An Access Control Language for Web Services

Emin Gün Sirer Ke Wang
Computer Science Department
Cornell University
Ithaca, NY 14853
{egs, kewang}@cs.cornell.edu

ABSTRACT
This paper presents an approach for formally specifying and enforcing security policies on web service implementations. Networked services in general, and web services in particular, require extensive amounts of code to ensure that clients respect site-integrity constraints. We provide a language by which these constraints can be expressed and enforced automatically, portably and efficiently. Security policies in our system are specified in a language based on temporal logic, and are processed by an enforcement engine to yield site and platform-specific access control code. This code is integrated with a web server and platform-specific libraries to enforce the specified policy on a given web service. Our approach decouples the security policy specification from service implementations, provides a mandatory access control model for web services, and achieves good performance. We show that up to 22% of the code in a traditional web service module is dedicated to security checking functionality, including checks for client sequencing and parameter validation. We show that our prototype language implementation, WebGuard, enables web programmers to significantly reduce the amount of security checking code they need to develop manually. The quality of the code generated by WebGuard from formal policy specifications is competitive with the latency of handcrafted code to within a few percent.

Categories and Subject Descriptors
D.4.6 [Operating Systems]: Security and Protection -- *access controls*

General Terms
Security

Keywords
Access control, Web services.

1. INTRODUCTION
Following the introduction of standardized protocols for hypertext transfer and the debut of browsers eight years ago, the web has become a critical part of our computing infrastructure. The number of web sites around the globe has been increasing exponentially, doubling roughly every six months [17]. Many application servers, web-programming languages, toolkits, libraries, modules and application servers have been proposed to facilitate the quick construction of such services. Indeed, it is possible to rapidly create versatile web sites by combining generic software, such as a web server, a database, a scripting language and some standard libraries, with site-specific code that encodes the unique functionality of a site. The current diversity of the web and its fast growth clearly demonstrates that this modular combination approach, whether it uses AOLServer, PostgreSQL, and Tcl, or Apache, Oracle and PHP, or IIS, SQL Server and .NET, or any of the myriad other combinations of application servers and site-specific code, is immensely successful.

Yet, this modular component approach fails to address security, a critical, cross-cutting concern in networked service design. Typically, it is entirely up to the web programmer to define and enforce security policies by implementing the appropriate security checks in site-specific code. This is highly error-prone for three reasons. First, a typical web server maps all requests from all clients into a single privileged entity, rendering standard operating system and database access control mechanisms useless. For example, requests from privileged users and unprivileged users alike get handled by threads belonging to a server process; consequently, user-based file access authentication and access checks to database tuples, performed by the operating system and the database server respectively, become ineffective. The programmer must explicitly insert the right checks into the web service code to authenticate every file and database access request from every client. In essence, every web programmer must be as diligent as the writer of privileged system programs that operate with the setuid bit set [27, 8, 6]. Second, the HTTP protocol is mostly stateless, making it difficult to check, for instance, the order in which certain actions need to occur, or to check and validate input parameters to POST methods. For example, a web-based auction site, which typically supports dozens of different kinds of actions, needs to check before each operation that the user performing the action has properly visited the login page and has been properly authenticated and that the input parameters are of the right type. Finally, the security policies, as well as site implementations, change frequently. Keeping the two up to date and matched with each other is an operationally difficult task, where the omission of a check can result in a security breach. As a consequence of these three properties, even highly visible and commercially backed web sites have had difficulty ensuring the security of the interfaces they expose on the web [9].

In this paper, we present an approach for automatically enforcing security policies on web sites. The main goal of our work is to automate and componentize security and access control services for web services; that is, to take security out of the domain of the web

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SACMAT '02, June 3-4, 2002, Monterey, CA, USA.
Copyright 2002 ACM 1-58113-496-7/02/0006…$5.00.

programmer, where it is error-prone, costly and difficult, and into the domain of automatic tools guided by a security administrator. Just as web programmers pick, choose and compose modular service implementations, so should they be able to specify, compose and modify security policies without having to rewrite site-specific code. Our goals for an automated scheme for security enforcement on web services include the following properties:

- **Secure**: The reference monitor implemented by the automatic enforcement engine must correctly implement a given policy, and be insurmountable by clients. It should be suitable for mandatory access control over untrusted clients.
- **Automatic**: Security enforcement should not depend on manual intervention and discretionary conformance by site programmers. Security policies should be modular, component-based, and aspect-oriented; they need to be imposed via automatic means to reduce site development time.
- **General purpose**: The language used to express security policies should be sufficiently general to express the common security concerns for web applications. Further, it should be concise to make the expression and modification of the security policy cheap and easy.
- **Portable**: The security enforcement mechanism should support the various different web server platforms. Changing the underlying platform should require no user intervention or changes to the security policy or the site-specific code.
- **Backwards compatible**: Existing site implementations should not need to be modified to use new automatic tools. No changes to the browsers should be necessary.
- **Performance**: The latency overhead of automatic security policy enforcement should be comparable to manually secured code.

