The Pennsylvania State University
The Graduate School

PYRENA: CLOSING THE SEMANTIC GAP FOR ACCESS CONTROL IN WEB-BASED CONTENT MANAGEMENT SYSTEMS

A Thesis in
Computer Science and Engineering
by
Adam Bergstein

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2014

The thesis of Adam Bergstein was reviewed and approved* by the following:

Trent Jaeger
Associate Professor of Computer Science and Engineering
Thesis Advisor

Patrick McDaniel
Professor of Computer Science and Engineering

Lee D. Coraor
Director of Academic Affairs

*Signatures are on file in the Graduate School.

Abstract

Large organizations that wish to publish content to the Web are increasingly turning to Content Management Systems (CMSs). A CMS allows a large set of members within the organization to author and publish content within their domains of expertise without having to consider the overarching organization or presentation of the content. Unfortunately, modern CMSs tend to be implemented as add-ons to webservers, leading to a *semantic gap* between the CMS and some types of content stored on the host machine. In this paper, we demonstrate how this semantic gap can leave sensitive files world-accessible in the widely used *Drupal* CMS. As a solution, we introduce Pyrena, a reference monitor that mediates file system access to resources owned by the CMS. We show that Pyrena maintains not only the well-known security properties of a reference monitor, but also does not break existing third-party access control plugins, or common use cases of the Drupal CMS.

Table of Contents

List of Figures

List of Tables

Acknowledgments

There are three people that absolutely must be acknowledged.

First, my advisor Dr. Trent Jaeger. Few people were interested in working with a full-time employee of the University that had a limited research background. You regularly took time out of your busy schedule to meet with me when other students were producing world class research. You have challenged me, mentored me, and taught me. I am grateful you took a chance on me and I thank you for all of your help.

Second, Steve McLaughlin and Hayaward Vijayakumar. This paper would not exist without your efforts to challenge my ideas and help me articulate my ideas. I cannot express how helpful your collaboration was, including the seemingly endless brainstorming and thrashing meetings we had and the paper reviews you performed. Your help made this effort relevant to the research community by defining the applicable security problems and how I could contribute to them. I thank you both for your assistance.

Third, I would like to thank Kathy Zimmerman and Karen Corl. Their job titles are Administrative Assistants, but their job descriptions should read ”Miracle Workers.” Without a doubt, I could not have determined how to navigate the program without their help, especially with my unique considerations as a full-time employee and not full-time student. Kathy and Karen were always available and willing to help, for which I am still grateful.

Dedication

This thesis is entirely dedicated to my family.

To my wife: thank you for sticking with me throughout this experience. My time spent on graduate work often asked you to do more around the house and pull me away from family activities. I'm looking forward to being around more.

To my mom: you are the model of hard work and dedication. Graduate work and full time employment was incredibly challenging. I was motivated by your day in and day out dedication to your family. My challenge paled in comparison to having a sick spouse, cooking dinner every night, finishing graduate school, and working several jobs simultaneously. Thank you for demonstrating to me what hard work and dedication truly looks like.

To my daughter: I have learned that nothing worthwhile in life comes easy. I sincerely hope that you love to learn and passionately dedicate yourself to your interests. It is easy to walk away when things become hard. To stand out, you have to be willing to go above and beyond. Don't be afraid to fail, just make sure you learn from your mistakes and press on.

Chapter 1

Introduction

Modern websites for organizations, such as universities, businesses, and governments, contain content contributed from a large number of users, called *content providers*. For example, a university website would have content providers in admissions, research programs, billing, athletics, and student government, to name a few. To assist in organizing this content, and to provide a friendly interface for viewing and modifying content, these websites are often built on Content Management Systems (*CMSs*). CMSs give content providers a uniform workflow for authoring and publishing content, while sparing them the details of presentation style and page organization, i.e., the addition of headers, footers, and navigation links.

CMSs enforce access control policies designed to constrain creation, deletion, editing, and viewing of content. For example, such a policy would give content providers in admissions the ability to edit content related to enrollment statistics, but not HOWTOs for connecting with the university's wireless network. Of course, these policies are only as good as their corresponding *enforcement mechanism*, the part of the CMS that checks all user operations against the policy and blocks disallowed operations. A flaw in the policy enforcement mechanism can be exploited by a malicious content provider to view or modify restricted content.

In this paper we consider the policy enforcement mechanism in the widely used *Drupal* CMS, and show that it does not achieve *complete mediation* of access attempts for content stored in plain files (as opposed to databases). Thus, content providers may be able to view or modify plain files uploaded by other users, even if they were not intended to be able to do so. This is due to a semantic gap that is fundamental to the CMS architecture. CMSs are most often implemented as plugins or scripts that run within an existing web server such as Apache [1]. Specifically, the web server acts as a middle layer between the CMS, and any CMS resources stored in the file system. Currently, there is no way for the CMS to instruct the web server how to handle requests for resources stored as plain files.

Previous solutions to this problems have been stop gap measures at best. For example, a content provider may choose to place plain files in a special *private* file system in which no sharing of any sort is allowed for files. This is simply too restrictive to be useful in practice, and differs from the way Drupal

treats non-file data, such as that stored in databases.

In this paper, we introduce Pyrena, a reference monitor for the Drupal CMS. Pyrena completely mediates access to Drupal content stored as plain files. Instead of requiring extensive modification to the CMS or web server, Pyrena embeds Drupal's access control policy into the file system access control primitives themselves, thus guaranteeing mediation of plain files owned by Drupal without any cooperation from the underlying web server. We believe a file system approach is advantageous, since the same mechanism can be applied to any CMS that leverages a file system for plain file access.

Our mechanism adds approximately ten percent overhead, has false positives only when non-standard access control mechanisms are introduced, and blocks plain file access for users that do not have permission to view comparable database content. This makes Pyrena more portable, lightweight, and performant than existing application-layer access control policies which suffer from the semantic gap problem.

This paper makes the following contributions.

- We identify inconsistent access control between database and file content in a CMS caused by a semantic gap between the CMS and the filesystem.
- We present Pyrena, the first general mechanism to close the semantic gap between CMSs, web servers, and filesystems. Practically speaking, Pyrena is also the only CMS-agnostic solution to the gap in plain file access control.
- Pyrena does not alter the core CMS system, but enhances the CMS to address the plain file limitation for both the current and future access control policy definition.

Because introducing Pyrena into an existing system may lead to compatibility issues, we evaluate a Pyrena-enhanced Drupal instance under a representative set of use cases and third-party plugins. Our results show that with few exceptions Pyrena does not cause abnormal behavior or inconvenience content providers or content consumers, in these different scenarios.

Chapter 2

Background

A Content Management System (CMS) is essentially a software platform allowing a large number of individuals within an organization to publish and consume content in the form of text, images, and downloadable files. CMSs are needed, as the traditional method of organizational website authoring typically places only a small number of expert maintainers in charge of content creation. CMSs on the other hand, allow for anyone in an organization to author new content. At first glance, this concept seems similar to that of a *wiki*. A key difference, however, is that wikis do not use *templates* in the same manner as CMSs. When a user creates a page in a wiki, there are no constraints on the combinations of text, images, tables, links, and associated files they may add. CMSs are less free form. Typically, a user creating a page in a CMS will first select a *content type* for the page. The content type then dictates the structure of the page, e.g., a title, some free form text, an image, and a link to an associated document.

There are a large variety of content management systems used in practice. The use cases for different CMSs includes large organization websites, blogs, structured wikis, and e-commerce sites. CMSs are often used in part because of their frameworks, which provide robust APIs and hooks developers can leverage to extend out of the box CMS behavior. The Drupal CMS is a popular CMS that has found usage in almost all applications. As of December 2013, there are nearly 1 million reported Drupal installations[1]. It provides a well defined interface for authoring, publishing, and viewing content. Thus, we choose it as the focus of our study. Drupal exists as a single component in a LAMP (Linux Apache MySQL PHP) system. A typical Drupal instance will run as a set of PHP scripts invoked by the Apache web server upon receiving a URL with a Drupal-specific prefix. Drupal stores content both in databases, e.g., MySQL, and on the local file system. As will be seen, content stored at the file system layer cannot be properly mediated by Drupal due to a semantic gap between Drupal, the web server, and the file system.

[1] `https://drupal.org/project/usage/drupal`

2.1 Example

Publically traded companies have strict regulations for reporting of investments. Such companies often have staff solely responsible for creating and approving investor reports. Plain files are commonly used for these reports, which are often *PDF* files. Within a CMS, a report file would need to be shared across many investment pages, such as the investment landing page, the investment report listing archive, and a page for the specific report. Most importantly, only investors staff should have access to all new investor pages and files before approval and a defined release date. Any other access should be blocked to the new pages and the new PDF file.

Drupal provides a logical grouping of similar content through the use of *content types*. Specific investor pages, implemented as *nodes* of the content type, would all share the access control policy defined for the content type. Content types define a data structure implemented as fields that the nodes inherit. One of the investor page nodes would have a field specifically designated for uploading the investor report PDF file. While the HTML for the investor page would be stored in Drupal's database, the uploaded investor report file would be stored on the file system. Drupal inherently provides two file system options, *public* and *private* that are specified within the field attributes of the content type.

The main problem with this example is the differential in access control of database-managed content and files in the file system. Drupal mediates all calls to the database and provides access control through the content type of the node. In this example, the new investor pages would have restricted access to only staff members granted permission until the public release date. The investor report, however, would need to reside in either the public or private file systems.

Use of the public file system would allow access to the plain file before staff approval through the file's URL. Use of the private file system would restrict file access only to one page, not shared on many similar pages. Ideally, a plain file would have comparable access restrictions to the file's associated pages. For this example, the file would inherit the same access control defined by the investor page content type. Neither the public or private filesystem would be effective for this example.

