Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications

Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich
Computer Systems Laboratory *CSAIL*
Stanford University *MIT*
{*mwdalton, kozyraki*}*@stanford.edu* *nickolai@csail.mit.edu*

Abstract

This paper presents *Nemesis*, a novel methodology for mitigating authentication bypass and access control vulnerabilities in existing web applications. *Authentication attacks* occur when a web application authenticates users unsafely, granting access to web clients that lack the appropriate credentials. *Access control attacks* occur when an access control check in the web application is incorrect or missing, allowing users unauthorized access to privileged resources such as databases and files. Such attacks are becoming increasingly common, and have occurred in many high-profile applications, such as IIS [10] and WordPress [31], as well as 14% of surveyed web sites [30]. Nevertheless, none of the currently available tools can fully mitigate these attacks.

Nemesis *automatically determines* when an application safely and correctly authenticates users, by using Dynamic Information Flow Tracking (DIFT) techniques to track the flow of user credentials through the application's language runtime. Nemesis combines authentication information with programmer-supplied access control rules on files and database entries to *automatically ensure* that only properly authenticated users are granted access to any privileged resources or data. A study of seven popular web applications demonstrates that a prototype of Nemesis is effective at mitigating attacks, requires little programmer effort, and imposes minimal runtime overhead. Finally, we show that Nemesis can also improve the precision of existing security tools, such as DIFT analyses for SQL injection prevention, by providing runtime information about user authentication.

1 Introduction

Web applications are becoming increasingly prevalent because they allow users to access their data from any computer and to interact and collaborate with each other. However, exposing these rich interfaces to anyone on the internet makes web applications an appealing target for attackers who want to gain access to other users' data or resources. Web applications typically address this problem through *access control*, which involves *authenticating* users that want to gain access to the system, and ensuring that a user is properly *authorized* to perform any operation the server executes on her behalf. In theory, this approach should ensure that unauthorized attackers cannot subvert the application.

Unfortunately, experience has shown that many web applications fail to follow these seemingly simple steps, with disastrous results. Each web application typically deploys its own authentication and access control framework. If any flaw exists in the authentication system, an authentication bypass attack may occur, allowing attackers to become authenticated as a valid user without having to present that user's credentials, such as a password. Similarly, a single missing or incomplete access control check can allow unauthorized users to access privileged resources. These attacks can result in the complete compromise of a web application.

Designing a secure authentication and access control system in a web application is difficult. Part of the reason is that the underlying file system and database layers perform operations with the privileges of the web application, rather than with privileges of a specific web application user. As a result, the web application must have the superset of privileges of all of its users. However, much like a Unix `setuid` application, it must explicitly check if the requesting user is authorized to perform each operation that the application performs on her behalf; otherwise, an attacker could exploit the web application's privileges to access unauthorized resources. This approach is ad-hoc and brittle, since these checks must be sprinkled throughout the application code whenever a resource is accessed, spanning code in multiple modules written by different developers over a long period of time. It is hard for developers to keep track of all the security policies that have to be checked. Worse yet, code written for other applications

or third-party libraries with different security assumptions is often reused without considering the security implications. In each case, the result is that it's difficult to ensure the correct checks are always performed.

It is not surprising, then, that authentication and access control vulnerabilities are listed among the top ten vulnerabilities in 2007 [17], and have been discovered in high-profile applications such as IIS [10] and WordPress [31]. In 2008 alone, 168 authentication and access control vulnerabilities were reported [28]. A recent survey of real-world web sites found that over 14% of surveyed sites were vulnerable to an authentication or access control bypass attack [30].

Despite the severity of authentication or authorization bypass attacks, no defensive tools currently exist to automatically detect or prevent them. The difficulty in addressing these attacks stems from the fact that most web applications implement their own user authentication and authorization systems. Hence, it is hard for an automatic tool to ensure that the application properly authenticates all users and only performs operations for which users have the appropriate authorization.

This paper presents *Nemesis*,[1] a security methodology that addresses these problems by *automatically tracking* when user authentication is performed in web applications without relying on the safety or correctness of the existing code. Nemesis can then use this information to *automatically enforce* access control rules and ensure that only authorized web application users can access resources such as files or databases. We can also use the authentication information to improve the precision of other security analyses, such as DIFT-based SQL injection protection, to reduce their false positive rate.

To determine how a web application authenticates users, Nemesis uses Dynamic Information Flow Tracking (DIFT) to track the flow of user credentials, such as a username and password, through the application code. The key insight is that most applications share a similar high-level design, such as storing usernames and passwords in a database table. While the details of the authentication system, such as function names, password hashing algorithms, and session management vary widely, we can nonetheless determine when an application authenticates a user by keeping track of what happens to user credentials at runtime. Once Nemesis detects that a user has provided appropriate credentials, it creates an additional HTTP cookie to track subsequent requests issued by the authenticated user's browser. Our approach does not require the behavior of the application to be modified, and does not require any modifications to the application's existing authentication and access control system. Instead, Nemesis is designed to secure legacy applications without requiring them to be rewritten.

To prevent unauthorized access in web applications, Nemesis combines user authentication information with authorization policies provided by the application developer or administrator in the form of access control rules for various resources in the application, such as files, directories, and database entries. Nemesis then automatically ensures that these access control rules are enforced at runtime whenever the resource is accessed by an (authenticated) user. Our approach requires only a small amount of work from the programmer to specify these rules—in most applications, less than 100 lines of code. We expect that explicitly specifying access control rules per-resource is less error-prone than having to invoke the access control check each time the resource is accessed, and having to enumerate all possible avenues of attack. Furthermore, in applications that support third-party plugins, these access control rules need only be specified once, and they will automatically apply to code written by all plugin developers.

By allowing programmers to explicitly specify access control policies in their applications, and by tying the authentication information to runtime authorization checks, Nemesis prevents a wide range of authentication and access control vulnerabilities seen in today's applications. The specific contributions of this paper are as follows:

- We present Nemesis, a methodology for inferring authentication and enforcing access control in existing web applications, while requiring minimal annotations from the application developers.
- We demonstrate that Nemesis can be used to prevent authentication and access control vulnerabilities in modern web applications. Furthermore, we show that Nemesis can be used to prevent false positives and improve precision in real-world security tools, such as SQL injection prevention using DIFT.
- We implement a prototype of Nemesis by modifying the PHP interpreter. The prototype is used to collect performance measurements and to evaluate our security claims by preventing authentication and access control attacks on real-world PHP applications.

The remainder of the paper is organized as follows. Section 2 reviews the security architecture of modern web applications, and how it relates to common vulnerabilities and defense mechanisms. We describe our authentication inference algorithm in Section 3, and discuss our access control methodology in Section 4. Our PHP-based prototype is discussed in Section 5. Section 6 presents our experimental results, and Section 7 discusses future work. Finally, Section 8 discusses related work and Section 9 concludes the paper.

[1] Nemesis is the Greek goddess of divine indignation and retribution, who punishes excessive pride, evil deeds, undeserved happiness, and the absence of moderation.

2 Web Application Security Architecture

A key problem underlying many security vulnerabilities is that web application code executes with full privileges while handling requests on behalf of users that only have limited privileges, violating the principle of least privilege [11]. Figure 1 provides a simplified view of the security architecture of typical web applications today. As can be seen from the figure, the web application is performing file and database operations on behalf of users using its own credentials, and if attackers can trick the application into performing the wrong operation, they can subvert the application's security. Web application security can thus be viewed as an instance of the confused deputy problem [9]. The rest of this section discusses this architecture and its security ramifications in more detail.

