a floppy disk or burning them to CD. Without security measures in place, this is probably the most convenient method of information theft available to an employee. Another method is saving files to public network or internet filespaces. While this may take more work to set up and access, it certainly a viable option as well. If an employee can print paper copies of company files, simply taking home these copies is an easy way to steal information. Also, emailing files would be an easy way to send company information to computers outside the company. There may be other more obscure methods for stealing information from within the development environment, but these are the most obvious, and thus the most important to protect against. When considering these problems on an application level, we must consider all the ways the application could allow these insecurities. Beyond that, however, we must also note how the application could make the files available to other programs with these insecurities. For example, if the program has the ability to launch other programs, as Eclipse does, this may create

other security holes. While I will attempt to fix only a small portion of these insecurities, the eventual goal is to remove all of them from the Eclipse environment.

Outline: Key Concerns and Solutions

In considering the problem of securing the Eclipse IDE, we will assume in many of the proposed solutions that there is some way to store files in a secure location, or at least a way to mark files as secure, for the purpose of identifying which files should be protected by Eclipse. If we assume the secure location option, this requires external operating system or file system support, ensuring that secure files are only accessed by appropriate programs and in appropriate ways. If we simply mark files as secure, with some sort of flag, for example, this might mean that the files are not truly secure outside of the Eclipse environment. This option, however, could possibly be handled within Eclipse itself by keeping some sort of registry of secure files. It could then check this registry any time it loaded a file, to see if that file should be secure. Other proposed solutions do not assume any sort of secure files, and thus are independent of the computing environment the program is run in. Their effectiveness, however, might still be compromised if the rest of the development environment was not properly secured, as previously mentioned. We will also assume in this discussion that Eclipse itself will only be modified by a system administrator, because if any user can modify Eclipse, they can easily remove any security features than might have been in place. Prevention these modifications could be as easy as not giving the user modify rights for the Eclipse program files.

In the outline below, security problems posed by the Eclipse environment are listed. Following each problem is a list of possible solutions, with the merits and problems posed by each solution. They are grouped by the type of insecurity posed.

Before the outline, however, there are several Eclipse features that should be explained before presenting the vulnerabilities posed by them. They are as follows:

Eclipse-Specific Features

1. The Eclipse workspace is the set of projects and files which are currently available for editing in Eclipse. Files can be imported and exported from the workspace to standard filespace locations. If a file is in the workspace, it is still in the standard file system. A project/file must be in the workspace to be edited within Eclipse.
2. The Eclipse navigator lists the project and files currently in the Eclipse workspace. It works much like Windows Explorer, as it shows projects and files in a hierarchical view, and allows drag and drop and copy and paste of files to different locations within the workspace. It also allows drag and drop and copy and paste to locations outside the workspace.
3. Eclipse's compare changes feature is a robust utility for comparing the differences between two different files, or even previous versions of the same file. It allows parts of one file which differ from the other to be copied in place of those differences, to "synchronize" the two documents.

Security Features Outline

I. Saving or moving files to a floppy disk/public network or internet filespace

   a. Save As... functionality may allow files to be saved to insecure locations or to removable disks.

Solution i: Completely disable Save As... option. With this feature disabled, it cannot pose a security threat. It may prevent the user from moving secure files to other secure locations, however.

Solution ii: Completely disable Save As... for secure files, allow Save As... for insecure files. This requires identification of secure files within Eclipse.

Solution iii: Restrict Save As... to only allow saving of secure files to secure locations, and not to removable drives. This allows Save As... to be used in secure ways instead of completely disabling it. This is the most complicated solution, however, because to do this, Eclipse must be able to identify which filespaces are secure, and which are not.

b. Export... functionality, which exports files in the Eclipse workspace to any location in the file system, may allow files to be saved to insecure locations or removable disks.

Solution i: Completely disable Export... option. This is essentially the same as Solution i to a., above.

Solution ii: Completely disable Export... option for secure files, allow it for insecure files. This is essentially the same as Solution ii to b, above.

Solution iii: Restrict Export... to only allow saving of secure files to secure locations, and not to removable drives. This is essentially the same as Solution iii to a., above.

c. The Eclipse navigator may allow files to be moved to insecure projects or insecure locations in the filespace, using its drag and drop or cut and paste functionality for files.

Solution i: Completely disable drag and drop and cut and paste functionality within the workspace navigator. This removes basic drag and drop and cut and paste functionality to all

locations, including secure locations. This may be overly restrictive to the user.

Solution ii: Completely disable drag and drop and cut and paste functionality within the workspace navigator for secure files. Allow such functionality for insecure files. This is still fairly restrictive to the user.

Solution iii: Restrict drag and drop and cut and paste to only allow moving of secure files to secure locations. Do not allow the moving of secure files to insecure projects or filespace locations. Allow insecure files to be moved to secure locations, but the must be marked as secure once moved. This requires identification of both secure project and secure filespace locations by Eclipse.

d. Eclipse's cut and paste functionality for text may allow information contained in secure files to be copied into insecure files in the workspace.

Solution i: Completely disable cut and paste within Eclipse. This would be extremely restrictive to basic productivity.

Solution ii: Restrict cut and paste to only allow pasting within the file the text had been cut from. This would still be quite restrictive to basic productivity.

Solution iii: Restrict cut and paste to only allow pasting of secure text into secure files. This requires identification of secure files/projects in the workspace. Allow cut and paste from insecure locations to all other locations, secure or insecure.

e. Eclipse's compare changes features may allow information in secure files to be copied into insecure files in the workspace and to files in other insecure locations.

Solution i: Completely disable compare changes functionality. This may be restrictive to basic productivity.

Solution ii: Disable compare changes functionality for secure files, allow for insecure files. This requires identification of secure files.

Solution iii: Restrict compare changes to only allow secure files to be compared with other secure files. This requires identification of secure files.

Solution iv: Restrict compare changes to allow comparisons of two secure files to work both ways, but only allow parts of insecure files to be incorporated into secure files when comparing the two, not vice versa. This requires identification of secure files.

II. Printing files

a. Eclipse's printing functionality allows employees access to paper copies of secure files.

Solution i: Completely disable printing from Eclipse. This is somewhat restrictive, but possibly not excessively.

Solution ii: Disable printing of secure files from Eclipse, but allow printing of insecure files. This is less restrictive, but may still be obstructive to productivity in some cases.

Solution iii: Allow printing, if not considered a large enough security risk.

Other Solutions: physical security measures could be used (not enforced by the program itself).

III. Making files available to other insecure programs

a. Eclipse's cut and paste functionality for text may allow information contained in secure files to be copied into insecure editors.

Solution i: Completely disable cut and paste within Eclipse. This would be extremely restrictive to basic productivity.

Solution ii: Restrict cut and paste to only allow pasting within Eclipse, not to other programs. This could be restrictive to productivity if other programs are often used in conjunction with Eclipse.

Solution iii: Restrict cut and paste to allow pasting of secure files in Eclipse to any file in other secure programs. This requires Identification of secure programs. It also creates a security hole, because it does not check if the file being edited in the other program is secure. Allow cut and paste from insecure files to all other programs, secure or insecure.

Solution iv: Restrict cut and paste to only allow pasting of secure files in Eclipse to secure files in other programs. This requires identification of secure programs, and whether the file it is editing is secure, which may be impossible without operating system assistance. Allow cut and paste from insecure files to all other programs, secure or insecure.

b. Eclipse's ability to register external editors, using plugin manifest files, for use with certain file types, and to open the system default editor for a file type may allow secure files to be edited in insecure editors.

Solution i: Completely disable external editor use within Eclipse. This may actually be a difficult solution to implement, because this functionality is highly integrated with Eclipse. Also, it may be excessively restrictive.

Solution ii: Do not allow the user to register external editors- only allow the system administrator to register external editors, and make it is his or her responsibility to ensure they are secure before registering them. System editors are not Eclipse's problem, so do not deal with them. This is probably the easiest solution to implement, but is somewhat restrictive.

Solution iii: Eclipse dynamically checks if external editors are secure, and only launches them if they are. This requires identification of secure editors, which may need operating system assistance. This is the most flexible solution to the problem.

IV. CD burning/Email

- Since Eclipse has no built in CD burning or email functionality, these should not be problems. However, it is not impossible that plugins could be created allowing this type of functionality, or creating other security risks. Thus it is important that any plugins that are added to Eclipse be checked for any insecurities they may introduce, and be modified (or simply not used) if such insecurities exist.

