EARs in the Wild: Large-Scale Analysis of Execution After Redirect Vulnerabilities

Pierre Payet
Ecole Superieure d'Informatique Electronique Automatique, Paris
payet@et.esiea.fr

Adam Doupé, Christopher Kruegel, Giovanni Vigna
University of California, Santa Barbara
{adoupe, chris, vigna}@cs.ucsb.edu

ABSTRACT

Execution After Redirect vulnerabilities—logic flaws in web applications where unintended code is executed after a redirect—have received little attention from the research community. In fact, we found a research paper that incorrectly modeled the redirect semantics, causing their static analysis to miss EAR vulnerabilities.

To understand the breadth and scope of EARs in the real world, we performed a large-scale analysis to determine the prevalence of EARs on the Internet. We crawled 8,097,283 URLs from 255,957 domains. We employ a black-box approach that finds EARs which manifest themselves by information leakage in the HTTP redirect response. For this type of EAR, we developed a classification system that discovered 2,173 security-critical EARs among 416 domains. This result shows that EARs are a serious and prevalent problem on the Internet today and deserve future research attention.

1. INTRODUCTION

The Internet and the Web have become an integral part in the lives of billions of people who routinely use online services to store and manage sensitive information. Unfortunately, the popularity of online services has attracted cybercriminals. An important class of attacks targets web applications: For example, in 2010, malicious activity targeting businesses' web applications increased 93% compared to the previous year [13]. Ideally, web applications would be impervious to leaks, attacks, or any other threat. However, due to the complexity of modern web architectures and web technologies, this goal is elusive. Even vulnerabilities such as SQL injection or cross-site scripting (XSS), which are well-known, are still frequently exploited and make up a significant portion of the vulnerabilities discovered every year [6, 27].

In this paper, we focus on a class of security flaws that is not as well-known as XSS and SQL injection vulnerabilities, but equally serious from a security point of view: Execution After Redirect [10].

Execution After Redirect, or EAR, is a type of security flaw that occurs when unintended code is executed after a call to a redirect function in a web application. The web application developer intends for the control flow of the web application to halt at the call to the redirect function, but, depending on the web application framework, the control flow does not halt. This misunderstanding of the web application framework semantics causes an EAR vulnerability because unintended code is executed and the web application sends an HTTP redirect response. As a result of the EAR, the unintended code execution can cause information leakage through the HTTP redirect response or can cause unauthorized database modification.

Previous research on EARs focused on detecting EARs in Ruby on Rails web applications via static analysis of the source code [10]. This approach detects unauthorized database modifications, but is limited to Ruby on Rails applications and requires access to the application code. This paper attempts to find information leakage EARs by looking at the HTTP redirect response.

An HTTP redirect response is used by a web application in the following way. When a user attempts to access a resource on a web application, the server can send an HTTP redirect response, indicating to the user to look elsewhere for the requested resource. Depending on the HTTP status code sent by the server[1], the browser will make a new request to the redirect location [12].

To assess the prevalence of information leakage EARs on the web at large, we performed a large-scale crawl of the web. During this crawl, we are looking for evidence of an information leakage EAR. Because information is leaked through the content of the HTTP redirect response, we look at the HTTP redirect response as external evidence of an information leakage EAR. From our crawl, we developed a classification system that classifies content of the HTTP redirect response, or simply redirect content, as benign or security critical. This classification system works in a black-box manner—with no access to the web application's code, only the HTTP redirect response content.

In summary, we provide the following contributions:

- A black-box approach to detecting a class of EARs (more precisely, information-leakage EARs).
- A classification tool based on the proposed approach that can automatically and efficiently identify EARs.
- A quantification of information-leakage EARs on the Internet based on a large-scale crawl of web applications. We found 2,173 likely vulnerable EARs across 416 domains.

2. EXECUTION AFTER REDIRECT

An Execution After Redirect is broadly defined as any unintended (from the perspective of the developer) server-side code that is executed after a redirect call. In this way, an EAR is an unintentional control flow vulnerability: The developer does not intend for

[1] `301`, `302`, `303`, `304`, or `307` all indicate a redirect.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'13 March 18-22, 2013, Coimbra, Portugal.
Copyright 2013 ACM 978-1-4503-1656-9/13/03 ...$10.00.

```
1 <?php
2 if (!$user->is_premium_member())
3 {
4   header("Location: /signup.php");
5 }
6 echo "Premium content that requires a subscription.";
7 ?>
```

Listing 1: Example of PHP code where execution continues after the redirection is triggered by the `header` function.

```
HTTP/1.1 302 Found
Server: Apache/2.2.3 (CentOS)
X-Powered-By: PHP/5.1.6
Set-Cookie: PHPSESSID=oj5intb9382pmevfm92pbm7bj7; path=/
Location: /login.php?auth=false
Content-Type: text/html; charset=ISO-8859-1

<html>
  <head><title>FootPlus: Player Statistics</title></head>
  <body>
    <div id="main_container">
      <b>Player Name: </b>Christopher Vigna<br/>
      <b>Position: </b>TE<br/>
      <b>Avg Yds: </b>XYZ<br/>
      <b>Avg Points: </b>X<br/>
      ... More Content ...
    </div>
  </body>
</html>
```

Listing 2: Raw HTTP response adapted from a real EAR our crawler found. The response code is `302`, yet there is content in the body. In this case, the content requires a paid subscription to access, yet it is sent in the response body.

the code to be executed. However, because of his misunderstanding of the redirection semantics of the web application framework, unintended code is executed anyway. Note that EARs defined in this way are bugs but not necessarily security-critical flaws. An EAR can become security-critical depending on *what* code is executed after a redirect. Therefore, the severity of an EAR is entirely application-specific.

