Interactive Sensitive Data Exposure Detection Through Static Analysis

Md Abu Obaida
Dept of Electrical and Computer Engineering
Texas A&M University
College Station, Texas
Email: tareqobaida@tamu.edu

Eric Nelson
Dept of Computer Science and Engineering
Texas A&M University
College Station, Texas
Email: ejn8411@tamu.edu

Rene Van Ee
Dept of Electrical and Computer Engineering
Texas A&M University
College Station, Texas
Email: ravanee@tamu.edu

***Abstract*—Data security has become an increasingly important topic as information stored and transmitted in electronic form has become the preferred method. In order to protect this information, developers need to follow, at minimum, some basic guidelines to secure this data from malicious attackers. However, security often takes a backseat while developing software and is either not implemented at all or is patched into the software at the end. Both are undesirable as it leave the software vulnerable to sensitive data leaks and lowers the overall quality of the application. These issues can be introduced by developers of any skill level. Existing static or dynamic analysis tool does not provide the functionality of interaction with user, which we believe can be an extremely valuable feature. We present Secure Sensitive Data (SSD) Eclipse IDE plug-in that can help bridge the gap in sensitive data leaks by aiding software developers interactively by pointing out issues in real-time and enforcing certain standards to protect sensitive data. Our SSD plug-in enforces five best practices and standards that should at minimum keep data encrypted and proper handling of sensitive data. By continuously involving and reminding the developers of the security implications, SSD helps to mitigate security flaws in future software applications.**

***Index Terms*—sensitive data, static analysis, interactive, Eclipse plug-in**

I. Introduction

Having data security vulnerabilities is not uncommon in programs written by most experienced programmers. This is not because that the programmer does not know about the secure programming practice, this is because sometimes he just forgets or skips unintentionally. This situation occurs because programmers have to deal with other pressures such as deadline, performance, security, requirement changes etc. A quote from Donald Knuth, the writer of TEX is relevant in this regard.

> "Here I did not remember to do everything I had intended when I actually got around to writing a particular part of the code. It was a simple error of omission, rather than commission. . . This seems to be one of my favorite mistakes: I often forget the most obvious things"[7]

This can be even worst for novice programmers who do not even have clear idea about securing sensitive data. Existing static and dynamic analysis tools work at the end of development period which requires re-analysis of whole program. It often causes much more analysis time and it's not a trivial task to fix all those security bugs.

Security features checking is often done by 3rd party security specialist companies. We believe only the programmer knows best about his program. It is much more easier for programmer to fix security bugs rather than 3rd party security specialist. It is much more convenient for programmer if he can get warning about potential security vulnerabilities at the development time. It provides the opportunity to detect the flaws at the earliest time and programmer can fix the issue much easily.

We present Secure Sensitive Data (SSD), which is an Eclipse IDE plug-in that provides interactive security suggestions to Java developers in real-time. This is accomplished by performing static analysis using the Eclipse API to traverse the application to search and check against the sensitive data vulnerabilities we chose to enforce. The five main checks we implemented involve forcing the developer to encrypt data using up-to-date encryption standards, promote proper handling of data by not allowing unnecessary copying of sensitive data, and proper transmission of data.

The SSD plug-in is an extension of the Application Security plug-in for Integrated Development Environment (ASIDE) [6]project currently being developed at the University of North Carolina at Charlotte. The ASIDE project is also an Eclipse IDE plug-in that performs static analysis. However, the SSD plug-in is concerned with sensitive data leakage, whereas ASIDE focuses mainly on vulnerabilities of incoming data, such as SQL injection and Cross Site Scripting.

A. Contribution

1) An interactive sensitive data exposure detection tool in the form of Eclipse plugin. The plugin can provide significant improvement in user experience incorporating static analysis to detect security vulnerability in incremental manner.
2) Concept of using Breadth First Search instead of Call Flow Graph
3) Simple technique to handle library calls in analysis.

4) Available as Eclipse plugin, easy to use.
5) User's freedom to run SSD in interactive mode or batch mode.

II. Related Work

Our project is based on work that interested us after reading about the ASIDE project at University of North Carolina at Charlotte. The ASIDE project is also an Eclipse IDE plugin that provides interactive analysis for computer security that keeps developers in the "security loop." The ASIDE plugin targets both the Java and PHP platforms. The ASIDE project is generally focused on protecting applications from incoming data and web application tainting attacks, such as unchecked access paths and cross site scripting.

Looking into the ASIDE project allowed us to become familiar with more work at the UNC Charlotte, particular the Open Web Application Security (OWASP) for web application security. The OWASP project's "Application Security Standard" (2014) [2] lists eight rules and standards to help ensure web applications promote security. Using this list, we generated our own list of five rules that we wished to implement in our SSD plugin.Another source of inspiration was "Finding Security Vulnerabilities in Java Applications withStatic Analysis" [8]. This paper helped understand the underlying principle of static analysis of Java applications in Eclipse. Even though the vulnerabilities and platform this paper targets don't coincide with our SSD plugin, the analysis of Java source code is similar to what SSD implements as far as static analysis using the Abstract Syntax Tree (AST) and using the related Java class to traverse the tree and search for specific items. The paper's analysis goes into detail of tainted object propagation analysis, which is of interest to our project as we also need to track the propagation of sensitive marked objects and how they are passed and modified to other objects and methods. Similar to our project, the purpose is to analysis the platform's source code and generate warnings to the developer in Eclipse.