Final Progress Report

Over the past academic year, I have worked with Eclipse to implement some of the security features needed to create a secure development environment. This has proved a challenging task, because of the complexity of Eclipse, and the various concepts and tools needed to integrate these features into Eclipse. The first semester of the year I spent working on understanding various parts of the Eclipse code base, and disabling Save As... functionality for all editors within Eclipse. Understanding the code was a complex task, but once I understood the code, this was a fairly simple modification. The problem with this was that it was overly restrictive to the user, and did not take into account any of the situations when Save As... should be allowed. Actually implementing such a restrictive feature modification in a work environment would be inadvisable, as Save As... is a valuable feature for any type of document editing.

My goal the second semester of the year was to create more flexible, user friendly security modifications. I spent most of this second semester working with modifying clipboard functionality within Eclipse. Instead of completely disabling the clipboard, I have modified it so that secure information can only be copied into secure files, and where insecure information can freely be copied into any file. This protects secure files while still allowing a developer to use the clipboard in many ways, instead of the complete removal of this extremely valuable tool. I have also revisited my Save As... modification, enabling it to work with the file's security setting to determine if Save As... should be disabled for that file.

As already mentioned, adding security features often forces a compromise of convenience to users, and thus the ideal solution minimizes the hindrances it places on the user while still enforcing strong security. I have tried to be aware of this compromise and find the best solutions to these problems in the modifications I have completed. Because the secure file system I assume this program would work with is somewhat hypothetical, and because of the general complexity of the Eclipse codebase, there are limits to the flexibility of my solutions, but I believe I have done the best I could given the time and resources at hand.

Assumptions for the Project

As noted previously, a complete, fool-proof security solution was not the goal of this project. As such, it is important to note the assumptions upon which I have based my work. These assumptions detail the type of computing environment I am assuming to be working within in creating security features. The first assumption is in regards to how secure files are stored. We assume that secure files are stored in either a secure file system or some other sort of filespace that protects files from unauthorized usage on an application-by-application basis. Thus, it makes sense to work to create security solutions on an application level, because the application must protect a secure file once it has been given access, and the application need not worry about its security measures being bypassed by opening the file in an insecure program.

We will also assume that there is no way for an application to tell which filespaces are secure and which are not. While this would probably not be the case in a real secure development environment, I do not have such an environment to work with.

This means there is no way to determine valid locations for secure documents to be saved, other than the location they are originally saved in. The next assumption is that other than files being protected from unauthorized access, there are no other security measures built into operating environment. This relates specifically to my work on Eclipse's interface with the system clipboard. We assume that any information placed on the system clipboard is available to any application, secure or insecure. Thus, passing secure data to the system clipboard allows the user to do essentially anything they want with that data, and should be prevented.

Finally, we assume that there is some way to determine the security setting of a file at or after load time. The specific method of determining this security setting is not important. Additional, we assume that we are not concerned with the sharing of information between various users' files. We will only have two security classifications of files: secure and insecure. Since the goal of the project is only to prevent the misuse of valuable company information, we are not concerned with the sharing of information within the company, only if that information leaves the company. Thus, the two security classifications are sufficient for our goals, in that they determine which files need to be protected. While in some situations keeping various users' files separate might be valuable, this type of feature is commonly built into the operating system, and is not the focus of this project.

Save As... Introduction

I began my work on Save As... in the first semester of this year. For my initial version of this modification, I simply disabled Save As... for all files edited within

Eclipse. This proved to be a complicated enough task, without worrying about marking certain locations as secure and allowing Save As... to work for those. A large portion of my work with the code consisted of searching for references to the functionality I wanted to change. I also had to learn about the design hierarchy of Eclipse to better understand how the changes I made would affect the functionality. After some searching, I found the isSaveAsAllowed() method declared in the IEditorPart.java file, which defines the org.eclipse.ui.IEditorPart interface. This method simply returns a Boolean, indicating whether Save As... can be performed with that editor.

The IEditorPart interface must be implemented when creating an editor for Eclipse. Thus all built in editors implement it, and all editors added as plugins must implement it as well. Since it is an interface, however, the IEditorPart interface cannot contain any defined methods, only their declarations. org.eclipse.ui.texteditor.AbstractTextEditor is a base class that implements IEditorPart and defines many of the methods needed for an editor in default, generic ways. All of Eclipse's built in editors extend this class, and simply override or add methods as needed. This includes Eclipse's built in text editor, defined in the org.eclipse.ui.editor.text.TextEditor class.

One of the first solutions I considered was to somehow prevent the overriding of the isSaveAsAllowed() method in AbstractTextEditor, which is set to return false. I was unsure of how this would work at first. I did not know how I would prevent the method from being overridden, because the ability to override other functionality is a key part of Eclipse architecture and its plugin-oriented design. Even if there was some way to do this, I was concerned that it might cause other problems in the program.

Compiling Eclipse

At the same time as I was considering the Save As... problem, I was attempting to compile the Eclipse source code. I knew that to implement and test any changes, I would need to compile my solution into a working program. While I was working on the problem of compiling Eclipse, I was also attempting to compile a sample plugin discussed in the Eclipse help files. I was considering the option of using a plugin to enforce security features at the time, and I figured I could better understand how plugins work by creating this sample plugin. This plugin would simply create a window with the text "Hello World" within the Eclipse environment. I was having problems, however, with satisfying file dependences for this plugin. I could not figure out why it could not find the files it needed, nor could I figure out how to add these files to the build path. I spent some time searching the Eclipse source code and program directory structure to find where these files were located. Finally, I discovered that the files to satisfy the dependencies were to be found in the .jar files that compose the codebase of the Eclipse application itself. I determined how to add these to the build path of the plugin project, and I was able to compile the plugin and eventually register it for use within Eclipse. Plugins are registered with plugin manifest files, written in xml, that tell Eclipse how to use the plugin.

The knowledge gained from creating the sample plugin would come in handy in compiling the Eclipse source code. First, however, I concerned myself with compiling the executable that launches Eclipse. I had come across the launchersrc.zip file in the source code, which contains the source for this executable, written in C. Included in this .zip file was a build.bat file, which, when ran, executed the commands needed to

compile the source code into a working executable. I tried to run the executable and received an error message indicating that it could not find a library it needed to run, startup.jar. I found the source code for this library, conveniently stored in startupsrc.zip. Using what I had learned from compiling the sample plugin, I was actually able to use my precompiled copy of Eclipse to import the files for the library, satisfy its dependencies, and export the compiled files to a .jar file for the Eclipse executable to use. I ran the executable again, and it found the startup.jar file, but returned another error message indicating another missing library. I continued this process of compiling libraries for the runtime.jar and boot.jar libraries.

Initial Save As... Modification

After compiling the three libraries, I decided to save time by simply copying over the precompiled libraries from my precompiled copy of Eclipse. I knew how to create these libraries, and so it was more important to now use this knowledge to make modifications to the Save As... functionality. I initially tried to disable Save As... simply in the TextEditor class, by changing its isSaveAsAllowed() method to return false. After compiling my modified copy of workbench.jar, which contains the org.eclipse.ui package, and placing it in the appropriate directory, I found that I had successfully disabled Save As... for the standard text editor.

At this point I still did not know how to better ensure that no files could use Save As..., no matter the Eclipse editor in use. The solution, however, turned out to be fairly simple. Since essentially all editors extend the AbstractTextEditor class, I decide to try declaring the isSaveAsAllowed() method in this class as final, and see how this affected

the functionality of editors that tried to override this method. In doing this, I found that editors that attempted to override the method could not be opened, because their classes could not be instantiated. Instead, a dialog box indicating the attempt to override the final method was displayed. It turned out that declaring the method as final essentially enforced this security feature on its own. To make these other editors run, I had to remove the attempt to override the method from the code, and put recompiled versions of them in the appropriate libraries.

While this provided one solution to the security issues posed by Save As..., it was a fairly restrictive solution. Learning enough about Eclipse to make this modification took most of the first semester, but when I had the chance to revisit Save As... in the second semester, I knew I could improve on this modification. First, however, I would work on implementing secure clipboard functionality for Eclipse, and on creating a security framework to support these security modifications.

Clipboard Introduction

While I did not make any modifications to the Eclipse clipboard in the first semester of the project, I still spent considerable time learning how modifications might be made. I began research by searching for locations of code referencing the clipboard in some way. I found occurrences of the java.awt.datatransfer.Clipboard object in the code which seemed promising at first, but it turned out that this Clipboard object is used for other purposes in the code.

The Eclipse tools newsgroup, found at news://www.eclipse.org/eclipse.tools, proved a valuable resource for understand the clipboard functionality of Eclipse. I posted

a question regarding the implementation of the clipboard on this newsgroup, and received a reply indicating that Eclipse's clipboard functionality is implemented with the SWT toolkit. I found the SWT source code, which is included with Eclipse. I spent a considerable amount of time at the end of the first semester searching the code to see how the clipboard is incorporated into the editors in Eclipse.