A security-critical EAR can compromise the environment of the web application in two ways: (1) permanently change the state of the web application or (2) leak sensitive information in the response. We call EARs that leak sensitive information in the response *explicit,* and those that do not leak any information *silent.* Frameworks, depending on their architecture, design, and language, may allow silent EARs, explicit EARs, or both [10].

Listing 1 shows an example of an EAR in server-side PHP code. Line 2 checks if the user is a premium member, and if not, the `header` function on Line 3 is invoked to redirect the user to a page where she can purchase a premium account. However, because `header()` does not halt execution, the control flow will continue and the premium content will be sent on Line 4. While this example is simplistic, we refer the interested reader to previous research for more complex examples [10].

Listing 2 shows a raw HTTP response adapted from an EAR we discovered in the wild. The EAR exists on a web application that sells American-football statistics, however a request to a page with the player's statistics leaks the data the application is selling. As shown in Listing 2, the response code is `302`, and there is a `location` header redirecting the user to the login page. However, the full content (in this case the player's statistics) is leaked after the headers. Note that this is equivalent to leaking the proprietary contents of the site's database to unauthenticated users.

An Execution After Redirect vulnerability can violate the confidentiality, integrity, or both of a web application. Confidentiality is violated through information leakage (as shown in Listing 2), where private data requiring an access fee is sent in the response. The integrity of the web application can be threatened when the server-side code continues to execute after failing an access-control check. In these attacks, an unauthorized user can change the state of the web application, often by modifying the database. An example of this type of vulnerability is a forum where a non-administrator user can change the title of a thread [10].

EAR vulnerabilities have been studied only recently. We found the first public EAR instance in the Common Vulnerabilities and Exposures (CVE) database in 2007, as `CVE-2007-2003`. There is also an entry[2] in the Common Weakness Enumeration (CWE) database (a community-developed list of software weaknesses) from 2008 called "Redirect Without Exit." However, even with this classification, no CVE entry is associated with this CWE entry. This lack of association shows that even if the community at large might have been exposed to the concept of EARs, they do not use this category to classify discovered EARs.

EARs have received little direct attention from the research community, nonetheless, there is evidence of EARs in some of the literature. For instance, Swaddler [8] found an EAR in the PHP BloggIt web application, however the authors did not identify it as such. In addition, as previously reported by Doupé et al. [10], Felmetsger et al. discovered an EAR in a Java web application and did not identify the vulnerability as an EAR [11]. Chaudhuri and Foster present a Ruby on Rails example, `UsersController`, that contains an unmentioned EAR vulnerability [5]. Finally, EAR vulnerabilities are subtle and can occur even in published peer-reviewed papers. For instance, Sun, et al. incorrectly model the semantics of redirection in PHP, thus ignoring and missing EAR vulnerabilities [23].

3. EAR DETECTION

We developed a black-box classification system to detect different types of EAR vulnerabilities. In the following, we describe the types of vulnerabilities that we identify and our detection technique.

3.1 White-Box vs. Black-Box

For automated vulnerability discovery, one can follow two major approaches: white-box and black-box testing.

White-box testing analyzes the application's source code. In this way, the testing tool obtains a view of the entire application, including all entry points. Typically, a control flow graph (CFG) is created, and data flow techniques are used to find paths from the points where external information (e.g., user input) is read (sources) to security-critical operations (sinks). If insufficient sanitization is performed along one of these paths, the application contains a vulnerability (e.g., a SQL injection if the sink was a SQL query).

While a white-box testing approach derives a full view of an application, it can only analyze applications written in the specific language that the tool targets.

Black-box analysis approaches operate without any knowledge of the internal working of the application: The internal state of the application and its source code are unknown. A black-box web vulnerability analysis tool sends requests to the web application and observes the associated responses. The idea is to detect vulnerabilities in a way that is independent of the underlying language.

Because they are not language-dependent, black-box tools are able to operate on a wide range of applications. In addition, black-box tools usually suffer from fewer false positives when compared to white-box tools. The lack of false positives is due to the black-

[2] `CWE-698`

box tool actually exercising the application and exploiting the vulnerability. However, black-box tools suffer from a discoverability problem: They cannot find a vulnerability in code that they fail to execute.

The black-box approach we took to detect information leakage EARs has the possibility of false positives. These false positives are due to the black-box nature of our approach: We do not know for certain if the information leaked is the result of an EAR or is from a legitimate HTTP redirect response. Moreover, the content of the leaked information along with the way the web application uses that information determines the severity of the EAR—therefore automatically determining the severity of the information leakage will have false positives.

3.2 Black-Box EAR Detection

The goal of our approach is to detect EARs in a black-box manner. EARs need not be directly visible externally: code executing after a redirect could, for instance, change the server-side state. We limit ourselves to analyzing the HTTP response of the web application. For an HTTP response to indicate an EAR vulnerability, two things must be true. First, by definition, the response must be an HTTP redirect response. Second, the HTTP redirect response content must divulge confidential information about the web application. Therefore, our system attempts to detect if the HTTP redirect response content divulges confidential information, thus indicating a potential EAR.