2.1 Authentication Overview

When clients first connect to a typical web application, they supply an application-specific username and password. The web application then performs an authentication check, ensuring that the username and password are valid. Once a user's credentials have been validated, the web application creates a login session for the user. This allows the user to access the web application without having to log in each time a new page is accessed. Login sessions are created either by placing authentication information directly in a cookie that is returned to the user, or by storing authentication information in a session file stored on the server and returning a cookie to the user containing a random, unique session identifier. Thus, a user request is deemed to be authenticated if the request includes a cookie with valid authentication information or session identifier, or if it directly includes a valid username and password.

Once the application establishes a login session for a user, it allows the user to issue requests, such as posting comments on a blog, which might insert a row into a database table, or uploading a picture, which might require a file to be written on the server. However, there is a *semantic gap* between the user authentication mechanism implemented by the web application, and the access control or authorization mechanism implemented by the lower layers, such as a SQL database or the file system. The lower layers in the system usually have no notion of application-level users; instead, database and file operations are usually performed with the privileges and credentials of the web application itself.

Consider the example shown in Figure 1, where the web application writes the file uploaded by user Bob to the local file system and inserts a row into the database to keep track of the file. The file system is not aware of any authentication performed by the web application or web server, and treats all operations as coming from the web application itself (e.g. running as the Apache user in Unix). Since the web application has access to every user's file, it must perform internal checks to ensure that Bob hasn't tricked it into overwriting some other user's file, or otherwise performing an unauthorized operation. Likewise, database operations are performed using a per-web application database username and password provided by the system administrator, which authenticates the web application as user `webdb` to MySQL. Much like the filesystem layer, MySQL has no knowledge of any authentication performed by the web application, interpreting all actions sent by the web application as coming from the highly-privileged `webdb` user.

2.2 Authentication & Access Control Attacks

The fragile security architecture in today's web applications leads to two common problems, authentication bypass and access control check vulnerabilities.

Authentication bypass attacks occur when an attacker can fool the application into treating his or her requests as coming from an authenticated user, without having to present that user's credentials, such as a password. A typical example of an authentication bypass vulnerability involves storing authentication state in an HTTP cookie without performing any server-side validation to ensure that the client-supplied cookie is valid. For example, many vulnerable web applications store only the username in the client's cookie when creating a new login session. A malicious user can then edit this cookie to change the username to the administrator, obtaining full administrator access. Even this seemingly simple problem affects many applications, including PHP iCalendar [20] and phpFastNews [19], both of which are discussed in more detail in the evaluation section.

Access control check vulnerabilities occur when an access check is missing or incorrectly performed in the application code, allowing an attacker to execute server-side operations that she might not be otherwise authorized to perform. For example, a web application may be compromised by an invalid access control check if an administrative control panel script does not verify that the web client is authenticated as the admin user. A malicious user can then use this script to reset other passwords, or even perform arbitrary SQL queries, depending on the contents of the script. These problems have been found in numerous applications, such as PhpStat [21].

Authentication and access control attacks often result in the same unfettered file and database access as traditional input validation vulnerabilities such as SQL injection and directory traversal. However, authentication and access control bugs are more difficult to detect, because their

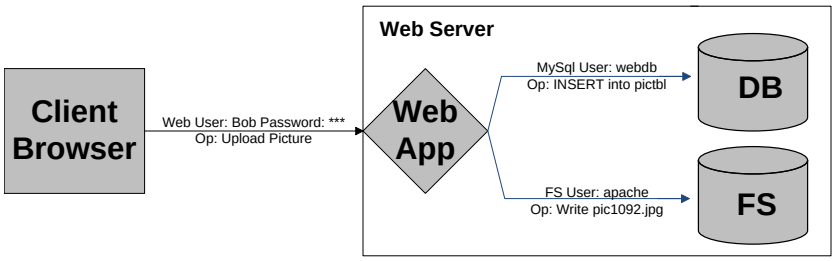

Figure 1: The security architecture of typical web applications. Here, user Bob uploads a picture to a web application, which in turn inserts data into a database and creates a file. The user annotation above each arrow indicates the credentials or privileges used to issue each operation or request.

logic is application-specific, and they do not follow simple patterns that can be detected by simple analysis tools.

2.3 Other Web Application Attacks

Authentication and access control also play an important, but less direct role, in SQL injection [27], command injection, and directory traversal attacks. For example, the PHP code in Figure 2 places user-supplied search parameters into a SQL query without performing any sanitization checks. This can result in a SQL injection vulnerability; a malicious user could exploit it to execute arbitrary SQL statements on the database. The general approach to addressing these attacks is to validate all user input before it is used in any filesystem or database operations, and to disallow users from directly supplying SQL statements. These checks occur throughout the application, and any missing check can lead to a SQL injection or directory traversal vulnerability.

However, these kinds of attacks are effective only because the filesystem and database layers perform all operations with the privilege level of the web application rather than the current authenticated webapp user. If the filesystem and database access of a webapp user were restricted only to the resources that the user should legitimately access, input validation attacks would not be effective as malicious users would not be able not leverage these attacks to access unauthorized resources.

Furthermore, privileged users such as site administrators are often allowed to perform operations that could be interpreted as SQL injection, command injection, or directory traversal attacks. For example, popular PHP web applications such as DeluxeBB and phpMyAdmin allow administrators to execute arbitrary SQL commands. Alternatively, code in Figure 2 could be safe, as long as only administrative users are allowed to issue such search queries. This is the very definition of a SQL injection attack. However, these SQL injection vulnerabilities can only be exploited if the application fails to check that the user is authenticated as the administrator before issuing the SQL query. Thus, to properly judge whether a SQL injection attack is occurring, the security system must know which user is currently authenticated.

3 Authentication Inference

Web applications often have buggy implementations of authentication and access control, and no two applications have the exact same authentication framework. Rather than try to mandate the use of any particular authentication system, Nemesis prevents authentication and access control vulnerabilities by automatically inferring when a user has been safely authenticated, and then using this authentication information to automatically enforce access control rules on web application users. An overview of Nemesis and how it integrates into a web application software stack is presented in Figure 3. In this section, we describe how Nemesis performs *authentication inference*.

3.1 Shadow Authentication Overview

To prevent authentication bypass attacks, Nemesis must infer when authentication has occurred without depending on the correctness of the application authentication system., which are often buggy or vulnerable. To this end, Nemesis constructs a *shadow authentication system* that works alongside the application's existing authentication framework. In order to infer when user authentication has safely and correctly occurred, Nemesis requires the application developer to provide one annotation—namely, where the application stores user names and their known-good passwords (e.g. in a database table), or what external function it invokes to authenticate users (e.g. using LDAP or OpenID). Aside from this annotation, Nemesis is agnostic to the specific hash function or algorithm used to validate user-supplied credentials.

To determine when a user successfully authenticates, Nemesis uses Dynamic Information Flow Tracking (DIFT). In particular, Nemesis keeps track of two bits

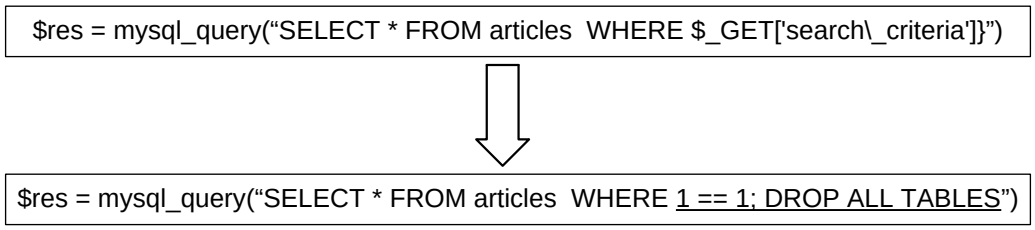

Figure 2: Sample PHP code vulnerable to SQL injection, and the resulting query when a user supplies the underlined, malicious input.

