A Survey of Three Proposed Techniques to Increase the Security of Java

COMPSCI725 Term Paper 2003
Qing-Shan Martin Lun

Abstract

The Java platform was designed from the outset to implement security as a default feature. However, the security of Java is lacking in several areas, including object encapsulation and ease of decompilation of bytecode. This paper introduces, explains and comments on the techniques proposed by three separate papers which address these security issues. The papers are: "Confined types" by Bokowski and Vitek; "Using class decompilers to facilitate the security of Java applications" by Tam and Gupta; "Automatic detection of immutable fields in Java" by Porat et al.

1 Introduction

The Java programming language represents one of the few computer languages designed from the outset to implement security features into its platform as a default feature. Despite this (or perhaps because of it), several extensions to its security mechanisms have been proposed. This paper introduces, explains and discusses the techniques described in three papers: "Confined types" by Bokowski and Vitek in 1999 [1]; "Using class decompilers to facilitate the security of Java applications" by Tam and Gupta, published in 2002 [3] and "Automatic detection of immutable fields in Java" by Porat, Biberstein, Koved and Mendelson in 2000 [2].

The three studied papers propose several differing approaches to extending Java's security features. In [1], the authors present a syntactical addition to the Java language to eliminate the leaking of references to sensitive objects to insecure or untrusted code. Porat et al in [2] present a system to detect mutable and immutable fields and classes in Java code to enable programmers to easily see where possible security risks may lie (amongst other purposes). Finally, in [3], Tam and Gupta present a system named "REVEAL" which combines Java decompilers and

obfuscators in a feedback mechanism to allow developers to more easily and effectively produce hard-to-decompile Java bytecodes.

In section 2 of this paper the proposed extensions described in [1, 2, 3] will be explained. In section 3, pertinent points from the papers' discussion and conclusions will be presented, the solutions introduced in each paper will be compared and independent discussion will be put forth. Finally, in section 4 the paper will be concluded with a summary of the ideas presented.

2 Proposed extensions

2.1 "Confined Types"

In their paper "Confined types", Vitek and Bokowski propose two additions to the Java programming language intended to eliminate the problem of untrusted code gaining use of or access to sensitive code. The authors describe the problem as one that is present in all object oriented languages – if a program contains references to "secure" objects, it is possible, without careful design and debugging, to leak these references to the outside world either directly or indirectly. An example of the problem leading to a real security breach in the JDK 1.1.1 implementation was presented (reproduced from [1] in figure 2.1a). [1]

```
private Identity[] signers;
…
public Identity[] getSigners( ) {
     return signers;
}
```

Figure 2.1a *– An example reproduced from figure 1 in [1], showing a code fragment from the JDK 1.1.1 implementation that exhibits a security breach due to the getSigners() method exposing a reference to an internal array of "secure", Identity objects. The breach can be eliminated by using Vitek and Bokowski's* confined types.

The above example shows a fragment of code that exposes a reference to an internal array of Identity objects (an *Identity* contains information about who has access rights to a class at runtime). The method getSigners() was deliberately made public to allow any Java applet to discover information about all the principles (defined in [1] as "entities whose actions must be controlled") known to the system. However, as the information was returned as a reference to the original array, malicious code was able to take advantage of the array's mutable nature add Identities at will – giving it access rights it should not be able to have. [1]

The authors of "Confined types" give an obvious solution to the problem without adding any additional language constructs – simply returning a shallow copy of the signers array instead of a reference of the internal array itself eliminates the problem. However, Vitek and Bokowski note that there is nothing in the Java language itself that could prevent a similar situation occurring elsewhere. As an answer to this observation, they present two new language constructs that can be used to eliminate the problem: *confined types* and *anonymous methods*. [1]

The constructs come in the form of two syntactic additions to the Java language; the keywords *confined* and *anon*. In the case of the former, by placing "confined" before a class declaration the programmer is able to sign the class as one which *must not* have any references to any instances of it exposed to code outside of its package. By doing this, source code can be statically checked at compile time to verify that no such marked class instances can be accessed by untrusted code. An example solution to the Identity problem using this construct is shown in figure 2.1b (as reproduced from [1]). [1]

```
confined class SecureIdentity … {
     …
     // the original Identity implementation
     …
}

public class Identity {
     SecureIdentity target;
     Identity(SecureIdentity t) { target = t; }
     …// public operations on identities;
}

private SecureIdentity[] signers;
…
public Identity[] getSigners( ) {
     Identity[] pub;
     pub = new Identity[signers.length];
     for (int i=0; i<signers.length; i++)
          pub[i] = new Identity(signers[i]);
     return pub;
}
```