Ideally, investor report files would have consistent access control provided by the CMS itself. But, the file system and web server resolve file access, not the CMS. Requests to investor reports in the public file system would never involve Drupal. In some manner, the file system must be instructed on how to perform the mediation. An ideal and CMS-agnostic solution would empower the file system to perform it's own mediation, as opposed to developing a CMS-specific approach. But, the CMS access control policy can change over time and the CMS provides the user authentication needed to properly mediate a request to a file. To protect investor report files, both the CMS and file system must work together to provide the necessary access control. The remaining sections outline these challenges in greater detail, providing the background necessary for understanding the design decisions of Pyrena.

2.2 Drupal Operations

In the Drupal CMS, users perform operations on *nodes*. The node is the basic container for content, and is specified by a unique node identifier. When a user chooses to view a node, the node is rendered by

Table 2.1. The operations supported by the Drupal CMS.

Return	Name	Parameters
node-id	create	uid, content-type, field-values
-	edit	uid, node-id, field-values
-	delete	uid, node-id
html	view	uid, node-id

Drupal into a single web page. Every node in a Drupal system has a single *content type*. A content type is a list of one or more *field types*, e.g., number, string, etc. A node is thus an instantiation of each field type in its content type. For example, if a content type contains a string field and a number field, then a potential node of that type would be `["John Doe", 393730591]`.

As shown in Table 2.1, Drupal provides users with four basic operations on nodes. Each operation requires a `uid` to identify the user performing the operation. The create operation takes a content type, and a set of values for the fields in that content type, and returns `node-id` of the newly created node. The edit operation takes a `node-id` and a set of new values for the fields in the specified node, and updates those field values in the specified node. The delete operation removes the specified node. Finally, the view operation renders a node into an HTML page that is returned to the user.

Typically, the `field-values` supplied in the parameter list to the create and edit operations will be stored in a database. In the example where the fields were string and number, the values would each be stored in their own database record of the appropriate type. However, fields may also contain files, such as spreadsheets and PDF documents. When a file is uploaded as a `field-value`, it is stored in the local file system. When the node containing that file is rendered, a link to the file is included in the field. As will be explained in the following section, following the link to the file is essentially an unmediated operation, thus raising the possibility for unmediated file access by unauthorized users.

2.3 Drupal Mediation and Semantic Gap

Drupal mediates each operation according to an access control policy. This policy is a list of entries of the form (operation, parameters) $\rightarrow$ {Allow, Deny} This satisfies the traditional definition of an access control policy as regulating which subjects can perform which operations on which objects. Mediation is done at node-granularity. Thus, if a user has permission to edit a node, then that user may modify any field within that node. Similarly, the right to view a node grants the viewer the ability to view the contents of all fields in that node.

Drupal uses a Role Based Access Control (RBAC) policy to mediate node access. This policy uses the following mappings:

$$\text{User}(\texttt{uid}) \rightarrow \text{Role} \xrightarrow{op} \text{Content Type} \rightarrow \text{Node}$$

Here, *op* is a set of zero to four of the operations listed in Table 2.1. Thus, for each Role possessed by a specific user's `uid`, the list of allowed operations for that role is specified to each defined content type in the system. If a new content type is created, the default policy is that no Role has permissions for any

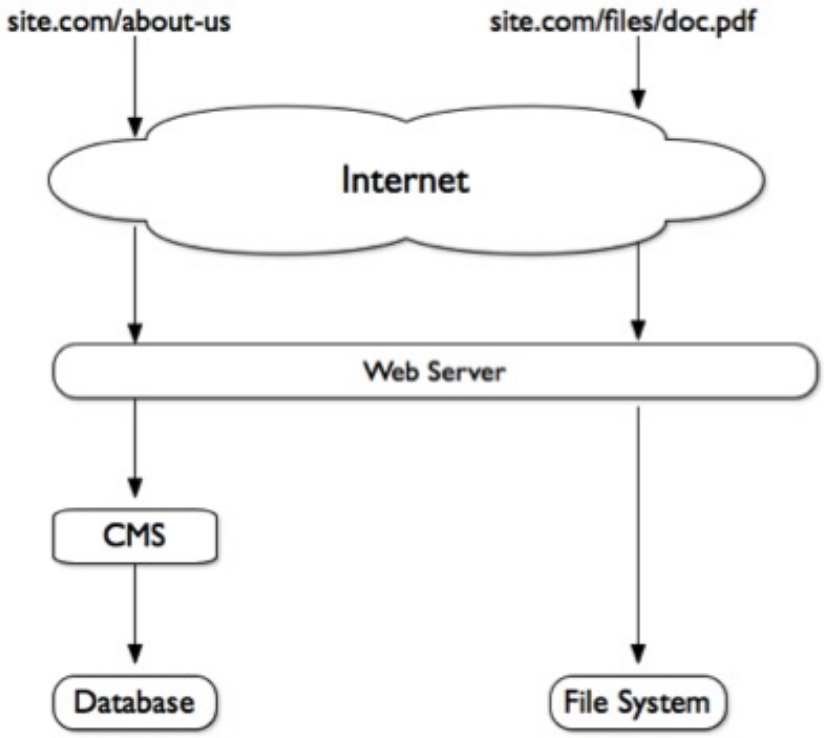

Figure 2.1. The steps when viewing nodes and viewing files without Pyrena. Note that there is no CMS mediation performed when accessing a plain file from it's URL.

operations on that content type. As a clarifying point, the mapping from Content Type to Node is not part of the Drupal access control policy, and is only shown here for completeness' sake. The Content Type to Node mapping is related to a node's structure, and is determined once during the create operation.

An important feature of this policy, when used strictly with database content, is that it does not support *sharing*. In this case, sharing means the containment of a single field value in more than one node. When database content is used, a unique copy of each field value is stored during a create or edit operation. This complicates matters when designing a policy for mediating access to files stored on the local file system, where sharing is the norm. That is, a local file in a Drupal system may be referenced from multiple nodes. Most often, this reference occurs in the form of a URL to the location of the file. Figure 2.1 demonstrates the differences between viewing nodes and viewing files. It is clear the CMS is capable of performing mediation for node-specific access.

Drupal, by default, has no mechanism for mediating access to individual field values. Thus, fields should only be accessible through nodes. If a user can access a field without first accessing its containing node, then the access cannot be mediated. The two basic types of field storage in Drupal are (i) databases, and (ii) files. Field entries stored in databases must be retrieved by Drupal during the rendering of a node, thus, access to database-resident fields is mediated. However, fields containing files values, e.g., a

spreadsheet or Microsoft Word Document, may be accessed directly on the file system due to the fact that Drupal, like most CMSs, runs on top of a web server.

Because the webserver has direct access to the local file system, files that were uploaded as field values can be accessed without accessing their containing nodes. This is known as a *semantic gap*, in which the webserver is unaware of the CMSs access control policy, and thus cannot enforce it. Additionally, neither the CMS or the file system maintain the metadata necessary to mediate files. The CMS holds the access control policy definition, but each specific CMS would need modified to support such a fundamental change. This serves as the motivation for a solution that empowers the web server and file system to perform it's own mediation, instead of altering every CMS that uses a file system for plain file access.

One possible remedy to this problem is to use *munged* file names. For example, if a user uploads `Report.pdf`, the file name on the local file system is changed to `Sgh7FKBu6UrzIG0T.pdf`. In order to access the file, the user must know the name, and thus, must have access to the node containing the link to the file. However, this is a weak form of access control in which the file name acts as a *capability* [2]. If the capability is accidentally leaked, then any user can access the file. Additionally, revoking access is more difficult, as the file name must be changed. A stronger method would be to generate a new random filename for each access. This too has several problems. It makes it impossible for a legitimate user to create a bookmark to the file. Additionally, it requires access to the containing node every time the file is to be accessed, thus doubling the amount of work on the user's and the CMS's parts.

A second possible remedy is to use the CMS database for plain file storage instead of the filesystem. This would break direct file URLs on the file system and require major CMS changes to support. The CMS would need to be altered to not only store the file, but create a new interface for file access. Such an interface would need new file-specific mediation and ensure the proper metadata was maintained for files. While this may be advantageous to do when creating a new CMS, this approach would require every existing CMS to be substantially changed.

Drupal attempts to supply a very coarse grained solution to this problem through the use of the *private file system.* The private file system enforces its own policy and file metadata to ensure that a given file can only be accessed by a link in a specified node. That is, the private file system reduces the treatment of files back to the treatment of database content. However, this does not support the sharing described earlier, and thus is inherently limited. For example, consider a content type for an article that contains a picture of the author. Using the private file system, every article written by the same author must have a unique copy of the author's picture associated with it. Thus, any viable solution must support sharing. The private file system blocks all sharing by restricting file access to one node. The original URL to the plain file is blocked, as the CMS reads the file into memory when the node is accessed. A one-time, randomized URL is generated for the private file access, in which the file contents are output from memory. Due to this restriction, plain file URLs traditionally used for file sharing would not work in the mediated private file system.

2.4 Threat Model

Due to the semantic gap between files and Drupal's access control policy, considerations for the threat model involve access of plain files. Apache intercepts requests made to the filesystem that store plain files. As such, the initial interface of threats start with Apache's handling of plain files. We assume the adversary knows the URLs of all sensitive plain files in the system.

User principles perform requests from web browsers to access plain files via URLs. One URL associates with one plain file. Users may be authenticated Drupal users or non-authenticated anonymous traffic. Authenticated users can be assigned roles, which define their capabilities with respect to nodes in the database. Similarly, capabilities are explicitly granted to anonymous users.