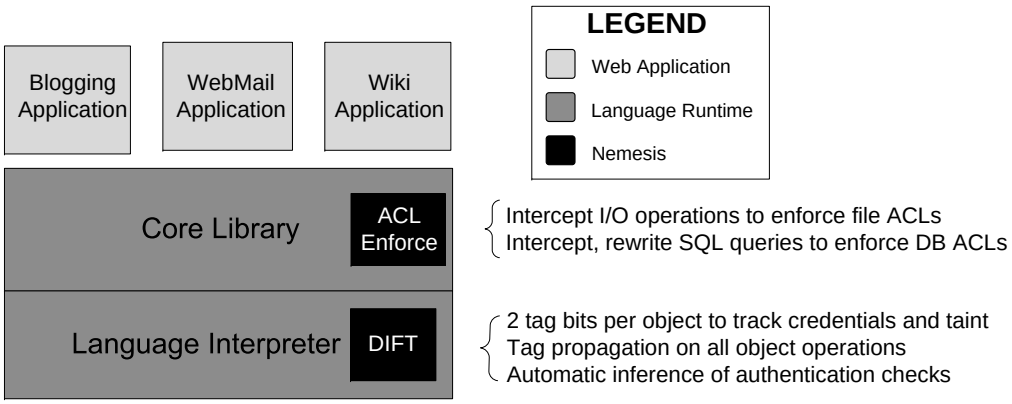

Figure 3: Overview of Nemesis system architecture

of *taint* for each data item in the application—a "credential" taint bit, indicating whether the data item represents a known-good password or other credential, and a "user input" taint bit, indicating whether the data item was supplied by the user as part of the HTTP request. User input includes all values supplied by the untrusted client, such as HTTP request headers, cookies, POST bodies, and URL parameters. Taint bits can be stored either per object (e.g., string), or per byte (e.g., string characters), depending on the needed level of precision and performance.

Nemesis must also track the flow of authentication credentials and user input during runtime code execution. Much like other DIFT systems [8, 15, 16], this is done by performing taint propagation in the language interpreter. Nemesis propagates both taint bits at runtime for all data operations, such as variable assignment, load, store, arithmetic, and string concatenation. The propagation rule we enforce is *union*: a destination operand's taint bit is set if it was set in any of the source operands. Since Nemesis is concerned with inferring authentication rather than addressing covert channels, implicit taint propagation across control flow is not considered. The rest of this section describes how Nemesis uses these two taint bits to infer when successful authentication has taken place.

3.2 Creating a New Login Session

Web applications commonly authenticate a new user session by retrieving a username and password from a storage location (typically a database) and comparing these credentials to user input. Other applications may use a dedicated login server such as LDAP or Kerberos, and instead defer all authentication to the login server by invoking a special third-party library authentication function. We must infer and detect both of these authentication types.

As mentioned, Nemesis requires the programmer to specify where the application stores user credentials for authentication. Typical applications store password hashes in a database table, in which case the programmer should specify the name of this table and the column names containing the user names and passwords. For applications that defer authentication to an external login server, the programmer must provide Nemesis with the name of the authentication function (such as ldap_login), as well as which function arguments represent the username and password, and what value the function returns upon authentication success. In either case, the shadow authentication system uses this information to determine when the web application has safely authenticated a user.

3.2.1 Direct Password Authentication

When an application performs authentication via direct password comparisons, the application must read the username and password from an authentication storage location, and compare them to the user-supplied authentication credentials. Whenever the authentication storage location is read, our shadow authentication system records the username read as the current user under authentication, and sets the "credential" taint bit for the password string. In most web applications, a client can only authenticate as a single user at any given time. If an application allows clients to authenticate as multiple users at the same time, Nemesis would have to be extended to keep track of multiple candidate usernames, as well as multiple "credential" taint bits on all data items. However, we are not aware of a situation in which this occurs in practice.

When data tagged as "user input" is compared to data tagged as "credentials" using string equality or inequality operators, we assume that the application is checking whether a user-supplied password matches the one stored in the local password database. If the two strings are found to be equal, Nemesis records the web client as authenticated for the candidate username. We believe this is an accurate heuristic, because known-good credentials are the only objects in the system with the "credential" taint bit set, and only user input has the "user input" taint bit set. This technique even works when usernames and passwords are supplied via URL parameters (such as "magic URLs" which perform automatic logins in HotCRP) because all values supplied by clients, including URL parameters, are tagged as user input.

Tag bits are propagated across all common operations, allowing Nemesis to support standard password techniques such as cryptographic hashes and salting. Hashing is supported because cryptographic hash functions consist of operations such as array access and arithmetic computations, all of which propagate tag bits from inputs to outputs. Similarly, salting is supported because prepending a salt to a user-supplied password is done via string concatenation, an operation that propagates tag bits from source operands to the destination operand.

This approach allows us to infer user authentication by detecting when a user input string is compared and found to be equal to a password. This avoids any internal knowledge of the application, requiring only that the system administrator correctly specify the storage location of usernames and passwords. A web client will only be authenticated by our shadow authentication system if they know the password, because authentication occurs only when a user-supplied value is equal to a known password. Thus, our approach does not suffer from authentication vulnerabilities, such as allowing a user to log in if a magic URL parameter or cookie value is set.

3.2.2 Deferred Authentication to a Login Server

We use similar logic to detect authentication when using a login server. The web client is assumed to be authenticated if the third-party authentication function is called with a username and password marked as "user input", and the function returns success. In this case, Nemesis sets the authenticated user to the username passed to this function. Nemesis checks to see if the username and password passed to this function are tainted in order to distinguish between credentials supplied by the web client and credentials supplied internally by the application. For example, phpMyAdmin uses MySQL's built-in authentication code to both authenticate web clients, and to authenticate itself to the database for internal database queries [23]. Credentials used internally by the application should not be treated as the client's credentials, and Nemesis ensures this by only accepting credentials that came from the web client. Applications that use single sign-on systems such as OpenID must use deferred authentication, as the third-party authentication server (e.g., OpenID Provider) performs the actual user authentication.

3.3 Resuming a Previous Login Session

As described in Section 2.1, web applications create login sessions by recording pertinent authentication information in cookies. This allows users to authenticate once, and then access the web application without having to authenticate each time a new page is loaded. Applications often write their own custom session management frameworks, and session management code is responsible for many authentication bypass vulnerabilities.

Fortunately, Nemesis does not require any per-application customization for session management. Instead, we use an entirely separate session management framework. When Nemesis infers that user authentication has occurred (as described earlier in this section), a new cookie is created to record the shadow authentication credentials of the current web client. We do not interpret or attempt to validate any other cookies stored and used by the web application for session management. For all intents and purposes, session management in the web application and Nemesis are orthogonal. We refer to the cookie used for Nemesis session management as the *shadow cookie*. When Nemesis is presented with a valid shadow cookie, the current shadow authenticated user is set to the username specified in the cookie.

Shadow authentication cookies contain the shadow authenticated username of the current web user and an HMAC of the username computed using a private key kept on the server. The user cannot edit or change their shadow authentication cookie because the username HMAC will no longer match the username itself, and the user does

not have the key used to compute the HMAC. This cookie is returned to the user, and stored along with any other authentication cookies created by the web application.

Our shadow authentication system detects a user safely resuming a prior login session if a valid shadow cookie is presented. The shadow authentication cookie is verified by recomputing the cookie HMAC based on the username from the cookie. If the recomputed HMAC and the HMAC from the cookie are identical, the user is successfully authenticated by our shadow authentication system. Nemesis distinguishes between shadow cookies from multiple applications running on the same server by using a different HMAC key for each application, and including a hash derived from the application's HMAC key in the name of the cookie.