III. SSD: Secure Sensitive Data

Secure Sensitive Data (SSD) is an interactive static analysis tool that we propose to remedy the problems as stated above. The tool will alert programmers during the development of Java servlets if their code has the potential to leak sensitive data. In the initial phase of this project, we gathered information about the problem. In our study, we found a list of rules to ensure sensitive data is not leaked in a web application by the Open Web Application Security Project (OWASP),[1] an organization that provides free articles on security and includes educational organizations. Because of the scope of the project and the time limits of the semester, we selected a subset of the most important from this list. The rules we selected to enforce are as follows:

1) Disabled client side caching
2) Sensitive data has to be encrypted
3) Garbage collection of expired sensitive data
4) No sensitive data passed as URL parameters
5) Sensitive data should not be stored unnecessarily

Given these rules, the tool will alert the user by providing icons and messages that explain the problems with the written code and notifications on how to fix them. The goal is both to improve the security of applications using the tool as well as to teach both experienced and novel programmers secure programming practices by interactively showing them the problems with the code they have just written.

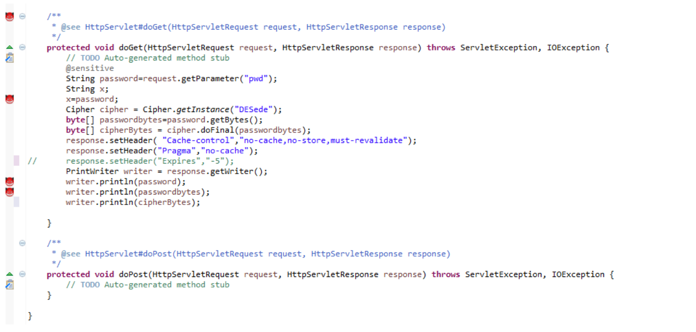
Figure-1 shows a sample of user editor while SSD is active. The first marker at the left side of editor is because the doGet method does not have proper cache setting while sensitive data is exposed. The line

```
response.setHeader("Expires","-5");
```

is commented out. The second marker warns about an unnecessary storing of sensitive data. Third and fourth markers warns about sensitive data exposure to outside world. As we can see the line

```
writer.println(cipherBytes);
```

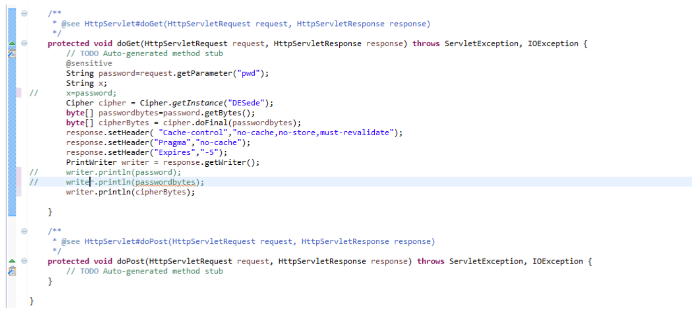
is not showing any warning, because the argument "cipherBytes" is encrypted properly.



```
    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // TODO Auto-generated method stub
        @sensitive
        String password=request.getParameter("pwd");
        String x;
        x=password;
        Cipher cipher = Cipher.getInstance("DESede");
        byte[] passwordbytes=password.getBytes();
        byte[] cipherBytes = cipher.doFinal(passwordbytes);
        response.setHeader( "Cache-control","no-cache,no-store,must-revalidate");
        response.setHeader("Pragma","no-cache");
//      response.setHeader("Expires","-5");
        PrintWriter writer = response.getWriter();
        writer.println(password);
        writer.println(passwordbytes);
        writer.println(cipherBytes);

    }

    /**
     * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
     */
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // TODO Auto-generated method stub
    }

}
```

Figure 1. SSD Warnings

Being warned by SSD, if user makes few modifications, warning markers will go away instantly as figure-2.



```
    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // TODO Auto-generated method stub
        @sensitive
        String password=request.getParameter("pwd");
        String x;
//      x=password;
        Cipher cipher = Cipher.getInstance("DESede");
        byte[] passwordbytes=password.getBytes();
        byte[] cipherBytes = cipher.doFinal(passwordbytes);
        response.setHeader( "Cache-control","no-cache,no-store,must-revalidate");
        response.setHeader("Pragma","no-cache");
        response.setHeader("Expires","-5");
        PrintWriter writer = response.getWriter();
//      writer.println(password);
//      writer.println(passwordbytes);
        writer.println(cipherBytes);

    }

    /**
     * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
     */
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // TODO Auto-generated method stub
    }

}
```

Figure 2. SSD warnings go away after modifications

IV. Background

This section will focus on giving the reader the background necessary to understand how the tool is implemented. There are two main ideas necessary to understand how the tool works internally. The first is the concept of an Abstract Syntax Tree (AST). Then next necessary concept is that of the Visitor pattern.

