Secure Coding with Information Flow Control

Mr. Aniket Patil, Mr. Harish Barapatre
aniket14976@yahoo.com
harish.barapatre@tasgaonkartech.com

Abstract-

Ruby is an interpreted scripting language for quick and easy object-oriented programming. In this document, we are providing information flow control (IFC) in Ruby. Information Flow Control (IFC) to track the flow of data end-to-end through the system. For this purpose, we use Ruby taint tracking system. Isolation of functional implementation and security restraint is provided by IFC, which supports easier development and maintenance. In this paper, Aspect Oriented Programming (AOP) affected the drawback of Ruby track. In this FlowRi.e. Information Flow Control for Ruby is library extending Ruby to keep IFC ancient using AOP via the Aquarium open source library.

Keywords:- Information Flow Control, Aspect Oriented Programming, Security

1. INTRODUCTION

In earlier stage of ruby we elaborate a web portal, in combination with Public Health, to grant access by brain cancer patients to their records. To track the flow of data through the system we used IFC by this we can keep data safe and also the authentication.

However, we came to realise certain limitations of the mechanisms we had deployed. For example, to enforce the required IFC policy, we manually inserted IFC checks at selected application component boundaries. In practice, objects and classes are the natural representation of application components within an object-oriented language and it seems natural to relate security concerns with those objects. We should therefore associate the primitives and mechanisms to enforce IFC with selected objects. Furthermore, we wish to be able to assign boundary checks on any class or object without further development overhead. We also wish to be able to exploit the inheritance property to define rules that apply to categories of objects (for example defining a boundary check for all possible children of I/O). We therefore decided to investigate the use of Aspect Oriented Programming (AOP), and selected the Aquarium library [53], instead of RubyTrack, to use with our Ruby implementation to provide IFC-aware web applications. We make some assumptions on the environment and the problems we are addressing. First, we assume that the developer is not adversarial; the aim is to protect against inadvertent disclosure of information through bugs within the application. Second, we focus on the design of web applications using a framework such as Sinatra or Rails to be, for example, deployed on a PaaS (Platform as a Service) cloud, using readily available languages/interpreters. Third, in this context, we assume the application’s host ensures that no data can be disclosed outside of the application. Finally, we assume that the infrastructure running the application is willing to accept performance overhead in exchange for increased security assurance. Other solutions can be envisioned for other circumstances, such as using a particular IFC-aware interpreter or running on an IFC-aware operating system.

The Ruby standard implementation provides no real multithreading support (more recent versions are starting to address this). Therefore, Ruby web servers tend to be multi-process rather than multi-threaded, which allows us to handle IFC rule violations effectively, and we fail completely any process violating an IFC constraint.
This is preferable to (per thread) exception handling which may generate implicit information flows [39].

Section 2 gives background on Aspect Oriented Programming and Information Flow Control. Section 3 then gives an overview of our work; we specify the basic principles governing flows and the primitives we added to the language to manipulate IFC labels. Section 4 describes our implementation. Section 5 presents a simple use case to demonstrate the simplicity of the approach then describes the web portal for brain cancer patients mentioned above. In section 6 we show that performance is similar to an equivalent solution and we argue that our approach provides better usability. Section 7 presents related work and section 8 summarises and concludes.

II. BACKGROUND

Our paper targets readers interested in both AOP usages and IFC implementations. Some may not be familiar with both topics so we give a brief introduction to each, indicating the relevant literature. The last subsection discusses problems that are not addressed in this paper and which are generally not addressed by library-level implementations of IFC.

A. Aspect Oriented Programming and Security

Aspect Oriented Programming was introduced in 1997 by Kiczales et al. [22]. It is a programming paradigm extending Object Oriented Programming (OOP) by allowing cross-cutting aspects to be expressed. An aspect is a piece of code named an advice together with a pointcut determining when it should execute. The pointcut is used to determine the join-points (object methods) where the advice code will be executed. Fig. 1 provides an illustration of this concept. In the original specification an advice could be executed before or after the join-point code is executed. The paradigm was later extended with an around advice [23] which has control over whether or not the join-point code should be executed.

