Static Detection of Access Control Vulnerabilities in Web Applications

Fangqi Sun Liang Xu Zhendong Su
University of California, Davis
{*fqsun, leoxu, su*}*@ucdavis.edu*

Abstract

Access control vulnerabilities, which cause privilege escalations, are among the most dangerous vulnerabilities in web applications. Unfortunately, due to the difficulty in designing and implementing perfect access checks, web applications often fall victim to access control attacks. In contrast to traditional injection flaws, access control vulnerabilities are application-specific, rendering it challenging to obtain precise specifications for static and runtime enforcement. On one hand, writing specifications manually is tedious and time-consuming, which leads to non-existent, incomplete or erroneous specifications. On the other hand, automatic probabilistic-based specification inference is imprecise and computationally expensive in general.

This paper describes the first static analysis that automatically detects access control vulnerabilities in web applications. The core of the analysis is a technique that statically infers and enforces *implicit access control assumptions*. Our insight is that source code implicitly documents intended accesses of each *role* and any successful *forced browsing* to a privileged page is likely a vulnerability. Based on this observation, our static analysis constructs sitemaps for different roles in a web application, compares per-role sitemaps to find privileged pages, and checks whether forced browsing is successful for each privileged page. We implemented our analysis and evaluated our tool on several real-world web applications. The evaluation results show that our tool is scalable and detects both known and new access control vulnerabilities with few false positives.

1 Introduction

Web applications often restrict privileged accesses to authorized users. While bringing the convenience of accessing a large amount of information and operations from anywhere into people's daily lives, web applications have opened a new door for attacks and the number of web-based attacks is on the rise. A Symantec Internet security threat report published in April 2011 points out that the volume of web-based attacks in 2010 increased by 93% over the volume observed in 2009[1]. Researchers of web security have focused their attention on injection vulnerability, which is the most common vulnerability in web applications. Although not as prevalent as injection vulnerability, access control vulnerability poses a more serious threat because of exposed privileges, and has started attracting the attention of researchers [7]. Compared with those in traditional software, access checks in web applications are harder to get right because of the stateless nature of the HTTP protocol. In traditional software, once a user has passed an authentication check, the system remembers the identity of the user until she logs out or a timeout event happens. This is not the case for web applications, which must parse each new HTTP request to identify a previously logged-in user. A statistics report published in 2007 states that 14.15% of the surveyed web applications suffer from vulnerabilities of insufficient authorization[2].

Traditional injection vulnerabilities such as Cross-Site Scripting (XSS) and SQL injection are not application-specific and have a clear and general definition [25]: an injection vulnerability exists when an untrusted input flows into a sensitive sink without proper sanitization. To detect injection vulnerabilities, it is sufficient to analyze individual pages separately to examine where untrusted user inputs can flow. In contrast, access control vulnerabilities are application-specific, and it is necessary to examine connections between pages.

Web application developers frequently make implicit assumptions of allowed accesses and protect privileged pages by hiding links to these pages from unauthorized users. However, security by obscurity is insufficient to prevent a determined and skilled attacker from accessing these pages, viewing sensitive data or performing dangerous operations. As an example, Business Wire used a

[1] http://www.symantec.com/business/threatreport
[2] http://projects.webappsec.org/f/wasc_wass_2007.pdf

web server to store files of important trade information, which were supposed to be accessible to registered members only. Although the URLs to these files were hidden in the presentation layer from unauthorized users, the date-based URLs were highly predictable. By simply accessing these privileged files, an investment bank Lõhmus Haavel & Viisemann profited over eight million dollars based on the disclosed trade information[3]. Similarly, in November 2010, Blooming News obtained and published valuable financial earnings data of Disney and NetApp to its subscribers hours before official data releases by predicting resource locations inside secure corporate networks. As yet another example, accesses to the videos of USENIX conference presentations are restricted to USENIX members for a short period after a conference. However, the authors of this paper were able to predict the author-name-based URLs of the videos and download a few videos as public users.

Researchers have proposed various static and dynamic analysis techniques [1, 7, 10, 13] to detect violations of application logic, including access control attacks. Unfortunately, these techniques have limited effectiveness on detecting access control vulnerabilities. Dynamic analyses have difficulty finding hidden pages and determining intended accesses for each role. Furthermore, sitemaps covered by dynamic executions tend to be shallow and incomplete as user inputs are usually limited. Despite that static analyses typically have better coverage, they often require good specifications in order to generate useful reports, whose false positives do not overwhelm users. In practice, deriving precise specifications is challenging, especially when diverse authentication and access control management schemes are in use. As manually writing specifications is time-consuming and probabilistic-based inference is error-prone, it is desirable to precisely infer implicit assumptions on intended accesses from the source code of applications.

In this paper, we use *role* to represent a unique set of privileges that a group of users has. Most web applications have at least three types of roles: the role for administrators, the role for normal logged-in users and the role for public or anonymous users. Access control checks must be performed before granting access to any privileged resource to prevent privilege escalation attacks. When implicit assumptions are not matched by explicit access checks, unauthorized accesses are possible.

We propose the first role-based static analysis to detect access control vulnerabilities with automatic inference on implicit access control assumptions. Our key observations are that each role represents a unique set of privileges, and intended accesses for each role are reflected in explicit links shown in the presentation layer of an application. Guided by these observations, our analysis automatically derives specifications on privileged accesses by comparing explicit links presented to different roles. It then directly accesses privileged pages for unprivileged roles, and examines whether these accesses are allowed to detect vulnerable pages which have missing or insufficient access checks. Our main contributions are:

- A formal definition of access control vulnerabilities in web applications.
- The first role-based static analysis which automatically detects access control vulnerabilities in web applications with minimal manual efforts.
- An implementation of our analysis which constructs intended per-role sitemaps. Given role-based specifications, our prototype can systematically explore feasible execution paths based on the satisfiability of constraints.
- An evaluation of our tool on real-world web applications. Our tool works on unmodified code, and is able to detect both new and known vulnerabilities before the deployment of web applications. The evaluation results show that our approach is scalable and effective, with few false positives.

The rest of the paper is organized as follows. We first use an example to illustrate the main steps of our approach (Section 2) and then present our formalization of access control vulnerability in web applications (Section 3). Section 4 describes our detailed algorithms. Section 5 presents the implementation details of our static analyzer, and Section 6 shows the effectiveness, coverage and performance of our analyzer on real-world web applications. Finally, we survey related work (Section 7) and conclude (Section 8).

2 Illustrative Example

Figure 1 shows a simple web application based on one of the real-world web applications in our test suite. For illustration, suppose that the application has two roles: role *a* for administrators and role *b* for normal users. In our approach, we require developers to only specify application entry points and role-based application states, which serve as the basis for automatically inferring the set of privileged pages. Suppose that in the given specifications, the entry sets for both roles are identical and contain only "index.php", and the value of $_SESSION["admin"] is specified as **true** for role *a* but **false** for role *b*. As we can see from the source code, only "functions.php" checks accesses. This file is included via PHP inclusion in both "index.php" and "user_delete.php", but not "user_add.php." Consequently, access checks are missing in "user_add.php" but present in the other three pages.