3.3 Classification of non-EARs

To develop this classification system, we manually analyzed the initial results from our large-scale crawl of the Internet. From this analysis, categories emerged. It is from these categories—developed by looking at actual content of redirect response—that we created the non-EAR and EAR classification categories.

In the following we describe the heuristics we use to make the distinction between legitimate content (non-EAR) and EAR content. We first identify as legitimate responses that are empty. Then we developed a number of string patterns that match text commonly used in legitimate redirections[3]. There is no standard format for such messages, but we found common patterns because the HTTP redirect responses are sent by a few well-known web frameworks. We also consider as legitimate responses in which the body of the HTTP redirect response is a near duplicate of the page the redirection leads to. This is because, obviously, no additional, sensitive information was revealed to the client. Then, we discard broken and malformed HTML content which typically contains page headers or navigation menus.

Framework Redirect
We examined a number of the most popular web frameworks and the web pages they produced as part of the redirection process. From these pages, we were able to extract a set of signatures (regular expressions) that help us identify such content.

Generic Redirect
In the general case, the content of a redirect message aims to warn a user of a changed location. Therefore, we manually developed regular expressions by analyzing a large corpus of data during our experiments. Overall, we developed 110 such signatures.

Irrelevant or Broken Content
These are HTTP redirect responses whose content is not syntactically correct or contains very little information. To estimate the value and extent of information in an HTTP redirect response, we first remove all the HTML tags (so that only text content is left). If less than n words are left, we consider the content to be irrelevant. With this irrelevant content, it is clear that server-side code has been executed but, due to the lack of information, we cannot automatically determine the severity of the flaw; so we classify these as non-security critical.

Near Duplicate
Near duplicate detection recognizes cases in which content of the redirect page is similar or identical to the redirect location's content. We utilize the well-known Normalized Compression Distance (NCD) to rapidly compute an approximation of the similarity between the HTTP redirect response content and the redirect location's content [7]. These are not EARs, because the information contained in the HTTP redirect response is very similar to the target page that is presented to the user.

3.4 Classification of EARs

Once we are left with content of the redirect page that is not obviously legitimate, we divide the remaining redirects using heuristics that attempt to identify different cases based on their severity. Note that our classification system processes the content of the redirect page in a precise order that will be explained in Section 3.5.

Error Message
Some of the content sent by the server contains error messages, which we wish to capture in a category. So we put in this category potential security-critical errors such as PHP errors and Java errors. Content of the redirect page in this category discloses information about paths of the server, details about the framework, the language and often, their version number. Black-box vulnerability scanners look for the same error messages when scanning a web application. We developed 11 regular expressions to classify the response content as an *Error Message.*

HTTPS Redirect
This category consists of responses from the server in which the HTTP redirect response contains a full web page *and* the redirect leads to a page served through a secure channel (via HTTPS). We consider this a security-critical EAR because, by redirecting to a secure channel, the web application is signaling that the content is sensitive. Yet, the web application sends some content in the clear.

Pre-Login Access
Responses from this category redirect to a login page. The intuition here is twofold: (1) when one tries to access a restricted resource the web application will redirect her to a login page, and (2) the restricted content of the redirect page will contain links to restricted content. Here, we attempt the capture the (application-specific) logic of a web application requesting authentication from a user.

To detect this type of security-critical EAR, we select links inside the response body, in addition to the one in the `Location` header, that also cause an HTTP redirect response. If, among these links, a significant amount (80%, explained in Section 3.6) leads to a login page, we classify them in this category.

Transparent Barrier
This category classifies responses which contain links that, when requested, redirect to the same page. Web application developers sometimes want the user to visit a certain page prior to allowing access to the full content of the web application. For example the user is redirected to the same page until she has "signed" a Terms of Use form. These cases are considered as a non-security–critical EAR because the information they limit the access to is available after visiting the special page.

[3]An example regular expression pattern is: `<title>(30[123] )?document moved ((permanently)|(temporarily))?</title>`

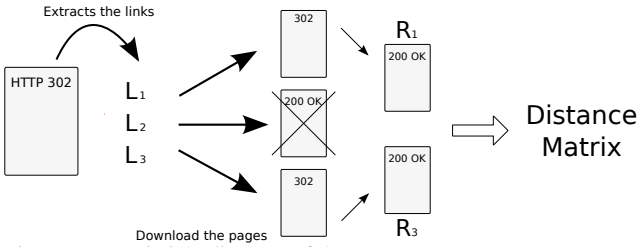


Figure 1: Heuristic's diagram of the *Transparent Barrier* category.

To classify content of the redirect page in this category, we extract all the links on the HTTP redirect response (L_1, L_2, and L_3 in Figure 1) and request the corresponding pages. From those requests that redirect, and contain content (R_1 and R_3 in Figure 1), we compute the NCD similarity of each content to every other content. If a certain percentage (discussed in Section 3.6) is very similar, then the original response is considered to be a *Transparent Barrier.* The intuition here is that we wish to capture instances where content is hidden behind a barrier.

Other

This category encompasses content of the redirect page that did not match any of the previous categories.

The previously-discussed categories provide a way to assess the potential severity of the content of the redirect page. Instances matching the *Framework, Generic Redirect,* and *Near Duplicate* categories, even though they send content after a redirect, are not EARs. However, EARs in the other categories point to possible logic flaws.