User principles upload plain files while authoring nodes. Since Drupal's access control policy defines the node types in which a user can author, this operation can be considered a part of the threat model. But, authoring nodes is already mediated by the CMS. Authoring and potentially uploading plain files present the context we need to associate plain files with their originally authored node type. But, we consider this threat to be properly mediated by Drupal.

The threat model includes a user performing a request to a plain file. Since the plain file originally was uploaded to a node, a user performing the request may or may not have access to the node. This access is mediated by the node's content type, in which Drupal's access control policy defines a list of authorized users and their capabilities for every content type. The threat exists when users access plain files that associate to nodes he or she do not have access to. Since system administrators can update Drupal's access control policy, any mediation that occurs to prevent the threat, must invoke Drupal to dynamically query it's access control policy to authorize requests.

2.5 The Reference Monitor Concept

The direct solution to Drupal's lack of mediation for access to plain files is the inclusion of a dynamic enforcement monitor. A dynamic enforcement monitor must check every access to a particular type of resource against a policy, and block any unauthorized access attempts. But how should one go about implementing such a monitor? As a starting point, any such monitor should satisfy the *reference monitor concept* [3]. A reference monitor must enforce three properties:

1. **Complete Mediation:** The monitor must sit on all possible code paths that could eventually access the protected resource.

2. **Tamperproofness:** The monitor should be protected from tampering by the user or program performing the resource access.

3. **Verifiability:** The monitor should be simple enough in design and implementation to be formally verified.

From the above description, it should be clear that the implementation of a reference monitor is a non-trivial process. Simply determining complete mediation is itself an error prone process, with bugs

being found in even highly scrutinized open source codebases like the Linux kernel [4, 5]. Futhermore, the task of placing a small set of mediation hooks is an ongoing problem [6]. Tamperproofness is itself defined relative to a threat model. Application level enforcement monitors will be subject to tampering from any lower layer of the system. Furthermore, the application cannot be exploitable in any way. As will be seen, the difficulty of achieving the reference monitor guarantees influences our design decision to leverage existing mediation at the file system level at the cost of pushing Drupal access control semantics into the operating system.

Chapter 3

Related Work

3.1 Summary

In this paper, we modify Drupal to enable the Apache webserver to act as a reference monitor that enforces complete mediation on Drupal's content. The reference monitor [3] is an authorization mechanism that enforces an access control policy whenever a subject (e.g., process) accesses an object (e.g., file). As described in Section 2.5, any request by a subject for an object will adhere to the access control policy if the reference monitor satisfies the three guarantees of: *(i)* providing complete mediation, *(ii)* being tamperproof, and *(iii)* being simple enough to aid verification of correctness [7].

Our focus in this work is on providing complete mediation. Verification of complete mediation has been attempted for several reference monitors. Zhang *et al.* [5] use simple manually-specified rules in the static analysis framework CQUAL to verify complete mediation for the Linux Security Modules (LSM) framework [8] in the Linux kernel. Edwards *et al.* [4] use a dynamic analysis of data structure access inconsistencies for the same purpose. Tan *et al.* [9] and Muthukumaran *et al.* [6] propose a static verification technique based on inferring security-sensitive structure member accesses. Instead of verifying complete mediation, other techniques have taken to detecting the absence of complete mediation, an attack called an *authorization* bypass. For example, Dalton *et al.* [10] and Sun *et al.* [64] propose techniques to detect bypasses in PHP web applications. On the strength of these techniques and studies, we assume the complete mediation offered by the Linux kernel and the Apache webserver for file access.

To enable enforcement of Drupal's files by the Linux kernel and the Apache webserver, we had to address the *semantic gap* that existed between these parties. That is, the Linux kernel and Apache webserver need to understand Drupal's subjects and objects in order to enforce access control on behalf of Drupal. Decentralized Information Flow Control (DIFC) systems [11, 12, 13, 14, 15] are another mechanism that enable this. For example, in the Flume [13] system, programs create their own labels or "tags". Objects are labeled using these tags. Access to objects is then enforced by the Linux kernel's LSM reference monitor using the program-defined tags. While we could have used such a system, it requires program code change and deployment on a system that supports DIFC.

3.2 Security Principles

3.2.1 Access Control Models

A critical component of Content Management Systems is access control. Operating System access control methods have often been suggested for use within web applications [16].

Role-based access control (RBAC) is a common access control method for web applications like Drupal. Park et al. [17] describe concepts about how RBAC is used within web applications. Many other models have been proposed which slightly alter the standard behavior of RBAC for other purposes [18].

CMS systems can leverage different semantics to alter access control. Such semantics include more fine-grained access control of user principles to capabilities or content-based access control like taxonomy. Many approaches have been proposed to use application logic or request context to apply more fine grained access control to traditional models [19] [20] [21] [22]. SecurOntology [23] developed an access control framework based on semantic relationships of capability labels. Yun and Seo [24] describe content management information retrieval for using semantic access control.

3.2.2 Trusted Verifiers

A simple question arises when securing CMS node content. Could another person manually review the content to ensure only proper links exist to plain files? One popular technique used in CMS systems is *publication workflows*. Administrators can configure a CMS to force all content to be manually reviewed by an editor which serves as a *trusted verifier*.

In this design, all authored content is sent for approval to editors who are authorized to publish content. This manual verification task is difficult, time consuming, and error prone, especially for editors to perform this task over a large numbers of content providers. With respect to plain files, an editor would need to have predefined knowledge of appropriate use of files that could be used as content. An example of a content workflow includes a user with role *author* creating only unpublished content for a user with role *editor* to moderate and publish.

Joshi et al. [25] describes the error-prone nature of content management processes that leverage workflow management systems (WFMS) to perform privacy-related decision making. This speaks to the motivation of this thesis, which advocates for systematic protection.

3.2.3 Authorization Bypass and Name Resolution

A CMS cannot perform complete mediation to plain files since the web server resolves requests to plain files. Since the CMS performs the authorization to access content, plain files bypass the CMS authorization.

There are a number of examples of tools developed to prevent authorization bypass in both traditional systems and web applications. The BlueBoX tool [26] proposes a sandboxing approach to detect authorization bypass vulnerabilities within system calls. The Rolecast tool [27] performs a static analysis of web applications to identify security operations and then performs authorization checks based on different roles.

Cai et al. [28] describe filesystem races with respect to name resolution vulnerabilities found within Unix. Chari et al. [29] outline privilege escalation concerns that leverage pathname manipulation within an Operating System. For web applications, Stone et al. [30] describe a name resolution vulnerability from a drive-by download attack using an iFrame.

3.3 Attacks

3.3.1 Web Application Vulnerabilities

Web applications, like CMSs, have unique considerations with respect to security. Several papers propose a canonical set of security metrics for web applications [31] [32]. Others address specific security concerns of content management systems [33] [34]. Based on these security metrics, common attacks include cross site scripting, SQL injection, and authorization bypass.

3.3.2 Drive-By Downloads

A drive-by download occurs when a user intentionally or unintentionally downloads a malicious file. This identifies a possible attack for plain files within a CMS.

Cranor et al. [35] outline the challenges of securing file content within a web application and propose a model and implementation to manage file access operations. Cova et al. [36] created an automated method for collecting metrics of drive-by download attacks found in JavaScript. The Prophiler tool [37] uses the metrics and then performs a static analysis on a webpage to identify the existence of vulnerabilities. EvilSeed leverages web search engine queries to collect a list of URLs that contain drive-by downloads by searching for identified metrics found in existing vulnerable web pages [38].

3.4 Existing Defenses

3.4.1 Static Analysis

Static analysis techniques build abstract models based on source code to verify specific behaviors of a program. For example, a static analysis could be performed on CMS source code to identify file operations. This analysis can formally prove aspects of source code. Models are based on identified code metrics and then built and updated throughout the simulation of the code. Wu and Offutt [39] used static analysis to identify static or dynamic components of a web application.

This concept has been used in web applications to try to identify vulnerabilities found in souce code. WebSSARI [40] is a tool developed that builds a lattice based on a static analysis and injects annotations for runtime protections. Xie and Aiken [41] identified security vulnerabilities by analyzing the relationship of specific code blocks with respect to intraprocedural and interprocedural data flows. Son and Shmatifov [42] developed SAFERPHP to perform a static analysis that identifies faulty logic vulnerabilities. FixMeUp [43] performs a static analysis to identify specific security sensitive operations that are missing access control checks.

3.4.2 Dynamic Analysis

Dynamic analysis evaluates a program over a set of test cases to verify the existance of specific behaviors. Our use of an automated testing framework is an example of a dynamic analysis. Given potentially malicious input, dynamic analysis executes a program to verify that the behaviors are adequately addressed within the program. This input should mimic an attacker using the program to identify vulnerabilities.

Artzi et al. [44] describe general bug finding techniques that use dynamic analysis of web applications. Liu [45] [46] [47] proposes a dynamic analysis which collects possible data flows within an uncompiled JSP web application.

Dalton et al. [10] developed a runtime tool *Nemesis* that tracks user authentication throughout web application data flows and identifies missing authentication checks based on programmer-supplied access control specification. Similarly, Felmetsger et al [48] developed an anomaly-detection tool Waler that uses a dynamic analysis to learn the expected execution behaviors of a web application.

3.4.3 Symbolic Execution

King [49] introduced symbolic execution as an analysis tool. Symbolic execution is a specific analysis approach that performs a simulation to identify all possible data flow paths in a program. This simulation often maps variables with a set of possible values to specific states within a program. It is often used to verify conditions and make assertions based on possible test cases. One such example of symbolic execution is *dynamic taint analysis* [50]. A common modeling approach for the simulation leverages *weighted pushdown systems* [51].