[3] http://www.whitehatsec.com/home/assets/WP_bizlogic092407.pdf

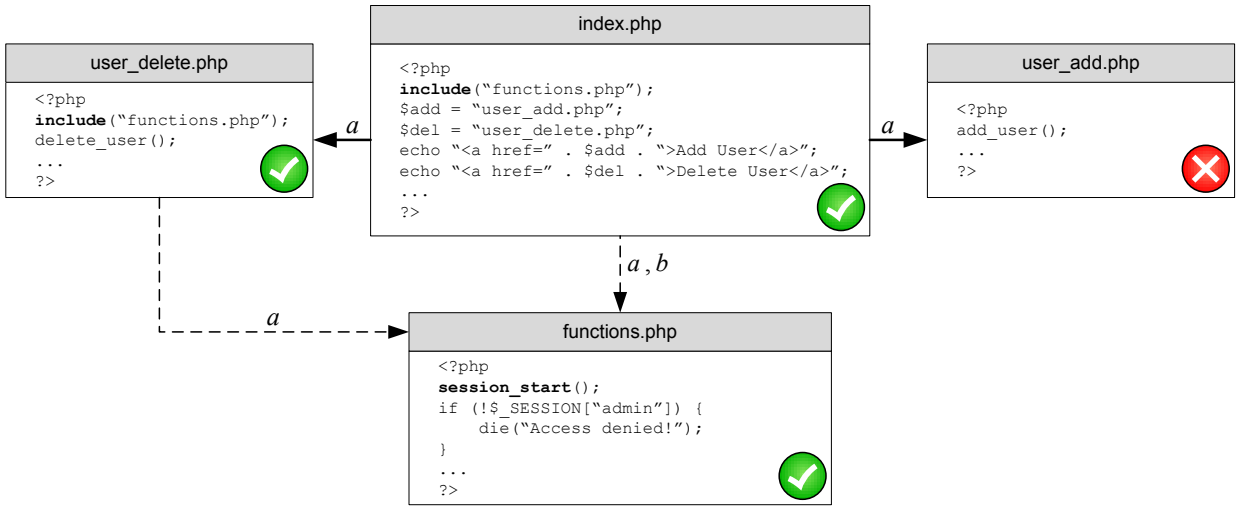

Figure 1: An Example of Access Control Vulnerability. Solid arrows represent explicit links, and dashed arrows represent inclusion relationship between pages. Arrows correspond to edges in sitemaps and are labeled with roles. The intended sitemap for privileged role a has four edges while the intended sitemap for role b has only one edge.

The first step of our analysis constructs per-role sitemaps with a worklist-based algorithm. Initially, worklists for both roles are ["index.php"]. While a worklist is not empty, our analysis pops a work node from the front of the worklist each time. Let us look at the sitemap construction for role a first. The first analyzed node is "index.php". From this node, users of role a can explicitly reach both "user_add.php" and "user_delete.php" via anchor tags, and "functions.php" via a file inclusion. Thus, our analysis adds three new edges in the sitemap and appends the newly discovered nodes to the worklist, which is now ["user_add.php", "user_delete.php", "functions.php"]. The second analyzed node is "user_add.php". This node can not reach any nodes, and thus our analysis pops "user_delete.php" and the worklist becomes ["functions.php"]. Role a can reach "functions.php" from "user_delete.php", and thus our analysis adds a new edge in the sitemap. Because "functions.php" is already in the worklist, it is not appended to the current worklist. Finally, our analysis pops "functions.php". This node can not reach any nodes and our analysis stops because the worklist is now empty. Now let us look at the sitemap construction for role b. The first popped node is still "index.php". However, role b can only explicitly reach "functions.php" via a file inclusion from this node. The links to "user_delete.php" and "user_add.php" are hidden from users of role b in "index.php" via the access check in "functions.php". Therefore, our analysis adds only one new edge and stops because the worklist is now empty. The edges of constructed per-role sitemaps are shown in Figure 1.

The second step of our analysis infers the set of privileged pages and attempts to access these pages directly to detect access control vulnerabilities. Comparing the sets of explicitly reachable nodes for role a and role b, our analysis infers that "user_add.php" and "user_delete.php" are privileged pages intended for users of role a only. Consequently, these two pages should have access checks to ward off users of role b. Unfortunately, only "user_delete.php" is safeguarded and "user_add.php" is left unprotected. Therefore, a direct access to "user_delete.php" fails, whereas a direct access to "user_add.php" succeeds, indicating that "user_delete.php" is guarded and "user_add.php" is vulnerable.

3 Approach Formulation

This section formulates our high-level approach. We define the notions of *role*, *explicit link*, *forced browsing*, *web application* and *access control vulnerability*, and present two assumptions we make with regard to roles and intended accesses.

Definition 1 (Role). A *role* $r \in R$ captures the set of allowed accesses for all users of role r where set R denotes roles that a web application has. Each role r represents a distinctive set of privileges.

Assumption 1 We assume that roles in R form a lattice $\langle R, \sqsubseteq \rangle$, where $\sqsubseteq$ denotes the ordering relationship between any two roles. Under this assumption, accessing a privileged resource as an unprivileged role is considered a privilege escalation attack. Roles at the same level of the lattice are not ordered by $\sqsubseteq$ as they may represent different sets of allowed accesses. The role for administrators is $\top$; the role for public users is $\bot$; and the role for

normal logged-in users lies in the middle of the lattice.

Definition 2 (Explicit Link). In a web application, there exists an *explicit link* from page n_i to a different page n_j when it is possible to jump to n_j via an explicit URL in n_i, incurring no exceptions or errors. URLs might appear in file inclusions, header redirections, HTML tags for anchors, forms, meta refresh headers, frames, iframes, scripts, images or links.

Definition 3 (Forced Browsing). *Forced browsing* is the act of directly accessing privileged pages rather than following explicit links in a web application. Attackers often harness brute force techniques to access hidden pages with predictable locations. We consider forced browsing successful when HTML pages presented to two different roles are identical, and no redirections, exceptions or errors occur during the page rendering process.

Definition 4 (Web Application). Let *node* represent a web page. Suppose that a web application contains k nodes. Given a user role $r \in R$, we abstract the *web application* as $P_r = (S_r, Q_r, E_r, I_r, \Pi_r, N_r)$, where

- *Entry set* S_r contains the entry nodes to the web application. We include index pages in all directories in the entry set. Different roles may have different entry sets.
- *State set* $Q_r = \{q_i \mid 0 \leq i < k\}$ is a set of application states. For each node n_i, an application state q_i captures critical information at that node. It might include session values, cookie values, request parameter values, database records, variable values or function return values.
- *Explicit edge set* $E_r = \{\langle n_i, n_j\rangle \mid 0 \leq i, j < k\}$. An explicit edge from node n_i to n_j exists iff n_i in state q_i contains an explicit link to n_j.
- *Implicit edge set* $I_r = \{\langle n_i, n_j\rangle \mid 0 \leq i, j < k\}$. An implicit edge from node n_i to n_j exists iff forced browsing enables one to jump to n_j from n_i in state q_i. Accesses via implicit edges are allowed but often unintended.
- *Navigation path set* $\Pi_r = \{(n_i)_{0 \leq i < l} \mid 0 < l < k \wedge n_0 \in S_r \wedge \langle n_i, n_{i+1}\rangle \in (E_r \cup I_r)\}$. It consists of all possible navigation paths for role r, including explicit edges as well as implicit edges.
- *Explicitly reachable node set* N_r consists of nodes that are reachable from application entries in S_r via explicit edges in E_r. It can be easily computed with a graph reachability analysis.

Assumption 2 For each node in a web application, if multiple roles can reach this node on navigation paths composed of only explicit edges, we assume that the privilege level required to access this node is determined by the least privileged role.

Definition 5 (Access Control Vulnerability). Let $a, b \in R$ denote two roles that can be ordered in a web application where role b is less privileged than role a, *i.e.*, $b \sqsubset a$. An *access control vulnerability* exists at node n when:

$$n \in N_a \wedge n \notin N_b \wedge \exists\, \pi_b \in \Pi_b\ (n \in \pi_b)$$

In this definition, destination node n is a privileged node intended to be accessible to role a but not role b. We use $n \in \pi_b$ to denote that n is on navigation path π_b. This node is vulnerable to access control attacks when a user of role b is able to access n via an allowed, but probably unintended, navigation path π_b.

4 Analysis Algorithm

In this section, we introduce the three major algorithms of our approach. Section 4.1 describes how our analysis automatically infers specifications of implicit access control assumptions and detects access control vulnerabilities from a high-level view. Section 4.2 shows the algorithm that we use to build per-role sitemaps. Finally, we present the detailed link extraction algorithm in Section 4.3.

4.1 Vulnerability Detection

Figure 2 presents the vulnerability detection algorithm which is the core of our approach. This algorithm infers privileged nodes from the source code of a web application and identifies nodes that are not properly protected.

```
DETECTVULS(Spec_a, Spec_b, reg)
 1  Vuls ← ∅
 2  nfa ← REG2NFA(reg)
 3  dfa ← NFA2DFA(nfa)
 4  N_a ← BUILDSITEMAP(Spec_a, dfa)
 5  N_b ← BUILDSITEMAP(Spec_b, dfa)
 6  Privileged ← N_a \ N_b
 7  for each n in Privileged
 8  do ⟨cfg_a, R_a⟩ ← GETCFG(n, Spec_a)
 9     ⟨cfg_b, R_b⟩ ← GETCFG(n, Spec_b)
10     if SIZEOF(cfg_a) = SIZEOF(cfg_b) and R_a = R_b
11        then Vuls ← Vuls ∪ {n}
12  return Vuls
```

Figure 2: Algorithm for Vulnerability Detection.