Figure 2.1b *– An example reproduced from figure 3 in [1], showing a solution to the problem demonstrated in figure 2.1a. The getSigners() method wraps each SecureIdentity in a new Identity object before returning them. That a SecureIdentity is not being returned directly is checked at compile time through the used of the* confined *keyword.*

By wrapping each SecureIdentity in the signers array in an Identity object (that defines only the public operations that can be performed on a SecureIdentity), the private data can be securely encapsulated. This is nothing special in itself – such a result can be achieved using standard Java – however, by marking the SecureIdentity class as confined, a full analysis can be made at compile time to verify that no references to the confined objects are exposed outside of the package. [1]

Vitek and Bokowski recognise that marking classes as confined is not enough to secure packages from reference leakage. In the case where a confined class inherits from an unconfined base class for example (in [1]'s model, *Object* is intrinsically unconfined), one might be able to downcast SecureIdentity to Object and pass a reference to an instance outside of the secure package. Though this could be

eliminated with dynamic checking, the authors constrained their solution to involve only static checking, thus; a further syntactic addition, the keyword *anon* was introduced. Vitek and Bokowski define an anonymous method to be one "that does not depend on the identity of the current instance to computer its value" [1].

By declaring a method as anon, the programmer states that the method can only use the reference "this" for "accessing fields and calling anonymous methods of the current instance" [1]. By restricting a method this way, one can guarantee at compile time that a method is not able to do anything that might break the security of a confined class.

2.2 "Automatic Detection of Immutable Fields in Java"

Porat et al in their paper "Automatic detection of immutable fields in Java" present an algorithm designed to detect the mutability of fields and classes in Java code. The algorithm runs statically (not at the run time of the code being tested) and can be used to test the mutability of any Java component. [2]

The algorithm described in [2] offers a possible solution to security breaches similar in nature to the example from [1] shown in figures 2.1a and 2.1b of this paper. In fact, Porat et al mention the same example as a possible usage of their algorithm, referencing the original article describing the security flaw [4]. By identifying variables that could be potentially be modified, their algorithm could be used by programmers to identify portions of Java code that could be vulnerable to attack, and allow them to rectify the problem by making the constructs in question immutable.

The algorithm was implemented in the form of a tool called "The Mutability Analyzer". The authors evaluated their tool by running it with the Java 2 JDK runtime library (*rt.jar*) – containing 4329 classes and 35999 methods – as input. They compared their results against the results produced by a reflection based tool, and found that the Mutability Analyzer identified approximately double the amount of immutable fields compared to the reflection based tool. The authors attribute this to their algorithm's ability to taken into account various "runtime accessibility constraints". [2]

2.3 "Using class decompilers to facilitate the security of Java applications"

Tam and Gupta in their paper "Using class decompilers to facilitate the security of Java applications" present a prototype system which integrates Java decompilers and obfuscators into a single application. The authors recognise the security risk presented by the standardised bytecode formats used to store compile Java applications and applets. Because of this, there are many decompilers available that are extremely effective at restoring readable source code from compiled Java bytecode. This poses a serious security risk as it allows attackers to do such things as reverse engineer the application or examine the source code and search it for security holes. [3]

Traditionally, a programmer requiring a hard to decompile Java application would use an obfuscator on the source code before compiling it – reducing fields and names to hard to read strings of characters – so when decompiled a human would have a difficult time interpreting the source. Tam and Gupta propose the use of their application "REVEAL" to allow interactive decompilation and obfuscation of Java applications. Their approach links the decompiler and obfuscator in a controlled feedback loop – the user is able to decompile bytecode, obfuscate it and decompile it again ad infinitum – with a library of different decompilers and obfuscators to see which produce the best results. Furthermore, REVEAL does not require the original source code to operate on, as it can be obtained by its built-in decompiler library. Figure 2.3a below (reproduced from [3]) shows the structure of REVEAL.