My main goal the second semester of the year was to modify the Eclipse clipboard to protect information in secure files modified within Eclipse, without disabling the use of the clipboard. In doing this, my research focused on the Standard Widget Toolkit (SWT). The SWT toolkit is a powerful and flexible package, which allows the creating of widgets that appear and function like native system widgets. This is done by using native widgets as often as possible. This means that while the actual implementation of SWT varies from operating system to operating system, the SWT API remains the same. There are versions of SWT (and Eclipse) for Windows, Linux, Solaris, and Mac OSX, among others. Where native implementations of SWT widgets are not available, they are emulated for that operating system. This allows Development in SWT to be portable while creating an application that seems system-specific.

The org.eclipse.swt.dnd.Clipboard object provides the standard SWT interface to the clipboard. The Clipboard class itself does not make native system calls, but makes calls to a system specific object, which contains native methods implemented with JNI. In windows, the org.eclipse.swt.internal.ole.win32.COM object provides this functionality. JNI (Java Native Interface) provides an interface to the SWT native library (a .DLL in windows), which is written in C. The Clipboard object provided two methods that allow data to be passed to and from the Clipboard: setContents(Object[]

data, Transfer[] dataTypes) and getContents(Transfer transfer). I focused on trying to understand these methods, so that they could be modified. However, I found their references to the COM object confusing, as they seemed to reference specific memory locations and not pass the clipboard information as parameters to these methods.

I considered at one point not modifying the Clipboard object, and instead modifying only the org.eclipse.swt.custom.StyledText object. This is a text widget which is used by the editors in Eclipse, forming the area in which files are displayed and edited. This object has cut, copy, and paste methods built in, which access the Clipboard object. The reasoning behind modifying StyledText instead of the Clipboard class was that by modifying the Clipboard class itself, I might affect other text widgets in Eclipse, such as those in preference dialogs. I did not know if StyledText was the only class accessing the Clipboard, and I was afraid that if I modified the Clipboard class it might create problems for other parts of Eclipse.

Before I started to modify any Eclipse or SWT code, I created sample program to better understand the usage of the clipboard object, and to test possible clipboard solutions. My first test program used the java.awt.datatransfer.Clipboard class, mainly to see if I might be able to use this clipboard class as part of my solution. One interesting feature of the java.awt.datatransfer.Clipboard class is that it can reference the system clipboard, but it can also reference a local clipboard internal to the application it is used in. The org.eclipse.swt.dnd.Clipboard object did not have this feature- it automatically interfaced with the system clipboard. In this first test program, I created two java.awt.datatransfer.Clipboard objects, one that referenced the system clipboard, and one that provided a local clipboard for the program. In this program, the user can type text in

a text box, and then, using buttons in the application window, select either the system or local clipboard, and then copy, cut and paste using that clipboard. The program functioned as expected, and data placed on the local clipboard was not available to other applications.

I also created a second test program to try using the SWT classes in conjunction with the java.awt.datatransfer.Clipboard class. In the second program, I used a java.awt.datatransfer.Clipboard object as the local clipboard, and an org.eclipse.swt.dnd.Clipboard object as the system clipboard. I also used SWT widgets to create this second program interface, where I used AWT widgets for the first, and created a more understandable user interface, that displays the currently selected clipboard's contents in one text box while allowing text to be edited in another. This test was successful as well, with the AWT clipboard serving as a local clipboard unavailable to other applications.

At this point, I realized that a possible method to avoid breaking parts of Eclipse through my modifications was to leave the original classes intact and create subclasses of those classes that needed to be modified. Because all original classes would be unchanged, they would perform exactly as expected, but the new subclasses would implement the desired security features. For example, the Clipboard class would be extended by a SecureClipboard class, and the StyledText object would be extended by a SecureStyledText class, and so on. I began my modifications this way, but soon realized that this method made things quite complicated. For example, when used within an Eclipse editor, the StyledText object is wrapped in an org.eclipse.jface.text.source.SourceViewer class. Thus, the SecureStyledText class

should be wrapped in a SecureSourceViewer class. However, the number of classes that reference SourceViewer is large, and the number of secure classes to add to Eclipse increased greatly as I continued up the code hierarchy. Another problem with this strategy was that some of the methods I needed to override were declared private, and thus I could not call those superclass methods. It seemed pointless to rewrite these methods for my secure classes, but I didn't want to change the access levels to these methods either, because this meant I would be modifying the original classes and creating new secure ones as well. Overall, this strategy seemed to create more problems than its resulting benefits warranted.

After reconsidering my options, I decided to add security functionality to existing SWT and Eclipse classes. The goal in doing this was to make this security functionality invisible to calling classes that did not want to access it, thus ensuring that these classes would continue to work as expected, but to provide the ability to activate security functionality when desired. There were several benefits to this approach: First, there were far fewer classes to modify, because many classes involved in an Eclipse editor do not need to directly deal with security, but access classes that need security features. By retaining the original class names, references to the modified classes often remained the same. Secondly, working within the original classes, I had access to all fields and methods of those classes, even private members. Also, by modifying the original classes, the security features are more highly integrated into Eclipse, and expanding this security to other aspects of the program becomes easier as well.

Security Framework

Before I could implement these security features, I needed to decide how to implement a security framework to support them. This framework would provide the security settings of files to the various secure classes within Eclipse, thus allowing them to handle those files appropriately. At the suggestion of my project advisor, I researched the java.security packages to ascertain whether they would be valuable in enforcing these security settings. I had begun to create my own classes to do this, but it seemed unnecessary to write code that provided similar functionality to already existing classes. Additionally, the Java security classes seemed to provide more powerful security features than any I might create.

As I researched the java.security classes, I hoped to file a way to assign permissions directly to the files to be edited in Eclipse. Unfortunately, this type of permission assignment is not part of the java.security classes. However, the java.security package does depend heavily on permission objects. There are many types of permissions included as part of the java.security package, some of which can be automatically checked by the virtual machine, if so enabled. These permissions range from file permissions like read, write and execute, to run time permissions such as creating a class loader, to security permissions like setting or modifying the current security policy. These permissions are enforced by running a program under a security manager. This can be done in the code itself, but can also be declared as a command line parameter. The security manager automatically loads the default java security policy and .java.policy file in the user's home directory, but can also load a specified policy file.

A policy file allows you to set java permissions. Permissions are assigned based on three possible criteria: the codebase (or the location from which the code is being loaded), the signature on the code, and the principal (or user) loading the code. The codebase is a URL representing is a directory or .jar file. Code is signed using a private key, which creates hash values for individual files. Signed files are stored in a .jar file, with their signatures stored in another file within the .jar. Principals can be user names in various operating systems, including Windows and Linux. Any or all of these criteria can be used when assigning permissions to files.

Assigning permissions based on codebase or principal is as simple as stating their values in the policy file, but assigning permissions based on file signatures is slightly more complicated. First, a key must be created, which will be stored in a keystore. A java keystore is a file that stores private and public keys. The *keytool* command line tool allows a user to create a keystore and add keys to it. Keystores and individual keys each have a password to access and use them, and keys store identifying information about the owner of the key. Once you have created a key, you can now sign the contents of .jar files, using the *jarsigner* command line tool. When using this tool, you specify the desired keystore, key, and the passwords for each, as well as the .jar file to be signed. The *jarsigner* tool then creates the signed .jar file using the specified key. If you intend to share this signed code, you must export a public key from the keystore to pass along as well, using the *keytool*. The person you are sharing the code must then import the key, again using the *keytool* to import the public key into their own keystore.

There are many permission classes included as part of Java, and many of these are automatically checked when running a program under the default security manager.

However, there was no built in permission class that did exactly what I needed it to do. Fortunately, a user can create their own permissions by extending existing permission classes. These user-created permission classes are not checked automatically, but are checked by calling the AccessController.checkPermission(Permission permission) method. AccessController is a final class, with static methods for checking and changing the current security context. checkPermission(Permission permission) checks if that permission is allowed by the policy file in effect. This method throws an AccessControlException if the permission is not granted for the current security context. Using a try/catch block, the checkPermission method can control how the program reacts based on whether the security is granted.

I would eventually create two permissions classes, which I added to the org.eclipse.swt.security package. This was a package I added to SWT to contain any new security classes not based off existing SWT classes. These permissions were SecureClipboardPermission, and SecureSaveAsPermission, intended to be checked when clipboard access and Save As... operations were performed, respectively. These permission classes extend the java.security.BasicPermission class. These classes' constructors, like all permissions, take a String as an argument, which helps to refine the permissions being granted. This being the case, I decided to pass the security setting of the files as a parameter to the permissions.