We have designed and implemented a technique for security enforcement on web sites that exhibits these properties. Specifically, in this paper, we present and evaluate a scheme for enforcing security policies on networked services via code generation and interposition. This general-purpose, portable, low overhead mechanism enables the enforcement of security policies on a web site by automatically generating the requisite access control checks from a policy specification written in a policy-specification language, and transparently interjects these checks across a web site. Typically the automatically generated enforcement code will be placed at the prologues and epilogues for each function, or script, exposed through a URL. The security enforcement code executes within the trusted context of the server host, protected from client tampering. Automatic generation of the reference monitor means that security specifications can be decoupled from the site implementation, developed and tested separately, and combined automatically. We show that our system achieves overheads within 1 to 2% of handcrafted, manually secured web services.

Our approach makes three contributions by automating the costly and error-prone task of manually inserting security checks in web services: (1) it provides a separation of the web site implementation from the high-level security policy, (2) it provides a general and versatile access control model for web services according to security policies specified in a domain-specific language, and (3) it demonstrates, through an implementation, that the requisite overhead for automatically generated security enforcement code can be less than a few percent of the overall transaction time.

The next section describes related work. Section 3 presents the overall architecture of our system and outlines our implementation WebGuard of the enforcement mechanism. Section 4 presents measurements that demonstrate that the enforcement mechanism is efficient, effective at reducing the task of the web programmer, and scales well under load. Section 5 summarizes our contributions and concludes.

2. RELATED WORK

Web systems are particular instances of networked systems, for which a wealth of access control models exist. Early work [3, 5, 15, 16, 20] has examined rigid, hierarchical models for securing the interface exposed by stand-alone operating systems against untrusted applications. Lattice-based access control models generalize the notion of security hierarchies [25]. We share with more recent work on access control models the insight that security policies need to be flexible and accommodating. Specifically, the domain and type enforcement model [1, 7] has been proposed as a practical mandatory access control system based on a partially ordered, non-hierarchical labelling system and an access mapping from correspondingly labelled execution domains. In a similar vein, role-based access control [14, 24] introduces the concept of roles for different subjects and grants access rights to subjects based on their roles. The role-based access control model has been applied to web services [2], where incoming requests are classified by the current active role of the originator. We note that this work is complementary to ours, in that these models can be expressed in our policy specification language. In this respect, our work is similar to logic-based access control frameworks [18], which describe an access control model capable of supporting different security requirements and multiple policies. We differ from previous approaches, however, in that the specification language we use naturally captures the notion of time, whereas indirect or external means are necessary to encode time-dependent behaviour and enforce sequencing constraints under non-temporal models and languages.

Related work on access control and system specification has examined how to fold temporal properties into system or security policy specifications. In [4], the researchers extend the role-based access control model with time-dependent functionality to yield Temporal RBAC (TRBAC). TRBAC supports periodic activations and deactivations of roles, and temporal dependencies among those activation and deactivations. Lamport's temporal logic of actions [21] provides a framework to reason about the time-dependent behaviour of complex software components. Finally, the guarded commands of Dijkstra [12] provide a technique for program verification. Our system shares some similarities with these approaches, in that temporal dependencies and inter-action dependencies are expressly modelled in our security policy using explicit predicates. We differ from these systems in our application domain of web services, our focus on automatically generating enforcement code, and our focus on portability, backwards-compatibility and performance.

Many others have examined the construction of efficient reference monitors for security enforcement. EM [26] provides a formal, automata-based definition of enforcement monitors that rely on execution prefixes, and SASI [13] provides an EM-based approach

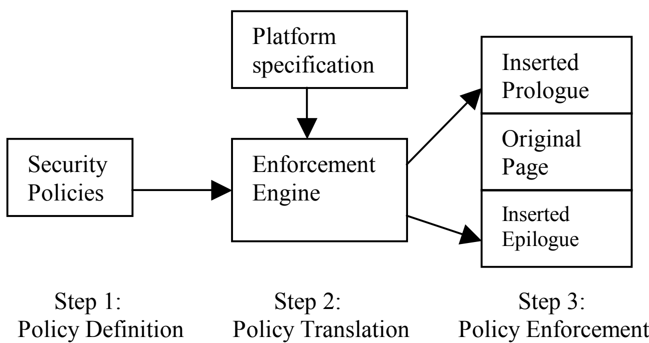
for performing security enforcement on a virtual machine interface. WebGuard builds on the prefix-based enforcement mechanism formalized in EM. It targets web services instead of a virtual machine and our implementation encodes security predicates as free-form predicates instead of as automata.