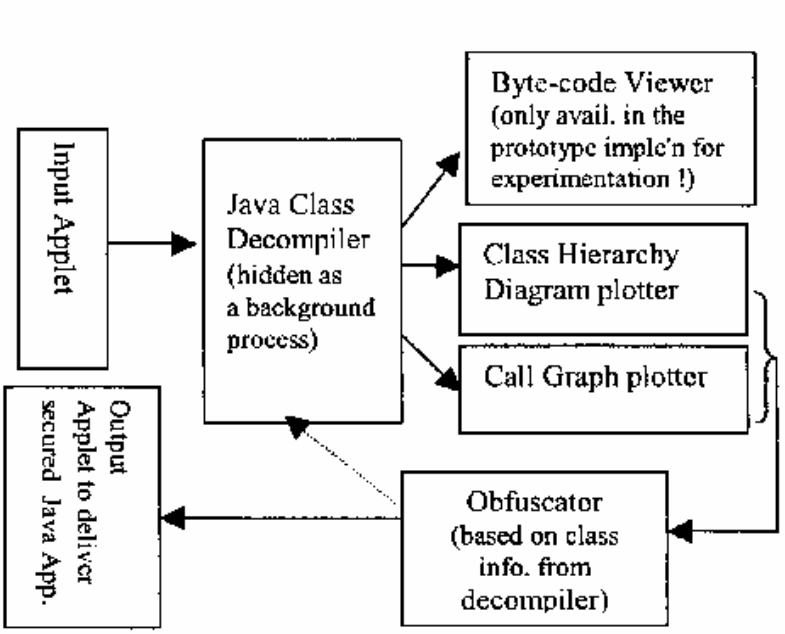

Figure 2.3a *– (reproduced from [3], page 155) The structure of the REVEAL system. A compiled Java application (applet) is input into a decompiler and after user examination of various visualisations (bytecode viewer, class hierarchy and call graph plotter) can be obfuscated. The process can be repeated until an acceptable obfuscation method is found.*

3 Discussion

In "Confined types", although the two syntactic additions presented seem simple at first glance, when considered more deeply their implementation requires much additional thought. In an OO language such as Java several different situations in which the new syntax could be used must be considered – for example inheritance, composition and other forms of code reuse. Vitek and Bokowski discuss extensively several different circumstances in which security could be breached, and show how the combination of confined types and anonymous methods avoids these problems. They provide detailed discussion of the rules that must be implemented by a compiler to correctly implement their solution. [1]

They identify two major weaknesses – that their solution defines a only flat protection model and that it can severely limit genericity. The first weakness is apparent in that only objects within a package can be protected; which may not be flexible enough for

some applications. They suggest use of *protection domains* to solve this problem – moving protection from the package level to named domains that contain a list of classes to be protected. The second weakness, that confined types limits genericity is of special interest because Java does not (as of 2003 (J2SDK 1.4) and the time of writing of [1]) support parameterised types. Because of this limitation, in order to store objects in a collection in Java one usually widens one's objects to class Object, and inserts them in a Collection class. As this is forbidden in many cases under the system presented in [1], custom wrapper Collection classes would required to be implemented, severely complicating user code. [1]

In "Automatic detection of immutable fields in Java", Porat et al note the advantages of their Mutability Analyzer application over the reflection based method. Firstly, and importantly, the Mutability Analyzer outputs the location of code found to potentially mutate objects. The authors state that this is important to allow developers to use the information to modify their code to remove unwanted mutations. They claim this feature is unique to the Mutability Analyzer tool. [2]

Although their results were impressive, Porat et al realise that their algorithm only operates statically – that is, it cannot detect discern the mutability of code that may only exhibit mutation at run time. They intend to implement "smart annotations" to allow the software to detect run time cases. However, no further detail is given on this matter. Finally, the authors state the possibility of extending the system to deal with "modular immutability analysis" – allowing the system combine the mutability results of several components to discern the mutability of an entire system. [2]

For the paper "Using class decompilers to facilitate the security of Java applications", Tam and Gupta conclude that their REVEAL prototype is the first system to combine the decompiler and obfuscator in a single interoperating package. They note several possible future extensions to their system: developing it into a centralised web-based system for secure Java applications; implementing incremental decompiling to increase efficiency and enhancement of the visualisations of the classes displayed to the user. [3]

The three papers examined in this report add to Java security in three separate and unique ways. Vitek and Bokowski attack the problem of untrusted code gaining access to or use of sensitive code by adding new syntax to the Java programming language itself. This method allows further compile time checking to be done which can detect an occurrence of the problem manifesting itself. On the other hand, Porat et al attack the same or similar problems by leaving the language syntax untouched, and applying extra analysis to the code to detect situations where the problem of unwanted mutation *might* occur. Porat et al's solution, while more broad, requires human analysis of the output from their algorithm to determine if it is indeed warning of a potential security in every case.