I created an org.eclipse.swt.security.Security class to store the String constants that would be parameters for the permission classes I created. I created two String constants, STANDARDSECURE and STANDARDINSECURE. These constants correspond to secure and insecure files, respectively. I decided to have only these two

security settings for files, because I wasn't dealing with various levels of security. However, for more complex security classification of files, with various security levels, more constants could be added and enforced in the policy file. By declaring constants for these settings, it means the actual values of the constants can be easily changed without having to the change references to the constants.

The next question was how to pass permissions, or rather, how to obtain the permissions for files and enforce those permissions. First, I dealt with passing permissions within Eclipse. I debated signing only specific files, and giving those signed files the appropriate security permissions. When a file was loaded into an Eclipse editor, a factory method would then be called, checking the permission of the file and loading it into the appropriate class. A factory method has the effect of allowing of any of a set of classes all satisfying a single interface to be chosen for creation at runtime. This would have the effect of allowing secure files to be loaded into a signed class, where insecure files would be loaded into an unsigned class. Otherwise, these classes would be functionally the same. When Java permissions are checked, they are recursively checked for all calling classes. If a class in the calling chain did not have the permission being checked, an AccessControlException would be raised. Thus, even if this class is not directly accessing secure features, as long as it is in the calling chain, its given permissions are checked.

I decided against this solution, however, because I did not see any particular class as lending itself to such an implementation. It also seemed overly complicated and unnecessary because it did not provide more security than the simpler method I decide to use instead. In this method, security settings are passed using the String constants

declared in the org.eclipse.swt.security.Security class. When files are loaded into an Eclipse editor, a permission String is set for that file, and then that string is passed to classes that need this setting. I decided to pass the security String instead of actual permissions because there are multiple permissions for each file, and instead of trying to pass all of them, a single string indicating the setting for all those permissions is simpler. Using Strings to pass security settings may seem less secure than assigning permissions solely by file signature, but this method uses both the String and the file signature to enforce the security. The permissions are still assigned in the policy file based on the file signature, which means that the files accessing these security features must be signed. That way, unless a user has access to the private key and password used to sign the files, they cannot change the way the security Strings are passed or the way the security features are accessed.

In addition to passing permissions within Eclipse, the security framework required a method to load the security settings of files into Eclipse. In an actual secure development environment, this would involve interfacing with the secure filespace to obtain these settings. Since we are not working with an actual secure filespace, I needed a stand in solution that would substitute for the secure filespace. In doing this, I wanted to ensure that an interface to an actual secure filespace could be added as needed. To do this, I started by creating an ISecurityRegistry interface, with only one method, getSecurity(IEditorInput input). This method takes an IEditorInput, which is a wrapper class that presents a document to an editor, and returns the security setting for that document. To implement this interface, I decided to use a simple text file to serve as a registry, storing security information on the files edited by Eclipse.

FileSecurityRegistry implements ISecurityRegistry using such a text file registry. The file format contains one entry per line, placing the path of the file on the line, followed by that file's security setting. An example of the registry file is shown below:

```
/test/secure.txt STANDARDSECURE
/test/mytest.java STANDARDSECURE
/test/anothertest.xml STANDARDSECURE
```

These security settings correspond to the constants declared in org.eclipse.swt.security.Security. Like references to these constants in the Eclipse code, the value of the constants can then be changed without having to change the security registry file. To implement this functionality, I used the java reflection classes. To retrieve the value of the constant listed in the registry file, for example, I had to create an instance of the org.eclipse.swt.security.Security class. I then had to call the getClass() method on that object to retrieve a Class object. This gave me access to the Class.getField(String field) method, which retrieves a Field object with the name given by the String parameter, if such a field exists. Given that Field object, I called the Field.get(Object object) method, which returns the value of that field for the given object, again if that field exists for the object. The code for this is shown below:

```
/**
 * Retrieves the security of an IFileEditorInput.
 *
 * @param input the file input to be checked
 * @return the security setting String for the input.
 */
private String getSecurity(IFileEditorInput input) {
  try {

    // create the BufferedReader object.
    registryBuffer = new BufferedReader(new FileReader(registryPath));

    // retrieve the path of the file being checked.
    IFile file = input.getFile();
```

```
        IPath path = file.getFullPath();
        String pathString = path.toString();

        /*
         * read through the registry until the editor file is found,
         * or until the end of the registry is reached.
         */
        while(registryBuffer.ready()) {

          // tokenize a line
          registryLine = new StringTokenizer(registryBuffer.readLine());

          // check if the path matches the editor file.
          if(registryLine.nextToken().equals(pathString)) {

            /*
             * if the path matches, get next token, and obtain the
             * value of the field in the Security class where
             * the field name matches that token.
             */
            return (String) securityObject.getClass().
                        getField(registryLine.nextToken()).
                        get(securityObject);
          }
        }
      }
      catch (IOException e) {
        // continue to default case
      }
      catch (NoSuchFieldException e) {
        // continue to default case
      }
      catch (IllegalAccessException e) {
        // continue to default case
      }

      /*
       *  if there is a problem retrieving the security setting,
       *  return the default security setting.
       */
      return defaultSecurity;

    }
```

You will notice that this is a private method which takes an IFileEditorInput. IFileEditorInput extends IEditorInput, and is used if the input is a file. Thus, in FileSecurity.getSecurity(IEditorInput input), we check if the input is a IFileEditorInput, and if so, call the private method above. If it is not a IFileEditorInput, then it is not a file, and there is currently no implementation to check its security.

This implementation opens the possibility for several different exceptions to be thrown. The first is an IOException, thrown if the registry file can not be accessed for some reason. The second is a NoSuchFieldException, which is thrown if the field requested by Class.getField(String field) does not exist. Third, an IllegalAccessException is thrown by Field.get(Object object) method if that field does not exist for the object. In addition to these exceptions, if there is no security setting listed for the file in the registry, this must be handled, and we must also return a value even if the IEditorInput is not a file. To handle all of these situations, a default return value was needed. I created a String field, defaultSecurity, to be returned in these cases. I decided to set this field's default value to Security.STANDARDINSECURE. By doing this, only secure files need to be listed in the registry file. The disadvantage of this is that if the registry became corrupted, secure files could be assigned an insecure security setting. The logic behind this choice, however, is that a greater number of insecure files exist on the system than secure files, and marking them all as such would be an excessive amount of work. This could be changed depending on the security level desired. Making the default security setting Security.STANDARDINSECURE is more convenient to the user, but making the default setting Security.STANDARDSECURE would offer a higher level of backup security, in case of a problem with the security registry. To make this easier to change, I added a setDefaultSecurity(String security) method to ISecurityRegistry and FileSecurityRegistry so that the default security could be changed without editing the class.

Since FileSecurityRegistry is intended as a temporary solution, I created a factory class to return this object, RegistryManager. RegistryManager has a getRegistry method,

which returns an implementation of ISecurityRegistry. In its current implementation, it simply returns a FileSecurityRegistry. The value of this is that there are no direct references to FileSecurityRegistry in the code, and thus a new security registry could be implemented by simply creating a new class implementing ISecurityRegistry and modifying RegistryManager.getRegistry() to return that class. This class also currently contains a string constant which points to the location of the registry file, and is passed to the FileSecurityRegistry constructor. This also hides the implementation of ISecurityRegistry from the rest of Eclipse.

For these security classes, I created a new package, org.eclipse.security. I chose not to place them in the org.eclipse.swt.security package because these classes provide higher level security functionality, designed to work at a higher level than SWT. For example, IEditorInput and IFileEditorInput are not parts of SWT but are instead declared in org.eclipse.ui. Creating references to IEditorInput within SWT would create a dependency that should not exist for such a toolkit. I also decide to name the registry file eclipse.registry, and to place this file in the org.eclipse.security package folder, storing the file closely to the class that accesses it.

org.eclipse.swt.dnd.Clipboard Modifications

With a security framework in place, I focused again on clipboard modifications. When I began to look into modifying the org.eclipse.swt.dnd.Clipboard object, I was concerned by the fact that the clipboard used direct memory access to pass information to and from the system clipboard. However, I realized that as long as I did not change the existing code, I could add other code to modify when data was passed or retrieved from

the system clipboard. The key modifications to the clipboard class were in the getContents(Transfer transfer) and setContents(Object[] data, Transfer[] dataTypes) methods, which retrieve and send data to the system clipboard, respectively. I knew that I needed to modify both how the Clipboard class interacted with the system clipboard, and also how the Clipboard class interacted with the rest of Eclipse. My goal was to make these changes to clipboard functionality as transparent as possible- that is, those changes should only be evident to the user when necessary enforce security. Thus, I outlined the following expected behaviors for the secure clipboard.