Some researchers have examined access control for web services. In particular, recent work [10, 11] has examined how to protect loosely structured XML data. Typically, however, web programmers are trusted not to write code that contains vulnerabilities by embedding the right checks into the site-specification. Guidelines offer tips on how to write secure web applications [28]. We guard against such attacks without cooperation from the web programmers and without modifying the site implementation.

3. SYSTEM ARCHITECTURE

The general-purpose mechanism we propose for enforcing security on web sites has three major components: a security policy, an enforcement engine, and the resulting, automatically generated enforcement code. First, web developers express a suitable security policy for a web site in the WebGuard security language, which is loosely based on temporal logic. This policy serves as an input to the enforcement engine, which also reads in a platform specification for the targeted web server platform. The enforcement engine generates site- and platform-specific security enforcement code from these two inputs. The generated code is then integrated into the web server. Figure 1 provides a brief outline of the end-to-end operation of our system. Because all the session related information is stored in a backend database, accessible to all servers, our security enforcement architecture works correctly across a web server cluster, found in a typical three-tiered web server. Note that our mechanism involves modifications only on the server side and places no requirements on client browsers. The security policy is independent of the site implementation as well as the platform, and thereby provides isolation from the platform and implementation, as well as portability to different platforms.

Figure 1. System Architecture



Below, we describe each of the steps involved in enforcing security on web services using this approach.

3.1 Security Policies

We use a domain-specific, special-purpose language to express security polices. Our language is draws upon features from temporal logic, a well-developed branch of modal logic. Temporal logic has been applied both to the specification and verification of program behavior [22], and to the specification of system behavior [21]. We borrow features from temporal logic to concisely capture time and sequence dependencies, which are common in web applications, in our policy specification language. For instance, users have to log in before reading their email, or users have to fill in form A before moving on to fill in form B. The WebGuard language can easily express such security requirements. Table 1 provides the simplified grammar for our policy specification language in Backus-Naur form.

```
Security-rule -> predicate-rule |
                 sequence-rule |
                 implication-rule
predicate-rule -> condition PREDICATES action
sequence-rule -> condition BEFORE action
implication-rule -> action [AND condition] IMPLIES
                    [EVENTUALLY (time)] (action| id=id)
condition -> term (OR term)*
term -> factor (AND factor)*
factor -> NOT word | word
word -> simple| (condition)
simple -> id op id | id ELEMENTOF setid |
          NOTNULL id | id | action
action -> http://sitename/path [([COOKIE | CONN] (id id)*)]
          | function | set-op
op -> =|<|>|>=|<=|!=
set-op -> ADD(id, setid)| REMOVE(id, setid) |
CREATE(setid)| DROP(setid)
```

Table 1. Grammar for the WebGuard Security Policy Specification Language

The types of access control specifications most commonly used by web applications consist of predicate rules, sequencing rules and implication rules. Predicate rules resemble guarded commands, as they specify that the action can only proceed if the condition is satisfied. Sequencing rules are used to express temporal dependencies on a user's actions in the past. Implication rules are used to specify dependencies on, or requirements from, future behaviour. For instance, implications can be used to specify that, following a user's visit to a page to initiate a transaction, either the user must visit another URL to complete the transaction, or the system ought to abort the transaction and clean up system state.

An *action* in our system corresponds either to a URL invocation initiated by a client, or to the execution of a server-side script initiated by the security module. We use familiar procedure call syntax for expressing both kinds of actions. Client-initiated URL invocations are distinguished by the prefix "*http://*", whereas server-side function calls are identified simply by the name of the library procedure. These server-side security enforcement libraries are provided by the platform-specific module, and implement commonly used functions, such as those for validating inputs, parsing cookies, and checking the cryptographic integrity of data stored in cookies or form variables.

In the discussion below, we provide a simplified running example from an e-publishing system. To authenticate a user, a web server typically will check a submitted password and issue a cryptographically encrypted authentication token. This operation

can be specified in our policy language with the following implication clause:

```
http://sitename/login(user userid, passwd
passwordid) AND
MD5Hash(passwordid) = Extract(user, "password",
user_col=userid) IMPLIES
CreateAuthToken(token_name, userid, passwordid)
```

This implication is an imperative, performed immediately when a user submits a valid password. The policy language also enables actions to be deferred for any amount of time. The EVENTUALLY clause will schedule a routine for future execution; this is typically used for cleaning up server-side state. In this example, once the policy issues an authorization token, predicate clauses can be used to enforce that only authenticated users can perform certain actions. This would be expressed in our policy specification language as simply:

```
ValidityCheck(authtoken) PREDICATES
http://sitename/rankarticle(Article ArticleID3)
```

Sequence clauses can be used to specify linear dependencies between user actions. For instance, suppose that, for a user to publish a submitted article, she needs to have read the article in the past and be a member of the site editors group. This can be naturally expressed in our security language with the following policy (for simplicity, we leave out the predicates that validate the authenticity of the submitted cookie, and concentrate on the sequence clause):

```
http://sitename/fetch_msg(Article
ArticleID1, COOKIE cookiename cookievalue)
AND (Extract(cookievalue, "userid", user)
ELEMENTOF editors)
AND ArticleID1 = ArticleID2
BEFORE
http://sitename/publish_article(Article
ArticleID2)
```

This clause also demonstrates variable scoping and argument passing, two of the more subtle aspects of our policy specification language.