3.5 Classification Pipeline

The goal of the classification process is to separate the security-critical EARs from the legitimate content. To do this, we applied each categorization to the content of the redirect page in a precise order to improve the accuracy of each category as well as the overall efficiency. A diagram of the classification pipeline is given in Figure 2.

The order we classify EARs is critical. This order acts as a pipeline; the content of the redirect page is placed in the first category that it matches. We apply *Empty Redirect* first because it is fast and allows us to reduce the remaining HTTP redirect responses by 49%. We then separate the *Framework Redirect* and after apply *Generic Redirect,* which removes 45% of the remaining HTTP redirect reponse.

Prior to extracting the security-relevant EARs, we noticed instances of HTTP redirect response that were not leaking information. We needed at this stage of the pipeline to apply the filter of a security-relevant category: *Error Message.* These potential errors messages are often not HTML or are short, thus we apply the *Error Message* category before the *Irrelevant* category.

Then, after the *Irrelevant* category, we filter the *HTTPS* responses. Disclosing information on a clear channel through the redirection can allow an eavesdropper to redirect the traffic from a login form running over SSL to a clear channel. The latter category is applied before the *Near Duplicate* category because *HTTPS* redirect response are frequently similar to the target body content, therefore the classification must occur before the *Near Duplicate* category.

After *Near Duplicate,* there are two more categories that are similar but each attempts to capture a different aspect of the HTTP redirect response content. The *Pre-Login Access* filter searches for information leakage, characterized by links within the HTTP redirect response content that redirects the user to a login page. On the other hand, the *Transparent Barrier* looks for the pattern when most of the HTTP redirect response redirect to the same page. In this way, *Transparent Barrier* is more general than *Pre-Login Access,* thus, we classify *Pre-Login Access* before *Transparent Barrier.* We make a distinction between the two categories because *Pre-Login Access* indicates a lack of authorization and is thus more severe than the *Transparent Barrier* category.

3.6 Classification Thresholds

Our EAR categorization requires setting thresholds for the *Irrelevant, Near Duplicate, Pre-Login Access,* and *Transparent Barrier* categories. Changing any of these parameters alters the false positives and false negatives of the EAR categorization process. In fact, the security-critical EARs are interspersed along the classification pipeline. To determine appropriate values for these thresholds we performed experiments—using a subset of the data we collected by crawling the Web (described in Section 4.1)—in which we varied each threshold to quantify its influence on the classification of its category. We first selected a random day of crawling for the classification subset, and then we manually categorized every content of the redirect page in the subset. Finally, for each category's threshold, we varied the threshold to understand how it affected the categorization.

Figure 3 shows a ROC curve for each of the four categories that use a threshold. A cross on the graph denotes where we chose the threshold. The rest of this section describes in more detail the threshold selection process for each categorization component's threshold.

3.6.1 Irrelevant.

The *Irrelevant* categorization component analyzes how much information the HTTP redirect response content contains, based on how much non-HTML content is present. The HTTP redirect response content will match this category if the content has less than the threshold number of non-HTML–markup words. We ran ten experiments, ranging the threshold from 10 to 100 words.

Figure 3(a) shows the ROC curve for the *Irrelevant* threshold. Because this category comes before the EAR categories in the pipeline, false positives are EARs that would be caught by the EAR categories; thus, false positives are costly. Therefore, we chose a value of 60 words for the *Irrelevant* threshold. As seen on the ROC curve, Figure 3(a), increasing this threshold will allow more false positives, which we are trying to minimize, and decreasing this threshold will reduce the true positive rate significantly.

3.6.2 Near Duplicate.

The *Near Duplicate* categorization component uses the Normalized Compression Distance as a threshold which ranges from 0 to 1. This threshold can be adjusted: Lower thresholds will match results that are more similar than a higher threshold.

The *Near Duplicate,* like the *Irrelevant* category, is classified in the pipeline before the EAR categories. Thus, any false positives that end up in this category are potential EARs that would be categorized in an EAR category. Therefore, we wish minimize false positives as much as possible.

Figure 3(b) shows the 10 experiments we ran, ranging the threshold from 0 to 1. As can be seen from Figure 3(b), our chosen threshold, 0.5, reduces the false positives while still classifying a good portion of similar content.

3.6.3 Pre-Login Access.

The threshold, in the *Pre-Login Access* categorization component, is the percentage of links on the page that also redirect to a

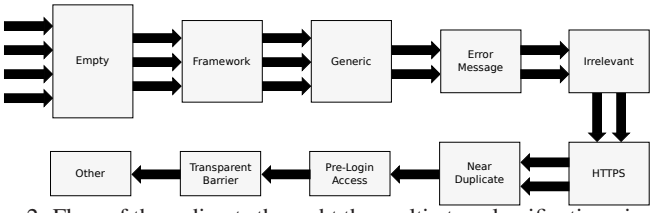
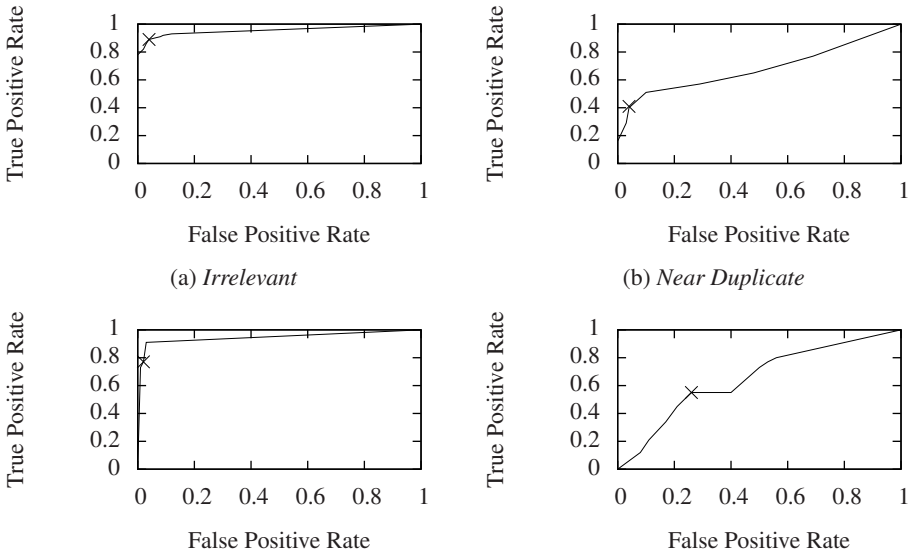