Let $Spec_a$ and $Spec_b$ denote specifications for role a and role b respectively. Initially, the set of vulnerable nodes *Vuls* is empty. First, this algorithm parses the regular expression *reg*, which captures HTML tags where a link might appear, into a non-deterministic finite automaton

(NFA). Then, the algorithm transforms the NFA into a deterministic finite automaton (DFA). Either NFA or DFA could be used for extracting links, and we chose DFA for its advantage on performance and the ease of FA state management.

Throughout this paper, we assume role a is more privileged than role b. Following Definition 4, we use N_a and N_b to denote the sets of explicitly reachable nodes for roles a and b respectively. Function BUILDSITEMAP, whose details are shown later in Section 4.2, computes these two sets. Relying on Assumption 2, the algorithm infers privileged nodes that are present in N_a but not in N_b (Line 6). For the example in Section 2, N_a ={"index.php", "user_add.php", "user_delete.php", "functions.php"} and N_b ={"index.php", "functions.php"}.

Access checks at privileged locations may be missing or insufficient. This algorithm analyzes each privileged node n twice with function GETCFG, once for role a to create an oracle for the intended server response (Line 8), and once for role b to emulate forced browsing (Line 9). Given a role r and a privileged node n, GETCFG returns a context-free grammar (CFG) cfg_r and the set of page redirections R_r.[4] The obtained cfg_r is an approximation of the dynamic HTML output of node n. We observe that when an access check succeeds, users are often granted accesses to sensitive information or operations; otherwise, they are redirected to another page, or presented with error messages or login forms. In the latter case, CFG sizes of the two roles are different because of the different HTML outputs that are presented. Consequently, if the sizes of the two CFGs or the two redirection sets differ, node n is considered guarded; otherwise, n may be vulnerable (Line 11). For the privileged page "user_delete.php" shown in Figure 1, $\text{SIZEOF}(cfg_a) \neq \text{SIZEOF}(cfg_b)$ and $R_a = R_b = \emptyset$, indicating that the page is guarded; for the privileged page "user_add.php", $\text{SIZEOF}(cfg_a) = \text{SIZEOF}(cfg_b)$ and $R_a = R_b = \emptyset$, indicating that the page is vulnerable.

4.2 Building Sitemaps

Function BUILDSITEMAP shown in Figure 3 builds a per-role sitemap with specifications $Spec_r$ for role r and the DFA *dfa*. We use a worklist-based algorithm to traverse nodes in a web application in a breath-first manner. Initially, both the visited node set *Visited* and the edge set E_r are empty, and the worklist *WkLst* is initialized with the entry set S_r specified in $Spec_r$ (Line 3).

In each iteration of the loop, function GETWORKNODE pops a working node n_i from the front of list *WkLst* and retrieves its associated state q_i from $Spec_r$ (Line 5) to find outgoing edges of this working node. Next, this algorithm constructs a CFG that represents the possible HTML outputs of node n_i (Line 6). Besides cfg_i, function

[4]Throughout this paper, CFG stands for context-free grammar rather than control-flow graph.

```
BUILDSITEMAP(Spec_r, dfa)
 1  E_r ← ∅
 2  Visited ← ∅
 3  WkLst ← GETENTRIES(Spec_r)
 4  while WkLst
 5  do ⟨n_i, q_i⟩ ← GETWORKNODE(WkLst, Spec_r)
 6     ⟨cfg_i, R_i, F_i⟩ ← CONSTRUCTCFG(n_i, q_i)
 7     L_i ← EXTRACTLINKS(cfg_i, dfa)
 8     N_j ← L_i ∪ R_i ∪ F_i
 9     for each n_j in N_j
10     do E_r ← E_r ∪ {⟨n_i, n_j⟩}
11     Visited ← Visited ∪ {n_i}
12     N ← ACTIVE(N_j) \ (Visited ∪ WkLst)
13     WkLst ← APPEND(WkLst, N)
14  return GETNODES(E_r)
```

Figure 3: Algorithm for Building Sitemaps.

CONSTRUCTCFG also returns the page redirection set R_i and the file inclusion set F_i as links in these two sets also contribute to outgoing edges in a sitemap. Then, function EXTRACTLINKS extracts a set of matched links L_i that are present in cfg_i based on *dfa* (Line 7). The details of EXTRACTLINKS are presented later in Section 4.3. The set of reachable nodes N_j for n_i is the union of L_i, R_i and F_i (Line 8). We conservatively include F_i in this union because included files may present sensitive information or operations. The algorithm adds an outgoing edge $\langle n_i, n_j \rangle$ to the explicit edge set E_r for each node $n_j \in N_j$ (Line 10) and then adds n_i to the visited node set (Line 11). To determine which nodes to analyze, we partition nodes into active nodes and inactive nodes, and only analyze active ones. Active nodes may have outgoing edges in a sitemap, whereas inactive nodes are dead ends. For example, a PDF file is considered an inactive node, while a PHP page is considered an active node. Finally, the algorithm adds the newly discovered active nodes to the worklist, excluding the ones that have been visited or are already in the worklist (Line 12, 13). The loop terminates when *WkLst* becomes empty, indicating that the construction of a per-role sitemap is complete. At this point, function BUILDSITEMAP returns the set of explicitly reachable nodes N_r based on E_r (Line 14). When work node n_i ="index.php" shown in Figure 1 is analyzed for role a in a loop iteration, L_i ={"user_delete.php", "user_add.php"}, $R_i = \emptyset$ and F_i ={"functions.php"}. Therefore, three new outgoing edges from "index.php" are added to E_a. In contrast, when "index.php" is analyzed for role b, $L_i = R_i = \emptyset$ and F_i ={"functions.php"}. In this case, only one new edge is added to E_b.

4.3 Link Extraction

We use C to denote a CFG, and F to denote an FA. In our setting, a CFG represents the dynamic HTML output of a node and an FA matches a single link-introducing HTML tag of various forms. Let $\mathscr{L}(C)$ be the set of words in the language for the CFG and $\mathscr{L}(F)$ be the set of words in the language for the FA. Suppose that function SUBSTR returns **true** only when w' is a substring of w. The output of EXTRACTLINKS on C and F is defined as follows:

$$\begin{aligned}\text{EXTRACTLINKS}(C,F) = \{\ w' \mid\ & w \in \mathscr{L}(C) \wedge \\ & w' \in \mathscr{L}(F) \wedge \\ & \text{SUBSTR}(w', w)\ \}\end{aligned}$$

We could use a straight-forward three-step approach to extract links. In the first step, we could use the standard CFG-reachability algorithm [20] to compute a CFG representing the intersection of the two languages for C and F', where F' matches HTML outputs that contain at least one link-introducing tag. The subtle difference between F' and F is that F' matches link-introducing tags as well as link-irrelevant HTML outputs, while F only matches link-introducing tags. In the second step, we could generate all possible HTML outputs of the CFG. In the third step, we could use an HTML parser to extract links from the generated HTML outputs. Nevertheless, this approach is not ideal for two reasons. The first is that the words of a CFG can be infinite and we can only generate a finite set of possible HTML outputs. The second is that the generated HTML outputs are likely being highly similar, and thus we may repetitively parse similar HTML outputs. For better performance, we designed a new algorithm that does not generate intermediate HTML outputs, but directly extracts links from the CFG.

In a CFG $\langle V, \Sigma, P, S_0 \rangle$, V is a finite set of variables (*i.e.* non-terminals); Σ is a finite set of terminals which is the alphabet of the language; $P = \{v \rightarrow rhs \mid v \in V \wedge rhs \in (V \cup \Sigma)^*\}$ is a finite set of grammar productions; and S_0 is the start variable. In an FA $\langle Q, \Sigma', q_0, \delta, Q_f \rangle$, Q is a finite, non-empty set of states; Σ' is the input alphabet; $q_0 \in Q$ is the start state; $\delta : Q \times \Sigma \rightarrow Q$ is the state-transition relation; and $Q_f \subseteq Q$ is the set of final states.