An advice is composed of a primitive to express when the advice should be executed (i.e., before, after, around), a pointcut describing where the advice should be executed and a block of instructions to specify the behavior of the advice. This is illustrated in Fig. 2 where we define an advice to be executed around a call to the method write of instances of File. The example is in Ruby and using the Aquarium library [53]. The parameters passed to the advice are the join-point to be executed, the object the method belongs to, and the arguments passed to the method. A pointcut can be made more expressive by using a regular expression (some implementations may not provide this, however this is provided by the library we are using) to define the methods and classes to which the advice should be applied. We can also specify a list of methods to be ignored, or implement different behavior either after a normal execution or if the method throws an exception.

B. Information Flow Control

It has long been argued that standard security techniques, such as firewalls and access control mechanisms, are not enough to prevent information leakage [9]. Indeed, it is beyond the scope of such mechanisms to determine whether, after the controls they impose, the information is used correctly. For example it is difficult to determine if the confidentiality of decrypted data is respected [39]. We therefore need to protect information flow, that is, how information is transmitted within and between applications. In 1975, Denning [8] proposed a model to track and enforce rules on information flow within a procedural language. In this model, variables are associated with security classes. The flow of information from a variable a to a variable b is allowed only if the security class of b noted b is higher than a. The security class of an function on a classified variable is noted .. This allows the no-read up, no-write down principle [4] to be implemented to enforce secrecy. By this means a traditional military classification (public, secret, top secret) can be implemented. A second security class can be associated with each variable to track integrity (quality of data) [6] during reading down and writing up. Using this model we are able to control and monitor information flow to ensure data secrecy and integrity.
In 1999 Myers [33] proposed security labels to replace the security classes of Denning’s model [9]. Clearance levels are considered too coarse-grained, permitting unnecessary access and were replaced by the “need-to-know” principle, also known as “Principle of Least Privilege (PoLP)” [41]. Labels are composed of tags representing categories of information or the nature of the information. The secrecy label is propagated with data between objects and the integrity label is used to define which data are allowed to flow into and within an object. Here, integrity relates to the trustworthiness of the source of any data rather than accidental corruption, for example, by hardware. A central authority is not needed in such a model since data flow policy is user-specified (discretionary) rather than centrally mandated. However, system support is needed at runtime for the continuous monitoring of data flows.

In this style of language a variable declaration can be augmented with an annotation to describe the policy associated with the data item. Examples can be seen in the solutions proposed by Denning [9] or Myers [33]. It is in these cases the programmers’ responsibility to not only understand the algorithm being implemented but also the desired security policy [57]. But the security constraints may not all be clear during the functional design phase and inconsistencies can arise at runtime. It is generally better to separate security concerns from functional ones, limiting the impact they have on each other in the engineered system. We decided in this work to explore the use of AOP to enforce IFC constraints specifically in order to provide this separation.

C. Implicit Flow and Covert Channels

This phenomenon is illustrated in Fig. 5. From the Denning model, briefly described in section B, we expect that $w = x$; that is, $x$ and $w$ are of the same security level. However, if we enforcing process sensitivity levels, we have $w = x \land z$ even if we know there is no $z \rightarrow w$. To address the concerns brought by the benevolent developer assumption, it has been suggested that an implicit flow can be prevented by the preemptive halting of program execution [2, 40]. However, this could prevent legitimate applications from terminating [2]. Therefore, to deal with potentially malicious code, variable-level runtime taint tracking can be combined with static analysis techniques [51].

At present in our project we do not consider implicit flow nor other covert channels [20] such as timing channels, storage channels [26, 27] or termination channels [52].

III. The FlowR IFC Model

IFC models are used to represent and constrain the flow of information within an application. In this paper, we focus on the aspects of the model
relating to a single application rather than a distributed, multi-application environment. In the DEFCon project [31], AOP was used with Java to enforce IFC by inserting IFC policy around selected methods. In FlowR, we extend those ideas by providing IFC at the level of objects, classes and methods, and provide basic primitives to enforce IFC.

Our approach is not specific to Ruby but can be used with any OO Language that supports AOP. Furthermore, our techniques can work with an arbitrary library, without programmers having to know about its inner workings, so requiring little effort from them.

A. Security labels

In order to monitor Information Flow we use labels. Our label model is inspired by that proposed by Efstathopoulos et al. [13].

Every tracked object is associated with two labels: a Receive label and a Send label. The Receive label is used to represent the type of information that is allowed to flow into an object, while Send labels are used to represent the nature of the information and its sensitivity. Send (S) labels are sticky, that is, they will propagate and taint any object they interact with, which ensures that no information can flow untracked. Receive (R) labels however do not propagate and concern only a single object or class.