In practice, when a user resumes a login session, the web application will validate the user's cookies and session file, and then authorize the user to access a privileged resource. When the privileged resource access is attempted, Nemesis will examine the user's shadow authentication credentials and search for valid shadow cookies. If a valid shadow cookie is found and verified to be safe, the user's shadow authentication credentials are updated. Nemesis then performs an access control check on the shadow authentication credentials using the web application ACL.

3.4 Registering a New User

The last way a user may authenticate is to register as a new user. Nemesis infers that new user registration has occurred when a user is inserted into the authentication credential storage location. In practice, this is usually a SQL INSERT statement modifying the user authentication database table. The inserted username must be tainted as "user input", to ensure that this new user addition is occurring on behalf of the web client, and not because the web application needed to add a user for internal usage.

Once the username has been extracted and verified as tainted, the web client is then treated as authenticated for that username, and the appropriate session files and shadow authentication cookies are created. For the common case of a database table, this requires us to parse the SQL query, and determine if the query is an INSERT into the user table or not. If so, we extract the username field from the SQL statement.

3.5 Authentication Bypass Attacks

Shadow authentication information is only updated when the web client supplies valid user credentials, such as a password for a web application user, or when a valid shadow cookie is presented. During authentication bypass attacks, malicious users are authenticated by the web application without supplying valid credentials. Thus, when one of these attacks occurs, the web application will incorrectly authenticate the malicious web client, but shadow authentication information will not be updated.

While we could detect authentication bypass attacks by trying to discern when shadow authentication information differs from the authenticated state in the web application, this would depend on internal knowledge of each web application's code base. Authentication frameworks are often complex, and each web application typically creates its own framework, possibly spreading the current authentication information among multiple variables and complex data structures.

Instead, we note that the goal of any authentication bypass attack is to use the ill-gotten authentication to obtain unauthorized access to resources. These are exactly the resources that the current shadow authenticated user is not permitted to access. As explained in the next section, we can prevent authentication bypass attacks by detecting when the current shadow authenticated user tries to obtain unauthorized access to a system resource such as a file, directory, or database table.

4 Authorization Enforcement

Both authentication and access control bypass vulnerabilities allow an attacker to perform operations that she would not be otherwise authorized to perform. The previous section described how Nemesis constructs a shadow authentication system to keep track of user authentication information despite application-level bugs. However, the shadow authentication system alone is not enough to prevent these attacks. This section describes how Nemesis mitigates the attacks by connecting its shadow authentication system with an access control system protecting the web application's database and file system.

To control what operations any given web user is allowed to perform, Nemesis allows the application developer to supply access control rules (ACL) for files, directories, and database objects. Nemesis extends the core system library so that each database or file operation performs an ACL check. The ACL check ensures that the current shadow authenticated user is permitted by the web application ACL to execute the operation. This enforcement prevents access control bypass attacks, because an attacker exploiting a missing or invalid access control check to perform a privileged operation will be foiled when Nemesis enforces the supplied ACL. This also mitigates authentication bypass attacks—even if an attacker can bypass the application's authentication system (e.g., due to a missing check in the application code), Nemesis will automatically perform ACL checks against the username provided by the shadow authentication system, which is not subject to authentication bypass attacks.

4.1 Access Control

In any web application, the authentication framework plays a critical role in access control decisions. There are often numerous, complex rules determining which resources (such as files, directories, or database tables, rows, or fields) can be accessed by a particular user. However, existing web applications do not have explicit, codified access control rules. Rather, each application has its own authentication system, and access control checks are interspersed throughout the application.

For example, many web applications have a privileged script used to manage the users of the web application. This script must only be accessed by the web application administrator, as it will likely contain logic to change the password of an arbitrary user and perform other privileged operations. To restrict access appropriately, the beginning of the script will contain an access control check to ensure that unauthorized users cannot access script functionality. This is actually an example of the policy, "only the administrator may access the admin.php script", or to rephrase such a policy in terms of the resources it affects, "only the administrator may modify the user table in the database". This policy is often never explicitly stated within the web application, and must instead be inferred from the authorization checks in the web application. Nemesis requires the developer or system administrator to explicitly provide an access control list based on knowledge of the application. Our prototype system and evaluation suggests that, in practice, this requires little programmer effort while providing significant security benefits. Note that a single developer or administrator needs to specify access control rules. Based on these rules, Nemesis will provide security checks for all application users.

4.1.1 File Access

Nemesis allows developers to restrict file or directory access to a particular shadow authenticated user. For example, a news application may only allow the administrator to update the news spool file. We can also restrict the set of valid operations that can be performed: read, write, or append. For directories, read permission is equivalent to listing the contents of the directory, while write permission allows files and subdirectories to be created. File access checks happen before any attempt to open a file or directory. These ACLs could be expressed by listing the files and access modes permitted for each user.

4.1.2 SQL Database Access

Nemesis allows web applications to restrict access to SQL tables. Access control rules specify the user, name of the SQL database table, and the type of access (INSERT, SELECT, DELETE, or UPDATE). For each SQL query, Nemesis must determine what tables will be accessed by the query, and whether the ACLs permit the user to perform the desired operation on those tables.

In addition to table-level access control, Nemesis also allows restricting access to individual rows in a SQL table, since applications often store data belonging to different users in the same table.

An ACL for a SQL row works by restricting a given SQL table to just those rows that should be accessible to the current user, much like a *view* in SQL terminology. Specifically, the ACL maps SQL table names and access types to an SQL predicate expression involving column names and values that constrain the kinds of rows the current user can access, where the values can be either fixed constants, or the current username from the shadow authentication system, evaluated at runtime. For example, a programmer can ensure that a user can only access their own profile by confining SQL queries on the profile table to those whose *user* column matches the current shadow username.

SELECT ACLs restrict the values returned by a SELECT SQL statement. DELETE and UPDATE query ACLs restrict the values modified by an UPDATE or DELETE statement, respectively. To enforce ACLs for these statements, Nemesis must rewrite the database query to append the field names and values from the ACL to the WHERE condition clause of the query. For example, a query to retrieve a user's private messages might be "SELECT * FROM messages WHERE recipient=$current_user", where $current_user is supplied by the application's authentication system. If attackers could fool the application's authentication system into setting $current_user to the name of a different user, they might be able to retrieve that user's messages.

Using Nemesis, the programmer can specify an ACL that only allows SELECTing rows whose sender or recipient column matches the current shadow user. As a result, if user Bob issues the query, Nemesis will transform it into the query "SELECT * FROM messages WHERE recipient=$current_user AND (sender=Bob or recipient=Bob)", which mitigates any possible authentication bypass attack.

Finally, INSERT statements do not read or modify existing rows in the database. Thus, access control for INSERT statements is governed solely by the table access control rules described earlier. However, sometimes developers may want to set a particular field to the current shadow authenticated user when a row is inserted into a table. Nemesis accomplishes this by rewriting the INSERT query to replace the value of the designated field with the current shadow authenticated user (or to add an additional field assignment if the designated field was not initialized by the INSERT statement).

Modifying INSERT queries has a number of real-world uses. Many database tables include a field that stores

the username of the user who inserted the field. The administrator can choose to replace the value of this field with the shadow authenticated username, so that authentication flaws do not allow users to spoof the owner of a particular row in the database. For example, in the PHP forum application DeluxeBB, we can override the author name field in the table of database posts with the shadow authenticated username. This prevents malicious clients from spoofing the author when posting messages, which can occur if an authentication flaw allows attackers to authenticate as arbitrary users.

