Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications

Marco Balduzzi*, Carmen Torrano Gimenez ‡, Davide Balzarotti*, and Engin Kirda* §

* Institute Eurecom, Sophia Antipolis
{balduzzi,balzarotti,kirda}@eurecom.fr

‡ Spanish National Research Council, Madrid
carmen.torrano@iec.csic.es

§ Northeastern University, Boston
ek@ccs.neu.edu

Abstract

In the last twenty years, web applications have grown from simple, static pages to complex, full-fledged dynamic applications. Typically, these applications are built using heterogeneous technologies and consist of code that runs both on the client and on the server. Even simple web applications today may accept and process hundreds of different HTTP parameters to be able to provide users with interactive services. While injection vulnerabilities such as SQL injection and cross-site scripting are well-known and have been intensively studied by the research community, a new class of injection vulnerabilities called HTTP Parameter Pollution (HPP) has not received as much attention. If a web application does not properly sanitize the user input for parameter delimiters, exploiting an HPP vulnerability, an attacker can compromise the logic of the application to perform either client-side or server-side attacks.

In this paper, we present the first automated approach for the discovery of HTTP Parameter Pollution vulnerabilities in web applications. Using our prototype implementation called PAPAS (PArameter Pollution Analysis System), we conducted a large-scale analysis of more than 5,000 popular websites. Our experimental results show that about 30% of the websites that we analyzed contain vulnerable parameters and that 46.8% of the vulnerabilities we discovered (i.e., 14% of the total websites) can be exploited via HPP attacks. The fact that PAPAS was able to find vulnerabilities in many high-profile, well-known websites suggests that many developers are not aware of the HPP problem. We informed a number of major websites about the vulnerabilities we identified, and our findings were confirmed.

1 Introduction

In the last twenty years, web applications have grown from simple, static pages to complex, full-fledged dynamic applications. Typically, these applications are built using heterogeneous technologies and consist of code that runs on the client (e.g., Javascript) and code that runs on the server (e.g., Java servlets). Even simple web applications today may accept and process hundreds of different HTTP parameters to be able to provide users with rich, interactive services. As a result, dynamic web applications may contain a wide range of input validation vulnerabilities such as cross site scripting (e.g., [4, 5, 34]) and SQL injection (e.g., [15, 17]).

Unfortunately, because of their high popularity and a user base that consists of millions of Internet users, web applications have become prime targets for attackers. In fact, according to SANS [19], attacks against web applications constitute more than 60% of the total attack attempts observed on the Internet. While flaws such as SQL injection and cross-site scripting may be used by attackers to steal sensitive information from application databases and to launch authentic-looking phishing attacks on vulnerable servers, many web applications are being exploited to convert trusted websites into malicious servers serving content that contains client-side exploits. According to SANS, most website owners fail to scan their application for common flaws. In contrast, from the attacker's point of view, automated tools, designed to target specific web application vulnerabilities simplify the discovery and infection of several thousand websites.

While injection vulnerabilities such as SQL injection and cross-site scripting are well-known and have been intensively studied, a new class of injection vulnerabilities called HTTP Parameter Pollution (HPP) has not received as much attention. HPP was first presented in 2009 by di Paola and Carettoni at the OWASP conference [27]. HPP attacks consist of injecting encoded query string delimiters into other existing parameters. If a web application does not properly sanitize the user input, a malicious user can compromise the logic of the application to perform either client-side or server-side attacks. One consequence of HPP attacks is that the attacker can potentially override existing hard-coded HTTP parameters to modify the behavior of an application, bypass input validation checkpoints, and access and possibly exploit variables that may be out of direct reach.

In this paper, we present the first automated approach for the discovery of HTTP Parameter Pollution vulnerabilities in web applications. Our prototype implementation, that we call PArameter Pollution Analysis System (PAPAS), uses a black-box scanning technique to inject parameters into web applications and analyze the generated output to identify HPP vulnerabilities. We have designed a novel approach and a set of heuristics to determine if the injected parameters are not sanitized correctly by the web application under analysis.

To the best of our knowledge, no tools have been presented to date for the detection of HPP vulnerabilities in web applications, and no studies have been published on the topic. At the time of the writing of this paper, the most effective means of discovering HPP vulnerabilities in websites is via manual inspection. At the same time, it is unclear how common and significant a threat HPP vulnerabilities are in existing web applications.

In order to show the feasibility of our approach, we used PAPAS to conduct a large-scale analysis of more than 5,000 popular websites. Our experimental results demonstrate that there is reason for concern as about 30% of the websites that we analyzed contained vulnerable parameters. Furthermore, we verified that 14% of the websites could be exploited via client-side HPP attacks. The fact that PAPAS was able to find vulnerabilities in many high-profile, well-known websites such as Google, Paypal, Symantec, and Microsoft suggests that many developers are not aware of the HPP problem.

When we were able to obtain contact information, we informed the vulnerable websites of the vulnerabilities we discovered. In the cases where the security officers of the concerned websites wrote back to us, our findings were confirmed.

We have created an online service based on PAPAS[1] (currently in beta version) that allows website maintainers to scan their sites. As proof of ownership of a site, the website owner is given a dynamically-generated token that she can put in the document root of her website.

[1]The PAPAS service is available at: `http://papas.iseclab.org`

In summary, the paper makes the following contributions:

- We present the first automated approach for the detection of HPP vulnerabilities in web applications. Our approach consists of a component to inject parameters into web applications and a set of tests and heuristics to determine if the pages that are generated contain HPP vulnerabilities.
- We describe the architecture and implementation of the prototype of our approach that we call PAPAS (PArameter Pollution Analysis System). PAPAS is able to crawl websites and generate a list of HPP vulnerable URLs.
- We present and discuss the large-scale, real-world experiments we conducted with more than 5,000 popular websites. Our experiments show that HPP vulnerabilities are prevalent on the web and that many well-known, major websites are affected. We verified that at least 46.8% of the vulnerabilities we discovered could be exploited on the client-side. Our empirical results suggest that, just like in the early days of cross site scripting and cross site request forgery [1], many developers are not aware of the HPP problem, or that they do not take it seriously.

The paper is structured as follows: The next section give an explanation of parameter pollution attacks and provides examples. Section 3 describes our approach and presents the main components of PAPAS. Section 4 presents and discusses the evaluation of PAPAS. Section 5 lists related work, and Section 6 briefly concludes the paper.

2 HTTP Parameter Pollution Attacks

HTTP Parameter Pollution attacks (HPP) have only recently been presented and discussed [27], and have not received much attention so far. An HPP vulnerability allows an attacker to inject a parameter inside the URLs generated by a web application. The consequences of the attack depend on the application's logic, and may vary from a simple annoyance to a complete corruption of the application's behavior. Because this class of web vulnerability is not widely known and well-understood yet, in this section, we first explain and discuss the problem.

Even though injecting a new parameter can sometimes be enough to exploit an application, the attacker is usually more interested in *overriding* the value of an already existing parameter. This can be achieved by "masking" the old parameter by introducing a new one with the same name.

For this to be possible, it is necessary for the web application to "misbehave" in the presence of duplicated parameters, a problem that is often erroneously confused with the HPP vulnerability itself. However, since parameter pollution attacks often rely on duplicated parameters in practice, we decided to study the parameter duplication behavior of applications, and measure it in our experiments.

2.1 Parameter Precedence in Web Applications

During the interaction with a web application, the client often needs to provide input to the program that generates the requested web page (e.g., a PHP or a Perl script). The HTTP protocol [12] allows the user's browser to transfer information inside the URI itself (i.e., GET parameters), in the HTTP headers (e.g., in the Cookie field), or inside the request body (i.e., POST parameters). The adopted technique depends on the application and on the type and amount of data that has to be transferred.