In a Receive label: An object with a tag in its Receive label is not allowed to receive information labelled with the tag t. An object with a tag t+ in its Receive label is allowed to receive such information. Receive labels are not changed by the flow of information.

In a Send label: An object with a tag t+ in its Send label is allowed to flow to an appropriately labeled destination object and the tag will propagate. An object with a tag t- in its Send label is allowed to flow to an appropriately labelled destination object, but the tag does not propagate. For details on tag propagation

B. Allowed flows and label propagation

We define the function ALLOW(A;B) which, given two entities A and B returns true if the flow is allowed and false otherwise. We also define the function PROPAGATE(A;B) which propagates the send label from A to B according to the definition we have just given. Jajodia et al. [19] specify that information flow occurs only if an object changes its state, i.e. changes the value of one or more of its attributes. However, this assumes that methods cannot be altered at run time [18], which is not the case in Ruby. Therefore we need to consider more possible flows.

Flow of information occurs on method call. A method call is the interaction of several entities: the caller C, the callee O, the method parameters p1 ; ::; pn and the returned value r. We distinguish two phases: the calling of the method and the returning phase. During the first phase the flow of information is as follows: C ! O, p1 !O, ...pn ! O. In the second phase the flows first O !r and then r ! C. It is important to note that at the end of the second phase we may have Sr 6= SO. This is due to the fact that class/object attributes may have different labels than the class/object they belong to and that there may be label operations within the execution of the method (a method performing an operation on the \ returned value of another method call, for example). Having different attribute labels may be useful when doing event processing such as in DEFCon [31].

IV. FlowR implementation:

We saw in section B, that flows are enforced in two phases: on method call and on method return. This corresponds exactly to the AOP standard around advice [23]. We describe the process in algorithm 2. O is the callee, C is the caller, M the method called, As is the set of attributes and join point isthe join point to be executed. We now describe the step described in algorithm 2:

1) We verify that information is allowed to flow from the caller to the method and we also verify that the information contained in the parameters is allowed to flow in the method;
2) We propagate the labels from the caller and the parameters to the callee;
3) We execute the join point;

Figure 6 Illustration of Label Inheritance

Flow of information occurs on method call. A method call is the interaction of several entities: the caller C, the callee O, the method parameters p1 ; ::; pn and the returned value r. We distinguish two phases: the calling of the method and the returning phase. During the first phase the flow of information is as follows: C ! O, p1 !O, ...pn ! O. In the second phase the flows first O !r and then r ! C. It is important to note that at the end of the second phase we may have Sr 6= SO. This is due to the fact that class/object attributes may have different labels than the class/object they belong to and that there may be label operations within the execution of the method (a method performing an operation on the \ returned value of another method call, for example). Having different attribute labels may be useful when doing event processing such as in DEFCon [31].
4) We propagate the method label to the returned value;  
5) We verify that the information contained in the returned value is allowed to flow to the caller;  
6) We propagate the returned value’s labels to the caller.

Figure 7 Example 1: Protecting Password

If the flows are found not to be allowed the program is aborted. We used the AOP library Aquarium [53] to implement this in Ruby. We place an advice around any public method of a tracked object (an object is considered to be tracked when there are tags associated with this object). Regardless of the actual object implementation we are therefore able to protect information flow.

In order to implement the concepts described in our model (section 3) we provided the API described in table 2. We have instructions to start and stop the tracking of basic variables. Indeed, in some cases, it may be required to activate tracking only on some portion of the code. For example, the loading of a large configuration file could be done before the tracking is activated in order to improve performance. Similarly, execute procedure untracked allows a single procedure to be executed with tracking deactivated.

Figure 8 Isolating a user’s data

Although untracked procedures are executed in Ruby safe mode, the programmer is relied upon to understand the IFC implication of executing a portion of code untracked. The manipulation of this API is illustrated in Fig. 9. In addition to this API we also support object methods manipulating their labels directly as this may be useful in some circumstances. In Fig. 7 we illustrate how such direct manipulation can be used to prevent information leak on standard output. In this example, we declare that credentials are not allowed to be displayed on the standard output and try to print a password that we previously associated with the credential tag, which causes the program to fail. A developer should be able to develop an application without initially being concerned about IFC, and with the ability to use a legacy library that was built without IFC in mind. Once the application is developed, the original developer, or another expert, can add IFC rules to ensure that the application behaves correctly with respect to information flow.

