Position paper: Confined Types

Boris Bokowski
Jan Vitek

1 Introduction

Writing secure code is hard. The steady stream of security defects reported in production code attests to the difficulty of the task. Software systems, such as the Java virtual machine, that permit untrusted code to mingle with authorized code raise the stakes much higher than more traditional systems as trust boundaries become thinner and fuzzier.

In object-oriented programming it is difficult to control the spread and sharing of object references. This pervasive aliasing makes it nearly impossible to know accurately who owns a given object, that is to say, which other objects have references to it [6]. The lack of ownership information [8] imposes a defensive programming style: since every method may have been called by an adversary, appropriate security checks that verify the caller's authority must be performed at method entry.

This state of affairs creates a tension between security and program efficiency. Placing dynamic security checks in the prologue of each method is not realistic for performance reasons. Instead, security conscientious environments offer dynamic security checks that can be called explicitly by the programmer [2, 10]. These checks are interspersed in the program logic and thus nothing short of full-fledged program verification can ensure that not a single check has been omitted somewhere with the potential of compromising the security of the entire system.

Reusability creates its own set of problems for security. Similar to the synchronization problems of concurrent programming, inheriting from classes that do not implement the same security policy may create security anomalies. Of course, from the point of view of the library designer, it is not possible to implement classes that are secure in all contexts. Even if one could do that, the overall performance of the library would be unacceptable in secure contexts.

To a large extent the problem for implementing security is one of defining interfaces between protection domains. As we stand now, a Java virtual machine may manage objects of many different protection domains — code loaded from different sources — but imposes no clear boundaries between these domains. So, from the security engineer's viewpoint, there is no well identified place where to put security checks.

One solution to this dilemma is to separate objects that are internal to a protection domain from external objects. Internal objects implement the behavior of the application without concern to security, while external objects are the interface between protection domains and must implement the security policy. Such a separation of concerns simplifies the life of the application programmer as the core of the system can be written without security checks. Moreover, it improves security as a smaller set of classes, the interface objects, become the focal point for security analysis.

Current object-oriented languages do not provide the means to enforce such a distinction between objects. While access modifiers can restrict how certain object types are manipulated [4] — by curtailing visibility of methods and fields — and also restrict the scope of types, object-oriented languages do not provide strong encapsulation [7]. They typically cannot control the scope of object *references*. Thus references to sensitive parts of an application may leak to other protection domains.

We propose *confined types* as an aid for writing secure code. Confined types are meant to be used for preventing internal objects from escaping their protection domain. Given some definition of domain, we give the following definition of *confinement*: a type is said to be confined in a domain if and only if all references to instances of that type originate from objects of the domain. Confined types differ from existing access control features in that they prevent references to *instances* from leaving a domain rather than restricting access on a *class* level. In effect, confined types enforce static scoping of dynamic object references.

Our proposal for achieving confinement in Java depends on static constraints on the definition and use of objects. Thus there is no extra runtime overhead and confinement is guaranteed at compile time — avoiding the need to worry about "confinement breach" exceptions. We extend Java with two additional modifiers, one for classes and one for methods, and enforce some restrictions. While certain programming tasks may be clumsier, we argue that these restrictions are mild and that reasoning about security is much simpler.

Confined types have been implemented with `CoffeeStrainer` [1], a framework for static checking of structural constraints in Java programs. In this implementation we have chosen the Java `package` as the unit of protection. Packages are well suited for this task as they group related classes and they already provide basic access control features. Our annotations do not affect program semantics, thus a valid program with confinement annotations behaves identically as the same program with no annotations.

2 The Class Signing Example

In Java, each class object (instance of class `Class`) stores a list of signers, which contains references to objects of type `java.security.Identity`, representing the principals under whose authority the class acts. This list is used by the security architecture to determine the access rights of the class at runtime. A serious security breach was found in the JDK 1.1.1 implementation which allowed untrusted code to acquire extended access rights [9]. The breach was due to a reference to the internal list of signers leaking out of the implementation of the security package into an untrusted applet.

The scenario is as follows. Assume a malicious applet loaded from the net. Without any trusted signatures, its access rights are strongly limited. But the JDK does allow the applet to get its own list of signers. Furthermore, an applet can find out about all the principals known to the system by calling a method of `java.security.IdentityScope`. The method that returned the list of signers of a class (implemented by a Java array object) accidentally returned a reference to the system's internal array. Since arrays are mutable data structures, the applet can then proceed to update the array to include signatures of principals known to the system and obtain access rights it should not have, thus opening the system to more serious attacks.

We first present a program fragment which exhibits the security problem described above, and then give a solution using confined types.