More recently, this approach has been applied to web applications. Kudzu is a symbolic execution tool used to secure client-side JavaScript [52]. Chaudhuri and Foster [53] used symbolic execution to identify cross-site scripting, cross-site request forgery and authorization bypass attacks with the Ruby on Rails framework.

3.4.4 Policy Analysis

Policy analysis takes a given policy and provides a mechanism for evaluating access control. Based on the *Satisfiability Modulo Theory* (SMT) class of decision problems, policies are often specified as constraints and tools can then formally evaluate the satisfiability of constraints. These tools often use first-order logic with a specified set of security requirements to identify misconfiguration of systems through conflicts in policy. Eshete et al. [54] evaluated misconfiguration of the servers used to run web applications.

Web applications, like CMS systems, often use a document standard for policy specification. Shannon et al. leverage a finite state machine with an SMT solver for analyzing Java web applications [55]. Sohr et al. [56] and Tonella et al. [57] propose tools that leverage UML-based policy specification and an analysis tool that leverages first-order logic. XACML is a widely used standard for web application policy specification. Many approaches were proposed that leverage XACML to evaluate web application policy [58] [59] .

3.4.5 Automated Testing

Automated testing is used to evaluate the behaviors of software against pre-existing use cases or by producing use cases inferred from some analysis. Yang et al. [60] breaks down a web application into components and proposes specific automated testing for each component.

Blackbox testing is one automated approach to test a running web application. Doupe et al [61] evaluate blackbox testing approaches used for web applications. Crawlers are often used to automatically identify the attack surface of web applications [62]. Huang et al. [63] also leveraged blackbox testing to observe web application behaviors with fault injection. Sun et al. [64] leverages a crawler to build role-based sitemaps to then perform automated testing. Each role attempts forced browsing of pages not found within the role's sitemap to identify missing access control.

Chapter 4

Design

4.1 Problem Statement

The fundamental problem being addressed is Drupal's lack of complete mediation for access to content stored in plain files on the local file system. There are two main causes for this. First and foremost, Drupal has no reference monitor for plain file access. Furthermore, there is no extension of the RBAC policy to plain files, and no means of supporting sharing. Additionally, no additional metadata is maintained for plain files to support such a policy. The second cause for Drupal's lack of complete mediation is its placement within the software stack. Drupal does not directly receive web requests. Instead it is invoked by a web server (e.g., Apache) that directly receives requests. When the web server receives a request for a file stored on the local file system, it can directly serve the file, with no access control check. In fact, in most cases, the web server is not even aware of what files contain content or were uploaded through Drupal nodes.

The second problem can be addressed easily enough by making a small modification to the web server's configuration to force all requests for Drupal owned files to be passed through Drupal. This can be accomplished via URL rewriting, for example. However, the first question is more fundamental, as the creation of a reference monitor that makes the above described guarantees is very difficult. Thus, the addition of a reference monitor to any given CMS is a non-trivial, and likely error-prone task. Thus, for our solution we take a different approach: *we focus on leveraging the existing file reference monitor in the underlying operating system's file system to mediate access to Drupal files.* In other words, as a part of any file access, either Drupal (the CMS) or the web server must eventually make some system call to the operating system as part of accessing the file. When this system call occurs, the file system will perform an access check equivalent to the enforcement of an application-level enforcement monitor.

The design choice of leveraging the file system reference monitor saves the trouble of implementing an application level reference monitor. However, it also introduces a new challenge: *How can the semantic information present at the CMS level being exported to the file system level?*. In other words, how can we inform the file system reference monitor of the Drupal level subjects (`uids`), objects (files),

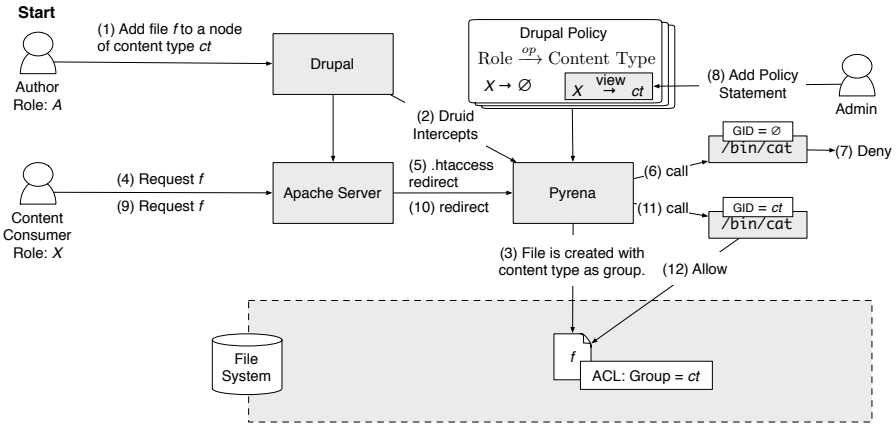

Figure 4.1. The flow of operations in content authoring, viewing, and administration using Pyrena. Note that Drupal policy updates do not access the file system layer.

and operations? This is complicated by the fact that most file-system level reference monitors do not directly support RBAC policies. Thus, we must translate the Drupal RBAC policy into an equivalently enforceable file system policy. Another important consideration is how the operating systems existing user management framework can be used to represent a Drupal user, and their corresponding roles. Finally, policy updates change and may be frequent, and thus should not require extensive system calls or ongoing modification of file metadata.

Another important notion that must be addressed is that of sharing. Sharing of files can occur in ad-hoc manners. For example, a content author may upload a file while creating a new node. A completely different author may then include a hyperlink to that file within an HTML field of a completely different node. This introduces an additional problem: *to whom should the file be accessible for reading?* Should it be accessible to those who can access the content type of the containing node? Should it be accessible to anyone who can read content from the node's creator? Or, should it be possible to create an arbitrary policy for each file? We will address this problem further in the following section.

We can summarize the challenges for our design of Pyrena as follows:

- The semantic gap between the CMS level access control policy and the file system level enforcement mechanism must be crossed.
- Any policy for mediating file access must enable *sharing.*
- Policy updates should not impose heavy burden on the host.

We address each of these problems in the following section.

4.2 Architecture

A typical solution to Drupal's lack of file mediation would be the creation of an application-level reference monitor. However, given the difficulty of reference monitor implementation, as described above, any such

implementation would require significant effort. One example is the use of automatic hook placement within the web application. This problem is very hard and the resources we are trying to protect are files. Plus, requests for plain file access are processed by Apache and not the web application. There would be only a limited advantage by using such an approach. Implementing a reference monitor for a particular CMS, such as Drupal does nothing to help solve the basic reference monitor difficulties for other CMSs that have completely different codebases. For example, determining which code paths to cover to achieve complete mediation in Drupal tells us nothing about how to achieve it in any other CMS. Other CMSs aside, there is little advantage to performing the exercise for Drupal alone, as the only resource being mediated are plain files, which are already protected by a file system reference monitor. Thus, we would like to leverage this reference monitor to protect Drupal content stored in plain files.

We choose to use the file system reference monitor for several reasons. First, significant effort has already been put into assuring the correctness and complete mediation of file system level code. Second, the use of the underlying file system allows this problem to be fixed for a large number of CMSs with little additional overhead per-CMS. Semantic gaps between application level components such as web servers and CMSs are not a problem, as all file requests must pass through the file system reference monitor, regardless of their origins. Thus, even if the Apache web server receives a request for a file containing Drupal content, it must go through the file system mediation, even if Drupal itself is completely bypassed. Finally, as will be seen, we demonstrate that the practical problem of exporting application level access control semantics to the file system requires only a small amount of effort, and thus is preferable to the method of application layer reference monitor implementation.

An example of the common CMS tasks involving Pyrena is shown in Figure 4.1. Steps 1-3 show the uploading of a file as part of node creation or editing. The containing node has content type ct, thus, the associated group is added to the file's access control list. Steps 4-7 show a failed attempt to access the file, by a content consumer with insufficient permissions. The `cat` utility is run with the group IDs available to users with role X. Step 8 shows the modification of the application-layer policy by a system administrator. It is important to note that because the *content type* is stored in the file's ACL, the file system need not be accessed for this step. This greatly increases the efficiency of policy changes involving content types for which there are many associate files. Finally, steps 9-12 show a successful access by the content consumer, after the policy modification in step 8. The policy change affects only the GIDs that the `cat` utility runs with, thus imposing virtually no performance overhead.

As described above, there are three basic challenges to leveraging the file system for mediation of plain files containing Drupal content: (i) Application level semantics must be represented within the file system's policy framework (ii) The chosen policy for file access must support file sharing in some useful way (iii) Updates to the application level policy should not impact system performance, or require time consuming updates to large numbers of files. We will address problem (ii) in Section 4.3 and problems (i) and (iii) in Section 4.4.

4.3 Pyrena Policy

Under Drupal's normal policy framework, each field value is tied to a specific node. This is because, with the exception of files, field values are stored in databases, and there is no clean method of sharing a single field value between Drupal nodes. Files, on the other hand, are more decoupled from their containing node. A file can be referenced in many hard-to-track ways such as through hyperlinks within nodes. I.e., an author can create a node that contains a large free-form text field. This field can contain some HTML, including a link to a plain file. In the most general cases, there is virtualy no way to determine if this link has been put in place. This gives rise to two issues. First, which users should able to access a plain file, and second, what functional problems can arise due to links from nodes to plain files.