A. AST

An Abstract Syntax Tree (AST) is a method for representing the structure of a program in a tree-like form. In addition, an AST, as the name implies, is abstract. This means that the AST can represent the structure of a program instead of being tied directly to the underlying text. This structure allows us to find patterns and extract meaning from programs. To illustrate, consider the following example:

- How do we interpret the statement: int x = y + 15 + 20;
- First we build an AST which will look as follows:

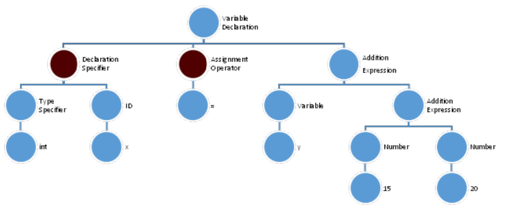
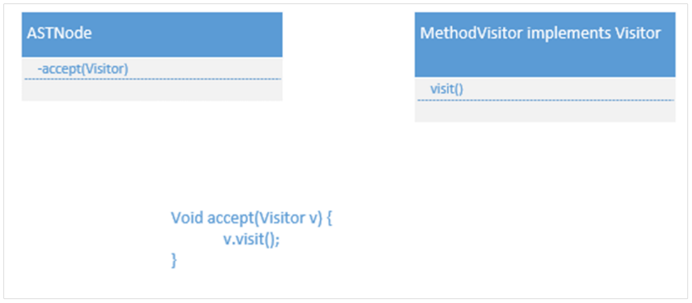


Figure 3. Sample AST

With this AST, we can now gain an understanding of what the statement means very easily. For instance, by looking at the root node we understand the statement is a variable declaration. In contrast, if we only looked at the text we would have to do a ton of manipulation to come to this understanding.

B. Visitor Pattern

The visitor design pattern is another important concept for our project. Since our project is implemented as an Eclipse plug-in, we are given the AST for the code that is written. Therefore, we need a way to perform some operations on the AST. This is where the visitor pattern enters the picture. Figure-4 is an example to illustrate:



```
Void accept(Visitor v) {
        v.visit();
}
```

Figure 4. Visitor Pattern

The first thing to notice is that each ASTNode has a function which takes a Visitor as a parameter. If we then declare a MethodVisitor class which implements the visitor interface, we can pass this to the ASTNode's accept function. Therefore, we write the code we want to run inside MethodVisitor's visit method and pass it to the ASTNode's accept method. If the ASTNode's type is a method then it can call the visit method of the MethodVisitor and the code we wrote in the visit method is run.

In this manner, we can run a specific function on each node and have different functions for different types of nodes. This allows us to perform some complex analysis on the underlying AST to locate potential sensitive data leakages.

V. Design

The design of SSD is based on Eclipse's OSGi R6 Framework specification [5]. SSD takes any Java servlet project as input and directly analyze source code without converting it to any intermediate representation. This enables SSD to maintain a real-time feedback with user interaction. SSD is able to detect potential sensitive data exposure in the project and checks if the sensitive data is properly protected or not. To identify sensitive data, the developer must tag the sensitive data with the @sensitive annotation. User must declare a variable as sensitive before performing any operation on it. The figure below shows a sample annotation.

```
@sensitive
String password=request.getParameter("pwd");
```

SSD takes advantages of Eclipse's abstract class ASTVisitor to visit all nodes containing the @sensitive annotation and marks them as sensitive data. Any other variable derived from sensitive will inherit the sensitive property, resulting in the same sensitive classification. SSD takes into consideration this inheritance and checks every possible way to derive secondary sensitive data from sensitive data and marks them sensitive as well. In doing so we consider the following scenarios:

1) Direct assignment
2) Indirect assignment
3) Method invocation having sensitive data as its argument. We consider the return variable if has any as sensitive as well if the method is a library call that is we don't have the source code. Otherwise we visit the body of the method to find the above two cases.

We believe we cover most of the cases of sensitive data flow by considering the above three steps checking without having full taint analysis. However, careful assigning of sensitive data can mitigate the risk of sensitive data flow.

One of the recommendations from OWASP [1] to prevent sensitive data exposure is not to store sensitive data unnecessarily. It was encouraged to discard it as soon as possible. SSD tries to detect unnecessary storing of sensitive data and creates warning for users. We consider direct assignment of sensitive variable to other variable is a violation of above mentioned practice. However, there can be situations where users may assign sensitive data for different purposes, our objective is to warn user to be careful when doing so.

We consider sending sensitive data without proper handling as sensitive data exposure. After careful consideration we found that checking the invocation of abstract class "java.io.Writer" is sufficient to detect data exposure to outside world. The "java.io.Writer" is the parent of all other classes responsible for writing output stream. It implies that any attempt to write output stream will be detected by our tool.

OWASP recommends disabling caching for pages that contain sensitive data[1]. In order to check if caching is disabled properly we check the following parameters in HTTP header settings.