Figure 4 shows our link extraction algorithm where function EXTRACTLINKS is the entry point. We use set VQW to store $\langle v, q, w \rangle$ tuples where v represents a CFG variable, q is an FA state and w is a partially matched link string. Completely matched links are stored in set *Words*. To begin with, this algorithm walks the CFG with the start CFG symbol S_0, the start FA state q_0, and the empty string which represents the terminals that have been partially matched (Line 38).

Function WALKTERMINAL is the only function that advances an FA state q to a new state q' based on the FA transition function δ and an input character t (Line 1). If

```
WALKTERMINAL(t, q, w)
 1  q' ← δ(q, t)
 2  if q' = q_0
 3     then return ⟨q_0, "" ⟩
 4  w' ← APPEND(w, t)
 5  if q' ∈ Q_f
 6     then Words ← Words ∪ {w'}
 7          w' = ""
 8  return ⟨q', w'⟩

WALKVAR(v, q, w)
10  VQW ← VQW ∪ {⟨v, q, w⟩}
11  RHS ← PRODUCTIONS(v, P)
12  if ISSIGMA(RHS) or RHS = ∅
13     then return {⟨q, w⟩}
14  QW ← ∅
15  for each rhs in RHS
16  do if ISEPSILON(rhs)
17        then QW ← QW ∪ {⟨q, w⟩}
18        else QW ← QW ∪ WALKSYMBOLS(rhs, q, w)
19  return QW

WALKSYMBOL(s, QW)
21  Result ← ∅
22  for each ⟨q, w⟩ in QW
23  do if ISTERMINAL(s)
24        then QW' ← {WALKTERMINAL(s, q, w)}
25        else if ⟨s, q, w⟩ ∈ VQW
26                then QW' ← {⟨q, w⟩}
27                else QW' ← WALKVAR(s, q, w)
28     Result ← Result ∪ QW'
29  return Result

WALKSYMBOLS(rhs = [γ], q, w)
31  QW ← {⟨q, w⟩}
32  for each s_i in [γ]
33  do QW ← WALKSYMBOL(s_i, QW)
34  return QW

EXTRACTLINKS(cfg = ⟨V, Σ, P, S_0⟩, fa = ⟨Q, Σ', q_0, δ, Q_f⟩)
36  VQW ← ∅
37  Words ← ∅
38  WALKVAR(S_0, q_0, "" )
39  return VALID(Words)
```

Figure 4: Algorithm for Link Extraction.

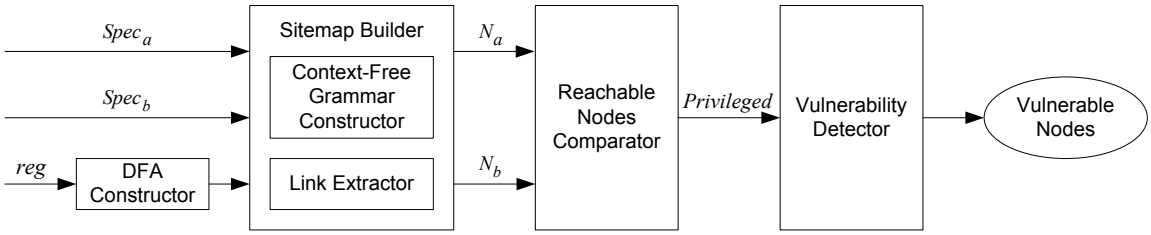

Figure 5: System Architecture.

q' is the FA start state q_0, which indicates a mismatch, the algorithm clears the partially matched terminals and returns (Line 3); otherwise, it appends t to w (Line 4) and examines q' again (Line 5). If q' is a final FA state in Q_f, the algorithm adds the completely matched link to *Words* (Line 6) and resets w' to the empty string. In this way, we filter out noises that are irrelevant to links in the CFG and only keep track of link-introducing HTML outputs.

Recursive function WALKVAR walks the grammar productions of variable v under an FA state q and a partially matched word w. Function PRODUCTIONS retrieves the set of productions which have v as the left-hand-side variable from the CFG production set P, and returns the set of right-hand sides *RHS* (Line 11). The different elements in *RHS* indicate how the dynamic HTML output might diverge for v. Function ISSIGMA checks whether a set is equivalent to the CFG alphabet Σ. A link of value Σ^* can point to any file in the application and therefore should be discarded. If *RHS* forms the alphabet or the empty set, the function returns the pair of unchanged q and w in a set (Line 13); otherwise, it walks the elements in set *RHS* one by one. In each loop iteration, if a right-hand side *rhs* has no symbols, the HTML output remains the same (Line 17); otherwise, the algorithm searches the set of new possible outcomes QW' with a call to function WALKSYMBOLS (Line 18).

Recursive function WALKSYMBOLS walks the symbols in list $[\gamma]$ in order. Consequently, links in the CFG are matched in the order of their appearances in a possible HTML output. Here $[\gamma] = (s_i)^* \wedge s_i \in (V \cup \Sigma)$, representing a sequence of right-hand-side symbols. For each symbol s_i in the list, the algorithm transitions the set of possible outcomes to a new set (Line 33).

Recursive function WALKSYMBOL walks a right-hand-side symbol s under each possible outcome $\langle q, w \rangle$. In each loop iteration, the algorithm first examines the symbol s (Line 23). If s is a terminal, the FA state is deterministically advanced via function WALKTERMINAL (Line 24). Otherwise, if the symbol is a variable, this algorithm recursively calls function WALKVAR for s (Line 27) when v is associated with a new q or a new w. The use of set *VQW* ensures the termination of the algorithm. This algorithm stops when all reachable grammar productions have been explored at least once. A concrete example of how this algorithm works is given in Section 5.2.2.

5 Implementation

As PHP is one of the most popular programming languages for web applications, we implemented our approach by extending Wassermann and Minamide's PHP string analyzer [21, 30], which is written in OCaml. The original PHP string analyzer was developed to detect injection vulnerabilities in web applications, and it analyzes individual pages in isolation and explores all execution paths. To detect access control vulnerabilities, we modified the string analyzer to build per-role sitemaps and examine connections between different pages. In particular, we introduced the concept of role into the static analyzer, added new specification rules for application states and entry sets, and strategically explored paths based on branch feasibilities. To explore only feasible execution paths, we keep track of both arithmetic constraints and string constraints. For arithmetic constraints, the analyzer consults a Satisfiability Modulo Theories (SMT) solver Z3 [8]; for string constraints, it consults a custom-built string constraint solver. Furthermore, we designed and implemented the algorithm shown in Figure 4 to efficiently extract explicit links from CFGs, added support for 176 built-in PHP functions, and modified both the specification lexer and parser to support specifications for the values of integers, floating-point numbers and strings.

Figure 5 shows our system architecture. A web application can have multiple roles, and our analysis compares a pair of ordered roles each time. Initially, the *DFA constructor* transforms the given regular expression *reg* into a DFA. The detection of access control vulnerabilities is carried out in two major steps. First, the *sitemap builder* explores the given web application based on parsed specifications and the DFA. Second, the *reachable nodes comparator* infers what privileged nodes are, and the *vulnerability detector* performs forced browsing to detect nodes that are vulnerable to access control attacks.

5.1 Specification Rules

In our analysis, specifications are parsed with a lexer and a parser. For each role r, we only require developers to specify the entry set S_r and the set of critical application

states Q_r. Multiple roles can share the same set of entry points. Either index pages or active pages with no incoming edges can be entry nodes. Index pages often have conventional names such as "index.php" and "index.html", and can be easily identified with a file scan; active pages with no incoming edges can be specified as entry nodes by developers. The types of application states that we support are listed in Definition 4. The state values that can be specified include abstract types and concrete values of built-in PHP types, and string values that can be represented by a regular expression. For function invocations, we allow developers to pinpoint an invocation by specifying the filename and line number where the invocation occurs. This is especially useful when function invocations return different values at different call sites.

Optionally, developers can explicitly specify a set of privileged nodes. In contrast to implicit navigation paths which involve forced browsing, explicit navigation paths are often tested more thoroughly. However, it is still possible that an allowed access to a sensitive node via an explicit navigation path of an unprivileged role is unauthorized, violating Assumption 2. In this case, when an unprivileged user can explicitly navigate to a privileged node, we would have false negatives. To solve this problem, we allow developers to explicitly specify privileged nodes. Such a node may be vulnerable to access control attacks even if it is explicitly accessible for both roles.