Figure 2: Flow of the redirects throught the multi-step classification pipeline.



(a) *Irrelevant* (b) *Near Duplicate*

(c) *Pre-Login Access* (d) *Transparent Barrier*

Figure 3: ROC curves of threshold experiments. A cross denotes where we chose our threshold.

login page. To test this threshold, we performed 10 experiments with different percentages.

For this category, we chose a threshold of 0.8, which is shown in Figure 3(c) as a cross. However, it appears in Figure 3(c) that there is a better threshold, which corresponds to 0.4. However, this is an unfortunate quirk of our testing data; there were only 15 *Pre-Login Access* redirect content in the testing data. The lack of data is a side-effect of the rarity of this category, in fact, our final classification found only 0.053% of these out of the total content of the redirect page. We chose 0.8 over 0.4 as the threshold because it is a tighter threshold and will allow less false positives into the *Pre-Login Access* category.

3.6.4 Transparent Barrier.

To determine the threshold for the *Transparent Barrier* categorization component, which controls how many links on an HTTP redirect response content must redirect to the same page for it to be classified as *Transparent Barrier,* we performed 10 experiments, ranging the threshold from 0 to 1. The results of the experiments are shown in Figure 3(d). We chose a value of 0.5 because increasing the threshold any further increases the false positive rate without increasing the true positive rate. One can see that after this point the ROC curve plateaus. This gives the category a false positive rate of 26% and a true positive rate of 55%.

4. EXPERIMENTAL EVALUATION

To investigate the prevalence of EARs on the Web, we performed a large-scale Web crawl looking for content sent after a redirection. We then classified the content of the redirect page according to the categories we developed.

Crawling Period	52 days
Average Speed	164,472 pages/day
Pages Downloaded	8,097,283
Total Domains	255,957
Total Redirections	1,708,771

Table 1: Statistics of a large-scale crawl of the Internet.

4.1 Crawling

We needed a web crawler to find a large number of content of redirect page on which to apply our black-box EAR detection. We required two properties: (1) the ability to store the content of the redirect page and (2) to execute JavaScript. To find such a crawler, we looked at available open-source projects, such as Nutch [4] and Heritrix [5]. Unfortunately, both did not index the redirection body's content and did not execute JavaScript. To the best of our knowledge, there was no open-source large-scale crawler that also executed JavaScript.

Due to the limitations of available crawlers, we developed our own large-scale, high-fidelity web crawler. We discover more URLs while executing JavaScript: On average, we find about 5% more links with JavaScript execution, which increases the search space for EARs.

Table 1 shows the size of the dataset collected in 52 days of crawling. We downloaded 8,097,283 pages from 255,957 distinct domains. Of these pages, 1,708,771, or 21.11% were HTTP redirect response.

Table 2 shows the results of our classification system on the 1,708,771 redirect pages. The results are shown by category, sep-

[4] `http://nutch.apache.org/`

[5] `http://sourceforge.net/projects/archive-crawler/`

Code	Empty	Framework	Generic	Irrelevant	N. Duplicate	Error	HTTPS	Pre-Login Access	T. Barrier	Other
301	337,089	4,487	312,719	22,014	1,524	82	228	137	1,083	2,843
302	477,757	47,350	450,002	30,290	798	340	596	766	833	792
303	14,284	4	2,353	367	1	23	0	1	1	7
Total	829,130	51,841	765,074	52,671	2,323	445	824	904	1,917	3,642
						Security-Critical EARs = 2,173			Benign EARs = 5,559	
	Total Legitimate = 1,701,039					Total EARs = 7,732				

Table 2: Classification of 1,708,771 redirect content that was collected over a period of 52 days.

```
You have an error in your SQL syntax; check the manual
    that corresponds to your
MySQL server version for the right syntax to use near ''
    at line 4<br />
  SELECT b.watch AS watch,b.friendtypeid AS friendtypeid,
  minutesago(u.logdate) AS age, u.username AS username,
  b.friendid AS userid, u.logdate AS logdate
  FROM friends b
  LEFT JOIN users u ON u.userid = b.friendid
  WHERE b.userid =
```

Listing 3: `302 Found` response body content leaking a SQL request adapted from a real EAR that our crawler found.

arated by HTTP response code, followed by the totals. Of the 1,708,771 HTTP redirect response we discovered in the wild, 40% are `301`, 59% are `302`, and 1% are `303`. A significant portion of the HTTP redirect response are concentrated in the *Empty* and *Generic* categories, as expected. However, there is a large amount of content of redirect page in the *Irrelevant* category with a total of 52,671 HTTP redirect response. Our classification system found 2,173 security-critical EARs spread out over 416 distinct domains; this result demonstrates that EARs are a serious problem that plagues real-world web applications.