Variable scoping and value binding occurs at each URL invocation. The enforcement engine identifies the user-initiated actions and generates code to record those parameters, such as ArticleID1 above, that will later be used in logic predicates (Our current implementation records *all* arguments, but a two-pass optimization or the construction of def-use chains can be used to record only the minimal set that is later used). The requirement that the published article match a previously read article is expressed by the predicate *ArticleID1= ArticleID2,* which relates the *fetch_msg* action in the past to the *publish_article* invocation. Note that the sequence clause is evaluated *after* all variable bindings have taken place; that is, *ArticleID2* is bound first to the parameters passed to the *publish_article* method, before the article equality predicate is evaluated. While this appears counter-intuitive, it results in concise specifications without auxiliary, temporary variables, and correct semantics.

Variable bindings are specified in terms of (*InputParameter VariableName*) pairs. In the example above, Article is the input parameter from the web page. Typically, it specifies a field name in a web form submitted by the HTTP POST command, or a parameter specified as part of the URL in an HTTP GET command. For instance, *q* is an input parameter to the invocation "*http://www.google.com/search?q=sacmat.*" The variable binding *http://www.google.com/search(q QueryString)* creates a new variable called *QueryString* and sets its value to "*sacmat*" when invoked as shown. While the InputParameter looks like a type specification, it is not a type but a parameter name. It relates variables used in the security policy to the web implementation and serves as the mapping function between information used in the policy and the concrete format in which it is passed from clients to the web server by the site implementation.

Web servers can get information from client side through three sources; consequently, there are three different kinds of variable definitions. The first deals with explicit parameters. Arguments that appear after a question mark in a GET URL, or those sent as query content in POST method are explicit parameters. Since they are so common, they require no special annotation in the policy. Clients can also pass information to the web server via browser cookies [19]. Such parameters are identified by a special COOKIE keyword. The information in cookies is assumed to be in a standard URL-encoded format, so the policy engine can easily retrieve and parse them – all cookies attached by the policy conform to this format. Finally, there is implicit information embedded in the connection, for example, the IP address of the client. The CONN keyword can be used to extract and bind variables to such information.

Overall, there are three main advantages to using a special-purpose language to express security policies, and to generate enforcement code automatically from security policies. First, this approach separates the policy expression from the particular web implementation. The security policies are the same no matter what web platform is used. Changes to the policy do not require modification of the site-specific code, a costly and error-prone undertaking. Second, it's natural to express web security policies in temporal logic. The interaction between a client and web services is typically sequential; the WebGuard language can naturally capture this key characteristic. Finally, the mapping between the security policy and the corresponding site is straightforward. Each exposed URL has a well-defined name, and three straightforward techniques for extracting and binding variables to all data passed to that URL.

3.2 Enforcement Engine

Our enforcement engine automates the task of converting security policies into access control code specific to a particular platform and web site. This translation process works much like a compiler. First, the enforcement engine parses the policy into an abstract syntax tree (AST), and then translates this AST into security checking code specific to a chosen platform and site implementation. To factor out dependencies on web platforms, we parameterize the enforcement engine with a translator from an abstract code generation interface to concrete web server code. This platform-specific translator converts the AST into a sequence of code in a language, such as Tcl, PHP or .NET, that is appropriate for the chosen server platform. Our implementation supports three diverse server platforms: .NET+IIS+SQL Server, AOLServer+PostgerSQL, and Apache+MySQL. The abstract interface to the translator isolates such platform dependencies from the core policy enforcement engine, and enables WebGuard to support a diverse selection of web server platforms

The code generated by the Enforcement Engine requires some run time support to carry out its tasks. For instance, user authentication and secure communication requires some common cryptographic functions for their operation. A run-time component, specific to the server platform, provides this functionality. Wherever possible, we simply redirect our runtime library calls to use the native implementation of such functions for maximum efficiency. Our runtime library needs to be loaded into the web server at start-up, but does not require any binary modifications to the server binary itself. Our work with policy translation has shown that this mechanism for automatic security enforcement is portable across multiple platforms.

3.3 Enforcement Code

The enforcement code generated by our enforcement engine is driven entirely by the security policy specification, and is thus entirely application-specific. However, there are two major axis of freedom for the implementation of this code that has a strong impact on its performance. Namely, the integration of the enforcement code with the site implementation, and the techniques it uses to represent server-side state determine the efficiency of the overall access control mechanism. In this section, we describe our implementation choices in these two areas.