5.2 Sitemap Builder

The sitemap builder has two components: the *context-free grammar constructor* and the *link extractor*. With these two components, our analysis constructs a CFG for each explicitly reachable node, and extracts links embedded in the CFG to find outgoing edges of the node.

5.2.1 Context-Free Grammar Constructor

For each web page, our analyzer first parses the page into an Abstract Syntax Tree (AST), and then transforms the AST into an Intermediate Representation (IR), distinguishing every variable occurrence. Interested readers can refer to Wassermann's work [30] for more details.

To build a per-role CFG, our analyzer explores the IR only when necessary by predicting branch feasibilities with an inter-procedural path-sensitive analysis. It analyzes statements in the IR in a top-down manner, updating path conditions for both string constraints and arithmetic constraints. For arithmetic constraints, our analyzer resorts to the integrated Z3 to check the satisfiability of constraints; for string constraints, it feeds possible values of string variables and their aliases to our string constraint solver in exchange of answers. Our prototype string constraint solver supports string constraints which may contain multiple variables, regular expressions, equality and inequality operators, and checks on string lengths. We tried to solve string constraints with HAMPI [15], but it does not support multiple string variables yet. When constraints of a conditional is unsolvable, the analyzer explores both branches, updating path conditions for both the true branch and the false branch. For each function call, our analyzer first checks its calling context and then explores the function only when the context is new. Next, it propagates constraints on the arguments and related global variables of the function call. The IR exploration terminates when all possible branches have redirections or exits, indicating that none of the unexplored branches are feasible. In our implementation, we do not consider different contexts of page accesses and assume the parameters of HTTP requests to be Σ^* unless specified. In this way, we analyze each page only once, making our analyzer scalable at the expense of obtaining over-approximations of outgoing edges.

```
function checkUser() {
  if (!isset($_SESSION["validUser"])
    || $_SESSION["validUser"] != true) {
    header("Location: login.php");
  }
}
checkUser();
sensitiveOperation();
```

Figure 6: An Example of Path Exploration.

Finding the targets of PHP includes is a non-trivial task. It requires value resolution of possible string variables that are used for filename construction. Furthermore, it is necessary to find the directories that a PHP include file may reside in. When resolving PHP include paths, the following steps are performed in order:

- The `include_path` in the configuration of a PHP application is checked first;
- If no matching file is found under `include_path`, the directory of the calling script is checked;
- If no matching file is found in the directory of the calling script, the current working directory is checked;
- If no matching file is found in the current working directory, the inclusion finally fails.

We illustrate our basic exploration strategy with a simple example shown in Figure 6 based on one of the web applications that we have analyzed. Function `checkUser` checks whether an access should be allowed for a given user. Function `SensitiveOperation` will only be executed when the user has passed the access check. Suppose that $\$\_SESSION$["validUser"] is a critical application state which determines the privileges of a role, and

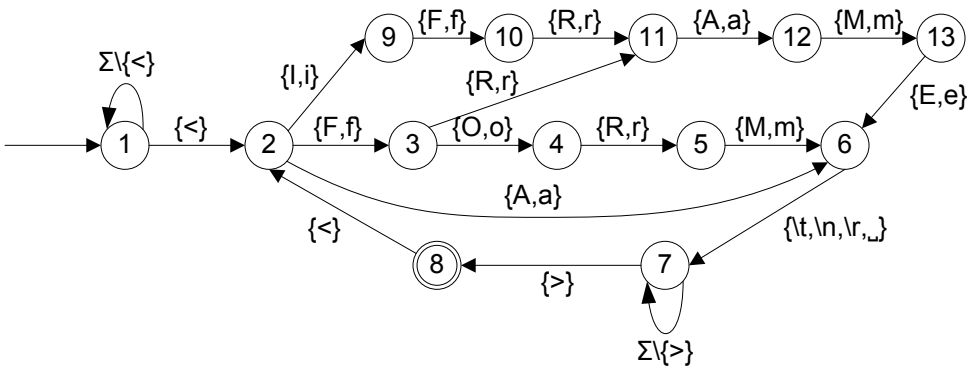

Figure 7: A Deterministic Finite Automaton Example.

its value should be specified as **true** for role a and **false** for role b. Our analyzer explores the statements of the IR in order. Besides function definitions, the first statement it encounters is the function call `checkUser()`. Therefore, it retrieves the corresponding function body and continues from the first statement in the function. Because the first statement is an `if` statement, the analyzer attempts to solve the satisfiability of constraints to determine branch feasibilities. If the given role is b, only the true branch is feasible. As the true branch has a header redirection, the analyzer stops exploring the statements after this function call. Otherwise, when the role is a, only the false branch is feasible, and the analyzer continues exploring the statements after this function call, and eventually reaches function call `SensitiveOperation()`.

Path sensitivity prevents us from exploring infeasible paths. For example, suppose we have predicate $\$x > 1$ in the current path condition when the exploration reaches an `if` statement, the branch target of which depends on a conditional $\$x < 0$. To determine the feasibilities of the two possible branches, our analyzer sends two queries to Z3. The first query appends the new constraint to the existing path condition, while the second query appends the negation of the new constraint to the existing path condition. Z3 will conclude that $(\$x > 1 \wedge \$x < 0)$ is unsatisfiable, but $(\$x > 1 \wedge \neg(\$x < 0))$ is satisfiable. Thus, only the false branch is feasible and our analyzer will not explore the infeasible true branch of the `if` statement.

5.2.2 Link Extractor

Our link extractor extracts links to different web pages within a given web application. Since we are interested in constructing sitemaps, our link extractor filters links that point to pages outside of the application. We did not reuse the implementation from the previous work [30], which is based on the standard graph-reachability algorithm, but instead implemented the new link extraction algorithm shown in Figure 4 to eliminate the need of computing intermediate HTML outputs. As an example, Figure 7 shows an FA which matches anchor, form, frame and iframe tags in HTML outputs based on a simple regular expression:

```
/<([Aa]
    |[Ff][Oo][Rr][Mm]
    |[Ii]?[Ff][Rr][Aa][Mm][Ee]
  )\s[^>]*>/
```

We only show state-advancing edges in Figure 7 and omit state-resetting edges. In this FA, the start state $q_0 = 1$ and the final state set $Q_f = \{8\}$. For any FA state, a state-resetting edge directs the current FA state back to the start FA state on input characters other than the ones shown on the state-advancing edges. We use the following simplified PHP code taken from one of our test subjects to show how our link extractor works.

```
echo "<div><a href="
    . $lang
    . ".php>Anchor</a></div>";
```

The above PHP code dynamically generates a link depending on the value of variable $\$lang$, which has three possible candidates: “english”, “spanish” and “french”. For this code, a CFG with five variables and seven grammar productions will be generated:

$$S_0 \rightarrow S_1 S_2$$
$$S_1 \rightarrow \text{“<div><a href=”}$$
$$S_2 \rightarrow S_3 S_4$$
$$S_3 \rightarrow \text{“english”} \mid \text{“spanish”} \mid \text{“french”}$$
$$S_4 \rightarrow \text{“.php>Anchor</div>”}$$

In this CFG, $V = \{S_0, S_1, S_2, S_3, S_4\}$ and S_0 is the start symbol. Note that S_3 has three associated grammar productions separated by bars. For the algorithm in Figure 4, the link extraction starts with function call WALKVAR($S_0, 1, \text{“”}$) (Line 38). Since S_0 maps to only one production, $RHS = \{[S_1 S_2]\}$ (Line 11) and our algorithm issues WALKSYMBOLS($[S_1 S_2], 1, \text{“”}$) (Line 18). Then, it

examines the symbols in list $[S_1 S_2]$ (Line 32) in order to derive the set of possible outcomes QW, the initial value of which is $\{\langle 1, \text{“”} \rangle\}$ (Line 31). Our algorithm sees that the first symbol S_1 is a variable and thus issues WALKVAR(S_1, 1, “”) (Line 27). For S_1, RHS ={“ <a href=”} (Line 11), and the algorithm issues WALKSYMBOLS(“ <a href=”,1,“”) (Line 18). Now our algorithm examines these terminals in order with function WALKTERMINAL. The first character is ‘<’, thus the algorithm transits the FA state from 1 to 2 along a state-advancing edge in Figure 7, and appends ‘<’ to w which is now “<”. The second character is ‘d’, thus the algorithm resets the FA state to the start state 1, and clears the matched terminals in w. The third character is ‘i’, thus the algorithm stays at the FA start state 1, and w is still the empty string. Our algorithm continues like this and by the time it gets to variable S_3, the FA is in state 7 with w =“<a href=”. For S_3, RHS ={“english”, “spanish”, “french”} (Line 11), and our algorithm walks these three elements one by one (Line 15). There are three possible outcomes, and thus the return value QW of WALKSYMBOLS(S_3, 7, “< *a* href=”) is $\{\langle 7, \text{“<a href=english”} \rangle, \langle 7, \text{“<a href=spanish”} \rangle, \langle 7, \text{“<a href=french”} \rangle\}$ (Line 19). Our algorithm continues until all the seven grammar rules have been explored. Upon termination, it returns {“english.php”, “spanish.php”, “french.php”} (Line 39).

