Detecting Privilege Escalation Attacks through Instrumenting Web Application Source Code

Jun Zhu, Bill Chu, Heather Lipford
University of North Carolina at Charlotte
Charlotte, NC 28223, USA
zhujunfirst1@gmail.com, {billchu, Heather.Lipford}@uncc.edu

ABSTRACT
Privilege Escalation is a common and serious type of security attack. Although experience shows that many applications are vulnerable to such attacks, attackers rarely succeed upon first trial. Their initial probing attempts often fail before a successful breach of access control is achieved. This paper presents an approach to automatically instrument application source code to report events of failed access attempts that may indicate privilege escalation attacks to a run time application protection mechanism. The focus of this paper is primarily on the problem of instrumenting web application source code to detect access control attack events. We evaluated false positives and negatives of our approach using two open source web applications.

Categories and Subject Descriptors
D.4.6 [**Security and Protection**]

General Terms
Security

Keywords
Unauthorized access; privilege escalation; application sensors; security.

1. INTRODUCTION
Violation of access control policies is a core computer security problem. A great deal of effort by the access control research community has focused on designing and implementing the right access control policies. However, this is a difficult challenge both for developers and for tools. We believe more research is needed on run-time protection against access control attacks. Experience based attack models, such as the kill chain model [14], show that a successful attack is preceded by a number of failed probing attempts. In this paper, we propose a method to instrument web application source code to detect attacks on access control so an attack kill chain can be stopped before it completes successfully.

Consider a fund transfer function of an online banking application illustrated in Listing 1. If the condition highlighted in italics failed, it means a user is attempting to transfer funds out of an account he/she does not own. For most online banking applications, this is strong evidence of an attack because it is unlikely caused by an innocent user. Such a user should not be allowed to interact with the application as he/she may successfully exploit unknown access control vulnerabilities. We describe an approach that automatically instruments the application code as illustrated in Listing 2, where the function call in bold reports this privilege escalation event to a run time protection mechanism. We refer to added code as *reporting code*, as it reports to an external agent.

```
public Response fundTransfer () {
if (owns([$_SESSION['user'], $_SESSION['accountNo']) {
// make the transfer
                                    }
else {
 // rejection transfer
}
```

Listing 1. An application event that signals unauthorized access.

```
public Response fundTransfer () {
if (owns([$_SESSION['user'], $_SESSION['accountNo']){
// make the transfer
                                    }
else {
sensor();
// rejection transfer
}
```

Listing 2. Instrument application to report access control attacks.

A good example of a run time application protection mechanism is the open source AppSensor project [13] which aims to detect possible attacking agents based on application specific information as illustrated in Figure 1. It is modeled after network intrusion detection/prevention frameworks, but using application as opposed to network events. A number of mechanisms are available for AppSensor to stop attackers, including blocking the attacker's IP address and suspending the attacker's access credentials.

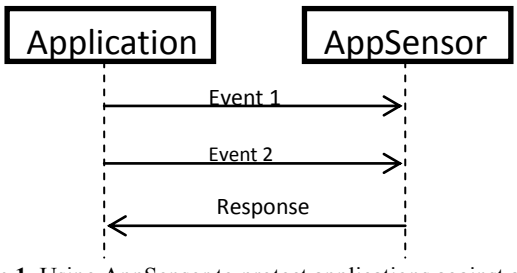

Figure 1. Using AppSensor to protect applications against attacks

AppSensor may make the determination that an application is under attack by correlating multiple events from a given

SACMAT'16, June 5-8, 2016, Shanghai, China.
 ISBN 978-1-4503-3802-8/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2914642.2914661

application or events from multiple applications as well as with reports from network sensors.

Relying on developers to manually insert reporting code as shown in Listing 2 is unlikely to lead to wide adoption. First, inserting such code is extra effort that is not part of the application specification and that will add to developer work load, both in terms of programming and testing to make sure the added reporting code does not change the application's business logic. Second, as part of normal application maintenance, reporting code must be updated and tested to make sure they are in sync with relevant application logic. We believe that developers should be focused on developing applications (and practice secure programming!). Instrumenting code to report access control attacks should be (a) done in an automated way, (b) without changing the intended functions of the application, and (c) having minimal performance impact. Furthermore, instrumented code could be added as part of code deployment so the developer will not have to be aware of added reporting code.