To answer the first question, each file is labeled with the content type of the node through which it was uploaded. Recall that plain files are uploaded through a field in a node during creation or editing. This node then contains an HTML link to the file. It is reasonable to assume that if a user can view the content of that node, then the user should also be allowed to access the file content. However, because of the possibility of hyperlinking, and because Drupal access control occurs at the granularity of content types, the file should be accessible to any user that can view nodes with the same content type as the file's origin node.

The second problem regards whether this will break any existing functionality or defy user expectations. In fact, it may. Consider that a file uploaded through a node of content type ct_1. Thus, the file also has label ct_1. A user with no access to ct_1, but with access to another type ct_2 can place an HTML link to the file in a node of type ct_2. However, when users with permission to view ct_2 content (but not ct_1) attempt to follow the link, they will be denied access to the file. The most straightforward solution to this problem is to directly monitor the creation of nodes. While it is not always possible to detect hyperlinks in code, e.g., through obfuscated javascript, Drupal provides mechanisms for only allowing basic HTML tags in text fields. Thus, a simple check can examine the targets of any anchor tags.

4.4 Crossing the Semantic Gap

The goal of Pyrena is to export Drupal's access control semantics into the existing file system, where they can be used to protect content stored in plain files. To do this, Pyrena must first represent the subjects, objects, and operations in Drupal's RBAC policy using the existing primitives in the host system. This must be done in such a way that access control decisions made by the file system alone are identical to those decisions that would be made if Drupal had a reference monitor that intercepted all URLs for plain file content.

To emulate CMS semantics, we consider Drupal's RBAC policy when viewing nodes. Any policy for plain files should mimic policies for nodes, which are defined by the node's content type. Each content type has explicit capabilities that can be granted to roles. A request is made to a node by an authenticated user or defaulted to anonymous access. Drupal's RBAC policy dereferences the user's capabilities through his or her assigned roles. Such capabilities define the set of content types users are able to access. This set of content types are used to enforce node access. A similar model for files would leverage these same

content type capabilities that are used for the access control of nodes.

For this mapping, we need to discuss the target model of the host OS's file system. Once we have both an application level policy model and a file system level model, can we describe how the former is mapped to the latter. Our model should be generic enough that it applies to the most widely used file systems typically found in servers running CMSs. Because the file system depends on the user and roles defined by external parts of the operating system, this must also be included in our model.

Our model assumes the presence of several basic primitives in the host OS file system and user management system. First, it is assumed that a basic Access Control List (ACL) mechanism is present. Ideally, this mechanism can support allow-by-group style rules. (Allow-by-user is also sufficient, but not optimal.) The ACL frameworks found in many implementation of the Berkeley File System (BFS), or the Access Control Entities (ACEs) in Microsoft Windows suffice. Second, the host user management system must support of spawning a subprocesses and changing its group ID, e.g., the effective group ID in UNIX.

Given the above requirements and policy described in Section 4.3, Pyrena maps an application layer policy to the file system primitives as follows. When a user uploads a file to a node in Drupal, the content type of the file is added as a group to the file's ACL. (There is a group for each content type defined in the host OS's user and role management system.) When a user later tries to access the file by sending a URL to the web server, the modification to the web server's configuration will cause the request to be passed to Drupal. At this point, Pyrena intercepts the request and spawns a new instance of the `cat` utility using the set of `gid`s corresponding to the Drupal user that initiated the web request. If the user has a GID corresponding to the one in the file's ACL, the access is allowed. Otherwise, it is denied.

The use of the ACL is required here, as some file systems only support a single group by default, and this group must be used by the server software stack. For example, in many web server systems, files will often have the group `www` or `www-data`, which is the effective group of the web server and other scripts or processes that it spawns. Replacing this field with the content type will often break the system functionality, hence the need for ACLs.

A more elegant design includes the use of the `setuid` utility and `SELinux`. In the current Pyrena design, Apache processes spawned from plain file requests are always run as a web-server specific user, like `www-user`. `HTACCESS` redirects the request back to the new CMS file interface for mediation. However, a streamlined approach is to leverage SELinux to intercept requests to plain files and switch the process to run as the OS user corresponding to the requestor's CMS user. Such an approach eliminates the need for Pyrena's file interface in the CMS, which currently spawns the `cat` utility as the OS user tied to the requestor's CMS user. Support for this design requires the `mod_selinux` package installed on the web server. Mod_selinux applies a custom policy to set the security context of a request. In this example, the policy would be specified by the CMS and enforced by SELinux on web server processes to plain files. Currently, the package is only available in the `Red Hat Fedora` Linux distribution. As such, Pyrena opts for a solution that can be adopted on more platforms.

Chapter 5

Evaluation

CMSs may be used for a large number of applications, each demanding their own special modifications to the vanilla CMS and special security policies. Of course, some of these modifications may not have been made with the expectation of a Pyrena-like mechanism being in place. Additionally, the default Drupal implementation assumes that files are either completely public or completely locked to a single node. We would thus like to evaluate whether Pyrena is compatible with these Drupal use cases. Furthermore, we evaluate the performance overhead of Pyrena on a vanilla Drupal installation. In this section, we perform both of these evaluations, starting with compatibility, followed by performance.

5.1 Compatibility

We wish to evaluate Pyrena's compatibility under typical Drupal use cases. To do this, we run browser-driven tests on Drupal instances running the two most common use cases, *organic communities* and *e-commerce*. (We detail both below.) This experiment is designed to elicit both false positives and false negatives. Here, a false positive is any legitimate access to file data that is blocked by Pyrena. Similarly, a false negative is a file access that should have been blocked, but was not. The goal is to first identify false positives and false negatives and try to understand in a general sense how the application-specific Drupal modifications caused them. Our results are promising, showing that Pyrena causes few false positives and no false negatives. Additionally, the same generic mechanism can be applied to address false positives.

This experiment uses a *behavioral driven* evaluation. In this type of experiment, user behavior is programmed according to a script that executes behaviors within common web browsers. No setup is done by hand or otherwise outside of the behavioral driven framework. Thus, all accounts, content types, and content created for the experiment is launched from our browser behavior driver, starting from a bare copy of each of the customized Drupal installations we consider.

To drive the experiments, we use the Behat framework[1]. Behat provides an English-like scripting language for describing the behaviors in each experiment. An example Behat script is shown in Figure 5.1.

[1] `http://behat.org/`

```
@api
Feature: Test role 1 and role 2
 Scenario: Two users with only role 1 and role 2 \
   respectively should not be able to access files
   Given I am logged in as a user with the "r1" role
   And I am on "node/add/ct1"
   And I fill in "test r1" for "Title"
   And I fill in "testing body" for "Body"
   And I attach the file "image.jpg" to "files[field_file_und_0]"
   And I fill in "test-r1" for "URL alias"
   And I press "Save"
   Then print current URL
   Then I am on "test-r1"
   Then I should see "file:"
   And I click on the first link in "div.field-name-field-file"
   Then print current URL
   #Then show last response
   #Then the response status code should not be 404
   Given I am logged in as a user with the "r2" role
   And I am on "test-r1"
   Then I should see "file"
   And I click on the first link in "div.field-name-field-file"
   Then print current URL
   #Then show last response
   Then I should see "Not Found"
```

Figure 5.1. An example Behat script to drive behavior of account and content creation.

The script is a use case in which role `r1` has permission to content type `ct1` and role `r2` does not. Given a user logged in that is assigned a role of `r1`, a node is created of content type `ct1` in which a file `image.jpg` is uploaded. Next, a user of role `r2` logs in and views the node, this test checks that they are shown "Not Found" when direct linking to `image.jpg`.

Our evaluation was carried out on two custom Drupal *distributions*. A distribution is a copy of vanilla Drupal with code and configuration modifications to tailor it to a specific use cases. We consider the two most popular Drupal distributions: *commons*, and *commerce*. The specifics of each are as follows.

Commons. Commons is a distribution for *organic communities*, similar to social networks. Organic communities share data differently than what is allowed by a vanilla Drupal distribution. In an organic community, as opposed to a role, users can administer permissions within their own community. Thus, a user can choose specific other users to view their content. This policy stacks on top of Drupal's existing RBAC policy. Thus, in order to view some content, a user must have the appropriate role in the RBAC policy, as well as be within the organic community. In short, the organic community creates policies that are equally or more restrictive than the vanilla RBAC policy. To date, organic communities is the second most downloaded Drupal distribution, with 86,000 downloads as of December 2013.

Commerce. Commerce is a distribution for implementing online stores. It implements payment gateways, integration with common shipping systems, and cart features. It comes with a stock set of users and

roles that are based on a vanilla Drupal RBAC policy. These users and roles are geared towards the tasks of creating new products and product categories, viewing and buying products, and adding informational pages. Commerce uses plain files for a number of reasons, the most common of which being product images. The private file system is not appropriate for this, as often, users will want to include a direct link to the product image, or a single image may be used to represent a family of products on different pages. Currently, commerce is the most popular Drupal distribution with over a quarter million downloads as of December 2013.

We performed two basic types of experiments, a *default*, and a *commons-specific* test. The default test is meant to test vanilla Drupal functionality under the customized distribution. The commons specific tests exercise functionality only available in the commons distribution, i.e., the creation of organic communities or new products. The test procedures for commons and commerce are as follows.

Default test case. The default test case is the same for both the commons and commerce experiments. It is as follows. First, three new roles are created `r1`, `r2`, and `r3`. Additionally, a user is created for each role. A content type called `article` is created, and `article` permissions are given to `r1` and `r2`. The `article` content type contains a field for a plain file upload. The test then logs in as `r1`, and creates a new node of content type `article`, uploading a file to the plain file field. We note that a user with `r2` should be able to access the file, and a user with `r3` should not. The test then logs in as a user with role `r2` and attempts to access the plain file. If the access is denied, a false positive is logged. Subsequently, the test logs in as a user with role `r3`, and attempts to access the file. If the access succeeds, a false negative is logged.