For the sake of simplicity, in the following, we focus on GET parameters. However, note that HPP attacks can be launched against any other input vector that may contain parameters controlled by the user.

RFC 3986 [7] specifies that the query component (or query string) of a URI is the part between the "`?`" character and the end of the URI (or the character "`#`"). The query string is passed unmodified to the application, and consists of one or more `field=value` pairs, separated by either an ampersand or a semicolon character. For example, the URI `http://host/path/somepage.pl?name=john&age=32` invokes the `verify.pl` script, passing the values `john` for the `name` parameter and the value 32 for the `age` parameter. To avoid conflicts, any special characters (such as the question mark) inside a parameter value must be encoded in its `%FF` hexadecimal form.

This standard technique for passing parameters is straightforward and is generally well-understood by web developers. However, the way in which the query string is processed to extract the single values depends on the application, the technology, and the development language that is used.

For example, consider a web page that contains a check-box that allows the user to select one or more options in a form. In a typical implementation, all the check-box items share the same name, and, therefore, the browser will send a separate homonym parameter for each item selected by the user. To support this functionality, most of the programming languages used to develop web applications provide methods for retrieving the complete list of values associated with a certain parameter. For example, the JSP `getParameterValues` method groups all the values together, and returns them as a list of strings. For the languages that do not support this functionality, the developer has to manually parse the query string to extract each single value.

However, the problem arises when the developer expects to receive a single item and, therefore, invokes methods (such as `getParameter` in JSP) that only return a single value. In this case, if more than one parameter with the same name is present in the query string, the one that is returned can either be the first, the last, or a combination of all the values. Since there is no standard behavior in this situation, the exact result depends on the combination of the programming language that is used, and the web server that is being deployed. Table 1 shows examples of the parameter precedence adopted by different web technologies.

Note that the fact that only one value is returned is not a vulnerability per se. However, if the developer is not aware of the problem, the presence of duplicated parameters can produce an anomalous behavior in the application that can be potentially exploited by an attacker in combination with other attacks. In fact, as we explain in the next section, this is often used in conjunction with HPP vulnerabilities to override hard-coded parameter values in the application's links.

2.2 Parameter Pollution

An HTTP Parameter Pollution (HPP) attack occurs when a malicious parameter P_{inj}, preceded by an encoded query string delimiter, is injected into an existing parameter P_{host}. If P_{host} is not properly sanitized by the application and its value is later decoded and used to generate a URL A, the attacker is able to add one or more new parameters to A.

The typical client-side scenario consists of persuading a victim to visit a malicious URL that exploits the HPP vulnerability. For example, consider a web application that allows users to cast their vote on a number of different elections. The application, written in JSP, receives a single parameter, called `poll_id`, that uniquely identifies the election the user is participating in. Based on the value of the parameter, the application generates a page that includes one link for each candidate. For example, the following snippet shows an election page with two candidates where the user could cast her vote by clicking on the desired link:

```
Url: http://host/election.jsp?poll_id=4568

Link1: <a href="vote.jsp?poll_id=4568&candidate=white">
         Vote for Mr. White</a>
Link2: <a href="vote.jsp?poll_id=4568&candidate=green">
         Vote for Mrs. Green</a>
```

Suppose that Mallory, a Mrs. Green supporter, is interested in subverting the result of the online election. By analyzing the webpage, he realizes that the application does not properly sanitize the `poll_id` parameter. Hence, Mallory

Technology/Server	Tested Method	Parameter Precedence
ASP/IIS	`Request.QueryString("par")`	All (comma-delimited string)
PHP/Apache	`$_GET["par"]`	Last
JSP/Tomcat	`Request.getParameter("par")`	First
Perl(CGI)/Apache	`Param("par")`	First
Python/Apache	`getvalue("par")`	All (List)

Table 1: Parameter precedence in the presence of multiple parameters with the same name

can use the HPP vulnerability to inject another parameter of his choice. He then creates and sends to Alice the following malicious Url:

```
http://host/election.jsp?poll_id=4568%26candidate%3Dgreen
```

Note how Mallory "polluted" the `poll_id` parameter by injecting into it the `candidate=green` pair. By clicking on the link, Alice is redirected to the original election website where she can cast her vote for the election. However, since the `poll_id` parameter is URL-decoded and used by the application to construct the links, when Alice visits the page, the malicious `candidate` value is injected into the URLs[2]:

```
http://host/election.jsp?poll_id=4568%26candidate%3Dgreen

Link 1: <a href=vote.jsp?poll_id=4568&candidate=green
        &candidate=white>Vote for Mr. White</a>
Link 2: <a href=vote.jsp?poll_id=4568&candidate=green
        &candidate=green>Vote for Mrs. Green</a>
```

No matter which link Alice clicks on, the application (in this case the `vote.jsp` script) will receive two `candidate` parameters. Furthermore, the first parameter will *always* be set to `green`.

In the scenario we discussed, it is likely that the developer of the voting application expected to receive only one candidate name, and, therefore, relied on the provided basic Java functionality to retrieve a single parameter. As a consequence, as shown in Table 1, only the first value (i.e., `green`) is returned to the program, and the second value (i.e., the one carrying the Alice's actual vote) is discarded.

In summary, in the example we presented, since the voting application is vulnerable to HPP, it is possible for an attacker to forge a malicious link that, once visited, tampers with the content of the page, and returns only links that force a vote for Mrs. Green.

Cross-Channel Pollution HPP attacks can also be used to override parameters between different input channels. A good security practice when developing a web application is to accept parameters only from the input channel (e.g., GET, POST, or Cookies) where they are supposed to be supplied. That is, an application that receives data from a POST request should not accept the same parameters if they are provided inside the URL. In fact, if this safety rule is ignored, an attacker could exploit an HPP flaw to inject arbitrary parameter-value pairs into a channel A to override the legitimate parameters that are normally provided in another channel B. Obviously, for this to be possible, a necessary condition is that the web technology gives precedence to A with respect to B.

[2]URLs in the page snippets have the injected string emphasized by using a red, underlining font.

HPP to bypass CSRF tokens One interesting use of HPP attacks is to bypass the protection mechanism used to prevent cross-site request forgery. A cross-site request forgery (CRSF) is a confused deputy type of attack [16] that works by including a malicious link in a page (usually in an image tag) that points to a website in which the victim is supposed to be authenticated. The attacker places parameters into the link that are required to initiate an unauthorized action. When the victim visits the attack page, the target application receives the malicious request. Since the request comes from a legitimate user and includes the cookie associated with a valid session, the request is likely to be processed.

A common technique to protect web applications against CSRF attacks consists of using a secret request token (e.g., see [20, 25]). A unique token is generated by the application and inserted in all the sensitive links URLs. When the application receives a request, it verifies that it contains the valid token before authorizing the action. Hence, since the attacker cannot predict the value of the token, she cannot forge the malicious URL to initiate the action.

A parameter pollution vulnerability can be used to inject parameters inside the existing links generated by the application (that, therefore, include a valid secret token). With these injected parameters, it may be possible for the attacker to initiate a malicious action and bypass CSRF protection.

A CSRF bypassing attack using HPP was demonstrated in 2009 against Yahoo Mail [10]. The parameter injection permitted to bypass the token protections adopted by Yahoo to protect sensitive operations, allowing the attacker to delete all the mails of a user.