For the rest of the paper we describe an approach that automatically instruments a web-based application to detect likely privilege escalation attack events. This is accomplished in two steps. First we describe an approach to identify candidate web pages to insert code to detect failed attempts to access sensitive information. Second, we present a model on how to automatically insert reporting code without impacting application logic. We report our evaluation results based on two open source web applications.

2. FORCED BROWSING DETECTION

In the context of web applications, privilege escalation often takes the form of forced browsing [6] where an attacker seeks to gain access by directly invoking a particular web page. Our approach is to instrument web application source code to report failed forced browsing attempts. However, not all access failures reliably signal forced browsing attacks.

In a web-based application a typical user navigates application functions by following links and menu items. For example, a micro blog application may require a user to login before writing comments. However, a comment link is always available on web pages even if the user has not logged in (for example see Twitter's web interface where the reply button is always visible). Clicking on the comment link may lead to an access control failure but it is not indicative of a forced browsing attack.

Forced browsing is defined as an action where a subject visits a web page without following available links and menu items. There are, however, scenarios where an innocent user may exhibit forced browsing behavior. The most common case is a user refreshing a protected page after session expiration. False alarms can be minimized by considering context information. For example malicious forced browsing differs from innocent cases in that an attacker often attempts forced browsing on multiple pages over a short period of time. Such context-based decisions can be made by the AppSensor. For example, it may suspend the user after multiple forced browsing attempts on different pages within five minutes, much like suspending user account after repeated failed login in attempts. We are not going to explore possible decision rules for the AppSensor in this paper. Instead, our focus is on automatic code instrumentation to report possible forced browsing events.

The road map for forced browsing detection is as follows. First we discuss how to identify code that implements access control in web-applications. Second we extract a sitemap of a web application as a graph where each node is a web page and edges represent URL links between pages. Third, we identify those web pages that are protected by access control logic, that is, links to such pages are only shown when access privileges are checked. We will insert code to report access control failures in such protected pages as forced browsing attempts.

2.1 Identify Access Control Checks

There is a significant body of research on identifying access control logic in applications. Approaches range from automatic detection using code mining [3,4,5,6,7,8,9] to interactive annotation by soliciting input from developers [1,2,12,16,17]. For our discussions here, we assume one can identify access control logic in the application source with reasonable amount of effort using such techniques. For empirical evaluation of our approach, we chose two open source web applications that have been used by other researchers to show that access control logic can be identified from source code. Specifically we assume the following elements are identified.

Security Sensitive Operation (SSO). Identifying SSOs is an important step in a secure software development lifecycle (SSDLC) [10] because knowing which data elements have security implications is critical for threat modeling [11].We consider SSOs as database operations (e.g. SELECT, INSERT, UPDATE, DELETE on specific database tables) for our work. This can be extended easily to include other operations such as file access.

Code implementing access control check. An access control check for a given SSO must satisfy several requirements: (1) it must be on the execution path from the program entry point to the SSO; (2) it must have the option of changing the execution path to deny access to the SSO. This can be accomplished by either (2.a) a set of Boolean expressions in a branch/conditional statement that lead to altering the execution path leading to the SSO involved, or (2.b) method invocations that could either throw exceptions or terminate the execution. Listing 3 shows a snippet of code from an application. Code that implements access control is shown in *italics*. Here function *require_login()* throws an exception if the user is not logged in and access to the SSO will be denied. Function *print_error* will never return after printing a message, again denying access to the SSO.

```
require_login($course, false, $cm);
if(isguestuser()){print_error('noguests','chat');}
if (!$chat_sid = chat_login_user($chat->id, 'sockets', $groupid, $course)) {
print_error('cantlogin');}
```

Listing 3. Example code snippet, access control checks are shown in *italics*.

2.2 Determine Candidate Webpages to Report Forced Browsing

Our first step is to construct a sitemap. Figure 2 shows part of a sitemap for Wheatblog, an open source PHP application. A sitemap shows possible navigation paths by following links inside web pages. It is a directed graph where each node is a web page. A link between node A and B indicates page A has an URL link pointing to page B. We distinguish two types of links. A link from page C to D is a *conditional* link if this link is displayed on

page C pointing to D only if certain access control checks are satisfied. They are shown as dashed links in Figure 2. Conditional links are often found in an index page where links pointing to pages accessing sensitive information are displayed after the user has been authorized. In other words web applications often "hide" links to pages accessing sensitive information until the access level of the user is determined.