- Cache-control. The value should be “no-cache,no-store,must-revalidate”
- Pragma.The value should be "no-cache"
- Expires. The value should be some unrealistic number, we consider any number <= 0 as good setting.

It is mandatory to encrypt all sensitive data before sending it. Current implementation of SSD checks if sensitive data is encrypted with Java’s built in “javax.crypto.Cipher”. However, there are few others libraries available for encryption. We plan to include all up to date encryption algorithms in future version of SSD.

Passing sensitive data as URL parameter is not encauraged at all. From Java servlet, URL is sent to client browser using “java.io.Writer” which is covered in the process of checking sensitive data exposure.

A. *Algorithm*

Figure-5 describes the algorithm of SSD. Whenever any AST Change event happens in Java editor, SSD triggers the analysis. Our analysis is based on AST traversing, listening to only AST change event enables to avoid triggering analysis when it is not required.

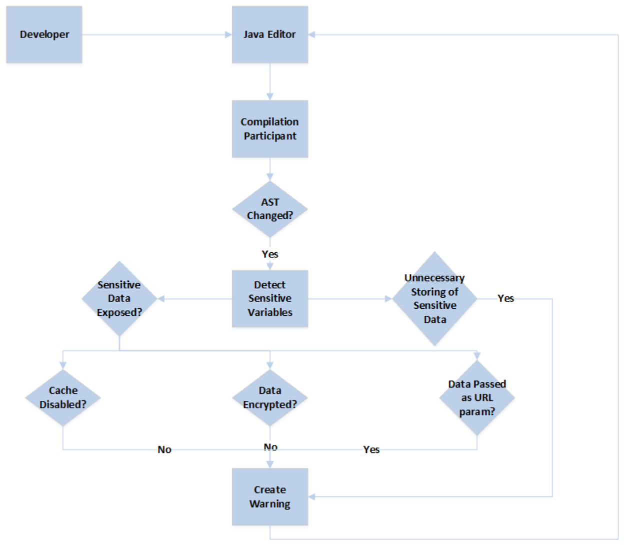


Figure 5. Flow Chart of SSD

SSD detects all the sensitive variables using user annotation. In next step it detects exposure of sensitive data and unnecessary storing of sensitive data. If it finds any unnecessary storing of sensitive data, it creates warnings and presents the warning marker in Java editor at appropriate location to the developer. Our primary objective is to detect sensitive data exposure. SSD traverses all related ASTs to detect potential exposures by searching “java.io.Writer”. If potential exposure found, SSD goes to next step to find the security settings, i,e; cache and encryption. Any deviation from recommended setting will result in creating warning marker in Java editor.

VI. Implementation

The Eclipse IDE allows the developer to extend the IDE functionality via plug-ins. Being an interactive tool, SSD takes the advantage of the extension points and API provided by JDT (Java Development Tooling) of Eclipse. JDT adds Java specific behavior to the generic platform resource model and contribute Java specific views, editors, and actions to the workbench. Using JDT API and extension points we can access Java editor in Eclipse and programmatically manipulate Java resources.

SSD comes in the form of Eclipse plug-in. Unlike most other interactive plug-ins it has the capability of starting and stopping the plug-in. This functionality becomes handy when the user is not dealing with any sensitive data and thus enables to avoid unnecessary overheads. SSD starts it’s analysis when user selects a project and click “Start SSD” menu. Being triggered by the click event it continues to analyze the code whenever any change is detected in the source code. The project information is collected from user selection and stored which is cleared when user clicks “Stop SSD” menu. Implementation of SSD is described in the following steps. We will use the example code from figure-6 to explain the implementation in rest of the document.

```
protected void doGet(HttpServletRequest
    request, HttpServletResponse response)
     throws ServletException, IOException
    {
@sensitive
String password=request.getParameter("pwd
    ");
String x;
x=password;
Cipher cipher = Cipher.getInstance("
    DESede");
byte[] passwordbytes=password.getBytes();
byte[] cipherBytes = cipher.doFinal(
    passwordbytes);
response.setHeader("Cache-control","no-
    cache,no-store,must-revalidate");
response.setHeader("Pragma","no-cache");
response.setHeader("Expires","-5");
PrintWriter writer = response.getWriter()
    ;
writer.println(password);
writer.println(passwordbytes);
writer.println(cipherBytes);
          }
```

Figure 6. Example Code

A. Sensitive Variable Detection

SSD get the Java project from user selection. A project consists of several packages. SSD extracts packages containing source files and converts each package to the object “ICompilationUnit”[3]. ICompilationUnit Represents an entire Java compilation unit which is a complete source file with Java extension. Each of the ICompilationUnit is parsed into Abstract Syntax Tree (AST). Visitor class visits ASTs and looks for the “VariableDeclarationFragment” AST node with sensitive annotation. Line 3 in the example code represents such a node. SSD marks each such variable as sensitive and stores in a set of sensitive variables that contains all other tagged sensitive variables.. Any other variable derived from sensitive variable is also sensitive and therefore is added to the sensitive list. An example derived sensitive property is shown on line-5 in the example code. This is an example of an assignment node. SSD marks “x” as sensitive and adds it to the set of sensitive variables. At the same time, it marks the node as “unnecessary storing of sensitive variable” because this is a direct assignment from sensitive variable to other variable. SSD also considers “InfixExpression” and “MethodInvocation” nodes to mark sensitive variables. For instance, “passwordbytes” in line no 7 from the example code is sensitive too because it’s derived from the sensitive variable “password”. Figure-xx is an example of “InfixExpression” expression where “derivedSensitive” is derived from an InfixExpression involving a sensitive variable “password”.

```
String derivedSensitive="this is"+
    password+"sensitive";
```