4.2 Enhancing Access Control with DIFT

Web applications often perform actions which are not authorized for the currently authenticated user. For example, in the PHP image gallery Linpha, users may inform the web application that they have lost their password. At this point, the web client is unauthenticated (as they have no valid password), but the web application changes the user's password to a random value, and e-mails the new password to the user's e-mail account. While one user should not generally be allowed to change the password of a different user, doing so is safe in this case because the application generates a fresh password not known to the requesting user, and only sends it via email to the owner's address.

One heuristic that helps us distinguish these two cases in practice is the taint status of the newly-supplied password. Clearly it would be a bad idea to allow an unauthenticated user to supply the new password for a different user's account, and such password values would have the "user input" taint under Nemesis. At the same time, our experience suggests that internally-generated passwords, which do not have the "user input" taint, correspond to password reset operations, and would be safe to allow.

To support this heuristic, we add one final parameter to all of the above access control rules: taint status. An ACL entry may specify, in addition to its other parameters, taint restrictions for the file contents or database query. For example, an ACL for Linpha allows the application to update the password field of the user table regardless of the authentication status, as long as the query is untainted. If the query is tainted, however, the ACL only allows updates to the row corresponding to the currently authenticated user.

4.3 Protecting Authentication Credentials

Additionally, there is one security rule that does not easily fit into our access control model, yet can be protected via DIFT. When a web client is authenticating to the web application, the application must read user credentials such as a password and use those credentials to authenticate the client. However, unauthenticated clients do not have permission to see passwords. A safe web application will ensure that these values are never leaked to the client. To prevent an information leak bug in the web application from resulting in password disclosure, Nemesis forbids any object that has the authentication credential DIFT tag bit set from being returned in any HTTP response. In our prototype, this rule has resulted in no false positives in practice. Nevertheless, we can easily modify this rule to allow passwords for a particular user to be returned in a HTTP response once the client is authenticated for that user. For example, this situation could arise if a secure e-mail service used the user's password to decrypt e-mails, causing any displayed emails to be tagged with the password bit.

5 Prototype Implementation

We have implemented a proof-of-concept prototype of Nemesis by modifying the PHP interpreter. PHP is one of the most popular languages for web application development. However, the overall approach is not tied to PHP by design, and could be implemented for any other popular web application programming language. Our prototype is based on an existing DIFT PHP tainting project [29]. We extend this work to support authentication inference and authorization enforcement.

5.1 Tag Management

PHP is a dynamically typed language. Internally, all values in PHP are instances of a single type of structure known as a *zval*, which is stored as a tagged union. Integers, booleans, and strings are all instances of the *zval* struct. Aggregate data types such as arrays serve as hash tables mapping index values to *zval*s. Symbol tables are hash tables mapping variable names to *zval*s.

Our prototype stores taint information at the granularity of a *zval* object, which can be implemented without storage overhead in the PHP interpreter. Due to alignment restrictions enforced by GCC, the *zval* structure has a few unused bits, which is sufficient for us to store the two taint bits required by Nemesis.

By keeping track of taint at the object level, Nemesis assumes that the application will not combine different kinds of tagged credentials in the same object (e.g. by concatenating passwords from two different users together, or combining untrusted and authentication-based input into a single string). While we have found this assumption to hold in all encountered applications, a byte granularity tainting approach could be used to avoid this limitation if needed, and prior work has shown it practical to implement byte-level tainting in PHP [16]. When multiple objects are combined in our prototype, the result's taint

bits are the union of the taint bits on all inputs. This works well for combining tainted and untainted data, such as concatenating an untainted salt with a tainted password (with the result being tainted), but can produce imprecise results when concatenating objects with two different classes of taint.

User input and password taint is propagated across all standard PHP operations, such as variable assignment, arithmetic, and string concatenation. Any value with password taint is forbidden from being returned to the user via echo, printf, or other output statements.

5.2 Tag Initialization

Any input from URL parameters (GET, PUT, etc), as well as from any cookies, is automatically tainted with the 'user input' taint bit. Currently, password taint initialization is done by manually inserting the taint initialization function call as soon as the password enters the system (e.g., from a database) as we have not yet implemented a full policy language for automated credential tainting. For a few of our experiments in Section 6 (phpFastNews, PHP iCalendar, Bilboblog), the admin password was stored in a configuration php script that was included by the application scripts at runtime. In this case, we instrumented the configuration script to set the password bit of the admin password variable in the script.

At the same time as we initialize the password taint, we also set a global variable to store the candidate username associated with the password, to keep track of the current username being authenticated. If authentication succeeds, the shadow authentication system uses this candidate username to set the global variable that stores the shadow authenticated user, as well as to initialize the shadow cookie. If a client starts authenticating a second time as a different user, the candidate username is reset to the new value, but the authenticated username is not affected until authentication succeeds.

Additionally, due to an implementation artifact in the PHP $setcookie()$ function, we also record shadow authentication in the PHP built-in session when appropriate. This is because PHP forbids new cookies to be added to the HTTP response once the application has placed part of the HTML body in the response output buffer. In an application that uses PHP sessions, the cookie only stores the session ID and all authentication information is stored in session files on the server. These applications may output part of the HTML body before authentication is complete. We correctly handle this case by storing shadow authentication credentials in the server session file if the application has begun a PHP session. When validating and recovering shadow cookies for authentication purposes, we also check the session file associated with the current user for shadow authentication credentials. This approach relies on PHP safely storing session files, but as session files are stored on the server in a temporary directory, this is a reasonable assumption.

5.3 Authentication Checks

When checking the authentication status of a user, we first check the global variable that indicates the current shadow authenticated user. This variable is set if the user has just begun a new session and been directly authenticated via password comparison or deferred authentication to a login server. If this variable is not set, we check to see if shadow authentication information is stored in the current session file (if any). Finally, we check to see if the user has presented a shadow authentication cookie, and if so we validate the cookie and extract the authentication credentials. If none of these checks succeeds, the user is treated as unauthenticated.

5.4 Password Comparison Authentication Inference

Authentication inference for password comparisons is performed by modifying the PHP interpreter's string comparison equality and inequality operators. When one of these string comparisons executes, we perform a check to see if the two string operands were determined to be equal. If the strings were equal, we then check their tags, and if one string has only the authentication credential tag bit set, and the other string has only the user input tag bit set, then we determine that a successful shadow authentication has occurred. In all of our experiments, only PhpMyAdmin used a form of authentication that did not rely on password string comparison, and our handling of this case is discussed in Section 6.

5.5 Access Control Checks

We perform access control checks for files by checking the current authenticated user against a list of accessible files (and file modes) on each file access. Similarly, we restrict SQL queries by checking if the currently authenticated user is authorized to access the table, and by appending additional WHERE clause predicates to scope the effect of the query to rows allowed for the current user.

Due to time constraints, we manually inserted these checks into applications based on the ACL needed by the application. ACLs that placed constraints on field values of a database row required simple query modifications to test if the field value met the constraints in the ACL.

In a full-fledged design, the SQL queries should be parsed, analyzed for the appropriate information, and rewritten if needed to enforce additional security guarantees (e.g., restrict rows modified to be only those cre-

ated by the current authenticated user). Depending on the database used, query rewriting may also be partially or totally implemented using database views and triggers [18, 26].

5.6 SQL Injection

Shadow authentication is necessary to prevent authentication bypass attacks and enforce our ACL rules. However, it can also be used to prevent false positives in DIFT SQL injection protection analyses. The most robust form of SQL injection protection [27] forbids tainted keywords or operators, and enforces the rule that tainted data may never change the parse tree of a query.