5.3 Vulnerability Detector

When the construction of per-role sitemaps is complete, our analyzer compares the two reachable node sets to infer privileged nodes. As HTML outputs presented to different roles are usually different, the set of privileged nodes is not empty in most cases. After obtaining the set of privileged nodes, our analyzer uses the same context-free grammar constructor again to approximate the outcomes of forced browsing. Finally, it compares derived redirection sets and the sizes of CFGs to determine whether forced browsing attemps are successful.

Even when forced browsing is successful, it is possible that the corresponding page does not contain any sensitive information or operations and is therefore considered safe. We observed that some pages used as file inclusions only contain function and class definitions. Such pages normally serve as inclusion files and are safe on their own. When the automatic vulnerability detection is over, we identify such safe pages with manual analysis, report them as false positives, and then mark the remaining pages as potentially vulnerable pages.

6 Empirical Evaluation

To evaluate the effectiveness and performance of our approach, we tested our tool on seven real-world PHP applications, two of which have patched versions. We picked these applications because they have reported vulnerabilities, which include injection vulnerabilities as well as access control vulnerabilities. The test subjects include both traditional web applications and Web 2.0 applications which use AJAX for client-server communications. The source code of all these PHP applications is publicly available. For each of the test subjects, we provide a specification file of at most ten lines. We ran all the tests on a PC with a quad-core CPU (2.40GHz) and 4 GB of RAM.

Subject	Files	LOC	
		PHP	HTML
SCARF	25	1,318	0
Events Lister	37	2,076	544
PHP Calendars	67	1,350	0
PHPoll	93	2,571	0
PHP iCalendar	183	8,276	0
AWCM	668	12,942	5,106
YaPiG	134	4,801	1,271

Table 1: Statistics on Evaluation Subjects.

Our tool supports multiple roles and each role should have a set of distinctive application states. Typically, the administrator role has the most privileges; the normal user role has necessary privileges for common user operations; and the public user role has the least privileges. Although our tool can detect access control violations for any two roles, we chose to detect access control violations between administrators and normal users for two reasons. First, the operations and information that administrators can access are of greater importance than those that normal users can access. Second, it is often difficult for attackers to legally obtain administrator accounts, but easy to obtain normal user accounts.

Table 1 shows the total number of files as well as the lines of code for each web application. For the two web applications that have patched versions, we only list the statistics for the patched versions in the table. The lines of code in each application are counted for both PHP and HTML, excluding comments and empty lines. Our analysis translates HTML code into equivalent PHP `echo` statements.

6.1 Analysis Results

Table 2 shows the analysis results for the nine web applications. Note that we include two versions of SCARF and AWCM for vulnerability analysis. Columns “Vulnerable” and “FP” denote the numbers of detected true vulnerabilities and manually confirmed false positives respectively. Column “Guarded” shows the number of privileged pages that are protected by access checks. The last four columns show numbers of explicitly reachable nodes and explicit edges in per-role sitemaps.

In summary, our tool found eight different access control vulnerabilities, four of which are previously unknown.

Project	Privileged	Vulnerable	FP	Guarded	Admin		Normal	
					Node	Edge	Node	Edge
SCARF	4	1	0	3	19	149	15	69
SCARF (patched)	4	0	0	4	19	149	15	69
Events Lister 2.03	9	2	2	5	23	113	14	26
PHP Calendars	3	1	0	2	19	35	19	30
PHPoll v0.97 beta	3	3	0	0	21	63	19	58
PHP iCalendar v1.1	1	0	0	1	51	292	50	292
AWCM v2.1	47	1	0	46	176	2,634	129	2,438
AWCM v2.2 final	47	0	0	47	180	2,851	133	2,612
YaPiG 0.95	11	0	0	11	54	260	44	154

Table 2: Vulnerability Analysis Results.

It only has two false positives and correctly reports 119 guarded pages as not vulnerable. We manually confirmed all vulnerabilities and false positives on deployed web applications. In addition, the by-products of our analysis, the generated per-role sitemaps, provide high-level views of the test subjects and can be useful for understanding or modifying the structures of these web applications.

6.1.1 SCARF

SCARF is the Standford Conference And Research Forum. A critical access control checks whether the value of $\$\_SESSION$["privilege"] equals "admin" in functions `is_admin` and `require_admin`.

Our tool detected a previously reported vulnerability (CVE-2006-5909). In this application, only users of role *a* are supposed to edit the configuration of the application in page "generaloptions.php". However, there is no access check for this edit privilege. Although the link is hidden from users of role *b*, they could still access and edit the configuration which affects the whole system. Our tool correctly reported the other three privileged pages "addsession.php", "editpaper.php" and "editsession.php" as guarded. Even if users of role *b* know the locations of these pages, forced browsing would fail because of the presence of access checks in these pages. The latest version of SCARF fixed the vulnerability, and this is reflected in the vulnerability analysis result for SCARF (patched).

6.1.2 Events Lister

Events Lister is a PHP application that allows users to manage their events. Function `checkUser` implements an access control by checking whether $\$\_SESSION$["validUser"] equals **true**.

Our tool found a new vulnerability in this application as well as a previously known one (CVE-2009-3168). We discovered that page "admin/setup.php" has no access checks and allows users of role *b* to repeatedly insert test events into the database of the application. It is even possible to create new tables in the database if none exists yet. The known vulnerability in page "admin/user_add.php" permits users of role *b* to add new users into the system. This privilege should only belong to users of role *a*. We consider the other two reports on privileged pages "admin/recover.php" and "admin/form.php" false positives. Page "admin/recover.php" allows users of role *b* to reset an administrator's password by sending a new password to the administrator's email address. Since only the administrator has access to her own email address, the password reset action does not pose any serious threats. Page "admin/form.php" contains an HTML form which is included in other container pages. On its own, this page does not expose any privileged operations or information, and is therefore considered safe. The notion of "safe" is sometimes a subjective matter. In a manual case study of another web application, we found that public users can view the list of all registered users with forced browsing. Such a list is also available for normal users and one can easily register for a normal user account. Consequently, it is unclear to us if the implicit access to the list of registered users is intended. As such, we would rather report such cases to developers for them to decide.

6.1.3 PHP Calendars

PHP Calendars is an online calendar management system. It protects privileged pages in the application by checking whether $\$\_SESSION$["admin"] equals "yes" in page "admin/access.php".

Our tool detected a known vulnerability (CVE-2010-0380) in page "install.php" of this application. The README file in this application warns administrators to delete this page after installation, but does not check if the file has indeed been deleted. If "install.php" exists in a deployed application, any users of role *b* could modify the configuration of the application by directly accessing this page. Because there is an explicit link to this page, we manually added this page to the privileged node set in the specification file. The other two privileged pages "ad-

min/import.php" and "powerfeed.php" are not vulnerable. Note that N_a is not necessarily a superset of N_b. In this application, $|N_a| = |N_b|$, but $N_a \neq N_b$.

6.1.4 PHPoll

PHPoll is an online poll system where only users of role *a* can pass access checks by providing correct values of $\$\_COOKIE[\$string\_cook\_login]$ and $\$\_COOKIE[\$string\_cook\_password]$. Note that the cookie-based access controls are safe in this case because unauthorized users have no knowledge of valid cookie values.