The following example demonstrates a simplified version of the Yahoo attack:

```
Url:
showFolder?fid=Inbox&order=down&tt=24&pSize=25&startMid=0
%2526cmd=fmgt.emptytrash%26DEL=1%26DelFID=Inbox%26
cmd=fmgt.delete

Link:
showMessage?sort=date&order=down&startMid=0
%26cmd%3Dfmgt.emptytrash&DEL=1&DelFID=Inbox&
cmd=fmgt.delete&.rand=1076957714
```

In the example, the link to display the mail message is protected by a secret token that is stored in the `.rand` parameter. This token prevents an attacker from including the link inside another page to launch a CSRF attack. However, by exploiting an HPP vulnerability, the attacker can still inject the malicious parameters (i.e., deleting all the mails of a user and emptying the trash can) into the legitimate page. The injection string is a concatenation of the two commands, where the second command needs to be URL-encoded twice in order to force the application to clean the trash can only after the deletion of the mails.

3 Automated HPP Vulnerability Detection with PAPAS

Our PArameter Pollution Analysis System (PAPAS) to automatically detect HPP vulnerabilities in websites consists of four main components: A browser, a crawler, and two scanners.

The first component is an instrumented browser that is responsible for fetching the webpages, rendering the content, and extracting all the links and form URLs contained in the page.

The second component is a crawler that communicates with the browser through a bidirectional channel. This channel is used by the crawler to inform the browser on the URLs that need to be visited, and on the forms that need to be submitted. Furthermore, the channel is also used to retrieve the collected information from the browser.

Every time the crawler visits a page, it passes the extracted information to the two scanners so that it can be analyzed. The parameter Precedence Scanner (P-Scan) is responsible for determining how the page behaves when it receives two parameters with the same name. The Vulnerability Scanner (V-Scan), in contrast, is responsible for testing the page to determine if it is vulnerable to HPP attacks. V-Scan does this by attempting to inject a new parameter inside one of the existing ones and analyzing the output. The two scanners also communicate with the instrumented browser in order to execute the tests.

All the collected information is stored in a database that is later analyzed by a statistics component that groups together information about the analyzed pages, and generates a report for the vulnerable URLs.

The general architecture of the system is summarized in Figure 1. In the following, we describe the approach that is used to detect HPP vulnerabilities and each component in more detail.

3.1 Browser and Crawler Components

Whenever the crawler issues a command such as the visiting of a new webpage, the instrumented browser in PAPAS first waits until the target page is loaded. After the browser is finished parsing the DOM, executing the client-side scripts, and loading additional resources, a browser extension (i.e., plugin) extracts the content, the list of links, and the forms in the page.

In order to increase the depth that a website can be scanned with, the instrumented browser in PAPAS uses a number of simple heuristics to automatically fill forms (similarly to previously proposed scanning solutions such as [24]). For example, random alphanumeric values of 8 characters are inserted into `password` fields and a default e-mail address is inserted into fields with the name `email`, `e-mail`, or `mail`.

For sites where the authentication or the provided inputs fail (e.g., because of the use of CAPTCHAs), the crawler can be assisted by manually logging into the application using the browser, and then specifying a regular expression to be used to prevent the crawler from visiting the log-out page (e.g., by excluding links that include the `cmd=logout` parameter).

3.2 P-Scan: Analysis of the Parameter Precedence

The P-Scan component analyzes a page to determine the precedence of parameters if multiple occurrences of the same parameter are injected into an application. For URLs that contain several parameters, each one is analyzed until the page's precedence has been determined or all available parameters have been tested.

The algorithm we use to test the precedence of parameters starts by taking the first parameter of the URL (in the form `par1=val1`), and generates a new parameter value `val2` that is similar to the existing one. The idea is to generate a value that would be accepted as being valid by the application. For example, a parameter that represents a page number cannot be replaced with a string. Hence, a number is cloned into a consecutive number, and a string is cloned into a same-length string with the first two characters modified.

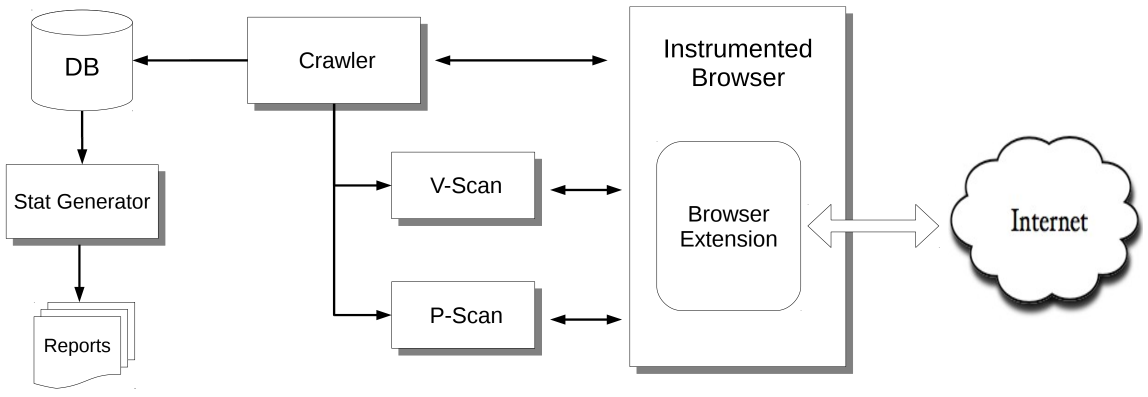

Figure 1: Architecture of PAPAS

In a second step, the scanner asks the browser to generate two new requests. The first request contains only the newly generated value `val2`. In contrast, the second request contains two copies of the parameter, one with the original value `val1`, and one with the value `val2`.

Suppose, for example, that a page accepts two parameters $par1$ and $par2$. In the first iteration, the first parameter is tested for the precedence behavior. That is, a new value new_val is generated and two requests are issued. In sum, the parameter precedence test is run on that pages that are the results of the three following requests:

```
Page0 - Original Url: application.php?
                        par1=val1&par2=val2
Page1 - Request 1:    application.php?
                        par1=new_val&par2=val2
Page2 - Request 2:    application.php?
                        par1=val1&par1=new_val&par2=val2
```

A naive approach to determine the parameter precedence would be to simply compare the three pages returned by the previous requests: If `Page1 == Page2`, then the second (last) parameter would have precedence over the first. If, however, `Page2 == Page0`, the application is giving precedence to the first parameter over the second.

Unfortunately, this straightforward approach does not work well in practice. Modern web applications are very complex, and often include dynamic content that may still vary even when the page is accessed with exactly the same parameters. Publicity banners, RSS feeds, real-time statistics, gadgets, and suggestion boxes are only a few examples of the dynamic content that can be present in a page and that may change each time the page is accessed.

The P-Scan component resolves the dynamic content problem in two stages. First, it pre-processes the page and tries to eliminate all dynamic content that does not depend on the values of the application parameters. That is, P-Scan removes HTML comments, images, embedded contents, interactive objects (e.g., Java applets), CSS stylesheets, cross-domain iFrames, and client-side scripts. It also uses regular expressions to identify and remove "timers" that are often used to report how long it takes to generate the page that is being accessed. In a similar way, all the date and time strings on the page are removed.

The last part of the sanitization step consists of removing all the URLs that reference the page itself. The problem is that as it is very common for form actions to submit data to the same page, when the parameters of a page are modified, the self-referencing URLs also change accordingly. Hence, to cope with this problem, we also eliminate these URLs.