Our current approach does not support byte granularity taint, and thus we must approximate this analysis. We introduce a third taint bit in the *zval* which we use to denote user input that may be interpreted as a SQL keyword or operator. We scan all user input at startup (GET, POST, COOKIE superglobal arrays, etc) and set this bit only for those user input values that contain a SQL keyword or operator. SQL quoting functions, such as `mysql_real_escape_string()`, clear this tag bit. Any attempt to execute a SQL query with the unsafe SQL tag bit set is reported as a SQL injection attack.

We use this SQL injection policy to confirm that DIFT SQL Injection has false positives and real-world web applications. This is because DIFT treats all user input as untrusted, but some web applications allow privileged users such as the admin to submit full SQL queries. As discussed in Section 6, we eliminate all encountered false positives using authentication policies which restrict SQL injection protection to users that are not shadow authenticated as the admin user. We have confirmed that all of these false positives are due to a lack of authentication information, and not due to any approximations made in our SQL injection protection implementation.

6 Experimental Results

To validate Nemesis, we used our prototype to protect a wide range of vulnerable real-world PHP applications from authentication and access control bypass attacks. A summary of the applications and their vulnerabilities is given in Table 1, along with the lines of code that were added or modified in order to protect them.

For each application, we had to specify where the application stores its username and password database, or what function it invokes to authenticate users. This step is quite simple for all applications, and the "authentication inference" column indicates the amount of code we had to add to each application to specify the table used to store known-good passwords, and to taint the passwords with the "credential" taint bit.

We also specified ACLs on files and database tables to protect them from unauthorized accesses; the number of access control rules for each application is shown in the table. As explained in Section 5, we currently enforce ACLs via explicitly inserted checks, which slightly increases the lines of code needed to implement the check (shown in the table as well). As we develop a full MySQL parser and query rewriter, we expect the lines of code needed for these checks to drop further.

We validated our rules by using each web application extensively to ensure there are no false positives, and then verifying that our rules prevented real-world attacks that have been found in these applications in the past. We also verified that our shadow authentication information is able to prevent false positives in DIFT SQL injection analyses for both the DeluxeBB and phpMyAdmin applications.

6.1 PHP iCalendar

PHP iCalendar is a PHP web application for presenting calendar information to users. The webapp administrator is authenticated using a configuration file that stores the admin username and password. Our ACL for PHP iCalendar allows users read access to various template files, language files, and all of the calendars. In addition, caches containing parsed calendars can be read or written by any user. The admin user is able to write, create, and delete calendar files, as well as read any uploaded calendars from the uploads directory. We added 8 authorization checks to enforce our ACL for PHP iCalendar.

An authentication bypass vulnerability occurs in PHP iCalendar because a script in the admin subdirectory incorrectly validates a login cookie when resuming a session [20]. This vulnerability allows a malicious user to forge a cookie that will cause her to be authenticated as the admin user.

Using Nemesis, when an attacker attempts to exploit the authentication bypass attack, she will find that her shadow authentication username is not affected by the attack. This is because shadow authentication uses its own secure form of cookie authentication, and stores its credentials separately from the rest of the web application. When the attacker attempts to use the admin scripts to perform any actions that require admin access, such as deleting a calendar, a security violation is reported because the shadow authentication username will not be 'admin', and the ACL will prevent that username from performing administrative operations.

6.2 Phpstat

Phpstat is an application for presenting a database of IM statistics to users, such as summaries and logs of their IM

Program	LoC in program	LoC for auth inference	Number of ACL checks	LoC for ACL checks	Vulnerability prevented
PHP iCalendar	13500	3	8	22	Authentication bypass
phpStat (IM Stats)	12700	3	10	17	Missing access check
Bilboblog	2000	3	4	11	Invalid access check
phpFastNews	500	5	2	17	Authentication bypass
Linpha Image Gallery	50000	15	17	49	Authentication bypass
DeluxeBB Web Forum	22000	6	82	143	Missing access check

Table 1: Applications used to evaluate Nemesis.

conversations. Phpstat stores its authentication credentials in a database table.

The access control list for PhpStat allows users to read and write various cache files, as well as read the statistics database tables. Users may also read profile information about any other user, but the value of the password field may never be sent back to the Web client. The administrative user is also allowed to create users by inserting into or updating the users table, as well as all of the various statistics tables. We added 10 authorization checks to enforce our ACL for PhpStat.

A security vulnerability exists in PhpStat because an installation script will reset the administrator password if a particular URL parameter is given. This behavior occurs without any access control checks, allowing any user to reset the admin password to a user-specified value [21]. Successful exploitation grants the attacker full administrative privileges to the Phpstat. Using Nemesis, when this attack occurs, the attacker will not be shadow authenticated as the admin, and any attempts to execute a SQL query that changes the password of the administrator are denied by our ACL rules. Only users shadow authenticated as the admin may change passwords.

6.3 Bilboblog

Bilboblog is a simple PHP blogging application that authenticates its administrator using a username and password from a configuration file.

Our ACL for bilboblog permits all users to read and write blog caching directories, and read any posted articles from the article database table. Only the administrator is allowed to modify or insert new entries into the articles database table. Bilboblog has an invalid access control check vulnerability because one of its scripts, if directly accessed, uses uninitialized variables to authenticate the admin user [3]. We added 4 access control checks to enforce our ACL for bilboblog.

In PHP, if the register_globals option is set, uninitialized variables may be initialized at startup by user-supplied URL parameters [22]. This allows a malicious user to supply the administrator username and password that the login will be authenticated against. The attacker may simply choose a username and password, access the login script with these credentials encoded as URL parameters, and then input the same username and password when prompted by the script. This attack grants the attacker full administrative access to Bilboblog.

This kind of attack does not affect shadow authentication. A user is shadow authenticated only if their input is compared against a valid password. This attack instead compares user input against a URL parameter. URL parameters do not have the password bit set – only passwords read from the configuration do. Thus, no shadow authentication occurs when this attack succeeds. If an attacker exploits this vulnerability on a system protected by our prototype, she will find herself unable to perform any privileged actions as the admin user. Any attempt to update, delete, or modify an article will be prevented by our prototype, as the current user will not be shadow authenticated as the administrator.

6.4 phpFastNews

PhpFastNews is a PHP application for displaying news stories. It performs authentication via a configuration file with username and password information. This web application displays a news spool to users. Our ACL for phpFastNews allows users to read the news spool, and restricts write access to the administrator. We added 2 access control checks to enforce our ACL for phpFastNews.

An authentication bypass vulnerability occurs in phpFastNews due to insecure cookie validation [19], much like in PHP iCalendar. If a particular cookie value is set, the user is automatically authenticated as the administrator without supplying the administrator's password. All the attacker must do is forge the appropriate cookie, and full admin access is granted.

Using Nemesis, when the authentication bypass attack occurs, our prototype will prevent any attempt to perform administrator-restricted actions such as updating the news spool because the current user is not shadow authenticated as the admin.

6.5 Linpha

Linpha is a PHP web gallery application, used to display directories of images to web users. It authenticates its users via a database table.

Our ACL for Linpha allows users to read files from the images directory, read and write files in the temporary and cache directories, and insert entries into the thumbnails table. Users may also read from the various settings, group, and configuration tables. The administrator may update or insert into the users table, as well as the settings, groups, and categories tables. Dealing with access by non-admin users to the user table is the most complex part of the Linpha ACL, and is our first example of a database row ACL. Any user may read from the user table, with the usual restriction that passwords may never be output to the Web client via echo, print, or related commands.

Users may also update entries in the user table. Updating the password field must be restricted so that a malicious user cannot update the other passwords. This safety restriction can be enforced by ensuring that only user table rows that have a username field equal to the current shadow authenticated user can be modified. The exception to this rule is when the new password is untainted. This can occur only when the web application has internally generated the new user password without using user input. We allow these queries even when they affect the password of a user that is not the current shadow authenticated user because they are used for lost password recovery.