For example in Figure 2 links to blog administration functions (nodes inside the dashed box) are displayed only if the user logged in is an administrator. Because web applications are multi-entrant, access control checks must be repeated in every page accessing sensitive information to prevent forced browsing attacks. For example, in Figure 2 any of the administrative web pages must implement access control checks to make sure the user is an admin. A common access control vulnerability is that one of the pages for an admin function did not contain proper access control check and therefore can be subject privilege escalation attacks.

An *unconditional* link from page A to page B means there is a web link displayed in page A referring to page B without the need to satisfy access control checks. They are shown as solid links in Figure 2.

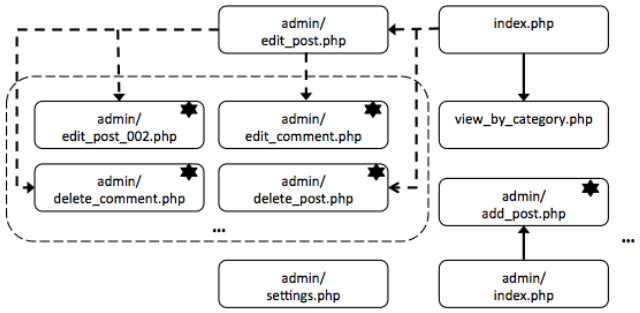

Figure 2. Part of the sitemap of Wheatblog (some pages are omitted as ellipsis due to page limit)

Web applications are built with intended execution paths. Intended starting pages (typically files named index.php in various directories) are assumed to be provided as input to constructing the sitemap. For example, in Figure 2, index.php and admin/settings.php are entry pages. The dashed box in Figure 2 contains a set of pages that are candidates for inserting code to report forced browsing events. This is because each page in that box has at least one SSO and it cannot be reached from an entry point via a path consisting of unconditioned links. That is to say, links to pages in the dashed box are protected by access control logic. We will describe how to identify conditional links in the next subsection. Failed access attempts reported by pages in the dashed box in Figure 2 are good indications of forced browsing.

2.2.1 Rough Sitemap Construction

The process of identifying candidate pages starts with building a rough site map by extracting links from the source files. We call it a 'rough' sitemap because the graph at this stage does not distinguish conditional vs. unconditional links. Input to this process is all web page source files (PHP files).

A PHP file refers to a file with a .php extension. A PHP file may contain function definitions as well as PHP executable programs. Some PHP files contain only function definitions. Such files serve as a function library. A PHP web page is a PHP file with a PHP executable program that is not part of any function definition. Execution of a PHP web page starts with the first statement in the file that is not part of a function definition.

The challenge to build a rough sitemap is to extract links out of web pages as links can be dynamically generated by PHP programs. For example, PHP API calls such as "echo" and "print" are often used to render links. Listing 3 and 4 show example code snippets displaying web links. For both cases, regular expressions can be used to match and locate HTML anchor, form, frame tags, etc. Listing 4 represents a static link, pointing to editsession.php. Listing 5 represents a dynamic link, because $lang is a string variable. Identifying its target involves resolving string variable $lang. Sun et. al. described a comprehensive approach for extracting links from PHP web pages [3], including extraction of most dynamic links. We use their method to identify the rough site map. For this paper, we take a somewhat simplified approach by only looking at static links. The rest of the algorithm for constructing the rough site map follows a fairly standard graph building process. It should be noted that the rough sitemap is represented as a multi-graph as it is possible to have parallel links from one page pointing to another page.

```
print "<a href='editsession.php'>(edit)</a>";
```

Listing 4. Static link example from SCARF

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

Listing 5. Dynamic link example

We use an example adapted from Wheatblog shown in Figure 2 to illustrate the algorithm. The first step of the rough sitemap construction is to put all known entry pages into a worklist queue. Then we pick a node from the worklist queue, suppose it is index.php. If the node has not been processed before, we extract all links from this PHP webpage. This process will return tuples, for example (index.php, admin/edit_post.php) and (index.php, admin/delete_post.php), indicating that index.php points to both admin/edit_post.php and admin/delete_post.php. For page nodes with no links identified on it, tuples (nodes, _) will be returned. Tuples representing links to web pages outside the application are discarded. Identified tuples will be added to the rough sitemap. Nodes admin/edit_post.php and admin/delete_post.php will be added to the worklist. We remove index.php from the worklist and add it to the visited set to prevent processing it again. This process continues until the worklist is empty. The resulting set of tuples is the rough sitemap.