After the pages have been stripped off their dynamic components, P-Scan compares them to determine the precedence of the parameters. Let `P0′, P1′`, and `P2′` be the sanitized versions of `Page0`, `Page1`, and `Page2`. The comparison procedure consists of five different tests that are applied until one of the tests succeeds:

I. Identity Test - The identity test checks whether the parameter under analysis has any impact on the content of the page. In fact, it is very common for query strings to contain many parameters that only affect the internal state, or some "invisible" logic of the application. Hence, if `P0′ == P1′ == P2′`, the parameter is considered to be ineffective.

II. Base Test - The base test is based on the assumption that the dynamic component stripping process is able to perfectly remove all dynamic components from the page that is under analysis. If this is the case, the second (last) parameter has precedence over the first if `P1′==P2′`. The situation is the opposite if `P2′ == P0′`. Note that despite our efforts to improve the dynamic content stripping process as much as possible, in practice, it is rarely the case that the compared pages match perfectly.

III. Join Test - The join test checks the pages for indications that show that the two values of the homonym parameters are somehow combined together by the application. For example, it searches `P2'` for two values that are separated by commas, spaces, or that are contained in the same HTML tag. If there is a positive match, the algorithm concludes that the application is merging the values of the parameters.

IV. Fuzzy Test - The fuzzy test is designed to cope with pages whose dynamic components have not been perfectly sanitized. The test aims to handle identical pages that may show minor differences because of embedded dynamic parts. The test is based on confidence intervals. We compute two values, S_{21} and S_{20}, that represent how similar `P2'` is to the pages `P1'` and `P0'` respectively. The similarity algorithm we use is based on the Ratcliff/Obershelp pattern recognition algorithm, (also known as *gestalt pattern matching* [28]), and returns a number between 0 (i.e, completely different) to 1 (i.e., perfect match). The parameter precedence detection algorithm that we use in the fuzzy test works as follows:

```
if ABS(S21-S20) > DISCRIMINATION_THRESHOLD:
  if (S21 > S20) and (S21 > SIMILARITY_THRESHOLD):
    Precedence = last
  else (S20 > S21) and (S20 > SIMILARITY_THRESHOLD):
    Precedence = first
  else:
    Unknown precedence
else:
  Unknown precedence
```

To draw a conclusion, the algorithm first checks if the two similarity values are different enough (i.e., the values show a difference that is greater than a certain *discrimination threshold*). If this is the case, the closer match (if the similarity is over a minimum *similarity threshold*) determines the parameter precedence. In other words, if the page with the duplicated parameters is very similar to the original page, there is a strong probability that the web application is only using the first parameter, and ignoring the second. However, if the similarity is closer to the page with the artificially injected parameter, there is a strong probability that the application is only accepting the second parameter.

The two threshold values have been determined by running the algorithm on one hundred random webpages that failed to pass the base test, and for which we manually determined the precedence of parameters. The two experimental thresholds (set respectively to 0.05 and 0.75) were chosen to maximize the accuracy of the detection, while minimizing the error rate.

V. Error Test - The error test checks if the application crashes, or returns an "internal" error when an identical parameter is injected multiple times. Such an error usually happens when the application does not expect to receive multiple parameters with the same name. Hence, it receives an array (or a list) of parameters instead of a single value. An error occurs if the value is later used in a function that expects a well-defined type (such as a number or a string). In this test, we search the page under analysis for strings that are associated with common error messages or exceptions. In particular, we adopted all the regular expressions that the *SqlMap project* [13] uses to identify database errors in MySQL, PostgreSQL, MS SQL Server, Microsoft Access, Oracle, DB2, and SQLite.

If none of these five tests succeed, the parameter is discarded from the analysis. This could be, for example, because of content that is generated randomly on the server-side. The parameter precedence detection algorithm is then run again on the next available parameter.

3.3 V-Scan: Testing for HPP vulnerabilities

In this section, we describe how the V-Scan component tests for the presence of HTTP Parameter Pollution vulnerabilities in web applications.

For every page that V-Scan receives from the crawler, it tries to inject a URL-encoded version of an innocuous parameter into each existing parameter of the query string. Then, for each injection, the scanner verifies the presence of the parameter in links, action fields and hidden fields of forms in the answer page.

For example, in a typical scenario, V-Scan injects the pair "`%26foo%3Dbar`" into the parameter "`par1=val1`" and then checks if the "`&foo=bar`" string is included inside the URLs of links or forms in the answer page.

Note that we do not check for the presence of the vulnerable parameter itself (e.g., by looking for the string "`par1=val1&foo=bar`"). This is because web applications sometimes use a different name for the same parameter in the URL and in the page content. Therefore, the parameter "`par1`" may appear under a different name inside the page.

In more detail, V-Scan starts by extracting the list $P_{URL} = [P_{U1}, P_{U2}, \ldots P_{Un}]$ of the parameters that are present in the page URL, and the list $P_{Body} = [P_{B1}, P_{B2}, \ldots P_{Bm}]$ of the parameters that are present in links or forms contained in the page body.
It then computes the following three sets:

- $P_A = P_{URL} \cap P_{Body}$ is the set of parameters that appear unmodified in the URL and in the links or forms of the page.

- $P_B = p \mid p \in P_{URL} \ \wedge \ p \notin P_{Body}$ contains the URL parameters that do not appear in the page. Some of these parameters may appear in the page under a different name.

- $P_C = p \mid p \notin P_{URL} \land p \in P_{Body}$ is the set of parameters that appear somewhere in the page, but that are not present in the URL.

First, V-Scan starts by injecting the new parameter in the P_A set. We observed that in practice, in the majority of the cases, the application copies the parameter to the page body and maintains the same name. Hence, there is a high probability that a vulnerability will be identified at this stage. However, if this test does not discover any vulnerability, then the scanner moves on to the second set (P_B). In the second test, the scanner tests for the (less likely) case in which the vulnerable parameter is renamed by the application. Finally, in the final test, V-Scan takes the parameters in the P_C group, attempts to add these to the URL, and use them as a vector to inject the malicious pair. This is because webpages usually accept a very large number of parameters, not all of which are normally specified in the URL. For example, imagine a case in which we observe that one of the links in the page contains a parameter "`language=en`". Suppose, however, that this parameter is not present in the page URL. In the final test, V-Scan would attempt to build a query string like "`par1=var1&language=en%26foo%3Dbar`".

Note that the last test V-Scan applies can be executed on pages with an empty query string (but with parameterized links/forms), while the first two require pages that already contain a query string.

In our prototype implementation, the V-Scan component encodes the attacker pair using the standard URL encoding schema[3]. Our experiments show that this is sufficient for discovering HPP flaws in many applications. However, there is room for improvement as in some cases, the attacker might need to use different types of encodings to be able to trigger a bug. For example, this was the case of the HPP attack against Yahoo (previously described in Section 2) where the attacker had to double URL-encode the "cleaning of the trash can" action.

Handling special cases In our experiments, we identified two special cases in which, even though our vulnerability scanner reported an alert, the page was not actually vulnerable to parameter pollution.

In the first case, one of the URL parameters (or part of it) is used as the *entire* target of a link. For example:

```
Url:  index.php?v1=p1&uri=apps%2Femail.jsp%3Fvar1%3Dpar1
                                                   %26foo%3Dbar
Link: apps/email.jsp?var1=par1&foo=bar
```

[3]URL Encoding Reference, `http://www.w3schools.com/TAGS/ref_urlencode.asp`

A parameter is used to store the URL of the target page. Hence, performing an injection in that parameter is equivalent to modifying its value to point to a different URL. Even though this technique is syntactically very similar to an HPP vulnerability, it is not a proper injection case. Therefore, we decided to consider this case as a false positive of the tool.