Figure 7. InfixExpression

B. Sensitive Data Exposure Detection

While visiting ASTs, SSD examines each “MethodInvocation” node to find if any method is member of the Java class “java.io.writer”. Lines 13 through 15 in the example code are examples of such method invocations. To do that SSD traverses backward to find the declaring class of the method and compare it with “java.io.writer”. If the method takes any sensitive variable as its argument, then SSD marks the node as a sensitive data exposure vulnerability. This analysis does not flag variables that have been verified as encrypted variables. Thus SSD provides a quick method of warning of sensitive data exposure to developer, allowing the developer to intervene and generate a solution to the vulnerability.

C. Is Sensitive Data Encrypted?

Encrypting sensitive data before sending is the most important measure to take to protect sensitive data. Current implementation of SSD examines the invocation of any method which is a member of the class “javax.crypto.Cipher”. In example code line no 8 is an example of encryption. Any sensitive data modified by such method is marked as encrypted, all others are marked unencrypted. Sending encrypted data does not create warning, all other instances create warnings. SSD analyzes the arguments and return type of such methods in doing so.

D. Checking Cache Settings

SSD not only provides data exposure but also detects if appropriate measures are not taken to protect sensitive data. One of the important measures to take is disabling client side caching. SSD checks cache settings by examining the method “setHeader”. Example of recommended settings is in line no 9,10,11 in example code. Each arguments of “setHeader” call is examined to get the settings of Cache. Any deviation of the recommended settings results in marker warning user about the pitfall.

E. Breadth First Search

Breadth fist search is used to traverse all the methods called from a doGet or doPost method. Calling of other methods come in the form of ClassInstanceCreation and MethodInvocation. Class instance creation implies calling the constructor of the class. BFS continues until each method reachable from is doGet or doPost method visited. The interesting part of using BFS is that, it can be used to make SSD more efficient while working interactively. SSD stores only the node information from a method. If a user makes changes inside a method, that particular method will be traversed only and node informations will be updated. SSD will use the stored informations for other methods thus can save a significant amount of analysis overhead. Currently full implementation of this concept is not available in SSD. For every change made by user SSD run analysis if it finds AST is changed.

F. User Interaction

Ability to interact with users makes SSD separate from all other static analysis tool for detecting sensitive data exposure. SSD takes advantage of Eclipes’s extension point “org.eclipse.core.resources.markers” to create markers in Java editor. It becomes visible instantly whenever SSD finds any issue with the code. User can take preventive measures and the result is reflected immediately in the editor. Additionally SSD creates entry in “Markers” view with useful informations which includes the possible reason for security vulnerability. Double clicking on the entry takes the user to the problematic line of code instantly.

G. Being Interactive

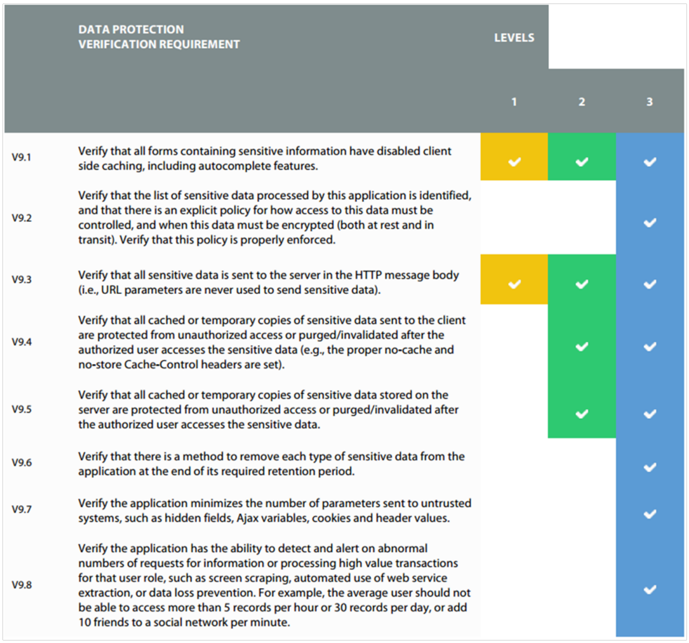
Eclipse provide an extension point compilationParticipant, which is used to participate in the compilation time. Auto build feature of Eclipse compile the program automatically in the background and user defined compilation participant is invoked in the compile time. Eclipse provide ReconcileContext from which SSD extracts the changed Java element. SSD runs the analysis if it finds any content changes or AST changes by examining changed Java element. Thus SSD will not trigger new analysis until any AST change event happens. Further researches can be carried out to make this process more optimized.