Our tool detected three new access control vulnerabilities in this application and we manually confirmed them on a deployed application of PHPoll. All three pages have no access checks. The first page "modifica_configurazione.php" allows users of role *b* to modify login IDs and passwords, truncate the configuration table, and insert new entries into the configuration table of the application. The second page "modifica_votanti.php" lets users of role *b* delete votes or update polls stored in the MySQL database. The third page "modifica_band.php" does not prevent users of role *b* from reading, updating, or deleting poll results from the database with POST requests. These access control vulnerabilities pose serious threats to the security of the application, yet they have not been reported to the best of our knowledge.

6.1.5 PHP iCalendar

PHP iCalendar is another calendar application which displays calendar information to users. The only privileged page is "admin.php", and it is guarded by an access check which examines the value of $\$HTTP\_SESSION\_VARS$["phpical_loggedin"].

This application does not have any access control vulnerabilities. As Table 2 shows, users of role *a* can reach 51 pages which include "admin.php", while users of role *b* can only reach 50 pages which exclude "admin.php".

6.1.6 AWCM

AWCM (AR Web Content Manage system) differentiates role *a* from role *b* by determining whether $\$\_SESSION$["awcm_cp"] equals "yes" in a PHP include file "control/common.php".

Our tool detected a previously known vulnerability (CVE-2010-1066) in "control/db_backup.php" which dumps all the database information onto a web page. The cause of this access control vulnerability is that "control/db_backup.php" includes "common.php" instead of "control/common.php". Since access checks are only present in "control/common.php" but not "common.php", page "control/db_backup.php" is not guarded and can be accessed via forced browsing. Most pages in the "control" directory are intended for administrators only and our tool detected 47 privileged nodes in total. Our tool correctly recognized the access checks in the other 46 privileged pages and only reported "control/db_backup.php" to be vulnerable. The latest version of AWCM fixed the vulnerability, and this is reflected in the analysis result shown in Table 2. Although this application is AJAX-heavy, our tool covered nearly 80% of the active nodes, indicating that a majority of the links appear in PHP and HTML code which can be well handled with our tool.

6.1.7 YaPiG

YaPiG (Yet Another PHP Image Gallery) validates passwords and determines the privilege level of users with an access check in function `check_admin_login`.

An interesting thing about YaPiG is that all the five unreachable pages result from an uncovered execution path. In our implementation, we assume that an HTTP parameter $\$v$ could have any values. Therefore, our tool infers that function call `isset($v)` returns **true** even if *v* is undefined. When a conditional depends on such a function call, the false branch is left unexplored. Our implementation does not yet support the specification of an optional value, which can either be defined or undefined.

6.2 Performance Evaluation

In our evaluation, we collect links that point to files within an application, excluding those that point to CSS files which are of no interest to us. Currently, we treat PHP, HTML and XML files to be active nodes and analyze them to extract links. A page can contain links to both active nodes and inactive nodes. Although inactive nodes do not provide sensitive operations, they may contain sensitive information and therefore should also be checked.

Table 3 shows the coverage and performance of our tool. Column "Entry" shows the number of specified entry nodes for each application. Column "Active" lists the number of all active nodes. Column "Orphan" lists the number of specified *orphan nodes* which are non-entry active nodes with no incoming edges. Column "Coverage" lists the coverage of our tool on active nodes in an application, excluding orphan nodes. We list the average numbers of variables and grammar productions of all CFGs for each web application. Note that the numbers are counted on CFGs that have been simplified with grammar-reachability analysis. The last column shows the total analysis time spent for each application in terms of seconds.

Active nodes may have outgoing edges and may not have any incoming edges. An active node with no incoming edges can be optionally specified as either an entry node or an orphan node. When it is specified as an entry node, it is analyzed in the sitemap construction process to find outgoing edges; when it is specified as an orphan node, which indicates that this node should be outside any sitemaps, it is excluded from the coverage calculation; when it is unspecified, it may affect the coverage

Project	Nodes			Context-Free Grammar		Coverage	Time (s)
	Entry	Active	Orphan	Variables	Productions		
SCARF	1	19	0	158	719	100.00%	6.02
SCARF (patched)	1	19	0	159	719	100.00%	6.01
Events Lister v2.03	4	23	5	100	2,083	100.00%	3.84
PHP Calendars	3	15	0	48	255	80.00%	5.09
PHPoll v0.97 beta	5	21	6	115	224	100.00%	4.26
PHP iCalendar v1.1	2	52	2	811	4,774	90.38%	760.62
AWCM v2.1	17	208	22	410	422	79.33%	89.48
AWCM v2.2 final	16	209	14	451	484	79.90%	108.51
YaPiG 0.95	7	59	3	332	532	91.53%	208.38

Table 3: Coverage and Performance Results.

Project	Time (s)		
	Admin Sitemap	Normal Sitemap	Forced Browsing
SCARF	3.15	1.70	1.15
Events Lister	2.29	1.00	0.53
PHP Calendars	1.81	1.67	1.61
PHPoll	2.39	1.54	0.33
PHP iCalendar	371.28	370.85	18.46
AWCM	55.36	49.11	3.85
YaPiG	85.59	44.91	77.86

Table 4: Analysis Time.

result. Let $Active$, $Orphan$ and $Reachable$ denote the sets of all active nodes, specified orphan nodes and explicitly reachable nodes respectively. We calculate the coverage as:

$$Coverage = \frac{|Reachable|}{|Active| - |Orphan|}$$

In our evaluation, we conservatively identify orphan nodes with a simple manual analysis and the obtained orphan sets may be incomplete, especially for large and complex applications. Therefore, the real coverages of our analysis might be better than the ones shown in the table because uncovered nodes might indeed be unreachable.

Our static analyzer achieved good coverage of active nodes: 100% for four applications, about 90% for two, and about 80% for the remaining three. The total analysis time listed in Table 3 demonstrates that our approach is scalable. For the smaller test applications SCARF, Events Lister, PHP Calendars and PHPoll, our tool finished within seven seconds; for the largest test application AWCM, our tool took less than two minutes to analyze the active nodes in the whole application. The analysis time for iCalendar is the longest because of the inlining of dynamic PHP files and the complexity of PHP code. As can be seen in Table 3, the number of grammar productions for PHP iCalendar is also the largest. We show the break down of analysis time in Table 4. Columns "Admin Sitemap" and "Normal Sitemap" list the time spent on constructing the sitemaps for roles a and b respectively. Column "Forced Browsing" shows the time spent on detecting access control vulnerabilities via forced browsing. It is obvious from the data in the table that building sitemaps consumes the majority of the analysis time.

6.3 Discussions

As we mentioned earlier, our prototype did not find all kinds of links in web applications. The major reason is that our prototype did not identify all the links generated by JavaScript code or HTML templates, or those constructed with unresolvable string variables. Extracting links from JavaScript code is especially challenging because of the dynamic features of the JavaScript language. Our prototype works better on traditional web applications than AJAX-heavy ones. Incorporating JavaScript analysis could possibly improve the coverage. Furthermore, our test applications may not be representative of general web applications.

What a node represents determines the granularity of the analysis. Our prototype treats a web page as a node, but the general approach still applies when the granularity is refined to functionalities within a page. Performing the analysis at a refined granularity would be especially useful for complex web pages which contain multiple functionalities within a single page. The techniques proposed by Halfond *et al.* [12] could be used to identify important parameters in web applications to distinguish functionalities. Because a privilege is often granted with a set of atomic database operations, advancing the granularity to the level of database operations might be too fine-grained.

Our prototype does not handle all object-oriented features in PHP. This prevents us from parsing some PHP pages in large PHP applications. We leave it as future work to enhance our static analyzer for additional object-

oriented features of the PHP language.

The current implementation of the string constraint solver is rudimentary. For either unsolvable constraints or non-determinism in a conditional, we conservatively explore both branches. This might lead to false negatives when infeasible paths for a less privileged role are explored. For access checks that involve non-determinism, such as password-based authentication and CSRF protection that uses random tokens, we rely on role-based specifications to determine which execution paths to explore. Non-determinism affects path explorations but not link extractions. Furthermore, when Assumption 2 does not hold, we would also have false negatives introduced by explicit accesses to privileged nodes.

Our tool generated false positives. Even when access checks are missing in hidden pages, these pages may not contain any sensitive information or operations and are therefore safe to access for any role in the application. We manually examined the analysis results and marked such safe pages as false positives.

7 Related Work

In this section, we discuss the most relevant work, including specification inference, workflow violation detection, privilege separation based on user roles, language-based approaches to secure web applications, and program analysis for web security.