The second case that generates false alarms is the opposite of the first case. In some pages, the entire URL of the page becomes a parameter in one of the links. This can frequently be observed in pages that support printing or sharing functionalities. For example, imagine an application that contains a link to report a problem to the website's administrator. The link contains a parameter `page` that references the URL of the page responsible for the problem:

```
Url:  search.html?session_id=jKAmSZx5%26foo%3Dbar&q=shoes

Link: service_request.html?page=search%2ehtml%3f
      session_id%3djKAmSZx5&foo=bar&q=shoes
```

Note that by changing the URL of the page, we also change the `page` parameter contained in the link. Clearly, this is not an HPP vulnerability.

Since the two previous implementation techniques are quite common in web applications, PAPAS would erroneously report these sites as being vulnerable to HPP. To eliminate such alarms and to make PAPAS suitable for large-scale analysis, we integrated heuristics into the V-Scan component to cross-check and verify that the vulnerabilities that are identified do not correspond to these two common techniques that are used in practice.

In our prototype implementation, in order to eliminate these false alarms, V-Scan checks that the parameter in which the injection is performed does not start with a scheme specifier string (e.g., `http://`). Then, it verifies that the parameter as a whole is not used as the target for a link. Furthermore, it also checks that the entire URL is not copied as a parameter inside a link. Finally, our vulnerability analysis component double-checks each vulnerability by injecting the new parameter *without* url-encoding the separator (i.e., by injecting `&foo=bar` instead of `%26foo%3Dbar`). If the result is the same, we know that the query string is simply copied inside another URL. While such input handling is possibly a dangerous design decision on the side of the developer, there is a high probability that it is intentional so we ignore it and do not report it by default. However, such checks can be deactivated anytime if the analyst would like to perform a more in-depth analysis of the website.

3.4 Implementation

The browser component of PAPAS is implemented as a Firefox extension, while the rest of the system is written in Python. The components communicate over TCP/IP sockets.

Similar to other scanners, it would have been possible to directly retrieve web pages without rendering them in a real browser. However, such techniques have the drawback that they cannot efficiently deal with dynamic content that is often found on Web pages (e.g., Javascript). By using a real browser to render the pages we visit, we are able to analyze the page as it is supposed to appear to the user after the dynamic content has been generated. Also, note that unlike detecting cross site scripting or SQL injections, the ability to deal with dynamic content is a necessary prerequisite to be able to test for HPP vulnerabilities using a black-box approach.

The browser extension has been developed using the standard technology offered by the Mozilla development environment: a mix of Javascript and XML User Interface Language (XUL). We use XPConnect to access Firefox's XPCOM components. These components are used for invoking GET and POST requests and for communicating with the scanning component.

PAPAS supports three different operational modes: *fast mode*, *extensive mode* and *assisted mode*. The fast mode aims to rapidly test a site until potential vulnerabilities are discovered. Whenever an alert is generated, the analysis continues, but the V-Scan component is not invoked to improve the scanning speed. In the extensive mode, the entire website is tested exhaustively and all potential problems and injections are logged. The assisted mode allows the scanner to be used in an interactive way. That is, the crawler pauses and specific pages can be tested for parameter precedence and HPP vulnerabilities. The assisted mode can be used by security professionals to conduct a semi-automated assessment of a web application, or to test websites that require a particular user authentication.

PAPAS is also customizable and settings such as scanning depths, numbers of injections that are performed, waiting times between requests, and page loading timeouts are all configurable by the analyst.

3.5 Limitations

Our current implementation of PAPAS has several limitations. First, PAPAS does not support the crawling of links embedded in active content such as Flash, and therefore, is not able to visit websites that rely on active content technologies to navigate among the pages.

Second, currently, PAPAS focuses only on HPP vulnerabilities that can be exploited via client-side attacks (e.g., analogous to reflected XSS attacks) where the user needs to click on a link prepared by the attacker. Some HPP vulnerabilities can also be used to exploit server-side components (when the malicious parameter value is not included in a link but it is decoded and passed to a back-end component). However, testing for server-side attacks is more difficult than testing for client-side attacks as comparing requests and answers is not sufficient (i.e., similar to the difficulty of detecting stored SQL-injection vulnerabilities via black-box scanning). We leave the detection of server-side attacks to future work.

4 Evaluation

We evaluated our detection technique by running two experiments. In the first experiment, we used PAPAS to automatically scan a list of popular websites with the aim of measuring the prevalence of HPP vulnerabilities in the wild. We then selected a limited number of vulnerable sites and, in a second experiment, performed a more in-depth analysis of the detected vulnerabilities to gain a better understanding of the possible consequences of the vulnerabilities our tool automatically identified.

4.1 HPP Prevalence in Popular Websites

In the first experiment, we collected 5,000 unique URLs from the public database of Alexa. In particular, we extracted the top ranked sites from each of the *Alexa's categories* [3]. Each website was considered only once – even if it was present in multiple distinct categories, or with different top-level domain names such as `google.com` and `google.fr`.

The aim of our experiments was to quickly scan as many websites as possible. Our basic premise was that it would be likely that the application would contain parameter injection vulnerabilities on many pages and on a large number of parameters if the developers of the site were not aware of the HPP threat and had failed to properly sanitize the user input.

To maximize the speed of the tests, we configured the crawler to start from the homepage and visit the sub-pages up to a distance of three (i.e., three clicks away from the website's entry point). For the tests, we only considered links that contained at least one parameter. In addition, we limited the analysis to 5 instances per page (i.e., a page with the same URL, but a different query string was considered a new instance). The global timeout was set to 15 minutes per site and the browser was customized to quickly load and render the pages, and run without any user interaction. Furthermore, we disabled pop-ups, image loading, and any plug-ins for active content technologies such as Flash, or

Categories	# of Tested Applications	Categories	# of Tested Applications
Internet	698	Government	132
News	599	Social Networking	117
Shopping	460	Video	114
Games	300	Financial	110
Sports	256	Organization	106
Health	235	University	91
Science	222	Others	1401
Travel	175		

Table 2: TOP15 categories of the analyzed sites

Silverlight. An external watchdog was also configured to monitor and restart the browser in case it became unresponsive.

In 13 days of experiments, we successfully scanned 5,016 websites, corresponding to a total of 149,806 unique pages. For each page, our tool generated a variable amount of queries, depending on the number of detected parameters. The websites we tested were distributed over 97 countries and hundreds of different Alexa categories. Table 2 summarizes the 15 categories containing the higher number of tested applications.

Parameter Precedence For each website, the P-Scan component tested every page to evaluate the order in which the GET parameters were considered by the application when two occurrences of the same parameter were specified. The results were then grouped together in a per-site summary, as shown in Figure 2. The first column reports the type of parameter precedence. *Last* and *First* indicate that all the analyzed pages of the application uniformly considered the last or the first specified value. *Union* indicates that the two parameters were combined together to form a single value, usually by simply concatenating the two strings with a space or a comma. In contrast, the parameter precedence is set to *inconsistent* when different pages of the website present mismatching precedences (i.e., some pages favor the first parameter's value, others favor the last). The *inconsistent* state, accounting for a total of 25% of the analyzed applications, is usually a consequence of the fact that the website has been developed using a combination of heterogeneous technologies. For example, the main implementation language of the website may be PHP, but a few Perl scripts may still be responsible for serving certain pages.