VII. Evaluation

We evaluated our tool on three different metrics: recommendation coverage, programmer improvement, and the overhead of the tool. The goal was to use these three metrics to identify if our tool is: correct, efficient, and usable.

A. *Recommendation Coverage*

Our initial idea was to analyze our tool and compare it with other similar types of tools which attempt to identify sensitive data leakage. However, we couldn't find a tool that would give an accurate comparison. By this we mean that most existing free tools don't attempt to detect vulnerabilities introduced by the programmer on the server side. Instead, these tools focus on ensuring that web application code is safe from common attacks on web applications such as XSS or SQL injection. Therefore, we instead focus on the coverage of recommendations from OWASP in its standard for web application security[2]. Specifically, we focus on section nine, which discusses the data protection requirements. A full listing of these requirements are as follows:



	DATA PROTECTION VERIFICATION REQUIREMENT	LEVELS		
		1	2	3
V9.1	Verify that all forms containing sensitive information have disabled client side caching, including autocomplete features.	✓	✓	✓
V9.2	Verify that the list of sensitive data processed by this application is identified, and that there is an explicit policy for how access to this data must be controlled, and when this data must be encrypted (both at rest and in transit). Verify that this policy is properly enforced.			✓
V9.3	Verify that all sensitive data is sent to the server in the HTTP message body (i.e., URL parameters are never used to send sensitive data).	✓	✓	✓
V9.4	Verify that all cached or temporary copies of sensitive data sent to the client are protected from unauthorized access or purged/invalidated after the authorized user accesses the sensitive data (e.g., the proper no-cache and no-store Cache-Control headers are set).		✓	✓
V9.5	Verify that all cached or temporary copies of sensitive data stored on the server are protected from unauthorized access or purged/invalidated after the authorized user accesses the sensitive data.		✓	✓
V9.6	Verify that there is a method to remove each type of sensitive data from the application at the end of its required retention period.			✓
V9.7	Verify the application minimizes the number of parameters sent to untrusted systems, such as hidden fields, Ajax variables, cookies and header values.			✓
V9.8	Verify the application has the ability to detect and alert on abnormal numbers of requests for information or processing high value transactions for that user role, such as screen scraping, automated use of web service extraction, or data loss prevention. For example, the average user should not be able to access more than 5 records per hour or 30 records per day, or add 10 friends to a social network per minute.			✓

Figure 8. Data Protection Verification Requirement

As discussed in the previous sections, there are five of these rules covered by SSD. Specifically, SSD covers rules V9.1 – V9.5. It can be seen that this qualifies the application that follows SSD's advice as a "Level 2" secure application. Level 2 is defined by the OWASP standard as follows:

> "An application achieves Level 2 (or Standard) verification if it also adequately defends against prevalent application security vulnerabilities whose existence poses moderate-to-serious risk."

In order to qualify for "Level 3 – Advanced" SSD would need to implement the additional 4 rules that were left for future work. The team has also discussed allowing the user to set a level of security that they would like their application to meet and configuring SSD to only produce warnings for the corresponding rules.

B. *Programmer Improvement*

The next area we focused on during the evaluation of our project was programmer improvement using SSD. Our goal for this portion of the evaluation was to identify if the tool was effectively teaching good security practices.

In order to evaluate this, we asked for the assistance of a group of 20 peers. Each peer was asked to write a simple Java servlet and to implement any security features that they thought should be in a production application. We then recorded which of the five rules they implemented correctly without assistance from SSD. The results are shown in figure-9.

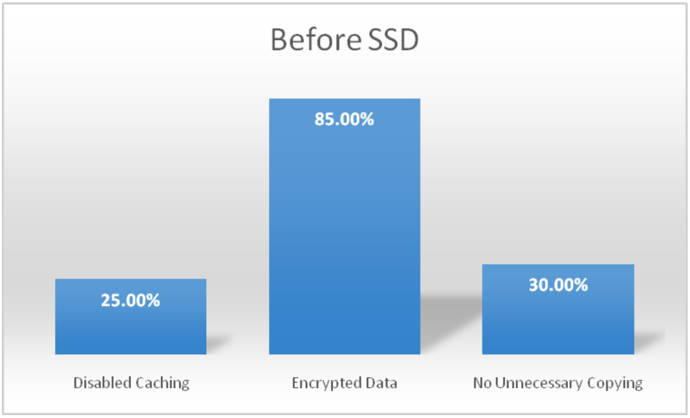


Figure 9. without assistance from SSD

The results of this test are very interesting. The first interesting point is that only five of the twenty peers knew how to disable caching. This number is surprisingly low and it can cause serious consequences if sensitive data is passed through the URL parameters. The next interesting point is that almost all of the peers knew to encrypt the data before sending it. The few that didn't might not have known how to encrypt the data or even worse ignored encrypting the sensitive data. Both sensitive data garbage collected and no sensitive data as URL parameter were given to every peer since Java has built in garbage collection and we asked the peers to simply write a response and not generate any requests to specific URLs. Finally, a surprisingly high number of peers made unnecessary copies of sensitive data.