2.2.2 Identify Conditional Links

Listing 6 shows an example of a conditional link to web page editsession.php. This link is shown on the screen if the user has admin rights. Identifying a link that is displayed conditionally is straightforward. Either the link is rendered as part of a branch statement (as indicated in Listing 6), or the rendering of the link is preceded by a function call that may not return due to either an exception or abort statement in the function. When considering exceptions, all subclasses of the PHP class Exception must be considered. The challenge is to determine whether such a condition in rendering the link is related to access control.

```
if (is_admin()) {
print " <a href='editsession.php'>(edit)</a>";
}
```

Listing 6. Rendering a link upon condition in SCARF

As mentioned in Section 3.1, we assume the code implementing access control checks has been identified. As part of one of the author's dissertation research [15], six open source PHP applications were studied for their implementation of access control. We discovered that many conditional links involve checking global variables (e.g. values associated with the web session) that are known to be involved in making access control decisions.

This observation has a reasonable explanation. Access control decisions often involve database queries. A common pattern to implement access control is to extract a given user's access privilege information from the database and store them in the web session and use session variables to perform both access control checks as well as determining what links should be displayed to a given user. We want to test how well this heuristic works in identifying conditional links in PHP web applications.

We use the term *critical global variable* to refer to trusted global variables that determine the course of an execution path. A global variable is trusted if it has never been assigned a value from an untrusted source in its data flow chain. In this paper we consider critical global variables as either part of a Boolean expression that are involved in a branch statement (if-then-else, or switch), or from functions that do not return due to exceptions or abort statements. We collectively refer to functions that may not return as function calls with abnormal returns. A third type of program construct, not considered in this work, may also be involved in determining execution outcome. That is conditions may be part of the WHERE clause in SQL statements. Monshizadeh et al. [9] have developed an approach to work with conditional logic in SQL statements. Our definition of critical global variables can be expanded to include these cases in the future.

Critical global variables can be directly extracted from Boolean expressions and switch statements because global variables can be easily identified directly or through dataflow analysis. We leverage open source software that constructs abstract syntax trees (AST) for PHP programs and perform data flow and control flow analyses using AST structures. For function calls with abnormal return we extract those global variables that cause the function to not return normally. We construct a function call graph and determine those global variables that are part of the decision to determine whether the function returns. We illustrate the process of critical global variable extraction using two examples from SCARF.

```
function require_admin(){
if(!is_admin())
        die("You don't have access to view it");
}
function is_admin() {
        if ($_SESSION['privilege'] == 'admin') return TRUE;
        else return FALSE;
}
```

Listing 7. Illustrative example from SCARF for security critical variable extraction

The first example concerns with extracting critical global variables from a function with abnormal return. Function *require_admin()* in Listing 7 returns if the user is an administrator. It aborts if the user is not an administrator by making the *die()* system call. Function *is_admin()* is called by function *require_admin()* to determine whether a user is an administrator. Definitions of both functions are shown in Listing 7. In order to extract critical variables associated with function *require_admin()* we construct a function call graph consisting of *require_admin()* invoking *is_admin()*. Global variables involved in making decisions whether *require_admin()* returns involves $_SESSION['privilege']. This variable is not tainted because it does not come from the user. Thus, the critical variable extracted from *require_admin()* is $_SESSION['privilege'].

```
$action = $_GET['action'];
$user_level = $_SESSION['user_level']; //data from global variables
if($user_level == 'Admin' && $action =='deleteAll') //annotated check
query("DELETE FROM sensitive_table "); //SSO
```

Listing 8. Example of Boolean expression as access control check

Listing 8 is another example from SCARF to illustrate the extraction of critical global variables from Boolean expressions. Consider the Boolean expression *$user_level == 'Admin' && $action =='deleteAll'*. First, a set of variables {*$user_level, $action* } is identified in the Boolean expression. We then perform dataflow analysis. One finds *$user_level* depends on global variable $_SESSION['user_level'], so it is added to the set of variables identified. Variable *$action* depends on global variable $_GET['action'], so it too is added to the set of variables identified. To obtain critical global variables, we remove variables that are not global, so variables *$user_level* and *$action* are removed. We further remove global variables that may contain untrusted data. In this example, because $_GET['action'] contains data from user input and it is widely regarded as a tainted source, $_GET['action'] is removed. Then the obtained set of critical variables is {$_SESSION['user_level']};

We define a *security critical global variable* as a critical global variable that is known to be involved in access control decisions. Recall that one of our assumptions is that we are given a set of annotated access control checks for security sensitive operations. Critical global variables extracted from these checks are security critical global variables. For example both the Boolean expression in Listing 8 and function *require_admin()* are identified access control checks. So the set of security critical variables associated with them is {$_SESSION['privilege'], $_SESSION['user_level']}.