It is the opinion of the author if this paper that although both [1] and [2] provide possible solutions to the problem of potentially unwanted class/object/field access or mutation, neither provides a sufficiently elegant or universal solution. Although by adding extra language features to Java in [1] adds much more control over the access to sensitive code, it does so at the cost of a considerable reduction of language genericity in cases where the new features are used. The authors do note this disadvantage – but perhaps it is more of a disadvantage than as is stated by them. One of the leading principles in software engineering is code reuse. However, by using these new features the programmer would be required to write custom "confined" container classes for confined objects, thus violating the code reuse principle. Although it may seem like a small price to pay, in a large piece of software many extra containers may need to be written, thus dramatically increasing the amount of supporting code and therefore the amount of possible bugs. Also, due to the somewhat complex nature of the language additions, developers may choose to ignore them altogether (similar to the use – or lack of use – of "const" in C++).

The solution to the problem under consideration provided by [2] does not cause the problems with genericity that [1] does, but introduces a new set. Although the developer may not need to deal with the complexity of new language constructs, he would be required to use and analyze the result produced by the Mutability Analyser tool. In a large application, the output produced could be enormous, and furthermore, a large proportion of it could be false negatives with respect to the specific security issue of code encapsulation in question.

A possible solution to these problems could be to combine the techniques described in both [1] and [2]. By implementing only a single new language construct, “confined”, and removing the concept of anonymous methods, one could considerably reduce the complexity of the language addition. However, by removing anonymous methods the compiler would no longer be able to enforce confinement. To correct this, it is proposed that the solution presenting in [2] could be incorporated into the compiler. Given the additional information that references to certain classes should be confined to a package, the algorithm in [2] might be able to be used to check the security of all references that would have previously been done by using programmer specified anonymous methods. Confinement breaches (not of the warp-core variety) could be reported as warnings by the compiler.

Finally, it is the opinion of the author of this paper that the REVEAL application presented in [3] could have advantages over using a traditional obfuscation system in the field of disabling reverse engineering of Java bytecode. However, the authors [3] did not consider any future extensions of their system involving decreasing the amount of user input. Rather than enhance the visualisations of the decompiled classes, it may be possible to eliminate them entirely and instead implement AI algorithms which could converge on a system of obfuscation that could produce the best results – without user intervention. For example, pattern matching between the unobfuscated decompiled source code and the obfuscated source code could be performed, and a measure of the difference between them could be developed. A search of different combinations of obfuscators and settings could then be performed to produce a result to maximize this difference measure. Such an automated iterative procedure could produce an optimal obfuscation.

4 Conclusion

This report studied three papers with the common goal of extending the security capabilities of the Java platform. Vitek and Bokowski proposed a syntactic extension of the language to enhance the security of object references within packages. Porat et al presented an algorithm and prototype application to analyze a Java application for

mutability of its classes and fields. Tam and Gupta presented an application to allow the user-controlled iterative refinement of obfuscation.

All three papers were found to have significant advantages over current methods, as well as disadvantages. The authors' conclusions and suggestions for future work were presented, and further independent conclusions and suggestions were put forth.

Finally, the author would like to acknowledge and thank Adam Johnson and Craig Carpenter for their reading of this paper and helpful comments.

5 References

[1] J. Vitek, B. Bokowski, *"Confined Types"*, ACM SIGPLAN Notices, Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. October 1999, Volume 34, Issue 10.

[2] S. Porat, M. Biberstein, L. Koved, B. Mendelson, *"Automatic Detection of Immutable Fields in Java"*, http://www.cas.ibm.com/archives/2000/proceed/cascon00/htm/english/abs/porat.htm, CASCON 2000.

[3] V. Tam, R.K. Gupta, *"Using class decompilers to facilitate the security of Java applications!"*, Proceedings of the First International Conference on Web Information Systems Engineering, 2000. Volume 1, pp 153 -158.

[4] Secure Internet Programming Group at Princeton University, *"HotJava 1.0 Signature Bug"*, http://www.cs.princeton.edu/sip/news/april29.html, 1997.