In Linpha, users may lose their password, in which case Linpha resets their password to an internally generated value, and e-mails this password to the user. This causes an arbitrary user's password to be changed on the behalf of a user who isn't even authenticated. However, we can distinguish this safe and reasonable behavior from an attack by a user attempting to change another user's password by examining the taint of the new password value in the SQL query. Thus, we allow non-admin users to update the password field of the user table if the password query is untainted, or if the username of the modified row is equal to the current shadow authenticated user. Overall, we added 17 authorization checks to enforce all of our ACLs for Linpha.

Linpha also has an authentication bypass vulnerability because one of its scripts has a SQL injection vulnerability in the SQL query used to validate login information from user cookies [13]. Successful exploitation of this vulnerability grants the attacker full administrative access to Linpha. For this experiment, we disabled SQL injection protection provided by the tainting framework we used to implement the Nemesis prototype [29], to allow the user to submit a malicious SQL query in order to bypass authentication entirely.

Using Nemesis, once a user has exploited this authentication bypass vulnerability, she may access the various administration scripts. However, any attempt to actually use these scripts to perform activities that are reserved for the admin user will fail, because the current shadow authenticated user will not be set to admin, and our ACLs will correspondingly deny any admin-restricted actions.

6.6 DeluxeBB

DeluxeBB is a PHP web forum application that supports a wide range of features, such as an administration console, multiple forums, and private message communication between forum users. Authentication is performed using a table from a MySQL database.

DeluxeBB has the most intricate ACL of any application in our experiments. All users in DeluxeBB may read and write files in the attachment directory, and the admin user may also write to system log files. Non-admin users in DeluxeBB may read the various configuration and settings tables. Admin users can also write these tables, as well as perform unrestricted modifications to the user table. DeluxeBB treats user table updates and lost password modifications in the same manner as Linpha, and we use the equivalent ACL to protect the user table from non-admin modifications and updates.

DeluxeBB allows unauthenticated users to register via a form, and thus unauthenticated users are allowed to perform inserts into the user table. As described in Section 3.4, inserting a user into the user table results in shadow authentication with the credentials of the inserted user.

The novel and interesting part of the ACLs for DeluxeBB are the treatment of posts, thread creation, and private messages. All inserts into the post, thread creation, or private message tables are rewritten to use the shadow authenticated user as the value for the author field (or the sender field, in the case of a private message). The only exception is when a query is totally untainted. For example, when a new user registers, a welcome message is sent from a fake system mailer user. As this query is totally untainted, we allow it to be inserted into the private message table, despite the fact that the identity of the sender is clearly forged. We added fields to the post and thread tables to store the username of the current shadow authenticated user, as these tables did not directly store the author's username. We then explicitly instrumented all SQL INSERT statements into these tables to append this information accordingly.

Any user may read from the thread or post databases. However, our ACL rules further constrain reads from the private message database. A row may only be read from the private message database if the 'from' or 'to' fields of the row are equal to the current shadow authenticated user.

We manually instrumented all SELECT queries from the private message table to add this condition to the WHERE clause of the query. In total, we modified 16 SQL queries to enforce both our private message protection and our INSERT rules to prevent spoofing messages, threads, and posts. We also inserted 82 authorization checks to enforce the rest of the ACL.

A vulnerability exists in the private message script of this application [6]. This script incorrectly validates cookies, missing a vital authentication check. This allows an attacker to forge a cookie and be treated as an arbitrary web application user by the script. Successful exploitation of this vulnerability gives an attacker the ability to access any user's private messages.

Using Nemesis, when this attack is exploited, the attacker can fool the private message script into thinking he is an arbitrary user due to a missing access control check. The shadow authentication for the attack still has the last safe, correct authenticated username, and is not affected by the attack. Thus, the attacker is unable to access any unauthorized messages, because our ACL rules only allow a user to retrieve messages from the private message table when the sender or recipient field of the message is equal to the current shadow authenticated user. Similarly, the attacker cannot abuse the private message script to forge messages, as our ACLs constrain any messages inserted into the private message table to have the sender field set to the current shadow authenticated username.

DeluxeBB allows admin users to execute arbitrary SQL queries. We verified that this results in false positives in existing DIFT SQL injection protection analyses. After adding an ACL allowing SQL injection for web clients shadow authenticated as the admin user, all SQL injection false positives were eliminated.

6.7 PhpMyAdmin

PhpMyAdmin is a popular web application used to remotely administer and manage MySQL databases. This application does not build its own authentication system; instead, it checks usernames and passwords against MySQL's own user database. A web client is validated only if the underlying MySQL database accepts their username and password.

We treat the MySQL database connection function as a third-party authentication function as detailed in Section 3.2.2. We instrumented the call to the MySQL database connection function to perform shadow authentication, authenticating a user if the username and password supplied to the database are both tainted, and if the login was successful.

The ACL for phpMyAdmin is very different from other web applications, as phpMyAdmin is intended to provide an authenticated user with unrestricted access to the underlying database. The only ACL we include is a rule allowing authenticated users to submit full SQL database queries. We implemented this by modifying our SQL injection protection policy defined in Section 5.6 to treat tainted SQL operators in user input as unsafe only when the current user was unauthenticated. Without this policy, any attempt to submit a query as an authenticated user results in a false positive in the DIFT SQL injection protection policy. We confirmed that adding this ACL removes all observed SQL injection false positives, while still preventing unauthenticated users from submitting SQL queries.

6.8 Performance

We also performed performance tests on our prototype implementation, measuring overhead against an unmodified version of PHP. We used the bench.php microbenchmark distributed with PHP, where our overhead was 2.9% compared to unmodified PHP. This is on par with prior results reported by object-level PHP tainting projects [29]. However, bench.php is a microbenchmark which performs CPU-intensive operations. Web applications often are network or I/O-bound, reducing the real-world performance impact of our information flow tracking and access checks.

To measure performance overhead of our prototype for web applications, we used the Rice University Bidding System (RUBiS) [24]. RUBiS is a web application benchmark suite, and has a PHP implementation that is approximately 2,100 lines of code. Our ACL for RUBiS prevents users from impersonating another user when placing bids, purchasing an item, or posting a bid comment. Three access checks were added to enforce this ACL. We compared latency and throughput for our prototype and an unmodified PHP, and found no discernible performance overhead.

7 Future Work

There is much opportunity for further research in preventing authentication bypass attacks. A fully developed, robust policy language for expressing access control in web applications must be developed. This will require support for SQL query rewriting, which can be implemented by developing a full SQL query parser or by using database table views and triggers similar to the approach described in [18].

Additionally, all of our ACL rules are derived from real-world behavior observed when using the web application. While we currently generate such ACLs manually, it should be possible to create candidate ACLs automatically. A log of all database and file operations, as well as the current shadow authenticated user at the time of the

operation, would be recorded. Using statistical techniques such as machine learning [14], this log could be analyzed and access control files generated based on application behavior. This would allow system administrators to automatically generate candidate ACL lists for their web applications without a precise understanding of the access control rules used internally by the webapp.

8 Related Work

Preventing web authentication and authorization vulnerabilities is a relatively new area of research. The only other work in this area of which we are aware is the CLAMP research project [18]. CLAMP prevents authorization vulnerabilities in web applications by migrating the user authentication module of a web application into a separate, trusted virtual machine (VM). Each new login session forks a new, untrusted session VM and forwards any authentication credentials to the authentication VM. Authorization vulnerabilities are prevented by a trusted query restricter VM which interposes on all session VM database accesses, examining queries to enforce the appropriate ACLs using the username supplied by the authentication VM.