All enforcement code generated by our enforcement engine operates in the prologue and epilogue code for web services. We structure our security enforcement mechanisms such that all security checks and state management operations occur at the entry and exit of web service invocations. The advantage of this approach is that it greatly simplifies the platform-specific component. Most web servers are typically structured as event-processing systems, and support interposition of event filters. Consequently, the platform specific code can register generated code snippets with the web server as event filters, and thus integrate them with the site implementation without having to parse the site-specific code and merge it with the implementation. The disadvantage of this approach is that the code we generate may recalculate or re-extract values that the site-specific code calculates or extracts anyway. We show in the evaluation section that this duplication is minimal and does not adversely affect the performance of our approach.

Typical security policies need to keep track of some, possibly extensive, state in order to make informed access control decisions. There are two major alternatives for state management in web services. A server-side approach stores all information, in particular, the variable bindings introduced by the security policy, in a permanent database on the server. This approach has the advantage that the state is easy to manipulate, secure from client tampering, and centralized such that a user accessing the same service via multiple devices perceives a single, consistent view. It has the disadvantage that it may pose a bottleneck on the server, and may not scale well. An alternative is to push and distribute the security-relevant state onto the clients. For instance, cookies could be used to store per-user histories and variable bindings. Naturally, any information kept by the clients needs to be made tamperproof and resilient against replay attacks, through the judicious use of encryption and digital signatures. The advantage of this approach is that it scales well under load. The disadvantage, however, is that a user who accesses the same service via two different browsers may see two different behaviours, depending on the security state stored in that browser. Worse, this approach enables users to destroy security-critical information, and possibly mount denial-of-service attacks. Finally, some users reject cookies, reducing the applicability of this technique. We chose the server-side approach for the security code we generate automatically. All information is kept securely in a database on the server side. For every valid session, the database keeps track of the recent, relevant security state that may be required when the user visits certain web pages in the future that use past variable bindings in their predicates. Outdated sessions are deleted periodically, so this state table can be kept to a reasonable size. In the next section, we evaluate the efficiency of our implementation mechanisms.

4. EVALUATION

This section evaluates the effectiveness, performance and scalability of our automatic security enforcement mechanism. These experiments are performed on OpenACS and .NET platforms. For OpenACS, we first examine the amount of security checking code in a popular web portal, and show that security checking poses a substantial burden for typical web programmers, and that our security policy specification mechanism can improve this significantly. Next, we give the average access time of a web page with and without our security enforcement, and show that the absolute overhead of our services is small. Finally, we compare the average access time of web services using handcrafted security checks versus access controls automatically generated by our mechanism. For the .NET platform, we first examine the average access time of a web page with and without security enforcement when the web server is under load. Then we also compare the overhead of automatically generated code with the hand-written one. These experiments results show that our approach is an effective and efficient way to enforce security on web services.

The first four experiments are performed on OpenACS-3.2.5+AOLserver-3.4+ PostgreSQL-7.1.2 on Linux 2.4.3-20mdk [23]. OpenACS is an open source portal and toolkit for scalable community-oriented web applications that has a large, active user community and is in use at many commercial sites. It is based on the highly optimized web server AOLserver, which is in production use at AOL. The hardware configuration consists of a 1.7 GHz Pentium server, with 1GB memory, 256Mbyte cache, 40GB hard disk, with two network interfaces that support a 100Mbit/sec LAN and 56K modem, respectively. Clients are similarly configured. All experiments indicate the mean of 10 measurements.

The first experiment shows that code for security checking accounts for a substantial fraction of a web site implementation. Table 2 provides a breakdown of the number of non-blank, non-comment

Table 2. Manually written security enforcement code accounts for a substantial fraction of programmer effort with current state of the art. The table shows a breakdown of total number of non-blank lines for site functionality and the percentage dedicated to security checking and enforcement.

Module Name	Security lines	Total lines	Percentage
Bulletin Board	1345	9372	14.35%
Chat	183	842	21.7%
User Groups	178	2708	6.57%
Total	1706	12922	13.20%

lines dedicated to security checking for three central modules in the OpenACS toolkit. Roughly 7 to 22% of the code in a web site exists solely for security enforcement. In the case of the OpenACS toolkit, all of these lines were written manually. While there is some structure to this code, it is neither uniform nor simple, thereby rendering a security audit of the toolkit difficult.