Commons. After manually installing the Commons distribution, we found a feature that enhanced Drupal's access control by further restricting access by assignment of a social group. This occurs by assigning users one or more social groups and assigning a social group to a node. This affects viewing nodes, as a node will be further restricted if assigned a social group. Drupal's standard role-based access control still verifies user access on a user's assigned roles and the role's capabilities with respect to the content type of a node. An additional access check is done to ensure a user is a member of the social group belonging to the social group of the node.

5.2 Implementation

Drupal, like many CMS systems, provides a framework to extend out-of-the-box functionality for specific business use cases. Pyrena has been implemented as a Drupal module. The CMS framework provides the hooks necessary to alter CMS behaviors, like file management, user management, and access control. Pyrena leverages the Drupal module to label uploaded files, define the new CMS-sponsored file access interface, and create the HTACCESS redirection to the new file interface. Recall in section 4.4, we describe a more elegant solution that leverages `SELinux` to switch the security context of the web request. While this would eliminate the need for Pyrena's CMS-sponsored file interface, support for the tt mod_selinux package is only available in select Linux distributions.

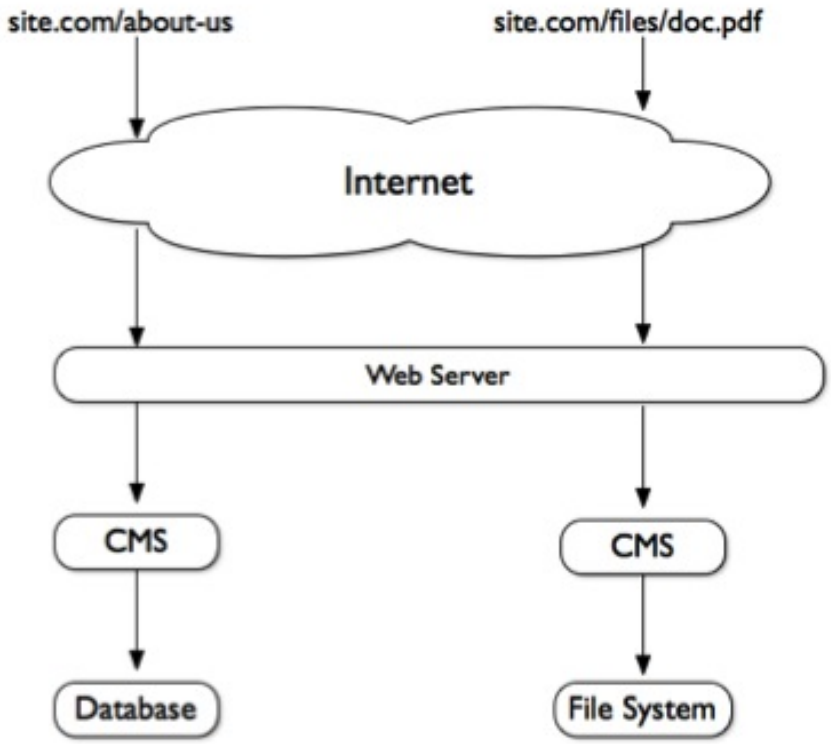

Figure 5.2. The steps when viewing nodes and viewing files with Pyrena. Note that there is a new CMS mediation performed when accessing a plain file from it's URL.

Within Pyrena's Drupal module, we alter the existing file access workflow found in figure 2.1. Recall Pyrena's operations found in figure 4.1, the CMS must be invoked at runtime when a request is made to a file to properly resolve the authenticated user and load the file contents as the corresponding OS user. The new workflow is defined in figure 5.2. This figure demonstrates how the CMS is invoked to perform mediation before the file system is accessed. Within the CMS file interface, the following steps must be performed:

1. Bootstrap the CMS
2. Resolve the CMS authenticated user
3. Run the *cat* command to load the file as the corresponding OS user
4. Return the result

Comparable modules could be set up for other CMS systems that leverage the file system as a data store. For our implementation, Pyrena was developed on the Linux distribution *Ubuntu* version 11 with the web server *Apache*, the database server *MySQL*, and the server-side scripting engine *PHP*. Drupal supports this suite of tools, like many other CMS systems.

Six separate Drupal instances were set up on the aforementioned platform for this experiment. We set up two instances of a basic Drupal installation, one with Pyrena and one without. We also set up two

instances of Drupal Commerce and Drupal Commons, respectively. The instances in which Pyrena was not installed served as a baseline for performance metrics and expected behaviors.

5.3 Results

The test cases and distributions described above are organized as follows.

- The default test case is run on vanilla Drupal with Pyrena.
- The default test case is run on commons with Pyrena.
- The commons test case is run on commons with Pyrena.
- The default test case is run on commerce with Pyrena.

The results for each are as follows.

Default on Vanilla Drupal. As expected, there were no false positive or false negative access control decisions under a vanilla Drupal distribution. This is mainly because the file system-level policy is the only one regulating the behavior of plain files. Furthermore, Pyrena's principal of typing each file to the content type of the node through which it was uploaded is consistent with the treatment of content stored in databases.

Default on Commons. The default test on commons also exhibited no false positives or false negatives. Recall that commons requires specific social groups to be defined, which can only *add more* restrictions to the existing Drupal RBAC policy. As the default test case does not support social group definitions, no policy inconsistencies were encountered.

Commons-specific on Commons. The commons specific test did elicit a single false positive. The false positive was introduced due to the additional authorization required for social groupings. Plain files are only annotated with node content types and therefore Pyrena lacked the context to provide the additional access control needed for commons. This is easily resolved, by creating a commons-specific implementation of Pyrena that also annotates files with social groupings and verifies social grouping membership upon plain file access. This demonstrates that Pyrena effectively presents a generic mechanism that can be tailored to address varying access control models in a CMS.

Default on Commerce. The commerce distribution did not exhibit any false positives or false negatives under the default test case. We conclude that in general, only distributions that directly stack new policy checks on top of the existing RBAC policy, such as commons, can cause incompatibilities when Pyrena is being used.

5.4 Performance

We now wish to evaluate the overhead introduced by Pyrena into a running Drupal system. We executed the default test above under both a vanilla Drupal install, and vanilla Drupal with Pyrena. We repeated the same performance evaluation for Commons and Commerce. We ran three instances of the default test

Table 5.1. Performance overheads for vanilla Drupal

Test	T_1	T_2	T_3	T_4	T_5
Overhead	5%	6%	6%	1%	4%

Table 5.2. Performance overheads for Commons

Test	T_1	T_2	T_3	T_4	T_5
Overhead	7%	8%	11%	2%	7%

with different parameters. In each instances, a different combination of users logged in, and files created were executed. We call these three tests $T_1 - T_3$. Additionally, we ran a test T_4 in which a user logs in and authors a node with no plain file uploaded. T_5 is a test specifically measuring the authoring of a node with a plain file uploaded. Therefore, comparing T_4 and T_5 measures the overhead added by annotating the file. The results are shown in Table 5.1, Table 5.2, and Table 5.3. The performance overhead of Pyrena over the baseline system is shown for each.

Each test was run for 20 iterations, and the three most outlying tests were thrown out. Thus, the table contains averages of 17 trials for each test. The main sources of over head are the handling of the Drupal application level policy by Pyrena, and the spawning of an additional process (`cat`).

Performance is directly proportional to the amount of time needed to bootstrap the CMS when accessing a file. The worst case demonstrated 15% overhead, primarily due to the inclusion of a new CMS file mediation interface. Recall, we leverage HTACCESS to redirect all plain file requests to a new CMS sponsored file interface within the CMS. This interface must bootstrap the entire CMS application to perform this mediation. The CMS must be fully bootstrapped to process the authenticated user performing the request and to leverage the framework to run the *cat command from the system.*

To optimize, CMS specific scripts could be developed to only bootstrap the necessary CMS subsystems needed for the file interface. While this is not a CMS-agnostic approach, it is only necessary to bootstrap the CMS authentication and perform a system call from the CMS. While it would be difficult to speculate about the amount of performance gain, it is clear performance gains can be made from removing the extraneous subsystems not pertinent to the new file interface.

Table 5.3. Performance overheads for Commerce

Test	T_1	T_2	T_3	T_4	T_5
Overhead	11%	15%	14%	3%	8%

Chapter 6

Future Work

We claim that our system modeling approach improves integrity of plain files used as content within a CMS, but it is clear that limitations still exist. Based on known limitations, there are opportunities for future work.

The most obvious limitation applies to existing systems that want to use this approach. This technique only works when plain files are uploaded. An existing system would have existing plain files and content. It would be challenging to accurately determine which content types to annotate to a plain file. And, multiple nodes may be currently linking to the file as content. Drupal supports a feature called revisioning, which maintains a log of changes to a node. This log could determine which node and it's content type saved the file first. Revisioning is optional, so this approach is no guarantee. Adding file-level access control could break access to users that do not have access to the file's original content type. This adds a layer of complexity and uncertainty to our approach, but presents opportunities for future work.