Even though the lack of a uniform behavior can be suspicious, it is neither a sign, nor a consequence of a vulnerable application. In fact, each parameter precedence behavior (even the *inconsistent* case) is perfectly safe if the application's developers are aware of the HPP threat and know how to handle a parameter's value in the proper way. Unfortunately, as shown in the rest of the section, the results of our experiments suggest that many developers are not aware of HPP.

Figure 2 shows that for 4% of the websites we analyzed, our scanner was not been able to automatically detect the parameter precedence. This is usually due to two main reasons. The first reason is that the parameters do not affect (or only minimally affect) the rendered page. Therefore, the result of the page comparison does not reach the discrimination threshold. The second reason is the opposite of the first. That is, the page shows too many differences even after the removal of the dynamic content, and the result of the comparison falls below the similarity threshold (see Section 3.2 for the full algorithm and an explanation of the threshold values).

The scanner found 238 applications that raised an SQL error when they were tested with duplicated parameters. Quite surprisingly, almost 5% of the most popular websites on the Internet failed to properly handle the user input, and returned an "internal" error page when a perfectly-legal parameter was repeated twice. Note that providing two parameters with the same name is a common practice in many applications, and most of the programming languages provide special functionalities to access multiple values. Therefore, this test was not intended to be an attack against the applications, but only a check to verify which parameter's value was given the precedence. Nevertheless, we were surprised to note error messages from the websites of many major companies, banks and government institutions, educational sites, and others popular websites.

HPP Vulnerabilities PAPAS discovered that 1499 websites (29.88% of the total we analyzed) contained at least one page vulnerable to HTTP Parameter Injection. That is, the tool was able to automatically inject an encoded parameter inside one of the existing parameters, and was then able to verify that its URL-decoded version was included in one of the URLs (links or forms) of the resulting page.

However, the fact that it is possible to inject a parameter

Parameter Precedence	WebSites	
Last	2,237	(44.60%)
First	946	(18.86%)
Union	381	(7.60%)
Inconsistent	1,251	(24.94%)
Unknown	201	(4.00%)
Total	5,016	(100.00%)
Database Errors	238	(4.74%)

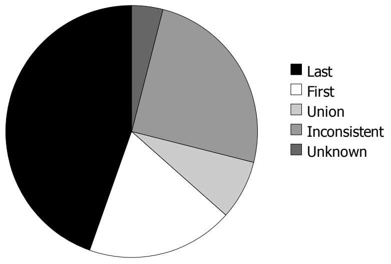

Figure 2: Precedence when the same parameter occurs multiple time

does not reveal information about the significance and the consequences of the injection. Therefore, we attempted to verify the number of *exploitable* applications (i.e., the subset of vulnerable websites in which the injected parameter could potentially be used to modify the behavior of the application).

We started by splitting the vulnerable set into two separate groups. In 872 websites (17.39%), the injection was on a link or a form's action field. In the remaining 627 cases (12.5%), the injection was on a form's hidden field.

For the first group, our tool verified if the parameter injection vulnerability could be used to override the value of one of the existing parameters in the application. This is possible only if the parameter precedence of the page is consistent with the position of the injected value. For example, if the malicious parameter is always added to the end of the URL and the first value has parameter precedence, it is impossible to override any existing parameter.

When the parameter precedence is not favorable, a vulnerable application can still be exploitable by injecting a new parameter (that differs from all the ones already present in the URL) that is accepted by the target page.

For example, consider a page `target.pl` that accepts an `action` parameter. Suppose that, on the same page, we find a page `poor.pl` vulnerable to HPP:

```
Url:  poor.pl?par1=val1%26action%3Dreset
Link: target.pl?x=y&w=z&par1=val1&action=reset
```

Since in Perl the parameter precedence is on the *first* value, it is impossible to override the `x` and `w` parameters. However, as shown in the example, the attacker can still exploit the application by injecting the `action` parameter that she knows is accepted by the `target.pl` script. Note that while the parameter overriding test was completely automated, this type of injection required a manual supervision to verify the effects of the injected parameter on the web application.

The final result was that at least 702 out of the 872 applications of the first group were exploitable. For the remaining 170 pages, we were not able, through a parameter injection, to affect the behavior of the application.

For the applications in the second group, the impact of the vulnerability is more difficult to estimate in an automated fashion. In fact, since modern browsers automatically encode all the form fields, the injected parameter will still be sent in a url-encoded form, thus making an attack ineffective.

In such a case, it may still be possible to exploit the application using a two-step attack where the malicious value is injected into the vulnerable field, it is propagated in the form submission, and it is (possibly) decoded and used in a later stage. In addition, the vulnerability could also be exploited to perform a server-side attack, as explained in Section 3.5. However, using a black-box approach, it is very difficult to automatically test the exploitability of multi-step or server-side vulnerabilities. Furthermore, server-side testing might have had ethical implications (see Section 4.3 for discussion). Therefore, we did not perform any further analysis in this direction.
To conclude, we were able to confirm that in (at least) 702 out of the 1499 vulnerable websites (i.e., 46.8%) that PAPAS identified, it would have been possible to exploit the HPP vulnerability to override one of the hard-coded parameters, or to inject another malicious parameter that would affect the behavior of the application.

Figure 3 shows the fraction of vulnerable and exploitable applications grouped by the different Alexa categories. The results are equally divided, suggesting that important financial and health institutions do not seem to be more security-aware and immune to HPP than leisure sites for sporting and gaming.

False Positives In our vulnerability detection experiments, the false positives rate was 1.12% (10 applications). All the false alarms were due to parameters that were used by the application as an entire target for one of the links. The heuristic we implemented to detect these cases (explained in Section 3.3) failed because the applications applied a transformation to the parameter before using it as a

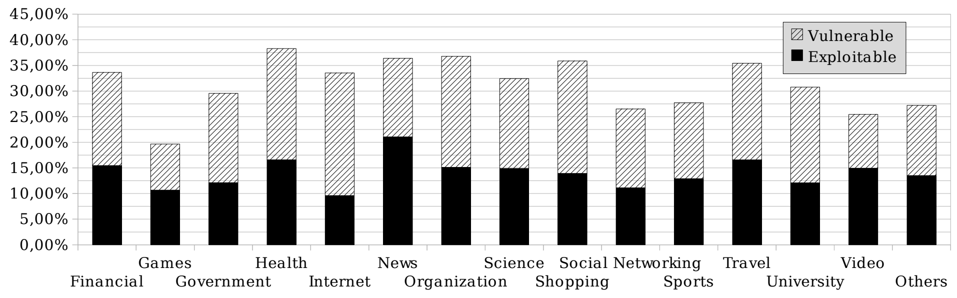

Figure 3: Vulnerability rate for category

link's URL.

Note that, to maximize efficiency, our results were obtained by crawling each website at a maximum depth of three pages. In our experiments, we observed that 11% of the vulnerable pages were directly linked from the homepage, while the remaining 89% were equally distributed between the distance of 2 and 3. This trend suggests that it is very probable that many more vulnerabilities could have been found by exploring the sites in more depth.

4.2 Examples of Discovered Vulnerabilities

Our final experiments consisted of the further analysis of some of the vulnerable websites that we identified. Our aim was to gain an insight into the real consequences of the HPP vulnerabilities we discovered.

The analysis we performed was assisted by the V-Scan component. When invoked in *extensive mode*, V-Scan was able to explore in detail the web application, enumerating all the vulnerable parameters. For some of the websites, we also registered an account and configured the scanner to test the authenticated part of the website.

HPP vulnerabilities can be abused to run a wide range of different attacks. In the rest of this section, we discuss the different classes of problems we identified in our analysis with the help of real-world examples.