2.1 The Security Breach in Detail

In Figure 1, the array `signers` is the system's internal array that contains references to instances of the class `Identity` (the principals). Modifying this array is definitely a dangerous operation but there are no provisions in the implementation of the array class for checking the authority of the caller in an update. The security breach is caused by the `getSigners()` method which returns a reference to the `signers` object.
The attacker need only call `getSigners()` to be able to freely update the system's signature array. A simple fix is to return a shallow copy of the internal array. Figure 2 makes the copy explicit. While this solves

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

Figure 1: Signatures without confined types

the particular problem, nothing guarantees that similar defects are not present in other parts of the package.

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

Figure 2: Ad-hoc fix of security problem

What is interesting about this example is that none of the standard Java protection mechanisms seem to help. Access modifiers and type abstraction are not relevant here. Restricting the use of the `Identity` objects would do no good as the attack does not interact with `Identity` objects, it only needs to acquire references to them and copy those references. Information flow control [11] does not apply either, since we do want to allow applets to read the signature information and to see identities known to the system. Finally, inserting dynamic checks in the array update operation, which is the point where the security policy is actually broken, is unrealistic as *all* array updates performed in the JVM would incur the cost of a dynamic check.

We now give a solution that guarantees that none of the key data structures used in code signing escape the scope of their defining package.

2.2 Class Signing with Confined Types

To prevent software defects such as the one outlined above, we propose to ensure that *references* to identity objects are confined to the `java.security` package. This is achieved by renaming the `Identity` class as `SecureIdentity` and declaring it *confined*. Intuitively, the meaning of confinement is that references to instances of a confined class, or to instances of any of its subclasses, cannot be disclosed to, or accessed by, other packages. That is to say, only the classes defined in package `java.security` may interact with `SecureIdentity` objects. In order to preserve the functionality of the original interface, we define a new class `Identity` which can be seen outside of the security package. This class implements the public methods of `SecureIdentity` and has a private reference to a `SecureIdentity` instance. `Identity` plays the role of a guard and encapsulates the real identity object [3, 5]. The `Identity` class is purely for external use, it is neither a subclass nor a superclass of `SecureIdentity` and thus cannot be confused with a `SecureIdentity` object within the security package. Any attempt to return a `SecureIdentity` object to an outside package will be caught at compile-time as a violation of confinement. Figure 3 outlines our solution.

```
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 3: Signatures with confined types.

The `getSigners()` method is similar to Figure 2. The important difference is that the type of the internal array `signers` is different from the type of the array that is being returned. The confinement constraints extend to arrays, thus if a type `A` is confined, then the array type `A[]` is confined as well. The `getSigners()` method allocates an unconfined array to which newly created objects of type `Identity` are copied. If `getSigners()` tried to return its internal array, a confinement breach error would be signaled.

This solution preserves the functionality of the original program, in fact outside code need not be aware of the existence of confined types. But from a security engineering point of view, attention is directed to the `Identity` class as it can be accessed by untrusted components, and may thus (if deemed necessary) include dynamic security checks.

This example shows that confined types help in developing secure code as they draw a strong demarcation line between internal representation objects and external interface objects.

3 Conclusion

Software security is difficult problem. We propose to control the dissemination of object references to ease the task of writing secure code, as the interface between components is sharper. Confinement is enforced by a set of syntactic constraint which can be verified statically. Thus, our proposal incurs no run-time overhead and all confinement violations are caught before running the program. We have implemented a confinement verifier for Java using CoffeeStrainer [1]. The verification procedure is modular as classes are analyzed individually. Furthermore, our extension is transparent, that is annotated classes can also be compiled by the standard Java compiler.

References

[1] B. Bokowski. CoffeeStrainer – statically checking structural constraints on Java programs. Technical Report TR-B-98-09, Freie Universität Berlin, Institut für Informatik, Dec. 1998.

[2] L. Gong. Java security architecture (JDK 1.2). Technical report, JavaSoft, July 1997. Revision 0.5.

[3] L. Gong. Guarding objects. In G. Vigna, editor, *Mobile Agents and Security*, volume 576 of *LNCS*, pages 1–23, Berlin, Germany, Aug. 1998. Springer.

[4] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[5] D. Hagimont, J. Mossière, X. R. de Pina, and F. Saunier. Hidden software capabilities. In *16th International Conference on Distributed Computing System*, Hong Kong, May 1996. IEEE CS Press.

[6] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4), Dec. 1992.

[7] J. Noble, J. Potter, and J. Vitek. Flexible alias protection. In *Proceedings of ECOOP'98*, Brussels, Belgium, July 20 - 24 1998.

[8] J. Potter, J. Noble, and D. Clarke. The ins and outs of objects. In *Australian Software Engineering Conference*, Adelaide, Australia, November 1998. IEEE Press.

[9] Secure Internet Programming Group. http://www.cs-.princeton.edu/sip/news/april29.html. 1997.

[10] J. Tardo and L. Valente. Mobile agent security and Telescript. In *IEEE CompCon*, 1996.

[11] D. Volpano and G. Smith. A type-based approach to program security. *Lecture Notes in Computer Science*, 1214, 1997.