Another limitation is the fixed access control modeling. We assume a one-to-one labeling between user and content type. This generic access model can be abstracted to work with many CMS of different designs. In the case of Drupal, plain file labels based on Drupal's originating content type and membership to the content type based on Drupal's users, roles, and permissions model. In the case of commons, the social group added complexity to a single membership model. One potential option is to add hooks into the plain file labeling and plain file access control of Pyrena. *Some CMSs like Drupal offer the ability to extend it's hook system within it's framework.*

One potential for future work is the implementation of user intent within a CMS. This model varies from the proposed systematic protection provided by Pyrena*'s reference monitor. And, this would likely include CMS-specific enhancements due to the varying CMS framework. However, such a strategy would grant authors the ability to specify sharing settings for plain files. Maintenance of plain file access control becomes a new concern of this approach. However, components of* Pyrena *could be used to produce a new solution within a CMS.*

Since plain files are being labeled, a secondary systematic protection could be considered. SELinux policy specification could be leveraged to restrict inappropriate file access from web server requests. The

protection mechanism would replace the current use of the cat utility within the plain file CMS interface. This design would require SELinux to be supplied a changing specification of the CMS access contol policy. And, SELinux would require a CMS-specific mechanism to derive the user associated to the web server's HTTP request. A request without a user would assume anonymous access, just like the current Pyrena *reference monitor. This approach would be evaluated as future work.*

Lastly, it is well known that a web server can run tools like anti-virus on a file system. But, using Pyrena*, plain files are labeled with CMS-specific metadata. This opens up new opportunities for system tools traditionally reserved for only file systems to be more involved in CMS security. Furthermore, a system would have more clues found in a CMS to solve a security incident. For example, CMS users could be notified or disabled within the CMS if files the user uploaded contain viruses. Or, CMS content could automatically be taken down from public if a link to a plain file is malicious.* Pyrena *presents an opportunity for both the system and the CMS to engage in a new way to enhance security within the entire system. Future work could identify the benefits of* Pyrena *with respect to collaborative security between a system and the CMS.*

Chapter 7

Conclusion

In this thesis, a vulnerability was identified within many common CMS systems and their approach to mediation of plain file access. The permitted out-of-the-box open file sharing could enable plain files to be used in ways the original author did not intend. This is due to a semantic gap between the file system and the CMS. We introduced Pyrena *to solve this limitation. A file-specific access control policy was proposed that mimicked other CMS content served from a database. To implement,* Pyrena *leveraged a technique for file system labeling that replicated the CMS access control policy onto the file system. Also,* Pyrena *redirects all requests to access plain files to a new file mediation interface within the CMS.*

To test this new defense, we selected the popular Drupal CMS. The concepts used in Pyrena*, while CMS-agnostic, were developed with Drupal and it's framework for this experiment. Drupal distributions were identified to test* Pyrena *for common CMS implementations. There was only one identified case in which* Pyrena *failed to properly mediate a file, which could be easily addressed by adding an additional file label. These typical CMS use cases were tested with a behavior-driven testing framework, Behat. This proved that* Pyrena *does not break the existing CMS functionality. Tests comparing Drupal with and without* Pyrena*, only demonstrated slight performance degradation.*

Pyrena *demonstrated a new generic technique to mediating the sharing of file assets within a content management system and opened the door to future opportunities to allow the Operating System to have a stronger role in security operations traditionally reserved to the content management system.*

Bibliography

[1] "The Apache web-server," `http://www.apache.org`.

[2] LEVY, H. M. (1984) *Capability-based Computer Systems*, Digital Press, available at `http://www.cs.washington.edu/homes/levy/capabook/`.

[3] ANDERSON, J. P. (1972) *Computer Security Technology Planning Study, Volume II*, *Tech. Rep. ESD-TR-73-51*, Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA.

[4] EDWARDS, A., T. JAEGER, and X. ZHANG (2002) "Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 225–234.

[5] ZHANG, X., A. EDWARDS, and T. JAEGER (2002) "Using CQUAL for Static Analysis of Authorization Hook Placement," in *Proceedings of the 11th USENIX Security Symposium*, pp. 33–48.

[6] MUTHUKUMARAN, D., T. JAEGER, and V. GANAPATHY (2012) "Leveraging "choice" to automate authorization hook placement," in *ACM Conference on Computer and Communications Security*, pp. 145–156.

[7] JAEGER, T. (2008) *Operating System Security*, Morgan & Claypool.

[8] WRIGHT, C., C. COWAN, S. SMALLEY, J. MORRIS, and G. KROAH-HARTMAN (2002) "Linux Security Modules: General Security Support for the Linux Kernel," in *Proceedings of the 11th USENIX Security Symposium*, USENIX Association, Berkeley, CA, USA, pp. 17–31.
URL `http://dl.acm.org/citation.cfm?id=647253.720287`

[9] TAN, L., X. ZHANG, X. MA, W. XIONG, and Y. ZHOU (2008) "AutoISES: Automatically Inferring Security Specifications and Detecting Violations," in *Proceedings of the 17th Conference on Security Symposium*, SS'08, USENIX Association, Berkeley, CA, USA, pp. 379–394.
URL `http://dl.acm.org/citation.cfm?id=1496711.1496737`

[10] DALTON, M., C. KOZYRAKIS, and N. ZELDOVICH (2009) "Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications," in *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, USENIX Association, Berkeley, CA, USA, pp. 267–282.
URL `http://dl.acm.org/citation.cfm?id=1855768.1855785`

[11] DENNING, D. (1976) "A Lattice Model of Secure Information Flow," *Communications of the ACM*, **19**(5), pp. 236–242.

[12] MYERS, A. C. (1999) *Mostly-Static Decentralized Information Flow Control*, *Technical Report MIT/LCS/TR-783*, Massachussetts Institute of Technology, ph.D. thesis.

[13] KROHN, M. N., A. YIP, M. BRODSKY, N. CLIFFER, M. F. KAASHOEK, E. KOHLER, and R. MORRIS (2007) "Information flow control for standard OS abstractions," in *Proceedings of the 21st ACM Symposium on Operating Systems Pr inciples*, pp. 321–334.

[14] EFSTATHOPOULOS, P., M. KROHN, S. VANDEBOGART, C. FREY, D. ZIEGLER, E. KOHLER, D. MAZIÈRES, F. KAASHOEK, and R. MORRIS (2005) "Labels and Event Processes in the Asbestos Operating System," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, ACM, New York, NY, USA, pp. 17–30.
URL http://doi.acm.org/10.1145/1095810.1095813

[15] ZELDOVICH, N., S. BOYD-WICKIZER, E. KOHLER, and D. MAZIÈRES (2006) "Making Information Flow Explicit in HiStar," in *OSDI*.

[16] KORNIEVSKAIA, O., P. HONEYMAN, B. DOSTER, and K. COFFMAN (2001) "Kerberized credential translation: a solution to web access control," in *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, USENIX Association, Berkeley, CA, USA, pp. 18–18.
URL http://dl.acm.org/citation.cfm?id=1251327.1251345

[17] PARK, J. S., R. SANDHU, and G.-J. AHN (2001) "Role-based access control on the web," *ACM Trans. Inf. Syst. Secur.*, **4**(1), pp. 37–71.
URL http://doi.acm.org/10.1145/383775.383777

[18] AL-KAHTANI, M. A. and R. SANDHU (2004) "Rule-Based RBAC with Negative Authorization," in *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, IEEE Computer Society, Washington, DC, USA, pp. 405–415.
URL http://dx.doi.org/10.1109/CSAC.2004.32

[19] KAPSALIS, V., D. KARELIS, L. HADELLIS, and G. PAPADOPOULOS (2005) "A context-aware access control framework for e-service provision," in *Industrial Technology, 2005. ICIT 2005. IEEE International Conference on*, pp. 932 –937.

[20] YI-QUN, Z., L. JIAN-HUA, and Z. QUAN-HAI (2007) "A General Attribute based RBAC Model for Web Service," in *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pp. 236 –239.

[21] LIU, M., H. QING GUO, and J. DIAN SU (2005) "An attribute and role based access control model for Web services," in *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*, vol. 2, pp. 1302 –1306 Vol. 2.

[22] SHEN, H.-B. and F. HONG (2006) "An Attribute-Based Access Control Model for Web Services," in *Proceedings of the Seventh International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '06, IEEE Computer Society, Washington, DC, USA, pp. 74–79.
URL http://dx.doi.org/10.1109/PDCAT.2006.28

[23] GARCÍA-CRESPO, A., J. M. GÓMEZ-BERBÍS, R. COLOMO-PALACIOS, and G. ALOR-HERNÁNDEZ (2011) "SecurOntology: A semantic web access control framework," *Comput. Stand. Interfaces*, **33**(1), pp. 42–49.
URL http://dx.doi.org/10.1016/j.csi.2009.10.003

[24] YUN, B.-H. and C.-H. SEO (2003) "Semantic-Based Information Retrieval for Content Management and Security," *Computational Intelligence*, **19**(2), pp. 87–110.

[25] SPAFFORD;, J. B. D. J. W. G. A. A. G. E. H. (2001), "SECURITY MODELS FOR WEB-BASED APPLICATIONS," .

[26] CHARI, S. N. and P.-C. CHENG (2003) "BlueBoX: A policy-driven, host-based intrusion detection system," *ACM Trans. Inf. Syst. Secur.*, **6**(2), pp. 173–200.
URL http://doi.acm.org/10.1145/762476.762477

[27] SON, S., K. S. MCKINLEY, and V. SHMATIKOV (2011) "RoleCast: finding missing security checks when you do not know what checks are," *SIGPLAN Not.*, **46**(10), pp. 1069–1084.
URL http://doi.acm.org/10.1145/2076021.2048146

[28] CAI, X., Y. GUI, and R. JOHNSON (2009) "Exploiting Unix File-System Races via Algorithmic Complexity Attacks," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, IEEE Computer Society, Washington, DC, USA, pp. 27–41.
URL http://dx.doi.org/10.1109/SP.2009.10

[29] CHARI, S., S. HALEVI, and W. VENEMA, "Where Do You Want to Go Today? Escalating Privileges by Pathname Manipulation ," .