We provide a more detailed breakdown of the security related code for one of the modules, and quantify the impact of automatically generating security enforcement code. Table 3 shows the number of security-related lines in the chat module. In this classification, we manually categorized the security code into "temporal" and "non-temporal" categories. The former refers to security checking code that relies on or enforces a certain sequence between service invocations, while the latter captures the amount of code dedicated to input validation and authentication. First, this table shows that web programmers spend roughly half their security enforcement effort on ensuring a desired temporal sequence on top of the stateless HTTP protocol. Our security policy specification language can, trivially and cogently, express such dependencies, as well as the more ad hoc, site specific security code in the non-temporal category. Overall, our automatic code generation can eliminate 56% of the code dedicated to security checking. This code is instead replaced by a specification of 29 lines.

Table 3. Breakdown of security-related code in the Chat module. Our approach can eliminate more than half of the manual effort required to secure a site.

Original System	Total lines	842
	Total lines for security checks	183
	Total lines for temporal checks	97
	Total lines for non-temporal checks	86
WebGuard	Total lines of security enforcement code in our policy	29
	Total lines written manually using automatic check generation	80
	Percent improvement in manual effort	56.3%

Interestingly, this classification effort revealed some instances of both redundant and missing checks in the OpenACS toolkit. As the toolkit evolved over time and was edited by multiple authors, some security checks were performed redundantly using alternative mechanisms, while others were dropped. Automatic enforcement of policies across a web site reduces the possibility for such incidents by decoupling the security policy from the implementation.

The next experiment examines the access latency with and without our automatic enforcement mechanism using a microbenchmark, and shows that our enforcement strategy does not imply a large overhead. The microbenchmark consists of a policy based on a sequence clause, which gets compiled down to two code snippets, to be executed from within the function prologue and epilogue, respectively. The web service being invoked is a null service that does not access the database; consequently, there are no operations that would mask the latency that code our enforcement engine generates. The prologue code performs a database lookup and a corresponding security check, while the epilogue performs a database insertion. Figure 2 shows the overhead of the overall benchmark, as well as the overhead of the prologue and epilogue in isolation. The measurements were performed under two separate network conditions. The LAN measurements refer to a 100 Mbit/sec Ethernet, while the Modem measurements were collected using a 56K serial modem. We report the observed client latency for 500 consecutive operations, normalized to the case where no security is performed on the server. Overall, the overhead associated with a security constraint is 2% for the modem case, where the latency of our security mechanism is negligible compared to network latencies. The LAN case represents a worst-case scenario for our scheme, yet the overhead we observe is less than 12% over an insecure, null call. Given that the standard deviation in these network measurements is around 5%, the latency of our approach is low, compared to an insecure web access.

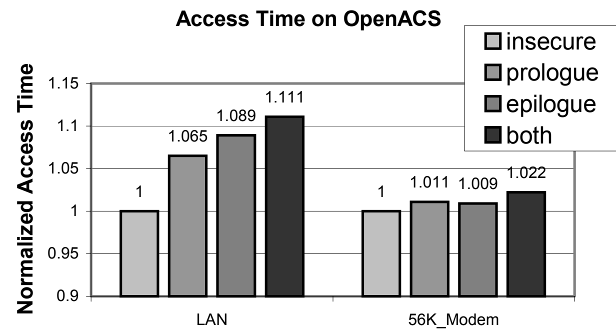


Figure 2. Access time for a single page access with and without security enforcement. The overhead of the automatically generated security checks is low: less than 12% for a 100 Mbit LAN, and negligible for modem users.

Next we examine the performance of our system under load to determine its scalability. Figure 3 shows the result of experiments that perform concurrent accesses on a web service from multiple users. We use five machines in the same lab with the web server connected by a 100Mbit/sec LAN to generate a continuous synthetic load on the web servers simultaneously. We simulate 5, 15, 25, and 50 concurrent clients, while all other experiment parameters are kept the same as the previous experiment. The access time for the original, insecure, null web service has been normalized to 1. Overall, the overhead introduced by automated security enforcement is within 18%. The sustained clickthrough rate for the server in the experimental set-up with 50 concurrent hosts corresponds to roughly 5.7 million hits/day.

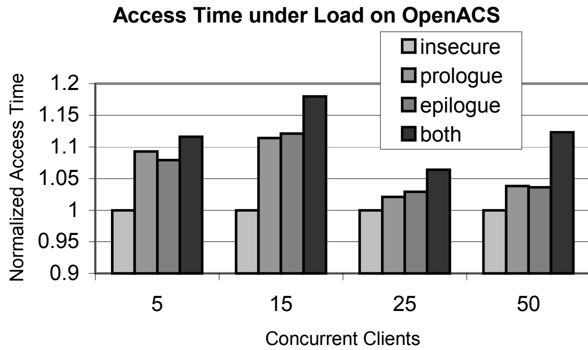


Figure 3. Average access time for multiple concurrent clients from LAN, with and without security enforcement. The overhead of automatically generated security checks is low, less than 18%, even under load.