A link is considered to be a *candidate for conditional link* if it is displayed as part of a branch statement (if-then-else statement or switch statement), or proceeded in the execution path by a function call that may not return.

We use the following steps to determine whether the link is a conditional link.

1. Extract a set of security critical global variables from the set of identified access control checks for SSOs, referred to as SCV_ssos.
2. For each link l in the rough sitemap that is a candidate for conditional link
   a. Extract a set of critical global variables for displaying l, referred to as CV_link_l.
   b. Compare CV_link_l with the set of security critical global variables for SSOs SCV_ssos, if the two sets share one or

more elements (variables), then the link is regarded as a conditional link.

We illustrate this process using the SCARF example in Listing 6. The if-branch *if(is_admin())* determines whether the link displaying code will be executed, so this link is a candidate for conditional links. Critical global variables are extracted for function call *is_admin()* which is defined in Listing 7. From our earlier description, we know the critical global variable for *is_admin()* is {$_SESSION['privilege']}. Since security critical global variables for application SCARF includes {$_SESSION['privilege'], $_SESSION['user_level']}, this link to *editsession.php* is a conditional link.

2.2.3 Identify Candidate Pages

Based on the rough sitemap constructed using steps described in Section 3.2.1 and the set of conditional links identified using steps described in Section 3.2.2, a candidate web page for inserting forced browsing events is a web page satisfying the following conditions: (1) the page has at least one SSO, and (2) there does not exist any navigation paths from any entry page to it with only unconditional links. In Figure 2, all page nodes with a star on their top right are pages with SSOs; six pages in the dashed box are identified as candidate pages.

2.3 Automatic Code Insertion

One of the authors studied six open source PHP applications as part of his dissertation research [blind ref]. Based on analysis of these empirical results, we created a model for automatic insertion of code to report forced browsing events.

When reporting a forced browsing event, application context information could be captured by calling APIs or retrieving from global variables. For example, in PHP, a session id could be obtained by calling API *session_id()*, host IP address could be obtained by retrieving from global variable *$_SERVER*. One could also provide a serialized object for session content along with time and date stamps. Insertion of code to report forced browsing must be done in such a way that does not impact the normal application flow. Throughout this paper, we use the function call **sensor()** to denote reporting code that is instrumented into the application.

Since **sensor()** is used to capture failure events of access control checks, it should be placed on all execution paths other than the path leading to execution of the SSO. As described earlier, access control checks could be a Boolean expression or a function calls with abnormal return. We did not observe any access control checks involving loops. We first describe code instrumentation for Boolean expression access control checks, followed by code instrumentation for function calls with abnormal returns.

2.3.1 Code Instrumentation for Boolean Expression Access Control Checks

We illustrate code insertion through examples. The left side of each listing is the code before inserting **sensor()**. Access control checks are shown in *italics*. The right side shows after code instrumentation. Newly inserted code is shown in **bold**.

In Listing 9, SSO is executed if *condition* is true. Forced browsing will be reported if *condition* fails, or access is denied. Function **sensor()** will be called right after the if statement containing the access control check.

if (*condition*){	if (*condition*){
SSO;	SSO;
}	}
	sensor();

Listing 9. If-then statement

Listing 10 shows an if-then-else statement where the SSO is executed if *condition_a* succeeds. Reporting code must be added in the else block before code indicated by "//some logic", as that logic may contain an exception.

if (*condition_a*){	if (*condition_a*){
SSO;	SSO;
} else {	} else {
//some logic	**sensor();**
}	//some logic
	}

Listing 10. If-then-else statement

List 11 shows a switch statement where the SSO is one of the cases. In this case instrumented code must be inserted into all other cases of the switch statement.