I. When copying data from insecure documents within Eclipse, that data should be passed to the system clipboard, making it available to all applications.

II. Pasting data from insecure documents, regardless of the target document's security, should be allowed, occurring as a standard paste.

III. When copying data from secure documents, that data should not be passed to the system clipboard, but instead stored within the secure clipboard in Eclipse. The system clipboard should be cleared when secure data is on the Eclipse clipboard, to preserve the appearance to the user of a single clipboard, which is inaccessible to other applications when secure data is stored on it.

IV. Pasting data from secure documents into secure documents within Eclipse should occur as a standard paste.

V. Pasting data from secure documents into insecure documents within Eclipse should not be allowed.

VI. When data is copied to the system clipboard from any other application, which is assumed to be insecure, this data should be available for pasting into both secure and insecure documents in Eclipse, occurring as a standard paste.

Before I handled these various cases, I needed to allow security constants to be passed to the clipboard. I did this by adding two new methods, getContents(Transfer transfer, String security) and setContents(Object[] data, Transfer[] dataTypes, String security), which added an additional string parameter to the methods already part of the Clipboard class. The original getContents(Transfer transfer) and setContents(Object[] data, Transfer[] dataTypes) were then renamed as internalGetContents(Transfer transfer) and internalSetContents(Object[] data, Transfer[] dataTypes). This way, the new getContents and setContents methods could call these methods when they wanted to interact with the system clipboard. In the new getContents and setContents methods, I used the AccessController.checkPermission(Permission permission) method to check the security parameter passed to the method, and defined the following functionality, corresponding with the outline of expected behavior above.

I. **Copying from insecure documents** - When the setContents method is called with the Security.STANDARDINSECURE parameter, the Object[] and Transfer[] parameters are saved in the private fields Object[] saveData and Transfer[] saveTransferAgents within the Clipboard class. Immediately afterwards, the SecureClipboardPermission is checked, and an AccessControllerException is thrown. In the catch block, a call is made to internalSetContents, which passes the data to the system clipboard.

II. **Pasting from insecure documents** - When the getContents method is called with any security parameter, internalGetContents is called and the data is stored in a local variable. If there is data on the system clipboard, this data is returned. Since in case I insecure data is passed to the system clipboard, this data would be returned.

III. **Copying from secure documents** - When the setContents method is called with the Security.STANDARDSECURE method, the Object[] and Transfer[] parameters are save as in case I, but no exception is thrown, and the entire try block is executed. This code passes a null object to the system clipboard, so it appears to the user that the secure data simply cannot be retrieve in other programs.

IV. **Pasting from secure documents to secure documents** - When the getContents method is called with the Security.STANDARDSECURE parameter, internalGetContents is called and stored as in case II. No exception is thrown, and the entire try block is executed. If there is no data on the system clipboard, an attempt is made to retrieve the appropriate data from the private field saveData, using saveTransferAgents to find the right data type. This data is returned, unless the appropriate data type cannot be retrieved, in which case the null object on the system clipboard is returned.

V. **Pasting from secure documents to insecure documents** - This is handled by case II. When the AccessControlException is thrown, the system clipboard data is returned instead of the data stored in the saveData field.

VI. **Pasting from other applications** - As noted in case IV, even if no exception is thrown, the system clipboard contents are retrieved and checked. If the data on the system clipboard is not a null object, this means that data has been placed on the system clipboard since the time secure data was copied to the clipboard, and this data is returned, whether or not the document being pasted into is secure.

The code for these methods is as follows:

```
/* Added 4-9-2003 by Jeff Woldan to replace original getContents method
 * Adds an additional parameter for the security settings with which
 * the clipboard object is being accessed.
 * original getContents method renamed to internalGetContents.
 * @see org.eclipse.swt.dnd.Clipboard
```

```
   * #internalGetContents(org.eclipse.swt.dnd.Transfer)
   */
  public Object getContents(Transfer transfer, String security) {

    // call the original getContents method to get system clipboard
    // contents.
    Object systemData = internalGetContents(transfer);

    // create permission object to be checked
    SecureClipboardPermission scPermission =
      new SecureClipboardPermission(security);

    try {
      // check permission, and if no exception is thrown,
      // check the system clipboard contents.
      AccessController.checkPermission(scPermission);

      // if the system clipboard contents are null,
      // return the data on the local secure clipboard.
      if(systemData == null) {
        // match type requested with type stored on clipboard.
        for (int i = 0; i < saveTransferAgents.length; i++) {
          if (transfer.getClass()
                == saveTransferAgents[i].getClass()) {
                return saveData[i];
          }
        }
      // if the system clipboard contents are not null, return this data.
      } else {
        return systemData;
      }
    }
    catch (AccessControlException e) {
      // insecure data request. do nothing, continue on...
    }

    // if an exception is thrown, return the system clipboard contents.
    return systemData;
  }


  /* Added 4-9-2003 by Jeff Woldan to replace original setContents method
   * Adds an additional parameter for the security settings with which
   * the clipboard object is being accessed.
   * original setContents method renamed to internalSetContents.
   * @see org.eclipse.swt.dnd.Clipboard#
   * internalSetContents(java.lang.Object[],
   * org.eclipse.swt.dnd.Transfer[])
   */
  public void setContents(Object[] data, Transfer[] dataTypes, String
  security) {

    // this "if block" copied from
    // internalSetContents(Object[] data, Transfer[] dataTypes) method.
    if (data == null
      || dataTypes == null
```

```
    || data.length != dataTypes.length) {
    DND.error(SWT.ERROR_INVALID_ARGUMENT);
  }

  // save data and dataTypes to "local clipboard"
  saveData = data;
  saveTransferAgents = dataTypes;

  // create permission object to be checked
  SecureClipboardPermission scPermission =
    new SecureClipboardPermission(security);

  try {
    // check permission, and if no exception is thrown,
    // pass a null string to the system clipboard.
    AccessController.checkPermission(scPermission);
    internalSetContents(
      new Object[] { null },
      new Transfer[] { TextTransfer.getInstance()});
  }
  catch (AccessControlException e) {
    //if an exception is thrown, send data to the system clipboard.
    internalSetContents(data, dataTypes);
  }

}
```

To preserve the original functionality of the clipboard object, I also replaced the original getContents(Transfer transfer) and setContents(Object[] data, Transfer[] dataTypes) methods. These methods would then call the getContents and setContents methods with the extra Security.STANDARDINSECURE parameter added on. By doing this, any classes accessing these methods without a security parameter would be classified as insecure access, thus giving the same functionality as originally expected from the clipboard before my modifications. In other words, this ensures backwards compatibility for code not making use of these security features.

org.eclipse.swt.custom.StyledText Modifications

Several classes needed to be modified to work with the security features of Clipboard class. The most important of these was the org.eclipse.swt.custom.StyledText class. As previously discussed, this class provides the text widget for Eclipse editors, and directly interfaces with the clipboard class for its cut, copy, and paste functionality. I decided to use the StyledText object to store the security setting for a given editor, which it would pass to the Clipboard object every time it called getContents or setContents. To do this, I added a private String field, mySecurity, to the StyledText object. To set this mySecurity field, I modified the StyledText constructor to receive an extra String parameter as its security setting, and set mySecurity to this String in the constructor.

To preserve the original functionality of StyledText, I replaced the original StyledText constructor. This constructor calls the modified StyledText constructor, with the default Security.STANDARDINSECURE as its security parameter. Much like with the Clipboard object, my intention was to ensure backwards compatibility with Eclipse classes unaware of the StyledText object's security features. I also added a completely new method to StyledText, the setSecurity(String security) method. This method allows the mySecurity field of the StyledText object to be set after creating the object. I originally included this feature for testing and demonstration purposes, and intended to remove it before my final solution, because I saw it as a security hole to allow the security setting of an editor to be changed.

After researching the code that creates and accesses the StyledText object, however, I realized that this setSecurity method would be more important than I thought. The problem is that in AbstractTextEditor, the default editor class, ISourceViewer object

that wraps the StyledText object is declared before any file is loaded into the editor. This means it is impossible to determine the security setting the StyledText object should have when it is created. In fact, not only was setSecurity essential to my modifications, I also added a getSecurity() method so that the security setting of the object could be returned for other security features.

org.eclipse.jface.text.TextViewer Modifications

The next class I needed to modify was the org.eclipse.jface.text.TextViewer class. TextViewer provides the implementation for the org.eclipse.jface.text.ITextViewer interface. ITextViewer is extended by org.eclipse.jface.text.source.ISourceViewer, and TextViewer is extended by org.eclipse.jface.text.source.SourceViewer, SourceViewer being the implementation of ISourceViewer. This is important because an ISourceViewer object serves as a wrapper class for the internal implementation of the AbstractTextEditor class. Included as part of TextViewer, and by extension SourceViewer, is the StyledText object for the editor. Since the StyledText object originates in TextViewer, I decided it made sense to modify TextViewer and ITextViewer, which would indirectly add this functionality to SourceViewer and ISourceViewer, as opposed to directly modifying those classes. Specifically, I needed to add methods to pass security settings to the StyledText object.