Finally, we compare the overhead of the security checking code generated by our enforcement engine to the overhead of handcrafted security enforcement code in the OpenACS toolkit. We examine an authentication benchmark, where a user's request to enter a chat room is checked for proper authentication and the request parameters are validated. The page is repeatedly accessed 200 times from the client. Figure 4 shows that the overhead of our automatic approach is negligible.

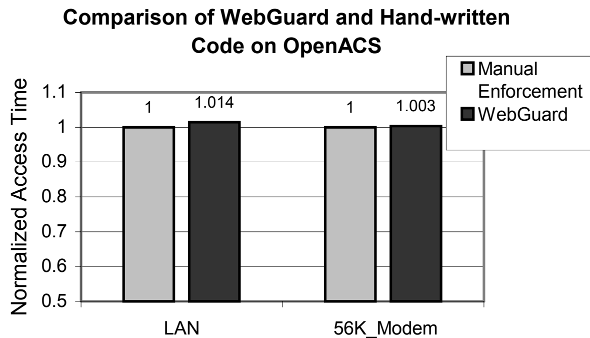


Figure 4. The overhead of automatic security enforcement compared to manually implemented security checks for an authentication microbenchmark under OpenACS.

We next examine the .NET platform and show that the WebGuard performs well across different web servers. We examine the scalability and comparing automatically generated code with hand-written code. For the scalability experiment, we use the same five machines as above to simulate concurrent client access from a LAN. The web server is running on a computer with the same hardware configuration as that running OpenACS, but with Windows 2000 Server, IIS5.0 and SQL Server 2000. Figure 5 shows the impact of security enforcement, normalized to an insecure implementation of the web page. The overhead introduced by automatically generated security enforcement code on Microsoft Server 2000 + .NET platform is less than 20% under load.

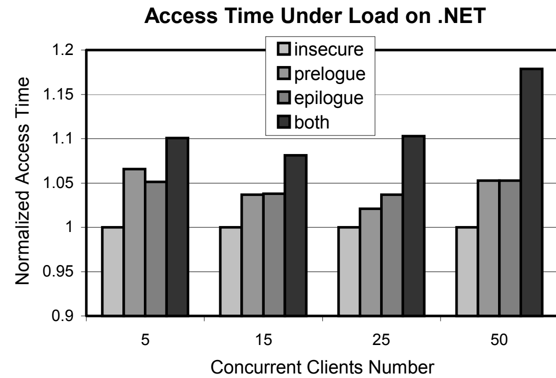


Figure5. Average access time for multiple con current clients from LAN, with and without security enforcement. The overhead of automatically generated security checks is less than 20% under load.

To compare the automatically generated code with hand-written code, we use a BEFORE rule micro-benchmark, where a user has to read one article before evaluate it. The experiment is performed inside a 100Mbps LAN. As above, the result is average of 10 measurements; each of them is 100 times continuous request to the web page. The result in figure 6 shows our mechanism also works well on .NET platform. The manual enforcement time is normalized to one. The overhead introduced by auto-generated code over manually written code is less than 5%.

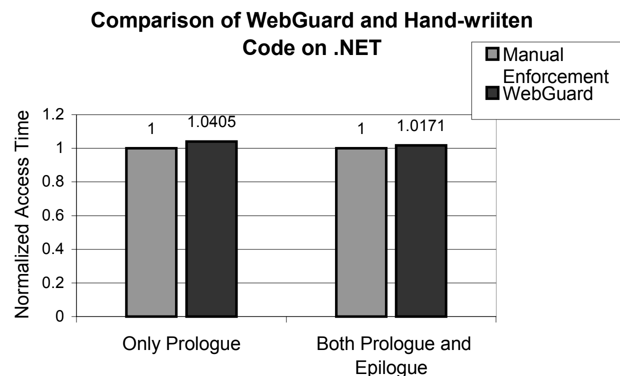


Figure 6. The overhead of automatic security enforcement using WebGuard compared to manually implemented security checks on .NET platform.

Two high-level observations account for the low overhead and high performance of our implementation. First, the temporal features in our language match the web domain well. Consequently, the quality of the access control code generated from policies expressed in this language is good. Second, the platform-specification enables our access control mechanism to take advantage of native functionality where possible, thus allowing the automatically generated code to use fast primitives while retaining portability.

The experiments above demonstrate that implementing security enforcement accounts for a large fraction of web site development. Automatic enforcement mechanisms, such as the approach presented here, can save substantial programming effort,

reduce errors, and perform about as well as hand-optimized code. This mechanism also enhances portability and can support different web server platforms.

5. CONCLUSION

This paper outlines an automatic, efficient and portable scheme for enforcing security policies on web services. We present quantitative measurements from an open source portal implementation to show that manual security checking code constitutes a sizeable fraction of a web site implementation. Our automatic approach saves the web developers from tedious and error-prone work, and improves the security of web sites by decoupling security policies from site-specific functionality. Our temporal logic-based policy specification is versatile, and naturally captures and enforces the linear flow between web pages that is a part of the web navigation model, but is not directly expressed or enforced by any other mechanism. Through a well-defined API, our code translation engine is portable across multiple server, operating system and database platforms. Our implementation shows that this mechanism can be realized efficiently with little overhead. Overall, we hope that effective, automated tools based on formally specified security policies can lead to quicker, cheaper and more secure web development.