The problems we identified affected many important and well-known websites such as Microsoft, Google, VMWare, About.com, Symantec, history.com, flickr, and Paypal. Since, at the time of writing, we have not yet received confirmation that all of the vulnerabilities have been fixed, we have anonymized the description of the following real-word cases.

Facebook Share Facebook, Twitter, Digg and other social networking sites offer a *share component* to easily share the content of a webpage over a user profile. Many news portals nowadays integrate these components with the intent of facilitating the distribution of their news.

By reviewing the vulnerability logs of the tested applications, we noticed that different sites allowed a parameter injection on the links referencing the share component of Facebook. In all those cases, a vulnerable parameter would allow an attacker to alter the request sent to Facebook and to trick the victim into sharing a page chosen by the attacker. For example, it was possible for an attacker to exploit these vulnerabilities to corrupt a shared link by overwriting the reference with the URL of a drive-by-download website.

In technical terms, the problem was due to the fact that it was possible to inject an extra *url-to-share* parameter that could overwrite the value of the parameter used by the application. For example:

```
Url:
<site>/shareurl.htm?PG=<default url>&zItl=<description>
               %26url-to-share%3Dhttp://www.malicious.com
Link:
http://www.facebook.com/sharer.php?
      url-to-share=<default url>&t=<description>&
      url-to-share=http://www.malicious.com
```

Even though the problem lies with the websites that use the share component, Facebook facilitated the exploitation by accepting multiple instances of the same parameter, and always considering the latest value (i.e., the one on the right).

We notified the security team of Facebook and proposed a simple solution based on the filtering of all incoming sharing requests that include duplicate parameters. The team promptly acknowledged the issue and informed us that they were willing to put in place our countermeasure.

CSRF via HPP Injection Many applications use hidden parameters to store a URL that is later used to redirect the users to an appropriate page. For example, social networks commonly use this feature to redirect new users to a page where they can look up a friend's profile.

In some of these sites, we observed that it was possible for an attacker to inject a new *redirect* parameter inside the registration or the login page so that it could override the hard-coded parameter's value. On one social-network website, we were able to inject a custom URL that had the effect of automatically sending friend requests after the login. In another site, by injecting the malicious pair into the registration form, an attacker could perform different actions on the authenticated area.

This problem is a CSRF attack that is carried out via an HPP injection. The advantages compared to a normal CSRF is that the attack URL is injected into the real login/registration page. Moreover, the user does not have to be already logged into the target website because the action is automatically executed when the user logs into the application. However, just like in normal CSRF, this attack can be prevented by using security tokens.

Shopping Carts We discovered different HPP vulnerabilities in online shopping websites that allow the attacker to tamper with the user interaction with the shopping cart component.

For example, in several shopping websites, we were able to force the application to select a particular product to be added into the user's cart. That is, when the victim checks out and would like to pay for the merchandise, she is actually paying for a product that is different from the ones she actually selected. On an Italian shopping portal, for example, it was even possible to override the ID of the product in such a way that the browser was still showing the image and the description of the original product, even when the victim was actually buying a different one.

Financial Institutions We ran PAPAS against the authenticated and non-authenticated areas of some financial websites and the tool automatically detected several HPP vulnerabilities that were potentially exploitable. Since the links involved sensitive operations (such as increasing account limits and manipulating credit card operations), we immediately stopped our experiments and promptly informed the security departments of the involved companies. The problems were acknowledged and are currently being fixed.

Tampering with Query Results In most cases, the HPP vulnerabilities that we discovered in our experiments allow the attacker to tamper with the data provided by the vulnerable website, and to present to the victim some information chosen by the attacker.

On several popular news portals, we managed to modify the news search results to hide certain news, to show the news of a certain day with another date, or to filter the news of a specific source/author. An attacker can exploit these vulnerabilities to promote some particular news, or conceal news that can hurt his person/image, or even subvert the information by replacing an article with an older one.

Also some multimedia websites were vulnerable to HPP attacks. In several popular sites, an attacker could override the video links and make them point to a link of his choice (e.g., a drive-by download site), or alter the results of a query to inject malicious multimedia materials. In one case, we were able to automatically register a user to a specific streaming event.

Similar problems also affected several popular search engines. We noticed that it would have been possible to tamper with the results of the search functionality by adding special keywords, or by manipulating the order in which the results are shown. We also noticed that on some search engines, it was possible to replace the content of the commercial suggestion boxes with links to sites owned by the attacker.

4.3 Ethical Considerations

Crawling and automatically testing a large number of applications may be considered an ethically sensitive issue. Clearly, one question that arises is if it is ethically acceptable and justifiable to test for vulnerabilities in popular websites.

Analogous to the real-world experiments conducted by Jakobsson et al. in [21, 22], we believe that realistic experiments are the only way to reliably estimate success rates of attacks in the real-world. Unfortunately, criminals do not have any second thoughts about discovering vulnerabilities in the wild. As researchers, we believe that our experiments helped many websites to improve their security. Furthermore, we were able to raise some awareness about HPP problems in the community.

Also, note that:

- PAPAS only performed client-side checks. Similar client-side vulnerability experiments have been peformed before in other studies (e.g., for detecting cross site scripting, SQL injections, and CSRF in the wild [24, 29]). Furthermore, we did not perform any server-side vulnerability analysis because such experiments had the potential to cause harm.

- We only provided the applications with innocuous parameters that we knew that the applications were already accepting, and did not use any malicious code as input.

- PAPAS was not powerful enough to influence the performance of any website we investigated, and the scan activities was limited to 15 minutes to further reduce the generated traffic.

- We informed the concerned sites of any critical vulnerabilities that we discovered.

- None of the security groups of the websites that we interacted with complained to us when we informed them that we were researchers, and that we had discovered vulnerabilities on their site with a tool that we were testing. On the contrary, many people were thankful to us that we were informing them about vulnerabilities in their code, and helping them make their site more secure.

5 Related work

There are two main approaches [14] to test software applications for the presence of bugs and vulnerabilities: white-box testing and black-box testing. In white-box testing, the source code of an application is analyzed to find flaws. In contrast, in black-box testing, input is fed into a running application and the generated output is analyzed for unexpected behavior that may indicate errors. PAPAS adopts a black-box approach to scan for vulnerabilities.

When analyzing web applications for vulnerabilities, black-box testing tools (e.g., [2, 8, 24, 33]) are the most popular. Some of these tools (e.g., [2]) claim to be generic enough to identify a wide range of vulnerabilities in web applications. However, recent studies ([6, 11]) have shown that scanning solutions that claim to be generic have serious limitations, and that they are not as comprehensive in practice as they pretend to be.

Two well-known, older web vulnerability detection and mitigation approaches in literature are Scott and Sharp's application-level firewall [30] and Huang et al.'s [17] vulnerability detection tool that automatically executes SQL injection attacks. Scott and Sharp's solution allows to define fine-grained policies manually in order to prevent attacks such as parameter tampering and cross-site scripting. However, it cannot prevent HPP attacks and has not been designed with this vulnerability in mind. In comparison, Huang et al.'s work solely focuses on SQL injection vulnerability detection using fault injection.

To the best of our knowledge, only one of the available black-box scanners, Cenzic Hailstorm [9], claims to support HPP detection. However, a study of its marketing material reveals that the tool only looks for behavioral differences when HTTP parameters are duplicated (i.e., not a sufficient test by itself to detect HPP). Unfortunately, we were not able to obtain more information about the inner-workings of the tool as Cenzic did not respond to our request for an evaluation version.