In contrast to Nemesis, CLAMP does not prevent authentication vulnerabilities as the application authentication code is part of the trusted user authentication VM. CLAMP also cannot support access control policies that require taint information as it does not use DIFT. Furthermore, CLAMP requires a new VM to be forked when a user session is created, and experimental results show that one CPU core could only fork two CLAMP VMs per second. This causes significant performance degradation for throughput-driven, multi-user web applications that have many simultaneous user sessions.

Researchers have extensively explored the use of DIFT to prevent security vulnerabilities. Robust defenses against SQL injection [15, 27], cross-site scripting [25], buffer overflows [5] and other attacks have all been proposed. DIFT has been used to prevent security vulnerabilities in most popular web programming languages, including Java [8], PHP [16], and even C [15]. This paper shows how DIFT techniques can be used to address authentication and access control vulnerabilities. Furthermore, DIFT-based solutions to attacks such as SQL injection have false positives in the real-world due to a lack of authentication information. Nemesis avoids such false positives by incorporating authentication and authorization information at runtime.

Much work has also been done in the field of secure web application design. Information-flow aware programming language extensions such as Sif [4] have been developed to prevent information leaks and security vulnerabilities in web application programs using the language and compiler. Unfortunately these approaches typically require legacy applications to be rewritten in the new, secure programming language.

Some web application frameworks provide common authentication or authorization code libraries [1, 2, 7]. The use of these authentication libraries is optional, and many application developers choose to partially or entirely implement their own authentication and authorization systems. Many design decisions, such as how users are registered, how lost passwords are recovered, or what rules govern access control to particular database tables are application-specific. Moreover, these frameworks do not connect the user authentication mechanisms with the access control checks in the underlying database or file system. As a result, the application programmer must still apply access checks at every relevant filesystem and database operation, and even a single mistake can compromise the security of the application.

Operating systems such as HiStar [32] and Flume [12] provide process-granularity information flow control support. One of the key advantages of these systems is that they give applications the flexibility to define their own security policies (in terms of information flow categories), which are then enforced by the underlying kernel. A web application written for HiStar or Flume can implement its user authentication logic in terms of the kernel's information flow categories. This allows the OS kernel to then ensure that one web user cannot access data belonging to a different web user, even though the OS kernel doesn't know how to authenticate a web user on its own. Our system provides two distinct advantages over HiStar and Flume. First, we can mitigate vulnerabilities in existing web applications, without requiring the application to be re-designed from scratch for security. Second, by providing sub-process-level information flow tracking and expressive access control checks instead of labels, we allow programmers to specify precise security policies in a small amount of code.

9 Conclusion

This paper presented Nemesis, a novel methodology for preventing authentication and access control bypass attacks in web applications. Nemesis uses Dynamic Information Flow Tracking to automatically detect when application-specific users are authenticated, and constructs a *shadow authentication system* to track user authentication state through an additional HTTP cookie.

Programmers can specify access control lists for resources such as files or database entries in terms of application-specific users, and Nemesis automatically enforces these ACLs at runtime. By providing a shadow authentication system and tightly coupling the authentication system to authorization checks, Nemesis prevents

a wide range of authentication and access control bypass attacks found in today's web applications. Nemesis can also be used to improve precision in other security tools, such as those that find SQL injection bugs, by avoiding false positives for properly authenticated requests.

We implemented a prototype of Nemesis in the PHP interpreter, and evaluated its security by protecting seven real-world applications. Our prototype stopped all known authentication and access control bypass attacks in these applications, while requiring only a small amount of work from the application developer, and introducing no false positives. For most applications we evaluated, the programmer had to write less than 100 lines of code to avoid authentication and access control vulnerabilities. We also measured performance overheads using PHP benchmarks, and found that our impact on web application performance was negligible.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback. Michael Dalton was supported by a Sequoia Capital Stanford Graduate Fellowship. This work was supported NSF Awards number 0546060 and 0701607.

References

[1] Turbogears documentation: Identity management. `http://docs.turbogears.org/1.0/Identity`.

[2] Zope security. `http://www.zope.org/Documentation/Books/ZDG/current/Security.stx`.

[3] BilboBlog admin/index.php Authentication Bypass Vulnerability. `http://www.securityfocus.com/bid/30225`, 2008.

[4] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proceedings of the 16th Annual USENIX Security Symposium*, 2007.

[5] M. Dalton, H. Kannan, and C. Kozyrakis. Real-World Buffer Overflow Protection for Userspace and Kernelspace. In *Proceedings of the 17th Annual USENIX Security Symposium*, 2008.

[6] DeluxeBB PM.PHP Unauthorized Access Vulnerability. `http://www.securityfocus.com/bid/19418`, 2006.

[7] Django Software Foundation. User authentication in Django. `http://docs.djangoproject.com/en/dev/topics/auth/`.

[8] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. *Annual Computer Security Applications Conference*, 2005.

[9] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Operating System Review*, 1988.

[10] Microsoft Internet Information Server Hit Highlighting Authentication Bypass Vulnerability. `http://www.securityfocus.com/bid/24105`, 2007.

[11] M. Krohn, P. Efstathopoulos, C. Frey, F. Kaashoek, E. Kohler, D. Mazières, R. Morris, M. Osborne, S. VanDeBogart, and D. Ziegler. Make Least Privilege a Right (Not a Privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, 2005.

[12] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[13] Linpha User Authentication Bypass Vulnerability. `http://secunia.com/advisories/12189`, 2004.

[14] E. Martin and T. Xie. Inferring access-control policy properties via machine learning. In *Proc. 7th IEEE Workshop on Policies for Distributed Systems and Networks*, 2006.

[15] S. Nanda, L.-C. Lam, and T. Chiueh. Dynamic multi-process information flow tracking for web application security. In *Proceedings of the 8th International Conference on Middleware*, 2007.

[16] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications using Precise Tainting. In *Proceedings of the 20th IFIP Intl. Information Security Conference*, 2005.

[17] Top 10 2007 - Broken Authentication and Session Management. `http://www.owasp.org/index.php/Top_10_2007-A7`, 2007.

[18] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical Prevention of Large-Scale Data Leaks. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, May 2009.

[19] phpFastNews Cookie Authentication Bypass Vulnerability. `http://www.securityfocus.com/bid/31811`, 2008.

[20] PHP iCalendar Cookie Authentication Bypass Vulnerability. `http://www.securityfocus.com/bid/31320`, 2008.

[21] Php Stat Vulnerability Discovery. `http://www.soulblack.com.ar/repo/papers/advisory/PhpStat_advisory.txt`, 2005.

[22] PHP: Using Register Globals. `http://us2.php.net/register_globals`.

[23] PhpMyAdmin control user. `http://wiki.cihar.com/pma/controluser`.

[24] Rice University Bidding System. `http://rubis.objectweb.org`, 2009.

[25] P. Saxena, D. Song, and Y. Nadji. Document structure integrity: A robust basis for cross-site scripting defense. *Network and Distributed Systems Security Symposium*, 2009.

[26] S. R. Shariq, A. Mendelzon, S. Sudarshan, and P. Roy. Extending Query Rewriting Techniques for Fine-Grained Access Control. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2004.

[27] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the 33rd Symposium on Principles of Programming Languages*, 2006.

[28] The MITRE Corporation. Common vulnerabilities and exposures (CVE) database. `http://cve.mitre.org/data/downloads/`.

[29] W. Venema. Taint support for php. `http://wiki.php.net/rfc/taint`, 2008.

[30] Web Application Security Consortium. 2007 web application security statistics. `http://www.webappsec.org/projects/statistics/wasc_wass_2007.pdf`.

[31] WordPress Cookie Integrity Protection Unauthorized Access Vulnerability. `http://www.securityfocus.com/bid/28935`, 2008.

[32] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.