As attributes are also objects it is also possible to assign labels to each attribute. This would represent the different security and confidentiality requirements of the different fields of this structured document. For example, medical records might be shared between medical professionals and social services. Some sensitive information such as HIV status may be restricted to medical professionals only, while more general information may be accessible to social services, for example to detect signs of child abuse. Another use of attribute labelling could be to build an event processing system as described in the DEFcon work [31].

V. Evaluation

Our tests measure the performance of our solution, FlowR, compared with an equivalent solution, that extends native Ruby with RubyTrack, developed for the SafeWeb project [17]. It is important to note the feature differences that explain the performance.
difference of FlowR when compared with RubyTrack. Our first series of tests concern computing intensive tasks. We demonstrate that FlowR does not perform significantly worse than its equivalent using RubyTrack. In addition, no performance optimization has been attempted for FlowR, which is beyond the scope of this paper.

A. Compute-intensive tasks

We designed two simple tests. The first consists of counting the number of words in text stored in a file on disk (“Les Contemplations” by Victor Hugo). The second test consists of calculating the first n prime numbers. The execution time of the native Ruby code is our time unit. We compare RubyTrack, FlowR and FlowR using untracked procedure calls (section 4). The results, show the same order of magnitude for RubyTrack and FlowR. We did not attempt to optimize performance, and the Aquarium library is known to suffer from performance issues [56]. This is because, at present, Aquarium applies advices at runtime whereas AspectJ and AspectC++ apply them at compile time. Furthermore, it is commonly accepted that performing IFC is generally not a good idea while performing computing intensive tasks. Using untracked procedure calls provides much better performance. This figure includes the switching of tracking on, off and on again which induces some overhead.

However, this overhead becomes negligible as the execution time becomes large. Therefore, untracked procedure calls can provide performance identical to native Ruby in the case of long computing intensive tasks.

B. Brain portal: a web portal application

In order to evaluate our library under realistic conditions we used the data store described in section 5. In order to evaluate the performance of our implementation we queried our data store 1000 times, asking for 50 different, randomly chosen data items. We compare the averaged values obtained with native Ruby, RubyTrack and FlowR.

We used the “thin” Ruby web server as it provides quite good performance. We first display an unlabelled static page to measure the influence of tracking without flow enforcement. RubyTrack and FlowR add an overhead of 7% and 12% respectively compared to native Ruby. The performance penalties for retrieving a medical record from our database are of the same order (10% and 15% respectively).

VI. CONCLUSIONS AND FUTURE WORK

Regardless of the precise implementation properties, we believe that the primitives we propose here are a natural way to express flow constraints within an application in an OO Language. We also believe that the AOP approach discussed in this paper is a good solution to providing IFC when control over the system running an application is not available. That is, our IFC runs above unmodified platforms as well as potentially extending unmodified applications. We assume a benevolent developer, which is standard for all library-level IFC implementations; and we do not support implicit flow tracking, again, the case for most library-level IFC. We will continue to evaluate the tradeoffs involved in taking the AOP approach compared with using more disruptive and less maintainable mechanisms that might provide higher security and performance. The Brain-Portal implementations using RubyTrack and FlowR have provided a first case study.

Our current implementation does not support multi-threading but this is not inherent to our proposed model. Rather, it is constrained by the AOP library implementation we used and the general poor support of real multithreading in the standard Ruby implementation. Another issue that may arise when using AOP to enforce IFC is when several AOP advice are implemented over the same object; for example enforcing IFC, logging and authentication.

In such an scenario it may be necessary to determine whether composition issues arise, as discussed in [12, 35]. It is important to note that our library does not require the rewriting of any code and therefore does not modify program behaviour, except when IFC constraint violation forces the process to abort. So when performance is a critical issue, the library can be used during development, to track unexpected data flows, and ignored in deployment. Again, a tradeoff is involved between performance and security.

In this paper we presented an IFC library implementation using AOP, with primitives to provide IFC concepts, and mechanisms to enforce IFC. We separated application functionality from security concerns. Programmers need not be aware of IFC during application development, and a security specialist can add IFC as a separate phase. This is good engineering practice and achieves better maintainability. However, we described our model informally and a more formal model would be required before substantial future work was carried out. We believe that using AOP to provide IFC has many advantages which we intend to evaluate further in future work, especially in the context of cloud deployment.
REFERENCES