The capability of automated tools in detecting vulnerabilities or bugs can only be as good as the specifications given to them. Since manually writing specifications is tedious, time-consuming and error-prone, a wide range of techniques have been proposed to automatically infer specifications from the source code of programs. For intrusion detection, Wagner and Dean [28] apply static analysis to derive a model of normal application behavior as an oracle. Based on the observation that bugs are deviant behavior [9], researchers have proposed probabilistic-based approaches [16, 26] to infer specifications from applications. However, without taking into account of roles in web applications, it is difficult to infer privileged pages which are only intended for a group of users.

Recently, workflow violations have attracted the interests of researchers. Nemesis [7] uses dynamic information flow tracking to detect authentication and access control vulnerabilities in web applications. It requires developers to specify access control lists for resources. Similarly, Hallé *et al.* [13] proposed a runtime enforcement mechanism to only allow navigations that conform to a state machine model specified by developers. Researchers have proposed various techniques to automatically infer correct workflows. Swaddler [6] first learns internal states of web applications, and then detects abnormal state violations at critical points. Targeting the detection of Ajax intrusion attacks, Guha *et al.* [11] leverage static analysis on client-side JavaScript code to infer expected server-side behavior. To detect multi-module vulnerabilities, MiMoSA [1] takes into account the interactions of different web pages. However, it is not always easy to distinguish an intended path from an unintended one because of flexible navigation paths that web applications allow. Its follow-up work Waler [10] uses a combination of dynamic analysis and symbolic model checking to first infer invariants from dynamic program executions, and then report violations of the invariants as logic vulnerabilities. From a high-level view, the likely invariants that Waler generates with heuristics are subject to errors. Furthermore, the inferred invariants may not always hold due to the limited coverage of dynamic analysis. Access control vulnerabilities can be considered a special case of workflow vulnerabilities where cross-role workflow assumptions are violated. Cross-role comparisons allow us to precisely reason about privileged pages in most cases.

To reduce least-privilege incompatibilities, researchers distinguish different user roles and separate privileges based on different roles. Aiming at identifying dependencies on `admin` privileges in traditional software applications, Chen *et al.* [4] run applications without `admin` privileges and collect dynamic execution traces. We take a step further and use roles to represent sets of privileges in web applications. In our setting, roles form a lattice and its height is not limited. To reduce developer's burden on securing web applications, the CLAMP project [23] prevents leakage of sensitive information by restricting the flows of user data and isolating the authentication module of an application. While they also minimize developers' effort, they secure web applications by modifying application code at critical points. Web application vulnerability scanners can also automatically detect access control vulnerabilities. However, they often build shallow and incomplete sitemaps, missing deep and invisible pages that are only accessible when valid form data are submitted. This undermines the capabilities of web scanners in both discovering privileged nodes as well as successfully performing forced browsing with valid form data.

Previous work has proposed language-based approaches to secure web applications in a principled way. SIF [5] accepts specifications either as program annotations at compile time, or as user requirements at run time to guarantee confidentiality and integrity with information flow analysis. Recently, Krishnamurthy *et al.* [17] presented an object-capability language for fine-grained privilege separation for web applications. Unfortunately, theses language-based approaches do not apply to the large set of legacy code that is not written in the newly designed languages.

In the past few years, researchers have focused their attention on detecting injection vulnerabilities in web applications with both static analysis [18, 19, 25, 27, 29, 30,

31, 32] and dynamic analysis [2, 3, 22, 24]. Similar to our static analyzer, Pixy [14] is also a static analyzer built to analyze PHP applications. It takes advantage of taint analysis to detect injection vulnerabilities with specifications on taint sources and sinks. Its implementation hinders it from scaling to large applications as Pixy has no support for include resolution and object-oriented features.

8 Conclusions

Developers should enforce access controls throughout web applications for every privileged page. This paper proposes a novel approach to detect access control vulnerabilities in web applications with minimal manual effort. Based on the observation that sitemaps presented to different roles are not identical, our analysis first automatically infers the set of privileged pages from the source code of a web application, and then detects access control vulnerabilities via forced browsing. We added support for role-based specification rules, and integrated constraint-solving capabilities with our static analyzer to systematically explore program paths. Our tool is able to achieve good coverage and scale to real-world applications. The evaluation results demonstrate that it is capable of detecting both unknown and known access control vulnerabilities in unmodified web applications with only a few lines of specifications. For future work, we plan to support additional language features of PHP, enhance the string constraint solver, and scale the analysis to larger web applications.

Acknowledgments

We thank the anonymous reviewers and Rob Johnson, the shepherd of this paper, for their useful and detailed comments. We also thank Earl T. Barr, Mark Gabel, Taeho Kwon, Zhongxian Gu and other people who gave helpful feedback on the overall approach and presentation of this work. We especially thank Gary Wassermann and Yasuhiko Minamide for developing the PHP string analyzer and answering our questions. This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, NSF TC Grant No. 0917392, and the US Air Force under grant FA9550-07-1-0532. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna. Multi-Module Vulnerability Analysis of Web-based Applications. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 25–35, 2007.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 387–401, 2008.

[3] W. Chang, B. Streiff, and C. Lin. Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 39–50, 2008.

[4] S. Chen, J. Dunagan, C. Verbowski, and Y.-M. Wang. A Black-Box Tracing Technique to Identify Causes of Least-Privilege Incompatibilities. In *Proceedings of Network and Distributed System Security Symposium*, 2005.

[5] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of the Conference on USENIX Security Symposium*, 2007.

[6] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 63–86, 2007.

[7] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing Authentication and Access Control Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium*, pages 267–282, 2009.

[8] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[9] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 57–72, 2001.

[10] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium*, pages 143–160, 2010.

[11] A. Guha, S. Krishnamurthi, and T. Jim. Using Static Analysis for Ajax Intrusion Detection. In *Proceedings of the International Conference on World Wide Web*, pages 561–570, 2009.

[12] W. G. J. Halfond and A. Orso. Automated identification of parameter mismatches in web applications. In *Proceedings of the Symposium on Foundations of software engineering*, 2008.

[13] S. Hallé, T. Ettema, C. Bunch, and T. Bultan. Eliminating Navigation Errors in Web Applications via Model Checking and Runtime Enforcement of Navigation State Machines. In *Proceedings of the International Conference on Automated Software Engineering*, pages 235–244, 2010.

[14] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (short paper). In *Proceedings of IEEE Symposium on Security and Privacy*, pages 258–263, 2006.

[15] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A Solver for String Constraints. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2009.

[16] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From Uncertainty to Belief: Inferring the Specification Within. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 12–12, 2006.

[17] A. Krishnamurthy, A. Mettler, and D. Wagner. Fine-Grained Privilege Separation for Web Applications. In *Proceedings of the International Conference on World Wide Web*, pages 551–560, 2010.

[18] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification Inference for Explicit Information Flow Problems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 75–86, 2009.

[19] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the Conference on USENIX Security Symposium*, pages 18–18, 2005.

[20] D. Melski and T. Reps. Interconvertbility of Set Constraints and Context-Free Language Reachability. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1997.

[21] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *Proceedings of the International Conference on World Wide Web*, pages 432–441, 2005.

[22] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the IFIP International Information Security Conference*, pages 372–382, 2005.

[23] 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 IEEE Symposium on Security and Privacy*, pages 154–169, 2009.

[24] R. Sekar. An Efficient Black-box Technique for Defeating Web Application Attacks. In *Proceedings of the Network and Distributed System Security Symposium*, 2009.

[25] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the Annual Symposium on Principles of Programming Languages*, pages 372–382, 2006.

[26] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically Inferring Security Specifications and Detecting Violations. In *Proceedings of the USENIX Security Symposium*, pages 379–394, 2008.

[27] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 87–97, 2009.

[28] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–168, 2001.

[29] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–41, 2007.

[30] G. Wassermann and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of International Conference on Software Engineering*, pages 171–180, 2008.

[31] Y. wen Huang, F. Yu, C. Hang, C. hung Tsai, D. T. Lee, and S. yen Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the International Conference on World Wide Web*, pages 40–52, 2004.

[32] Y. Xie and A. Aiken. Static Detection of Security vulnerabilities in Scripting Languages. In *Proceedings of the Conference on USENIX Security Symposium*, 2006.