To get a handle on how effective our heuristics are, we evaluate the false positives of our classification approach. To evaluate the false positives, we randomly sampled 100 HTTP redirect responses from each security-critical EAR category and manually verified them.

Table 3 shows the results from the manual evaluation of the random sampling. The *HTTPS* and *Error Message* categories are accurate with 3% and 0% of false positives, respectively. The false positives for the *Error Message* category is so low because we are searching for specific error messages that are unlikely to appear in legitimate pages. The *HTTPS* category still contains unfiltered generic or irrelevant content. Because our heuristic to detect generic content is not complete, we have false positives that display information for the user.

The *Pre-Login Access* category has a false positive rate of 13%. The uniqueness of each web application remains the main cause of these false positives. *Irrelevant* responses end up in the *Pre-Login Access* category because they had sufficient non-HTML words to bypass the *Irrelevant* threshold, but did not leak any information. Finally, the *Pre-Login Access* category suffers from the problem of a login form being present on every page in a web application.

In an effort to improve the security of the web sites in our crawl, we contacted 50 web sites, via email or contact form, that contained security-critical EAR vulnerabilities that our crawl discovered. Five developers replied to our vulnerability notification. Of these, two developers fixed the EAR vulnerability and three thanked us for the notification and said that they would look into the vulnerability. We believe this shows that this is a vulnerability that is misunderstood by developers yet they believe that it is a serious enough vulnerability to fix.

4.2 Interpretation

As previously mentioned, the content of redirect page is diverse. We saw a variety of content: non-sensitive content, page duplication, half-formed HTML pages, just JavaScript (no HTML) pages, navigation bars with no content, and unidentifiable (junk) content.

Category	Percentage False Positive
HTTPS	3%
Error Message	0%
Pre-Login Access	13%

Table 3: Percentage of false positives calculated by manual analysis of a random sampling of 100 redirects of each security-critical EAR category.

A confusing type of content of redirect page we found were empty pages that contain only white spaces and tabulations. We cannot definitively determine what is causing this, but we believe that some code execution happened on the server because the indentation appears to be source code. Here, without additional information from the developer we cannot attest if it is a bug.

Next, we discuss the content that ended up in each security-critical EAR category.

4.2.1 *HTTPS Redirect.*

The web application developer wants to secure data transmission when she sends information over SSL. However, due to an EAR, this private information is sent in the clear for content of redirect page that was classified as *HTTPS Redirect.* Here, the privacy and confidentiality of the data is compromised if an attacker can eavesdrop on the conversation.

Two particular, security-critical, examples stand out from our dataset. The first is an HTTP redirect response that we analyzed from a web application selling financial trading services. This response gives access to a (private) information request form in the clear. Although we did not try—due to ethical considerations—we suspect that the web application would allow us, as non-authenticated users, to submit the form and request the information.

The second example HTTP redirect response is similar to the *Pre-Login Access* category: The content of the redirect page leaks a film's review form that is only available to registered users on a popular film database. However, the application sends us this form in the content of the redirect page.

4.2.2 *Error Message.*

The *Error Message* category outputs sensitive information about the technology used by the web application. We found in this category Java stack traces, SQL errors, and PHP source code. Listing 3 shows a SQL error EAR that we adapted from one found in the wild. These types of information leakage ease the work of an attacker who is attempting to exploit the web application thanks to information about local paths or SQL query syntax.

4.2.3 *Pre-Login Access.*

Inside the *Pre-Login Access* category, the most common type of EAR was the one where content of the page was displayed even though we were not authenticated. The web application detects our lack of access rights, and it redirects us to a login page. The application is aware that we are not authenticated (as evidenced by the redirection) but still outputs the content of the web page. Listing 2 shows a canonical example from this category. This web

application sells sports statistics. Our crawler discovered an EAR that lets anyone access the private content.

In a similar fashion, we found an EAR on a social networking web application where anyone can access a user's profile inside the content of the redirect page, thus violating that user's privacy. A conclusion that can be drawn from this type of EAR is that it indicates that the authentication mechanism of the web application is flawed. The harm is specific to the web application and can affect the web application's owner or the user as well.

4.2.4 *Transparent Barrier.*

Even though *Transparent Barrier* is a category that contains benign EARs—that is, there is HTTP redirect response content and server-side execution, however the information leaked is not security-critical—the results of this category surprised us. Often, the body of the HTTP redirect response is outdated versions of the web application. Instead of removing the content, developers set a redirection but keep the old web application version. It is clear that the web developers do not want the content to be accessible, however, the old content is leaked in the response.

Another example is a domain name that has been sold. The domain then redirects, via a `301` response code, to a different domain. We believe this is done for SEO purposes, as the `301` is permanent and transfers the so called "Google juice," or search engine importance, to the new domain. However, the former domain still runs the code of the old web application and sends the old web application's output in the content the redirect page. We also saw instances where the barrier required a user to select a country, supply the user's age, or accept Terms of Use while still sending the data to which the barrier prevents the access.