Next, we asked the same peers after some time to write another simple Java servlet. This time we asked them to do so while SSD was running within Eclipse. The results of this test are shown if figure-10.

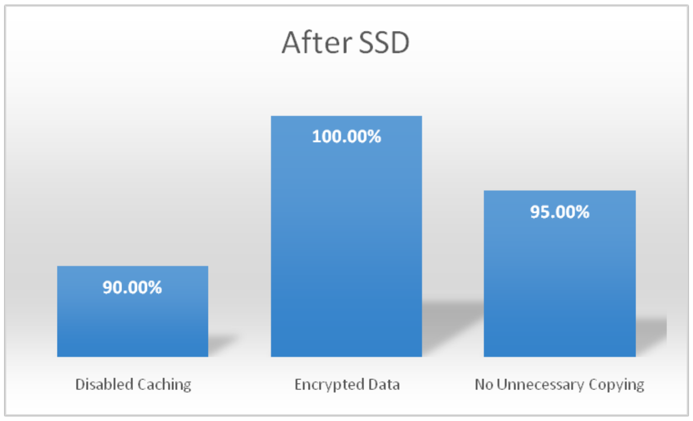
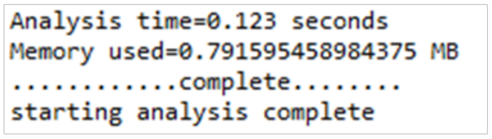


Figure 10. with assistance from SSD

This time almost every peer implemented each of the security rules. However, we still had a few peers who didn't implement caching. This means that the peer ignored the warnings provided by SSD. As we can see, this table shows that SSD effectively teaches programmers to write more secure code by providing visual notifications and explanations of how to fix problems.

C. Overhead

The final metric we used to evaluate the system is a measurement of the overhead for running SSD in real time. We were able to track both the time taken and the memory used for the plug-in. We found that there is actually very little overhead. The figure below shows the time and memory usage on a small project with SSD.



```
Analysis time=0.123 seconds
Memory used=0.791595458984375 MB
............complete........
starting analysis complete
```

Figure 11. Overhead Measurement

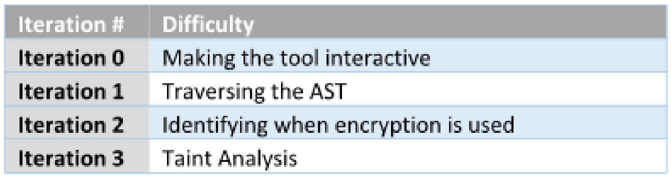
This is most likely due to the algorithm used to analyze a project. First, we are only analyzing the methods doGet and doPost recursively. Therefore, if there is a more complex function that is never invoked in any method inside of doGet or doPost, then we can simply ignore that method. Second, we are only worried about methods which actually write data back in a response. Therefore, if we find a method that doesn't write any data then we can ignore it.

Finally, we only ever need two passes of the program to complete our analysis. The first visit we gather the variables annotated with @sensitive. The next pass we actually look for security flaws. The algorithm we use for analyzing behaves linearly in both time and memory. This is because we are visiting only changed AST, while user keep developing his code. Therefore, we need at most a linear amount of both memory and time.

We concluded from the above analysis that our tool is effective because it was able to teach programmers good security practices. In addition, we concluded that our tool is correct because we cover the rules from OWASP that qualify an application that takes SSD's advice as a "Level 2" application. Finally, we conclude that our tool is usable because the overhead is relatively small and the feedback is almost instantaneous.

VIII. Difficulties Faced

This project presented many different challenges. First, we had to identify how we were going to implement the features that we proposed at the beginning of the semester. Through a series of iterations we were able to come up with a few algorithms that would allow us to identify potential security flaws. During each iteration, we identified a major roadblock to our goals. They are listed in figure-12.



Iteration #	Difficulty
Iteration 0	Making the tool interactive
Iteration 1	Traversing the AST
Iteration 2	Identifying when encryption is used
Iteration 3	Taint Analysis

Figure 12. Difficulties

Initially, we were concerned with finding a method to make the tool interactive. This was an important goal for the project because this would allow programmers to learn good security practices as they are writing the code. We solved this issue by studying existing solution to problems close to ours. We found the ASIDE project, which is implemented as an Eclipse plugin and decided to follow the same suit.

The next major roadblock we ran into was traversing the AST. In order to extract some information from the source, we have to build the AST first then traverse it in a BFS fashion. This allows us to build relationships between variables and how they are used within the web application.

Another major roadblock we faced was identifying when encryptions was used. We came to the conclusion that we had to limit the supported libraries for encryption as it is impossible to enumerate all encryption routines. Therefore, we decided to only support Javax.crypto for this semester and adding support for more libraries to future versions.

Finally, we had to determine how we would handle using taint analysis. If a sensitive variable is assigned to another variable it too should be considered as sensitive. In full taint analysis we would be able to determine these sensitive variables at any level of recursion or through method calls. However, we decided that for this semester we would only perform partial taint analysis in order to allow us to complete the project this semester.