As with the StyledText object, I initially thought that I would pass the security settings in the object constructor, and added two constructors that added an additional security parameter to the original constructors. After realizing this would not work, I added the setSecurity(String security) and getSecurity() methods. The setSecurity and

getSecurity methods simply call the corresponding method for the StyledText object. I did not remove the constructors I added, however, because they are in no way harmful to the functionality of the code, and might be useful for additional security features.

org.eclipse.ui.texteditor.AbstractTextEditor Modifications

To complete the clipboard security feature, I needed to modify org.eclipse.ui.texteditor.AbstractTextEditor Modification to retrieve the security setting for a file from an ISecurityRegistry object, and pass that setting to ISourceViewer, and indirectly, its StyledText object. After examining the class, I found that the initializeSourceViewer(IEditorInput input) method takes an IEditorInput parameter and initializes the ISourceViewer with the document content. This seemed like the perfect place to add the initialization of security settings. The security settings are obtained by calling RegistryManager.getRegistry() to obtain an ISecurityRegistry object, and then calling ISecurityRegistry.getSecurity on the IEditorInput object. The results of this are then passed to the ISourceViewer.setSecurity method. The code I added is as follows, input is the IEditorInput object, fSourceViewer is the ISourceViewer object:

```
/* Added 4-18-2003 by Jeff Woldan
 * gets an instance of ISecurityRegistry, retrieves the
 * security setting for input, and passes that security
 * setting to fSourceViewer.
 */
ISecurityRegistry registry = RegistryManager.getRegistry();
fSourceViewer.setSecurity(registry.getSecurity(input));
```

Since all of Eclipse's built in editors extend AbstractTextEditor, this modification had the effect of incorporating this security feature into all of Eclipse's editors, including

the standard text editor and java code editor. This security modification also worked with a 3rd party XML editor I had added to the program. While the source code for that editor was unavailable, it apparently still extends AbstractTextEditor, and thus its functionality was modified as well.

Save As... Revisited

After completing my modifications to the Clipboard, I returned to my Save As... modifications to improve upon what I had done first semester. With my security framework in place, and the security setting of an editor being stored in the StyledText object, I planned to modify the isSaveAsAllowed() method of AbstractTextEditor to return a value based on the security setting of the editor. However, as I examined some of the classes that extend AbstractTextEditor, I realized a problem with this modification. The org.eclipse.ui.texteditor.StatusTextEditor class extends AbstractTextEditor, and expects isSaveAsAllowed to always return false. I could find no real documentation on this class, and so I was unsure of its use, but I didn't want to cause unexpected side effects by changing the functionality of a class I didn't understand. Instead, I decided to implement my solutions in the classes of specific editors.

This was not a problem for the two key editors in Eclipse, the standard text editor and the java editor. The text editor is implemented by the org.eclipse.ui.editors.text.TextEditor class, and the java editor is implemented by the org.eclipse.jdt.internal.ui.javaeditor.CompilationUnitEditor class. Both define isSaveAsAllowed to be true by default, but it proved fairly simple to modify these methods. In these methods, the ISourceViewer.getSecurity() method is called to get the

security String for the object, and then AccessController.checkPermission method is called on a SecureSaveAsPermission object, constructed with the security String. If no exception is raised, isSaveAsAllowed() returns false, but if an AccessControlException is raised, it returns true. This way, Save As... is disabled only for secure files, and insecure files have full Save As... functionality. The code for the changes to these two files is essentially the same, and is as follows:

```
/*
 * Modified 4-23-2003 by Jeff Woldan
 * to only allow save as if the document is insecure.
 *
 * @see IEditorPart#isSaveAsAllowed()
 */
public boolean isSaveAsAllowed() {

  //retrieve the ISourceViewer Object.
  ISourceViewer sourceViewer = getSourceViewer();

  /*
   * create a SecureSaveAsPermission Object, retrieving
   * the security parameter from the ISourceViewer object.
   */
  SecureSaveAsPermission ssaPermission =
    new SecureSaveAsPermission(sourceViewer.getSecurity());

  try {
    //Check permission, if no exception is raised, return false;
    AccessController.checkPermission(ssaPermission);
    return false;
  }
  catch (AccessControlException e) {
    //do nothing, continue on...
  }

  //If an exception is raised, return true.
  return true;
}
```

Incorporating Code Modifications into Eclipse

After modifying all of this code, it had to be added to the Eclipse application. After my work modifying Save As... functionality the first semester, most of the additions were easy to make. For most of the packages I had modified, it simply involved recompiling the package, exporting a .jar file for the package, and replacing the old .jar file in the Eclipse plugins directory with the new .jar file. There were a few complications, however. The first of which was that some of these packages required certain resource files to be included in the package, to provide dialog and labels. These files were not included with the source code, so I had to go through the old .jar files and copy the resource files into the new package. The worst of these was the org.eclipse.jdt.ui package, which included 42 such files.

All but one of my modifications were part of an existing package. For org.eclipse.security, the one package that was not, I had to create its own plugin directory and plugin manifest file. I followed the form of other plugins, and created a directory named org.eclipse.security_2.1.0 in the plugins directory. In addition to the .jar file and the file registry I created, two other files are included in this directory. The plugin.xml file is the plugin manifest; this file makes the plugin available to Eclipse. It includes the plugin name, the name of the .jar file where the plugin code is located, the package prefixes for the plugin, and a list of other plugins it is dependent on. I based my plugin manifest file off other plugin manifests that seemed similar. The contents of the file are shown below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<plugin id="org.eclipse.security" name="%pluginName" version="2.1.0"
 provider-name="%providerName">
 <runtime>
```

```
    <library name="security.jar">
      <export name="*" />
      <packages prefixes="org.eclipse.security" />
    </library>
  </runtime>
  <requires>
    <import plugin="org.eclipse.core.resources" />
    <import plugin="org.eclipse.core.runtime" />
    <import plugin="org.eclipse.ui.workbench" />
    <import plugin="org.eclipse.swt" />
  </requires>
</plugin>
```

In addition to the plugin manifest, I created a plugin.properties file. This file lists values of variables in the plugin manifest file. It is standard to use the %pluginName and %providerName variables in the plugin manifest, and the plugin.properties file simply defines these variables. Some plugin.properties files are much more complicated, but this was all that was required for this plugin. The contents of the file are shown below:

```
pluginName= Eclipse Security Support
providerName= Jeff Woldan
```

I also had to modify the plugin manifest file for the org.eclipse.ui.workbench.texteditor package, which includes the AbstractTextEditor class. My modifications had made it dependent on the org.eclipse.security package, so I added an <import plugin="org.eclipse.security" /> declaration to the file in the <requires> block.

After adding the modifications to Eclipse, I had to enable the security features I had implemented. This involved modifying the code for the Eclipse executable, to load the security manager and the eclipse security policy file. The source code for the executable was provided, so changing this was just a matter of finding the code where the JVM is called and modifying it. After searching through the source code, I found an

array called reqVMarg that declared to list the arguments passed to the virtual machine. I followed the format of the other parameters, where a literal string was preceded immediately by an L, and modified the declaration as follows:

```
static wchar_t*  reqVMarg[] = { L"-cp", startupJarName, L"-
Djava.security.manager", L"-Djava.security.policy=eclipse.policy",
L"org.eclipse.core.launcher.Main", NULL };
```

By adding the -Djava.security.manager and -Djava.security.policy=eclipse.policy parameters, the JVM would then be called with these security features loaded. At this point, I had decided to name the policy file for Eclipse eclipse.policy, and place it in the directory with the Eclipse executable, where no path would be needed to specify its location.

Next, I had to sign files and assign permissions. First, I generated a key named eclipseFiles in the eclipsestore keystore, using *keytool*. I set the key and keystore password to "honors." Obviously, in a real security solution, a more complicated password would be chosen, and would not be shared with users. With my key created, I signed every .jar file that comprises Eclipse with the eclipseFiles key, using *jarsigner*. It was probably not necessary that every file be signed, because some file need no special permissions to run, but this seemed easier than determining which ones needed a signature. Finally, I modified the eclipse.policy file using *policytool*, to add the permissions needed to run Eclipse with security features enabled. While I was originally concerned I would have to add an exorbitant number of permissions to allow Eclipse to function normally, this proved to not be the case. The default security manager runs code

under the "sandbox" model, which basically allows the code to do whatever it wants as long as it does not affect other parts of the environment. This meant that permissions I had to grant were those allowing access to external resources, such as the SWT .dll containing its native calls, and also access to files to be edited. While I used *policytool* to modify the file, it is a standard text file that can be read and modified with a text editor. The complete file contents are shown below:

```
/* AUTOMATICALLY GENERATED ON Wed Apr 23 22:26:49 EDT 2003*/
/* DO NOT EDIT */

keystore "file://C:/temp/Eclipse/eclipsestore";

grant signedBy "eclipseFiles" {
  permission java.lang.RuntimePermission "loadLibrary.swt-win32-2128",
    signedBy "eclipseFiles";
  permission org.eclipse.swt.security.SecureClipboardPermission
    "StandardSecure", signedBy "eclipseFiles";
  permission java.io.FilePermission "C:\\temp\\Eclipse\\workspace\\-",
    "read, write, delete, execute";
  permission java.lang.RuntimePermission "getProtectionDomain", signedBy
    "eclipseFiles";
  permission java.io.FilePermission "C:\\temp\\Eclipse\\plugins\\-",
    "execute", signedBy "eclipseFiles";
  permission java.lang.RuntimePermission "createClassLoader", signedBy
    "eclipseFiles";
  permission java.lang.RuntimePermission "*", signedBy "eclipseFiles";
  permission java.io.FilePermission "c:\\temp\\Eclipse\\eclipse.exe",
    "execute";
  permission java.io.FilePermission "<<ALL FILES>>", "read", signedBy
    "eclipseFiles";
  permission java.util.PropertyPermission "*", "read, write", signedBy
    "eclipseFiles";
  permission java.io.FilePermission "C:\\temp\\Eclipse\\workspace",
    "write", signedBy "eclipseFiles";
  permission org.eclipse.swt.security.SecureSaveAsPermission
    "StandardSecure", signedBy "eclipseFiles";
};
```

After completing modifications to the eclipse.policy file, Eclipse was ready to run with my security features enabled.

Contributing to the Eclipse Project

One of my goals for this project was to contribute to the Eclipse project in some way. In the first semester of the project I did this by submitting a bug fix. The fix I ended up working on was not related to security issues, but it was valuable just making a contribution to the Eclipse project. In doing so, I hoped to learn more about working within the open source community, and perhaps gain respect from the Eclipse community. By gaining such respect, the hope was that any security-related modifications I suggested might be taken more seriously. The first step in fixing a bug in Eclipse was to choose a bug. This required searching the Eclipse Bugzilla database. To narrow my search, I decided to focus on those bugs marked as "help wanted," indicating a request for help from the open source community. This seemed like a good place to start, and gave me a more manageable list of bugs to look over. I set aside several possible bugs that sounded both interesting and doable, and finally narrowed it down to one. The description for this bug can be found at the following site http://dev.eclipse.org/bugs/show_bug.cgi?id=2273. The basic idea behind this bug is that an external editor can be registered with Eclipse for editing certain file types using a plugin manifest file. Unfortunately, as currently designed, Eclipse simply calls the external editor using the command specified in the plugin manifest, and tacks the name of the file to be opened to the end of the command. This could create problems for editors where, for example, option flags must follow the file name. Currently there is no support for this. The idea suggested by this bug report is to use %f as a file name placeholder in the command specified in the plugin manifest. This would allow the file name to be placed anywhere in the command.

Once I had decided on this bug, I began research to find the code that would need to be modified. I started by looking for the plugin parsing code, so I could find how the plugin information is stored. I posted a message on the Eclipse newsgroup regarding this, but received no reply. I also did some research on XML to better understand the plugin manifest file, and to learn if the manifest template could be modified to better accommodate the change in command format. After learning more about XML and how DTD descriptions work, I decided against modifying the plugin structure. After more research, I determined that Eclipse uses the Xerces XML parser, which is part of the Apache open source project. Since it used this type of generic parser, it didn't seem to make sense to modify this either. At this point I tried to focus on finding the part of the code where the plugin information was parsed and used in the program. I ended up finding this by simply searching for files with the words external editor in them, and coming across org.eclipse.ui.internal.misc.ExternalEditor. This turned out to be the class where the method to load a file in the external editor is found, and would be part of the code I would modify. After researching the data types used in that class, I determined that org.eclipse.ui.internal.registry.EditorDescriptor is the class that defines the data type where the editor information is stored, and that EditorDescriptor.getFileName() returns the command to load the editor, which is stored in a string. I decided that %f would work well as a placeholder, because it was both intuitive and would not otherwise be used in an editor command. I needed to either modify this method, create another method to handle the %f placeholder, or simply handle the results of getFileName() differently.

My initial thoughts on solving this problem were to simply modify the EditorDescriptor.getFileName() method to return the command with the file name

inserted in the appropriate place. Thus all changes could be internal to EditorDescriptor, and the ExternalEditor class would not need to be modified. The immediate problem with this idea was that getFileName() takes no parameters, and there would be no way replace the %f placeholder with the actual name of the file without other changes. I could have simply modified ExternalEditor to replace the %f in the string returned by getFileName(), but I needed to create a uniform and consistent solution, so the fix would be of the quality that could be incorporated into the Eclipse code base. Thus while it would be a quick fix for the problem, it might create other problems, for example, if getFileName() were called elsewhere. It also meant that getFileName() would return a currently unexecutable command when it contained the %f placeholder, and this just seemed like bad coding practice. I decided instead that the file placeholder should be handled internally in EditorDescriptor. This meant that the file name would need to be passed as an argument to the method that would return the appropriate command, with the file name inserted.

I decided the best way to handle this was to create a new method, EditorDescriptor.getCommand(String aFile). I could have overloaded getFileName() to work with a file name parameter, but since the functionality differed I decided it made more sense to create another separate method. This method takes the name of the file to be opened and returns the command to open that file. I designed it to handle both commands containing the placeholder and those not containing it, to make it backwards compatible. The method works as follows. It searches the command string (stored in EditorDescriptor) for a %f placeholder, and if found, it replaces it with the actual file name. It will actually replace all the occurrences of %f with the file name, if for some

reason the file needed to be mentioned more than once in the command. It will not support opening multiple files with one call, but this was never supported by Eclipse, so it is not a problem. If %f is not found in the string, the file name is simply appended to the end of the string. The result is returned, and then can be called as-is. I modified EditorDescriptor.getFileName() as well. For tradition commands (with no %f placeholder), it works the same as always. For commands with a %f placeholder, getFileName() returns the first word in the string, which should be the name of the program without any placeholders or other options. Thus if getFileName() is called on an external editor that is accessed by a command containing the %f placeholder, it will still return a usable file name. It might be possible that this would not work in some cases, but I believe it is the best solution. To account for a more complicated command where the first word in the string is not the editor file name, there would need other information listing the file name separate from the command to call it. This information is not provided in the plugin manifest, so it is not really an option. I originally thought of having getFileName() return the command up to the %f placeholder, to allow any flags appearing before the placeholder to be included in the command, but I decided against that, because it did not see why the flags before the placeholder should be included and not those after the placeholder. On the other hand, it did not make sense to include everything but the placeholder, because this might place flags in places that would not make sense when called. I took considerable time choosing the way I wanted to implement this feature, and I believe I picked the best compromise between a hack fix and rewriting excessive amounts of the Eclipse code base, neither of which would be acceptable to the Eclipse project.

After modifying EditorDescriptor, I had to modify ExternalEditor to use the new getCommand method. This ended up being somewhat more complicated than I expected at first, because ExternalEditor uses the results of the getFileName() method to retrieve the absolute path of the editor, which it then uses to call the program. To solve this, I took the absolute path, removed the last occurrence of the file name of the program to open, and appended the results of getCommand(path), where path is the file name, to the path. I made sure it only removed the last occurrence, because the file name could also appear as a directory, earlier in the path. This is somewhat awkward, but is the best option without excessive modifications to other code.

I was able to submit my bug fix by creating an attachment to the bug report (http://dev.eclipse.org/bugs/show_bug.cgi?id=2273). This automatically sends an email to the bug reporter notifying him or her of the attachment. I simply included the two files I modified, and gave instructions as how to apply them. I received a response to my file submission fairly quickly. Unfortunately, it turned out I was working with a somewhat older version of Eclipse. I had version 2.0.1, which I had downloaded at the beginning of the semester. The newest version, 2.1.0, actually had one major difference. The ShellCommand class, part of SWT, which was previously used to invoke a command line call. Apparently if it threw an exception, it was not caught in time, and thus was replaced by the Runtime.getRuntime().exec(String []) method that is built into Java. I had to modify my fix to work with this newer version of the code, and resubmit it. When another user working on the project reviewed it, the comment I received concerned the use of %f as a file placeholder. This user was concerned that %f would not be a valid flag in some operating systems. I was unsure as to whether this was the case or not, so I