4.2.5 *Other.*

The *Other* category was the last stage of the pipeline; any content that did not match one of the other categories landed here. This category contains diverse content, however we will focus on two types: legitimate HTTP redirect responses and security-critical EARs.

Legitimate content is, for example, an HTTP redirect response that displays enough words to bypass the *Irrelevant* category's threshold and does not match a regular expression of the *Generic* category. This misclassification is due to the fact that our regular expressions are focused on English language content, while the Internet is international. We could improve this by creating regular expressions that target a diverse set of languages.

The security-critical EARs were not classified in the proper category for one of two reasons. They were either application-specific error messages, where the error message was too specific to match the more generic regular expressions of the *Error Message* category. The other security-critical EARs that were in *Other* are those that should have been in *Pre-Login Access,* however, the HTTP redirect responses did not have enough links leading to a login page to match the *Pre-Login Access* threshold.

We do not consider the *Other* category to be security-critical because we were unable to broadly categorize the security relevance of these EARs. However, because legitimate content was already classified earlier in the pipeline by the *Empty, Framework, Generic, Irrelevant,* and *Near Duplicate* categories, the content in the *Other* category has a large percentage of potentially security-critical EARs. In fact, a random sampling of the *Other* category found 59% security-critical EARs. This result shows that the *Other* category can be a valuable source of potentially vulnerable EARs.

4.3 Limitations

Due to the black-box approach we took to detecting Execution After Redirect vulnerabilities, we are limited to discovering explicit EARs only. Our approach does not attempt to infer the web application's state, and, therefore, we only find EARs that leak information in the content of the redirect page.

Our classification method uses heuristics that were built by manually analyzing the initial data collected, and, as a result, our approach is prone to both false positives and false negatives. The main reason for false negatives is the diversity of content of redirect page; however, we believe that our heuristics capture the essential semantics of information leakage EARs. False positives will occur because each web application is different.

False positives that appear in all EAR categories are the typical redirect message whose body is in a non-English language. Even if our regular expressions could still capture most of the generic content, every web application developer has the ability to customize the content sent in the HTTP redirect body. Thus, these legitimate HTTP redirects are misclassified and can land in *HTTPS, Pre-Login Access, Transparent Barrier,* and *Other,* depending on the features present in the web application. Nonetheless, they are more frequent inside the default category *Other.*

Misclassified *Error Message* responses are found in *Irrelevant* and *Other* categories because we are not exhaustive in the search of these errors. False negatives of the *Pre-Login Access* category can be found in *HTTPS* and *Other*. The *HTTPS* misclassification is due to the order of our pipeline: *HTTPS* categorization is done before *Pre-Login Access. Pre-Login Access* redirect content can end up in *Other* due to the threshold.

Even though our approach has both false positives and false negatives, the categorization process significantly reduces the amount of content of redirect page that an analyst must review in order to discover security-critical EARs.

Because we built the heuristics after in-depth manual analysis of a subset of the crawling data, our categories are biased towards trends we saw in this subset. It is possible that the frequency of these categories is not representative of the whole Web. However, we constructed the categories to be broad enough to enclose essential features of the security-critical EARs.

5. RELATED WORK

The work presented in this paper is at the intersection of multiple topics of security research: vulnerability discovery, black-box vulnerability detection, and logic flaws.

There have been many approaches to automatically detect security vulnerabilities in web applications. Most approaches track the information flow through the application, using static or dynamic analysis, and offer an overview of how secure the application is regarding a known vulnerability. Many approaches attempt to discover vulnerabilities via white-box analysis of the web application's source code [1, 3, 5, 10, 17–19, 22, 23]. Among the white-box approaches, our work is closely related to Doupé et al. work which detected EARs in Ruby on Rails web applications via static analysis of the source code [10].

We use a black-box approach to detect EARs in web applications. Another approach to detecting vulnerabilities in web applications is using a black-box web vulnerability scanner, which actively attempts to exploit the web application to discover vulnerabilities. Huang et al. developed one of the first tools for black-box analysis of security vulnerabilities [16]. Other tools were developed to improve black-box web vulnerability scanners [2, 15, 20], and attempts were made to evaluate the capabilities of open-source and

commercial black-box web vulnerability scanners [4, 9, 14, 24, 25]. Unlike black-box web vulnerability scanners, we did not fuzz the web applications, instead, we passively analyzed web applications to understand the prevalence of EARs on the Internet.

Because EARs are a logic flaw in a web application, our work is related to research on logic flaws. Felmetsger et al. developed a white-box tool to assess logic flaws in web applications [11]. They were able to find an EAR in GIMS, an HR web application, through dynamic analysis and symbolic model checking. Wang et al. analyzed logic flaws in how web applications use Cashier-as-a-Service APIs [26]. Li and Xue used a gray-box approach to detect state violation attacks against web applications [21], while Sun et al. used static analysis to detect the same type of vulnerability in PHP web applications [23].

In comparison, our approach is focused on vulnerability discovery and is intended to find a specific logic flaw through a black-box approach. However, to our knowledge, no other research has been done to understand the prevalence of EARs on the Internet. Without being intrusive, from the point of view of the web application, we managed to discover vulnerable EARs on the Internet.

6. CONCLUSION

We have shown, using a novel approach, that Execution After Redirect vulnerabilities, despite being relatively obscure, are widespread on the Internet. These vulnerable web applications are leaking sensitive information to anyone who asks for it. And, because we used a passive approach, it may be possible to leverage deeper interaction with the web application while crawling to discover even more EARs. We hope that this work raises awareness of this prevalent security vulnerability.