[30] STONE-GROSS, B., M. COVA, C. KRUEGEL, and G. VIGNA (2011) "Peering Through the iFrame," in *Proceedings of the International Conference on Computer Communications (INFOCOM) Mini Conference*, Shanghai, China.

[31] POPA, M. (2009) "Detection of the Security Vulnerabilities in Web Applications," *Informatica Economica*, **13**(1), pp. 127–136.
URL http://ideas.repec.org/a/aes/infoec/v13y2009i1p127-136.html

[32] BOJA, C. E. and M. DOINEA (2010) "Security Assessment of Web Based Distributed Applications," *Informatica Economica*, **14**(1), pp. 152–162.
URL http://EconPapers.repec.org/RePEc:aes:infoec:v:14:y:2010:i:1:p:152-162

[33] VAIDYANATHAN, G. and S. MAUTONE (2009) "Security in dynamic web content management systems applications," *Commun. ACM*, **52**(12), pp. 121–125.
URL http://doi.acm.org/10.1145/1610252.1610284

[34] MEIKE, M., J. SAMETINGER, and A. WIESAUER (2009) "Security in Open Source Web Content Management Systems," *Security Privacy, IEEE*, **7**(4), pp. 44 –51.

[35] CRANOR, C. D., R. ETHINGTON, A. SEHGAL, D. SHUR, C. SREENAN, and J. E. VAN DER MERWE (2003) "Design and implementation of a distributed content management system," in *Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '03, ACM, New York, NY, USA, pp. 4–11.
URL http://doi.acm.org/10.1145/776322.776326

[36] COVA, M., C. KRUEGEL, and G. VIGNA (2010) "Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code," in *Proceedings of the World Wide Web Conference (WWW)*, Raleigh, NC.

[37] CANALI, D., M. COVA, C. KRUEGEL, and G. VIGNA (2011) "Prophiler: A Fast Filter for the Large-Scale Detection of Malicious Web Pages," in *Proceedings of the World Wide Web Conference (WWW)*, Hiderabad, India.

[38] INVERNIZZI, L., S. BENVENUTI, M. COVA, P. MILANI-COMPARETTI, C. KRUEGEL, and G. VIGNA (2012) "EvilSeed: A Guided Approach to Finding Malicious Web Pages," in *Proceedings of the IEEE Symposium on Security and Privacy*, S. Francisco, CA.

[39] WU, Y. and J. OFFUTT (2002) *Modeling and Testing Web-based Applications*, *Tech. rep.*, George Mason University.

[40] HUANG, Y.-W., F. YU, C. HANG, C.-H. TSAI, D.-T. LEE, and S.-Y. KUO (2004) "Securing web application code by static analysis and runtime protection," in *Proceedings of the 13th international conference on World Wide Web*, WWW '04, ACM, New York, NY, USA, pp. 40–52.
URL http://doi.acm.org/10.1145/988672.988679

[41] XIE, Y. and A. AIKEN (2006) "Static detection of security vulnerabilities in scripting languages," in *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, USENIX Association, Berkeley, CA, USA.
URL http://dl.acm.org/citation.cfm?id=1267336.1267349

[42] SON, S. and V. SHMATIKOV (2011) "SAFERPHP: finding semantic vulnerabilities in PHP applications," in *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, PLAS '11, ACM, New York, NY, USA, pp. 8:1–8:13.
URL http://doi.acm.org/10.1145/2166956.2166964

[43] (2013) *Fix Me Up: Repairing Access-Control Bugs in Web Applications*, 20, Network and Distributed System Security Symposium (NDSS).

[44] ARTZI, S., A. KIEZUN, J. DOLBY, F. TIP, D. DIG, A. PARADKAR, and M. D. ERNST (2008) "Finding bugs in dynamic web applications," in *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, ACM, New York, NY, USA, pp. 261–272.
URL http://doi.acm.org/10.1145/1390630.1390662

[45] LIU, C.-H., D. C. KUNG, P. HSIA, and C.-T. HSU (2000) "Structural Testing of Web Applications," in *Proceedings of the 11th International Symposium on Software Reliability Engineering*, ISSRE '00, IEEE Computer Society, Washington, DC, USA, pp. 84–.
URL http://dl.acm.org/citation.cfm?id=851024.856240

[46] ——— (2000) "Object-Based Data Flow Testing of Web Applications," in *Proceedings of the The First Asia-Pacific Conference on Quality Software (APAQS'00)*, APAQS '00, IEEE Computer Society, Washington, DC, USA, pp. 7–.
URL http://dl.acm.org/citation.cfm?id=786446.786478

[47] LIU, C.-H. (2006) "Data flow analysis and testing of JSP-based Web applications," *Inf. Softw. Technol.*, **48**(12), pp. 1137–1147.
URL http://dx.doi.org/10.1016/j.infsof.2006.06.003

[48] FELMETSGER, V., L. CAVEDON, C. KRUEGEL, and G. VIGNA (2010) "Toward automated detection of logic vulnerabilities in web applications," in *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, USENIX Association, Berkeley, CA, USA, pp. 10–10.
URL http://dl.acm.org/citation.cfm?id=1929820.1929834

[49] KING, J. C. (1976) "Symbolic execution and program testing," *Commun. ACM*, **19**(7), pp. 385–394.
URL http://doi.acm.org/10.1145/360248.360252

[50] SCHWARTZ, E. J., T. AVGERINOS, and D. BRUMLEY (2010) "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, IEEE Computer Society, Washington, DC, USA, pp. 317–331.
URL http://dx.doi.org/10.1109/SP.2010.26

[51] LI, X., D. SHANNON, I. GHOSH, M. OGAWA, S. P. RAJAN, and S. KHURSHID (2008) "Context-Sensitive Relevancy Analysis for Efficient Symbolic Execution," in *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, Springer-Verlag, Berlin, Heidelberg, pp. 36–52.
URL http://dx.doi.org/10.1007/978-3-540-89330-1_4

[52] SAXENA, P., D. AKHAWE, S. HANNA, F. MAO, S. MCCAMANT, and D. SONG, "A Symbolic Execution Framework for JavaScript," .

[53] CHAUDHURI, A. and J. S. FOSTER (2010) "Symbolic security analysis of ruby-on-rails web applications," in *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, ACM, New York, NY, USA, pp. 585–594.
URL http://doi.acm.org/10.1145/1866307.1866373

[54] ESHETE, B., A. VILLAFIORITA, and K. WELDEMARIAM (2011) "Early Detection of Security Misconfiguration Vulnerabilities in Web Applications," in *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pp. 169 –174.

[55] SHANNON, D., I. GHOSH, S. RAJAN, and S. KHURSHID (2009) "Efficient symbolic execution of strings for validating web applications," in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, DEFECTS '09, ACM, New York, NY, USA, pp. 22–26.
URL http://doi.acm.org/10.1145/1555860.1555868

[56] SOHR, K., M. DROUINEAUD, G.-J. AHN, and M. GOGOLLA (2008) "Analyzing and Managing Role-Based Access Control Policies," *Knowledge and Data Engineering, IEEE Transactions on*, **20**(7), pp. 924 –939.

[57] TONELLA, P. and F. RICCA (2002) "Dynamic model extraction and statistical analysis of Web applications," in *Web Site Evolution, 2002. Proceedings. Fourth International Workshop on*, pp. 43 – 52.

[58] KOLOVSKI, V., J. HENDLER, and B. PARSIA (2007) "Analyzing web access control policies," in *Proceedings of the 16th international conference on World Wide Web*, WWW '07, ACM, New York, NY, USA, pp. 677–686.
URL http://doi.acm.org/10.1145/1242572.1242664

[59] HU, H., G.-J. AHN, and K. KULKARNI (2011) "Anomaly discovery and resolution in web access control policies," in *Proceedings of the 16th ACM symposium on Access control models and technologies*, SACMAT '11, ACM, New York, NY, USA, pp. 165–174.
URL http://doi.acm.org/10.1145/1998441.1998472

[60] YANG, J.-T., J.-L. HUANG, F.-J. WANG, and W. C. CHU (1999) "An Object-Oriented Architecture Supporting Web Application Testing," in *23rd International Computer Software and Applications Conference*, COMPSAC '99, IEEE Computer Society, Washington, DC, USA, pp. 122–.
URL http://dl.acm.org/citation.cfm?id=645981.674607

[61] DOUPÉ, A., M. COVA, and G. VIGNA (2010) "Why Johnny can't pentest: an analysis of black-box web vulnerability scanners," in *Proceedings of the 7th international conference on Detection of intrusions and malware, and vulnerability assessment*, DIMVA'10, Springer-Verlag, Berlin, Heidelberg, pp. 111–131.
URL http://dl.acm.org/citation.cfm?id=1884848.1884858

[62] HUANG, Y.-W., C.-H. TSAI, T.-P. LIN, S.-K. HUANG, D. T. LEE, and S.-Y. KUO (2005) "A testing framework for Web application security assessment," *Comput. Netw.*, **48**(5), pp. 739–761.
URL http://dx.doi.org/10.1016/j.comnet.2005.01.003

[63] HUANG, Y.-W., S.-K. HUANG, T.-P. LIN, and C.-H. TSAI (2003) "Web application security assessment by fault injection and behavior monitoring," in *Proceedings of the 12th international conference on World Wide Web*, WWW '03, ACM, New York, NY, USA, pp. 148–159.
URL http://doi.acm.org/10.1145/775152.775174

[64] SUN, F., L. XU, and Z. SU (2011) "Static detection of access control vulnerabilities in web applications," in *Proceedings of the 20th USENIX conference on Security*, SEC'11, USENIX Association, Berkeley, CA, USA, pp. 11–11.
URL http://dl.acm.org/citation.cfm?id=2028067.2028078