switch (*expr*) {	switch (*expr*) {
case label1*:*	case label1*:*
SSO;	SSO;
break;	break;
case label2*:*	case label2*:*
break;	**sensor();**
}	break;
	default:
	sensor();
	break;
	}

Listing 12. Switch branch

2.3.2 Code Insertion for Function Call Access Control Checks

We illustrate code insertion for function calls with abnormal returns through an example adapted from SCARF (Listing 13). The top part of the listing shows the access control check (function *require_logggedin()*) and the SSO. The bottom part shows the definition of function *require_loggedin()* which aborts if the user's identify (email) is unknown. Since access control check *require_loggedin()* dominates the path flowing to the SSO, access control fails when *require_loggedin()* fails. Thus reporting code should be inserted in *require_login()* right before the abortion of normal execution, as shown on the right half of Listing 13. There may exist multiple abnormal returns, each needs to be instrumented with a call to **sensor()**. Function *require_login()* may invoke another function that contains abnormal return. If that is the case, that function needs to be analyzed using the same process describe in this section.

```
require_loggedin();
query("INSERT INTO comment …); //SSO

function require_loggedin() {
if (getEmail() === FALSE) {
die ("You must be logged in");
//abnormal return
}}
```

```
require_loggedin();
query("INSERT INTO comment …); //SSO

function require_loggedin() {
if (getEmail() === FALSE) {
    sensor();
die ("You must be logged in");
//abnormal return
}}
```

Listing 13. Function call with abnormal return

3. EVALUATION

We performed a proof-of-concept evaluation using two PHP open source projects Wheatblog and SCARF that were used in previous research on access control implementations [6,7], [9,10,11,12]. We choose these applications in part because it has been shown that access control logic can be identified from their source code. Wheatblog is a blogging application with over 4000 lines of code. SCARF is a conference paper discussion forum with over 1300 lines of code. We seek to answer the following questions: (a) what is the likelihood that instrumented code will generate false positive forced browsing reports: i.e. a forced browsing event is incorrectly detected; and (b) false negatives: a forced browsing event is not reported.

Table 3 and Table 4 lists all the PHP web pages in Wheatblog and SCARF respectively, each page is assigned a number which will be referenced in subsequent discussions. For example, in row three of Table 3 is Wheatblog's page admin/add_post.php. Pages that are identified as targets for forced browsing are listed in *italics* in Tables 3 and 4. This page is subsequently referred to as page 3 of Wheatblog. Tables 5 and 6 summarize the sitemap for Wheatblog and SCARF respectively. Each row in these tables represents a page, e.g. in table 5 the web page in row 3, Wheatblog's admin/add_post.php, has one SSO; Wheatblog's page 17 has an unconditional link pointing to Wheatblog's page 3. Wheatblog's pages 10-14 and 18 have conditional links pointing to Wheatblog's page 3. Pages that are not pointed to by any other pages are regarded as entry pages as these pages may be published in public web pages as entry points to the application.

Pages identified as a candidate for reporting forced browsing are underlined. Figure 3 shows a part of the constructed sitemap. Dashed links represent conditional links, solid links for unconditional links, and dashed box represents pages with SSO. In Figure 3, page admin/add_link.php is a candidate page, because one can only navigate to it via conditioned links.

Table 3. Wheatblog web pages, italics represent forced browsing targets

No.	PHP executable file
1	*admin/add_category.php*
2	*admin/add_link.php*
3	*admin/add_post.php*
4	*admin/delete_category.php*
5	*admin/delete_comment.php*
6	*admin/delete_link.php*
7	*admin/delete_post.php*
8	*admin/edit_categories.php*
9	*admin/edit_comment.php*
10	admin/edit_post.php
11	*admin/edit_post_002.php*
12	admin/manage_categories
13	admin/manage_links.php
14	*admin/manage_links_002.php*
15	admin/manage_posts.php
16	admin/manage_users.php
17	admin/index.php
18	includes/header.php
19	index.php
20	add_comment.php
21	view_by_permalink.php
22	view_by_category.php
23	view_by_archive.php
24	view_by_title.php
25	archive.php
26	update_archive.php
27	list_category.php
28	registration.php
29	view_links.php
30	admin/settings.php

Table 4. SCARF web pages, italics reprent forced browsing targets.