7. REFERENCES

[1] An, J., Chaudhuri, A., Foster, J.: Static Typing for Ruby on Rails. In: Proceedings of the 24th IEEE/ACM Conference on Automated Software Engineering (ASE'09). pp. 590–594. IEEE (2009)

[2] Balduzzi, M., Gimenez, C., Balzarotti, D., Kirda, E.: Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications. In: Proceedings of the 18th Network and Distributed System Security Symposium (2011)

[3] Balzarotti, D., Cova, M., Felmetsger, V.V., Vigna, G.: Multi-module vulnerability analysis of web-based applications. In: Proceedings of the 14th ACM conference on Computer and communications security. ACM (2007)

[4] Bau, J., Bursztein, E., Gupta, D., Mitchell, J.: State of the Art: Automated Black-Box Web Application Vulnerability Testing. In: Security and Privacy (SP), 2010 IEEE Symposium on. pp. 332–345. IEEE (2010)

[5] Chaudhuri, A., Foster, J.: Symbolic Security Analysis of Ruby-on-Rails Web Applications. In: Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10). pp. 585–594. ACM (2010)

[6] Christey, S., Martin, R.A.: Vulnerability Type Distribution in CVE. Tech. rep., MITRE Corporation (May 2007)

[7] Cilibrasi, R., Vitanyi, P.: Clustering by Compression. Information Theory, IEEE Transactions on 51(4), 1523 – 1545 (april 2005)

[8] Cova, M., Balzarotti, D., Felmetsger, V., Vigna, G.: Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In: Proceedings of the 10th international conference on Recent advances in intrusion detection. vol. 4637. Springer (2007)

[9] Doupé, A., Cova, M., Vigna, G.: Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. In: Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA). Bonn, Germany (July 2010)

[10] Doupé, A., Boe, B., Kruegel, C., Vigna, G.: Fear the EAR: Discovering and Mitigating Execution After Redirect Vulnerabilities. In: Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011). Chicago, IL (October 2011)

[11] Felmetsger, V., Cavedon, L., Kruegel, C., Vigna, G.: Toward Automated Detection of Logic Vulnerabilities in Web Applications. In: Proceedings of the USENIX Security Symposium. Washington, DC (August 2010)

[12] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1 (June 1999), `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`

[13] Fossi, M., Egan, G., Haley, K., Johnson, E., Mack, T., Adams, T., Blackbird, J., King Low, M., Mazurek, D., McKinney, D., Paul, W.: Internet security threat report. Tech. rep., Symantec Corp (April 2011), vol. 16

[14] Grossman, J.: Challenges of Automated Web Application Scanning. Blackhat Windows 2004 (2004)

[15] Halfond, W., Choudhary, S., Orso, A.: Penetration testing with improved input vector identification. In: Software Testing Verification and Validation, 2009. ICST'09. International Conference on. pp. 346–355. IEEE (2009)

[16] Huang, Y.W., Huang, S.K., Lin, T.P., Tsai, C.H.: Web Application Security Assessment by Fault Injection and Behavior Monitoring. In: Proceedings of the 12th international conference on World Wide Web. pp. 148–159. WWW '03, ACM, New York, NY, USA (2003)

[17] Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.: Securing Web Application Code by Static Analysis and Runtime Protection. In: Proceedings of the 13th international conference on World Wide Web. pp. 40–52. WWW '04, ACM, New York, NY, USA (2004)

[18] Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In: Proceedings of the 2006 IEEE Symposium on Security and Privacy. pp. 258–263 (2006)

[19] Jovanovic, N., Kruegel, C., Kirda, E.: Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In: Proceedings of the 2006 workshop on Programming languages and analysis for security. ACM (2006)

[20] Kals, S., Kirda, E., Kruegel, C., Jovanovic, N.: Secubat: a Web Vulnerability Scanner. In: Proceedings of the 15th international conference on World Wide Web. pp. 247–256. ACM (2006)

[21] Li, X., Xue, Y.: BLOCK: A Black-box Approach for Detection of State Violation Attacks Towards Web Applications. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC 2011). Orlando, FL

[22] Livshits, B., Lam, M.: Finding Security Vulnerabilities in Java Applications with Static Analysis. In: Proceedings of the 14th conference on USENIX Security Symposium - Volume 14. USENIX Association (2005)

[23] Sun, F., Xu, L., Su, Z.: Static Detection of Access Control Vulnerabilities in Web Applications. In: Proceedings of the 20th USENIX conference on Security. pp. 11–11. SEC'11, USENIX Association, Berkeley, CA, USA (2011)

[24] Suto, L.: Analyzing the Accuracy and Time Costs of Web Application Security Scanners (2010)

[25] Vieira, M., Antunes, N., Madeira, H.: Using Web Security Scanners to Detect Vulnerabilities in Web Services. In: Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on. pp. 566–571. IEEE (2009)

[26] Wang, R., Chen, S., Wang, X., Qadeer, S.: How to Shop for Free Online: Security Analysis of Cashier-as-a-Service Based Web Stores. In: Security and Privacy (SP), 2011 IEEE Symposium on. pp. 465 –480 (may 2011)

[27] Williams, J., Wichers, D.: Owasp top 10 - 2010 the ten most critical web application security risks. Tech. rep., The OWASP Community (2010)