The injection technique we use is similar to other black-box approaches such as SecuBat [24] that aim to discover SQL injection, or reflected cross site scripting vulnerabilities. However, note that conceptually, detecting cross site scripting or SQL injection is different from detecting HPP. In fact, our approach required the development of a set of tests and heuristics to be able to deal with dynamic content that is often found on webpages today (content that is not an issue when testing for XSS or SQL injection). Hence, compared to existing work in literature, our approach for detecting HPP, and the prototype we present in this paper are unique.

With respect to white-box testing of web applications, a large number of static source code analysis tools (e.g., [23, 31, 34]) that aim to identify vulnerabilities have been proposed. These approaches typically employ taint tracking to help discover if tainted user input reaches a critical function without being validated. We believe that static code analysis would be useful and would help developers identify HPP vulnerabilities. However, to be able to use static code analysis, it is still necessary for the developers to understand the concept of HPP. Previous research has shown that the sanitization process can still be faulty if the developer does not understand a certain class of vulnerability [4].

Note that there also exists a large body of more general vulnerability detection and security assessment tools (e.g., Nikto [26], and Nessus [32]). Such tools typically rely on a repository of known vulnerabilities and test for the existence of these flaws. In comparison, our approach aims to discover previously unknown HPP vulnerabilities in the applications that are under analysis.

With respect to scanning, there also exist network-level tools such as nmap [18]. Tools like nmap can determine the availability of hosts and accessible services. However, they cannot detect higher-level application vulnerabilities.

In comparison to the work we present in this paper, to the best of our knowledge, no large-scale study has been performed to date to measure the prevalence and the significance of HPP vulnerabilities in popular websites.

6 Conclusion

Web applications are not what they used to be ten years ago. Popular web applications have now become more dynamic, interactive, complex, and often contain a large number of multimedia components. Unfortunately, as the popularity of a technology increases, it also becomes a target for criminals. As a result, most attacks today are launched against web applications.

Vulnerabilities such as cross site scripting, SQL injection, and cross site request forgery are well-known and have been intensively studied by the research community. Many solutions have been proposed, and tools have been released. However, a new class of injection vulnerabilities called HTTP Parameter Pollution (HPP) that was first presented at the OWASP conference [27] in 2009 has not received as much attention. If a web application does not properly sanitize the user input for parameter delimiters, using an HPP vulnerability, an attacker can compromise the logic of the application to perform client-side or server-side attacks.

In this paper, we present the first automated approach for the discovery of HPP vulnerabilities in web applications. Our prototype implementation called PArameter Pollution Analysis System (PAPAS) is able to crawl websites and discover HPP vulnerabilities by parameter injection. In order to determine the feasibility of our approach and to assess the prevalence of HPP vulnerabilities on the Internet today, we analyzed more than 5,000 popular websites. Our results show that about 30% of the sites we analyzed contain vulnerable parameters and that at least 14% of them can be exploited using HPP. A large number of well-known, high-profile websites such as Symantec, Google, VMWare, and Microsoft were among the sites affected by HPP vulnerabilities that we discovered. We informed the sites for which we could obtain contact information, and some of these sites wrote back to us and confirmed our findings.

We hope that this paper will help raise awareness and draw attention to the HPP problem.

Acknowledgments This work has been supported by the POLE de Competitivite SCS (France) through the MECANOS project and by the French National Research Agency through the VAMPIRE project. The work has also received support from the Secure Business Austria in Vienna.

References

[1] C. A. A-2000-02. Malicious HTML Tags Embedded in Client Web Requests, 2000. http://www.cert.org/advisories/CA-2000-02.html.

[2] Acunetix. Acunetix Web Vulnerability Scanner. http://www.acunetix.com/, 2008.

[3] I. Alexa Internet. Alexa - Top Sites by Category: Top. http://www.alexa.com/topsites/category.

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

[5] D. Bates, A. Barth, and C. Jackson. Regular Expressions Considered Harmful in Client-Side XSS Filters. In *19th International World Wide Web Conference. (WWW 2010)*, 2010.

[6] J. Bau, E. Burzstein, D. Gupta, and J. C. Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *Proceedings of IEEE Security and Privacy*, May 2010.

[7] T. Berners-Lee, R. Fielding, and L. Masinter. Rfc 3986, uniform resource identifier (uri): Generic syntax, 2005. http://rfc.net/rfc3986.html.

[8] Burp Spider. Web Application Security. http://portswigger.net/spider/, 2008.

[9] Cenzic. Cenzic Hailstormr. http://www.cenzic.com/, 2010.

[10] S. di Paola and L. Carettoni. Client side Http Parameter Pollution - Yahoo! Classic Mail Video Poc, May 2009. http://blog.mindedsecurity.com/2009/05/client-side-http-parameter-pollution.html.

[11] A. Doupé, M. Cova, and G. Vigna. Why Johnny Cant Pentest: An Analysis of Black-Box Web Vulnerability Scanners. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131, 2010.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, hypertext transfer protocol – http/1.1, 1999. http://www.rfc.net/rfc2616.html.

[13] B. D. A. G. and M. Stampar. sqlmap. http://sqlmap.sourceforge.net.

[14] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall International, 1994.

[15] W. G. J. Halfond and A. Orso. Preventing SQL injection attacks using AMNESIA. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, 2006.

[16] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4), October 1988.

[17] Y. Huang, S. Huang, and T. Lin. Web Application Security Assessment by Fault Injection and Behavior Monitoring. *12th World Wide Web Conference*, 2003.

[18] Insecure.org. NMap Network Scanner. http://www.insecure.org/nmap/, 2010.

[19] S. Institute. Top Cyber Security Risks, September 2009. http://www.sans.org/top-cyber-security-risks/summary.php.

[20] A. B. C. Jackson and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *15th ACM Conference on Computer and Communications Security*, 2007.

[21] M. Jakobsson, P. Finn, and N. Johnson. Why and How to Perform Fraud Experiments. *Security & Privacy, IEEE*, 6(2):66–68, March-April 2008.

[22] M. Jakobsson and J. Ratkiewicz. Designing ethical phishing experiments: a study of (ROT13) rOnl query features. In *15th International Conference on World Wide Web (WWW)*, 2006.

[23] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy*, 2006.

[24] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. SecuBat: A Web Vulnerability Scanner. In *World Wide Web Conference*, 2006.

[25] N. J. E. Kirda and C. Kruegel. Preventing Cross Site Request Forgery Attacks. In *IEEE International Conference on Security and Privacy in Communication Networks (SecureComm), Baltimore, MD*, 2006.

[26] Nikto. Web Server Scanner. http://www.cirt.net/code/nikto.shtml, 2010.

[27] OWASP AppSec Europe 2009. *HTTP Parameter Pollution*, May 2009. http://www.owasp.org/images/b/ba/AppsecEU09_CarettoniDiPaola_v0.8.pdf.

[28] J. Ratcliff and D. Metzener. Pattern matching: The gestalt approach. *Dr. Dobbs Journal*, 7:46, 1988.

[29] D. Reading. CSRF Flaws Found on Major Websites: Princeton University researchers reveal four sites with cross-site request forgery flaws and unveil tools to protect against these attacks, 2008. `http://www.darkreading.com/security/app-security/showArticle.jhtml?articleID=211201247`.

[30] D. Scott and R. Sharp. Abstracting Application-level Web Security. *11th World Wide Web Conference*, 2002.

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

[32] Tenable Network Security. Nessus Open Source Vulnerability Scanner Project. `http://www.nessus.org/`, 2010.

[33] Web Application Attack and Audit Framework. `http://w3af.sourceforge.net/`.

[34] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *15th USENIX Security Symposium*, 2006.