did not respond to this comment. I later received notice through Bugzilla that this bug was not a high concern of the Eclipse UI team, and would not be addressed until more pressing bugs were taken care of. This was fine with me, because I wanted to focus on security modifications in the second semester of the project.

After completing my security modifications, I planned to submit my work to those in the Eclipse community who might be interested. I did this by posting an email to the general development mailing list of the Eclipse project, eclipse-dev@eclipse.org. In this email, I gave a brief description of the work I have done, and said that anyone who was interested could contact me to get copies of the files and documentation for the project. I have no expectation that my work would be incorporated as part of the Eclipse code base, but I figured there might be individuals who would want to see what I had done as a possible way of implementing these types of features. My most ambitious objective would be that someone would see my work as a start of a more comprehensive security framework for Eclipse.

Conclusions

The security features and security framework I implemented this year are only a start to the features needed for a fully secure IDE. However, they are significant in their integration within exist Eclipse and the model they provide for other security features. Neither solution is perfect, but given the resources and time I had to do this project, I believe they represent an impressive accomplishment.

While my initial solution to Save As... was quite restrictive, the work I did for that modification provided a strong background for my work in the second semester.

When revisited, I improved upon the solution greatly, simply using the security framework I had created. If I were to improve upon this security feature, the next step would be to identify secure file locations and allow secure files Save As... functionality to those locations. This would probably involve modifying the Save As... dialogs, which is not a part of the code that I dealt with.

While the security framework I implemented is not extremely complicated, it provided the functionality needed for the security features I implemented, and could be easily expanded to add greater functionality and support other security features. To add support for other features, the most obvious step would be adding new security permissions, which could be checked by those features. It should be fairly simple to incorporate interaction with another type of security registry, using the ISecurityRegistry interface. However, the most important improvement to the security framework would be to incorporate it into the lower level file-loading features of Eclipse. For example, by the time an editor receives an IEditorInput object, the file has most likely already been opened by Eclipse. Depending on the implementation of a secure filespace/registry, it might be necessary to retrieve security settings when the file is opened. Implementing this lower level functionality would be the next step in improving the security framework.

My modifications to Eclipse's interface with the clipboard are my most comprehensive. It takes its flexibility a step farther than my Save As... modification, by allowing secure files to use the clipboard in secure ways, instead of completely disabling this functionality. Assuming there are no other secure applications in the development environment, this is the ideal solution to this security hole. However, in a real secure development environment, there would be other secure applications. The next step in

improving the clipboard security features would be allowing pasting between applications for secure files. This, however, would most likely require operating system level functionality for passing security settings, and for establishing "trust" between applications. This is beyond the scope of the work I've done, but would be an interesting extension to the project.

Considering the magnitude of the task at hand, the progress I made on developing a secure IDE was considerable. While these features only comprise a few pieces in a fairly large puzzle, they are each formidable accomplishments on their own. In addition, I valued the opportunity to contribute to the Eclipse project and learn more about working on an open source project. Overall, the work I have done supports the idea of using Eclipse as a secure IDE and the viability of such security features.

General Conclusions

The problem of security in a development environment is not an easy one. In different situations, the security needs of a company may vary greatly. Not only that, but certain resources, whether they are physical or software safeguards, may not be available or affordable to all companies. Thus in addition to the conflict between security and convenience to users, there is also the conflict between security and cost. When developing security related software, there are similar concerns. The most foolproof solutions are often the most restrictive to users. By simply disabling any feature that might create a security hole, there can be no security hole in that feature. This, however, could be quite restrictive to productivity. The most user friendly solutions that still provide maximum security modify features so that they can be used in any secure way, but no insecure ways. Ensuring that this occurs can be a daunting task for a software developer, because these solutions must account for all possible insecurities while still enabling all other uses. Sometimes however, these types of solutions are not an option, because the time or resources required may not be available.

There are other modifications that may allow some functionality of features while preventing that feature from being used in all insecure ways as well as some secure ways. As seen in the security features outline, often these solutions are easier to implement than those that allow maximum secure functionality. Another option is to plug the largest or most problematic security holes, and leave smaller ones open, considering them a reasonable risk. In some cases these solutions may be the best for the situation at hand.

In working with Eclipse, I have outlined many options for securing it. While any of these solutions might be acceptable for a particular development environment, many would not be satisfactory for a commercially viable IDE. We've already stated that Eclipse, no matter how well secured, is not a comprehensive security solution on its own. If it were to be packaged as part of a system-wide security solution, however, the security features we would need to implement would be those that offer the best security while still being as user-friendly as possible. These features should be highly configurable as well, so the security could be tailored to a site's individual needs. In doing this, a secure development product could be created.

I believe Eclipse holds considerable promise as part of a secure development environment. The extensible, open source nature of Eclipse makes it the ideal platform for implementing a secure development solution at low cost. Its extensible nature also shows promise for implementing secure interactions with other parts of the development environment. Eclipse's many features and the ability to add new features as plugins indicates it is robust enough for use as the sole IDE in a secure development environment. While it would be considerable work to modify, any security solution is complicated, and using Eclipse would be much easier than creating an entirely new IDE. The work I have done provides a strong contribution to the features needed to create such a solution.

Appendix

Offline Resources

Included with this report is a CDR containing a modified copy of Eclipse, along with the complete source code for Eclipse. Instructions for installing and running Eclipse, and links to copies of the modified files can be found in the README.htm file.

Online Resources

http://webstu.messiah.edu/~jw1197/honorsprojects1
Webpage for the first semester of the project

http://webstu.messiah.edu/~jw1197/honorsprojects2
Webpage for the second semester of the project

The existence of these pages is subject to the existence of my Messiah College webspace, which could be deleted at any time after graduation. No replacement location for this information has yet been established.

Project Sources

- http://www.eclipse.org - The Eclipse project home page.
    - http://www.eclipse.org/documentation/html/plugins/org.eclipse.platform.doc.isv/doc/guide/swt.htm - Summary information on SWT.
    - http://www.eclipse.org/articles/StyledText%202/article2.html - Discussion of SWT's implementation of native system calls.

- http://www.eclipse.org/articles/StyledText%201/article1.html, http://www.eclipse.org/articles/StyledText%202/article2.html - Explanations of the functionality of the StyledText object.
- http://www.eclipse.org/documentation/html/plugins/org.eclipse.platform.doc.isv/doc/reference/api/ - Eclipse platform API.
- http://www.eclipse.org/articles/Article-API%20use/eclipse-api-usage-rules.html - How to use the Eclipse API.
- http://www.eclipse.org/articles/Understanding%20Layouts/Understanding%20Layouts.htm - An explanation of widget layouts in Eclipse.
- http://www.eclipse.org/eclipse/faq/eclipse-faq.html. - Eclipse project FAQ.

- http://dev.eclipse.org - Eclipse development resources.
  - http://dev.eclipse.org:8080/help/help.jsp - Eclipse program help files.
  - http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-swt-home/dev.html - SWT development resources.
  - http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-swt-home/faq.html - SWT FAQ.
  - http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-swt-home/snippits/snippet94.html - Example code accessing the clipboard through SWT.

- http://eclipsewiki.swiki.net/ - The Eclipse Wiki Wiki web.

- http://java.sun.com – Java home page
  - http://java.sun.com/docs/books/jni/ - Online text on Java Native Interface (JNI).
  - http://java.sun.com/j2se/1.4.1/docs/api/ - Java 1.4.1 API.
  - http://java.sun.com/j2se/1.4/docs/guide/security/index.html - Java security enhancements documentation.
  - http://java.sun.com/j2se/1.4/docs/guide/security/spec/security-spec.doc.html - Java platform security architecture.

- http://java.sun.com/j2se/1.4/docs/guide/security/permissions.html - Java permissions.
- http://java.sun.com/j2se/1.4/docs/guide/security/doprivileged.html - Java API for privileged blocks.
- http://developer.java.sun.com/developer/TechTips/1999/tt0414.html - Example code accessing the clipboard through AWT.
- http://java.sun.com/docs/books/tutorial/security1.2/TOC.html - Tutorials and examples of Java security.
- http://java.sun.com/docs/books/tutorial/reflect/index.html - Tutorial on Java reflection classes.

- http://www.javaworld.com – Articles and tutorials on Java.
  - http://www.javaworld.com/javaworld/javaqa/2001-05/02-qa-0511-factory.html - Discussion of factory methods.
  - http://www.javaworld.com/javaworld/javaqa/2000-12/01-qa-1208-abstract.html - An example of a factory method.