**Title:** On the Systematic Design of Web Languages

**Performing Organization:** Naval Postgraduate School, Center for Information Systems Security Studies and Research (NPS CISR), Department of Computer Science, Monterey, CA, 93943

**DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution unlimited

**ABSTRACT**
On the Systematic Design of Web Languages

Dennis Volpano  Geoffre y Smith

“Semanticists should be the obstetricians, not the coroners, of programming languages.” -John Reynolds

Recently there has been phenomenal growth in the use of the Int ernet through the World Wide Web. Especially interesting is the emergence of so-called “web languages”, such as Java\textsuperscript{1} [4, 5], which support the development of programs that can be downlo aded to a client’s machine for execution by a web browser. This brings power and flexibility to web applications, but it also brings well-known end-point security problems, such as the threat of integrity and unauthorized disclosure attacks [1]. Yet, although security has been an issue in web language design, there appears to be no formal characterization of the kinds of security properties that one can expect of programs written in these languages.

A web language should, we believe, be designed so that all programs are guaranteed to satisfy explicitly-stated security properties. To this end, we advocate a more systematic approach to the design of secure web languages. One begins with a core language for which some set of desired security properties can be shown to hold with respect to a formal semantics. The language is then incrementally extended to include new features, as necessary. At each step, the designer has an obligation to establish that the security properties have been preserved.

For example, one kind of attack a web language should guard against is a disclosure attack where a malicious program downloaded by a client attempts to make the contents of certain private files public. In response, one might use discretionary access control mechanisms to simply forbid access to sensitive information, or to limit access to output devices. But such restrictions would likely render many applications useless.

For example, a downloaded mail tool would need access to a client’s sensitive mail folder and to a nonsensitive, publicly-writable directory like /tmp for message composition and replies. However, granting such access rights requires care, since a mail tool might now be able to copy a client’s entire mail folder to /tmp.

\textsuperscript{1}Java is a registered trademark of Sun Microsystems Inc.
There is a form of static analysis, however, that can be applied to programs to protect against disclosure attacks while allowing utilities like mail tools to access files and directories at different sensitivity levels. It is called secure information flow analysis and it was pioneered by Dorothy Denning [2]. We briefly describe how a type system can be imposed on a simple imperative language to guarantee secure information flow. In part, we illustrate how well type theory can be applied to address a security problem, but our main point is that the security theorem stated below actually guides the language designer in evaluating potential language extensions.

To begin with, we suppose that we have a partial order of security classes \( \tau \). For simplicity, we may consider just two classes, \( L \) (low) and \( H \) (high), where \( L \) represents nonsensitive information and \( H \) represents sensitive information. There is a natural subtyping among these classes, based on the observation that \( L \) information can safely be considered to be \( H \) information, but not vice versa. Thus, \( L \leq H \).

To type the phrases of the language we use three forms of types: \( \tau \) (for expressions), \( \tau \ var \) (for variables), and \( \tau \ cmd \) (for commands). For example, we might have \( x : L \ var \) to indicate that \( x \) is a variable that holds values of type \( L \). The rule for typing an assignment \( x := e \) requires that \( x : \tau \ var \) and \( e : \tau \). So if \( y : H \ var \), the assignment \( y := x \) is legal, since the value of \( x \) can be coerced to \( H \), but the assignment \( x := y \) is illegal. Also, the assignment \( x := 17 \) is legal, since literals are taken to have type \( L \).

More subtly, we also want the conditional

\[
\text{if } even(y) \text{ then } x := 0 \text{ else } x := 1
\]

to be illegal, because this results in an implicit flow of information from \( y \) to \( x \). To achieve this, we give commands types of the form \( \tau \ cmd \). A command \( c \) has type \( \tau \ cmd \) only if every variable assigned to in \( c \) has type at least \( \tau \). Note that \( cmd \) is an antimonotonic type constructor: \( \tau \ cmd \leq \tau' cmd \iff \tau' \leq \tau \). The rule for typing assignments says that if \( x : \tau \ var \) and \( e : \tau \), then \( x := e : \tau \ cmd \). The rule for typing conditionals says that if \( e : \tau, c : \tau \ cmd, \) and \( c' : \tau \ cmd \), then if \( e \) then \( c \) else \( c' \) has type \( \tau \ cmd \). The conditional \( \text{if } even(y) \text{ then } x := 0 \text{ else } x := 1 \) is illegal, then, because \( even(y) \) only has type \( H \), and \( x := 0 \) does not have type \( H \ cmd \).

The value of this type system is given by a security theorem, which is a property known as noninterference [3]. It asserts, intuitively, that no information can flow from \( H \) variables to \( L \) variables [6]:

**Theorem.** Suppose that program \( c \) is well typed, and \( \mu \) and \( \nu \) are two memories that agree on all \( L \) variables. If \( c \) terminates successfully when run on memories \( \mu \) and \( \nu \), yielding memories \( \mu' \) and \( \nu' \), respectively, then \( \mu' \) and \( \nu' \) agree on all \( L \) variables.

Many challenges remain before a truly secure web programming environment can be developed. However, we believe that theorems of this kind are necessary in any web language that hopes to provide secure web computing. By taking security into account at the outset of language design, one can
get a handle on what kind of language is possible if a particular security property is expected to hold. For instance, notice that the security theorem above requires \( c \) to terminate successfully when run on memories \( \mu \) and \( \nu \). This means that if it should terminate abnormally on \( \mu \) or \( \nu \) then the potential for interference exists in any language with the ability to detect such status. Unfortunately this capability can be found in languages that return the exit status of a process or generate and handle exceptions. So extending the simple language with such features can be dangerous. The point here is that the systematic approach leads us to the discovery.

References