No	PHP executable file
1	*editpaper.php*
2	*addsession.php*
3	*editsession.php*
4	useroptions.php
5	comments.php
6	*generaloptions.php*
7	header.php
8	showpaper.php
9	showsessions.php
10	index.php
11	fogot.php
12	login.php
13	install.php
14	register.php

Table 5. Wheatblog sitemap (Candidate pages underlined)

No.	SSOs	Pages having unconditional links to it	Pages having conditional links to it

1	1		12
2	1		13
3	1	17	10-14, 18
4	1		12
5	2		10
6	1		13
7	1		15,19
8	1		12
9	1		10
10	0		5,15,19
11	1		10
12	0	17	4,8,10-14,18
13	0	17	2,6,10-14,18
14	1		13
15	0	17	7,9-14,18
16	3	17	10-14,18
17	0	Entry page	Entry page
18	0	3,5,10,11,17,19-25,27-29	12-16,26,30
19	0	Entry page	Entry page
20	2	21	
21	0	19,20,22-24	
22	0	18,19,21,23,24, 27	15
23	0	25	26
24	0	18	
25	0		26
26	0	Entry page	Entry page
27	0	Entry page	Entry page
28	0	18	
29	0	Entry page	Entry page
30	1	Entry page	Entry page

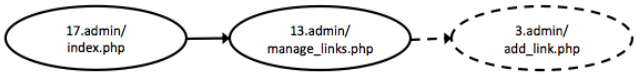

Figure 3. Admin/add_link.php is a candidate page.

Wheatblog's page 3, admin/add_post.php, on row 3 in Table 5, on the other hand is not a candidate page because it is pointed to by entry pages admin/index.php (page 17). This means an unauthorized user could access admin/add_post.php by clicking a visible link on page admin/index.php.

Among 30 Wheatblog pages, 10 pages are considered as candidate pages for reporting forced browsing events. Results for SCARF are similarly reported in Table 6, with 4 candidate pages out of 14 web pages. All candidate pages require administrative privileges and are "hidden" behind administrative login pages.

Table 6. SCARF sitemap (Candidate pages underlined)

No	SSOs	Pages having unconditional links to it	Pages having conditional links to it
1	8		7-9
2	1		7
3	6		9
4	3	7	5
5	3	8	7
6	3		7
7	0	1-6,8-14	
8	0	9	1,3
9	0	3,10	7
10	0	Entry page	Entry page
11	0	12	
12	0	5,7	
13	0	10	
14	0	7,10,12	

The accuracy of the algorithm described in section 3 is evaluated based on false positives and false negatives. A false positive is when the algorithm miss-identifies a page as only reachable by conditioned links. False positives could occur due to inaccuracy of our approach to identify conditioned links (section 3.22). We examined all candidate pages in Tables 5 and 6 by source code review and did not find any false positive cases. False positives are certainly possible. We have at least two places to address potential false positives. First the heuristics we used in section 3.2.2 can be refined. Second, decision rules in AppSensor may be tuned to seek multiple sources of evidence before taking preventive measures.

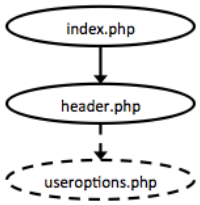

Figure 4. A false negative case

A false negative refers to a situation where a forced browsing event would be missed. We examined every page in both Wheatblog and SCARF. Based on intended application functions we determine whether a forced browsing attack could occur on each page. We found one false negative case in SCARF as illustrated in Figure 4. SCARF is a public discussion forum for published conference papers. All papers are publicly available for discussions. However, in order to participate in discussions, one must be logged in. The false negative case involves the link between page 7 and 4. Page useroptions.php, containing user options, should be displayed for authenticated users only thus viewing it without first logged in could indicate a forced browsing attack.

Close examination shows that SCARF uses condition isset($_SESSION['email']) to check whether a link to page 4 should be displayed. The critical global variable for displaying this link is $_SESSION['email']. However this global variable is not involved in any access control checks to access database resources. Therefore our algorithm did not regard this link as a conditional link. We found that global variable $_SESSION['priviledge'] was used to check for authenticated users in the application logic for access control. There is a relationship between $_SESSION['priviledge'] and $_SESSION['email'] in that they were assigned values from the source, namely a logged in user. It may be possible to mitigate such false negative situations by performing more sophisticated data flow analysis on dependencies of session variables to discover that certain groups of variables are "equivalent" because they are fed from the same source. We plan to investigate this in future research.