ACKNOWLEDGMENTS

We would like to thank Fred B. Schneider and the anonymous referees for their valuable feedback on earlier drafts of this paper.

REFERENCES

[1] L. Badger and D. F. Sterne and D. L. Sherman and K. M. Walker. Practical Domain and Type Enforcement for UNIX. In *IEEE Symposium on Security and Privacy,* Oakland, California, May 1995, 66-77.

[2] J. Barkley, A. Cincotta, D. Ferraiolo, S. Gavrila, and D.R. Kuhn. Role Based Access Control for the World Wide Web. In *Proceedings of the 20th National Information System Security Conference*, NIST/NSA, 1997.

[3] D. Bell and L. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report MTR-1997, MITRE, Bedford, MA, 1975.

[4] E. Bertino, P. A. Bonatti, E. Ferrari. TRBAC: A Temporal Role-based Access Control Model. In *Proceedings of the Fifth ACM Workshop on Role-based Access Control.* July 2000.

[5] K. J. Biba. Integrity Constraints for Secure Computer Systems. Technical Report ESD-TR76-372, USAF Electronic System Division, Bedford, Massachusetts, April 1977.

[6] M. Bishop. How to Write a Setuid Program. ;login: The USENIX Association Newsletter, 12(1):5--11, Jan./Feb. 1987.

[7] W.E. Boebert and R.Y. Kain, A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conf.*, Gaithersburg, MD, 1985.

[8] S. Bunch. The Setuid Feature in UNIX and Security. In *Proceedings of Tenth National Computer Security Conference*, September 1987, 245-253.

[9] CNet. Microsoft plugs Hotmail security hole. http://news.cnet.com/news/0-1003-200-6941020.html, August 2001.

[10] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati. Securing XML Documents. In *Proceedings. of the 2000 International Conference on Extending Database Technology*, Konstanz, Germany, March 27-31, 2000.

[11] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati. XML Access Control Systems: A Component-Based Approach. In *Proceedings of IFIP WG11.3 Working Conf. on Database Security*, Schoorl, The Netherlands, August 21-23, 2000.

[12] E.W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communication of the ACM*, vol. 18, num. 8, 453-457, Aug 1975.

[13] U. Erlingsson and F. B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, Caledon Hills, Sept.1999.

[14] D. Ferraiolo and R. Kuhn. Role-Based Access Control. In *Proceedings of the 15th National Computer Security Conference*, Baltimore, Maryland, 1992, 554-563.

[15] J. Goguen and J.Meseguer. Security Policies and Security Models. In *Proceedings of the 1982 IEEE Symp. on Research in Security and Privacy*, IEEE Computer Society Press, 1982.

[16] G.Graham and P.Denning. Protection: Principles and Practice. In *Proceeding of Spring Joint Computer Conf.*, AFIPS Press, 1972.

[17] M. Gray. Web Growth Summary. http://www.mit.edu/people/mkgray/net/, December 1997.

[18] S. Jajodia, P. Samarati, and V.S. Subrahmanian. A Logical Language for Expressing Authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. Oakland, CA, USA: IEEE Press, 1997. 31-42.

[19] D. Kristol, L. Montulli. HTTP State Management Mechanism. Request for Comments RFC-2965, Internet Engineering Task Force, October 2000.

[20] B. Lampson. Protection. In *Proceedings of 5th Princeton Symposium on Information Sciences and Systems*, March 1971. Reprinted in *ACM Operating Systems Review*, 8(1) 1974.

[21] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems,* 16(3), 872-923, 1994.

[22] Z. Manna and A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag: Heidelberg, Germany, 1992.

[23] OpenACS Documentation. http://www.openacs.org/, December 2001.

[24] R. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role Based Access Control Models. *IEEE Computer*, 29(2), Feb.1996.

[25] R. S. Sandhu. Lattice-based Access Control Models. *IEEE Computer*, 26(11): 9--19, November 1993.

[26] F. B. Schneider. Enforceable Security Policies. TR 98-1664, Dept. of Computer Science, Cornell Univ., Ithaca, NY, 1998.

[27] D. J. Thomsen and J. T. Haigh. A Comparison of Type Enforcement and Unix Setuid Implementation of Well-formed Transactions. In *Proceedings of Sixth Annual Computer Security Applications Conf.,* Tucson, Arizona, December 1990, 304-312.

[28] Whitehat, Inc. Secure Web Programming, http://www.whitehatsec.com/, December 2001.