IX. Limitations

- Current implementation considers "javax.crypto.Cipher" as the only class responsible for encryption which not necessarily true. There are several others libraries and

methods available. We plan to include all reliable encryption methods in the future work.
- We have implemented data flow analysis instead of full taint analysis. Deploying "points-to" [4]analysis can make SSD to be more precise and less false alarms generating. On the other hand points-to analysis will produce more time overhead which is can be a bottle neck for SSD to be scalable to large projects.
- All possible ways of data exposure is not covered. We consider only "java.io.Writer" class as responsible for sending data to client browser, but there could be other methods for sending data.
- Being a static analysis tool, SSD inherits limitations from static analysis. SSD has to cover all possible ways of program execution, which results in false positives. However, SSD does not change any source code and the user has full freedom to ignore warnings generated by SSD. We believe false negatives is more important than false positives in interactive tools. SSD emphasizes on detection of every possible data exposure instead of reducing false positives.

X. Future Work

While the SSD plug-in can prove to be a useful tool in combating sensitive data leakage, we believe that much more can be done to help developers become more aware of data vulnerabilities. To further the cause of the SSD project, future revisions will expand the plug-in to traverse the application more in-depth. This means that the plug-in should perform full taint analysis. By implementing full taint analysis, we would be able to better track how sensitive data is being passed throughout the application. As sensitive data variables are assigned to other variables or passed into functions, we can continually track this data and further variables will be able to inherit the sensitive annotation, allowing for fuller coverage of the application.

Currently, our plug-in in relies on the developer to mark sensitive data without our custom annotation (@sensitive). We understand that for larger projects and teams, keeping track of the annotation may place a burden on the developer. However, we believe that we can accurately infer sensitive data variables automatically and thus remove the need for the annotation entirely. Using pattern recognition and analyzing the general structure of applications that transmit and receive data (certain functions and library are always present), we believe that automatic sensitive data inference can be possible.

As mentioned before, one security measure we enforce is requiring the developer to use only current and up-to-date encryption standards. As of now, the plug-in only enforces using the javax.crypto library. We understand that this is a limitation of the plug-in as applications and developers can use many and multiple different valid and up-to-date encryption standards. For a future version, we would like to extend our list of valid encryption libraries to include more options such as Legion or The Bouncy Castle Java library. We would also like to implement a method to automatically pull the latest revisions of each encryption library from their respective websites.

As the SSD project grows, we believe that it can easily be extended to other platforms besides Java. Many applications deployed on different platforms, such as PHP and ASP, suffer from the same vulnerabilities that Java developers face. While keeping our project an Eclipse IDE plug-in, we can support multiple platforms using our generic underlying static analysis foundation and merely extend the vulnerabilities to the specifics of each platform.

XI. Conclusions

SSD has complete and functional implementations of all five tasks we originally set out to cover. While our project is still in the early stages before full deployment and expanding our analysis to provide more coverage, our working implementation is a stepping stone to larger things. The plug-in, at its current state, is roughly 2000 lines of code which we contributed. While using the plug-in is the only way to truly understand what we have accomplished, we believe our project is the first interactive eclipse plug-in that specifically covers detecting and mitigating sensitive data vulnerabilities. Some of the highlights of the SSD project are the fully interactive suggestions to the developer in real-time (or batch mode if preferred) and the ability to identify and fix sensitive data vulnerabilities before the deployment of the applications (saving the developer and company time and money).

Appendix A
Individual Contribution

- Md Abu Obaida: 36%
- Eric Nelson: 32%
- Rene Van Ee: 32%

Acknowlegment

We would like to thank Dr. Guofei Gu for his kind support and guidance.

References

[1] OWASP.Top 10 2013-A6-Sensitive Data Exposure, 23 June 2013. https://www.owasp.org/index.php/Top_10_2013-A6-Sensitive_Data_Exposure

[2] OWASP.Application Security Verification Standard (2014), 2014. https://www.owasp.org/images/5/58/OWASP_ASVS_Version_2.pdf

[3] Eclipse. Eclipse documentation - Current Release, 2014. http://help.eclipse.org/luna/index.jsp

[4] O. Lhot ak. Spark: A exible points-to analysis framework for Java. Master's thesis, School of Computer Science, McGill University, Montr eal, Qu ebec, Canada, Feb. 2003.

[5] OSGi. OSGi Alliance Specifications, 2014. http://www.osgi.org/Specifications/HomePage

[6] Jing Xie, Bill Chu, Heather Richter Lipford, and John T. Melton. Aside: Ide support for web application security. In Proceed- ings of the 27th Annual Computer Security Applications Conference, ACSAC '11, pages 267{276, New York, NY, USA, 2011. ACM. ISBN 978-1- 4503-0672-0. doi: 10.1145/2076732.2076770. URL http://doi.acm.org/10.1145/2076732.2076770.

[7] D. E. Knuth. The errors of tex. Softw. Pract. Exper., 19:607–685, July 1989.

[8] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14,SSYM'05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association. http://dl.acm.org/citation.cfm?id=1251398.1251416.