4. CONCLUSION AND FUTURE WORK

Our research aims at providing run time protection against privilege escalation attacks by automatically instrumenting application source code to report possible forced browsing attacks. The main technical contribution of this paper is to outline a method to automatically identify web pages that may be the target of forced browsing attacks. Our method is based on an observed pattern of how developers implement access control in PHP web applications. The work presented in this paper is the first step in our research plan: providing a proof of concept evaluation and prototyping to establish the feasibility of this approach. We evaluated our method using two open source PHP web applications and found this approach promising. No false positives were found. There was one false negative and we discussed how this might be mitigated through more in-depth data flow analysis of dependencies among global variables.

Our future research plan includes (a) more empirical evaluation of applications of different types and complexity, (b) looking at other heuristics to detect unauthorized access attempts, (c) more rigorous description of the algorithm for identifying unconditional links, including fully considering conditions in SQL statements, with the purpose of making sure all possible cases are thoroughly considered, (d) more thorough analysis of the correctness of inserted reporting code and its impact on performance, and (e) examining strategies for decision making rules for AppSensor to detect unauthorized accesses.

5. ACKNOWLEDGMENTS

NSF grants: 1129190, 1318854.

6. REFERENCES

[1] Zhu, J. Xie, J. Lipford, H., Chu, B. December 2013. Support Secure Programming In Web Applications through Interactive Static Analysis. Journal of Advanced Research. Elsevier.

[2] Xie, J., Chu, B., Lipford, H.R., Melton, J.T., 2011.ASIDE: IDE support for web application security. In Proceedings of the 27th Annual Computer Security Applications Conference ACM, 267-276.

[3] Sun, F., Xu, L., Su, Z. (2011, August). Static Detection of Access Control Vulnerabilities in Web Applications. In USENIX Security Symposium.

[4] Son, S., Shmatikov, V., 2011. SAFERPHP: Finding semantic vulnerabilities in PHP applications. In Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security, ACM.

[5] Tan, L., Zhang, X., Ma, X., Xiong, W., Zhou, Y., 2008. AutoISES: Automatically Inferring Security Specification and Detecting Violations. In USENIX Security Symposium, 379-394.

[6] Dalton, M., Christos K., Nickolai Z.. "Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications." USENIX Security Symposium. 2009.

[7] Son, S., Mckinley, K.S., Shmatikov, V., 2011. Rolecast: finding missing security checks when you do not know what checks are. In ACM SIGPLAN Notices, ACM, 1069-1084.

[8] Gauthier, F., Lavoie, T., Merlo, E. (2013, December). Uncovering access control weaknesses and flaws with security-discordant software clones. In Proceedings of the 29th Annual Computer Security Applications Conference (pp. 209-218).

[9] Monshizadeh, M., Prasad N., Venkatakrishnan V. N., 2014. "Mace: Detecting privilege escalation vulnerabilities in web applications." In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014.

[10] Howard, M., Lipner, S. (2006) The Security Development Lifecycle. Microsoft Press.

[11] Shostack, A. Threat Modeling, Design for Security. 2014. John Wiley & Sons Inc.

[12] Zhu, J., Chu, B., Lipford, H., Thomas, T. "Mitigate Access Control Vulnerabilities through Interactive Static Analysis", in *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies (SACMAT). ACM, 2015.*

[13] Watson, C., Groves, D. and Melton, J., *AppSensor* OWASP Foundation, 2014.

[14] Hutchins, E., Clopper, M., and Amin, R. "Intelligence Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion kill Chains", White paper Lockheed Marin Corporation, http://www.lockheedmartin.com/content/dam/lockheed/data/corporate/documents/LM-White-Paper-Intel-Driven-Defense.pdf

[15] Zhu, J. *Interactive Static Analysis for Application Security* Ph.D. Dissertation, UNC Charlotte June, 2015.

[16] Tomas, T., Smith, J., Murphy-Hills, E., Chu, B., and Lipford, H. "A study of Interactive Code Annotation For Access Control Vulnerabilities" in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC15)* Oct. 2015.

[17] Smith, J. Johnson, B. Murphy-Hill, E., Chu, B. Lipford, H. "Questions Developers Ask While Diagnosing Potential Security Vulnerabilities with Static Analysis" in *ACM SIGSOF FSE*, Oct 2015.