HTTP Parameter Pollution Vulnerabilities

in Web Applications

Is your web application protected against HTTP Parameter Pollution? A new class of injection vulnerabilities allows attackers to compromise the logic of the application to perform client and server-side attacks. HPP can be detected and avoided. But how?

What you will learn…

- what is HTTP Parameter Pollution (HPP)
- how to spoil HPP flaws in web applications
- how to prevent HPP in web developing

What you should know…

- basic understanding of web technologies and languages
- web security knowledge is a plus

In the last twenty years, web applications have grown from simple, static pages to complex, full-fledged dynamic applications. Web applications can 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* (XSS) and SQL injection (SQLi). According to the OWASP Testing Guide v3, *The most common web application security weakness is the failure to properly validate input coming from the client or environment before using it. This weakness leads to almost all of the major vulnerabilities in web applications [...]*. Several kind of injection flaws exist and they are usually strictly related to the specific metalanguage used by the subsystems: XML Injection, SQL Injection, LDAP Injection, etc. Each application layer uses a specific set of technologies and a characteristic contextual language.

In 2009, *Luca Carettoni* and *Stefano di Paola* introduced a new class of web vulnerabilities called HTTP Parameter Pollution (HPP) that permits to inject new parameters inside an existing HTTP parameter. Lately, in 2010, *Marco Balduzzi* of the International Secure Systems Lab at EURECOM investigated the problem and developed a system, called *PAPAS*, to detect HPP flaws in an automated way. He used PAPAS to conduct a large-scale study on popular websites and discovered that many real web applications are affected by HPP flaws at different levels.

This article discusses why and how applications may be vulnerable to HTTP Parameter Pollution. By analyzing different attacking scenarios, we introduce the HPP problem. We then describe PAPAS, the system for the detection of HPP flaws, and we conclude by giving the different countermeasures that conscious web designers may adopt to deal with this novel class of injection vulnerabilities.

Parameter Precedence

In the context of websites, when the user's browser wants to transfer information to the web application (e.g. a server-side script), the transmission can be performed in three different ways. The HTTP protocol allows to provide input inside the URI query string (GET parameters), in the HTTP headers (e.g. within the Cookie field), or inside the request body (POST parameters). The adopted technique depends on the application and on the type and amount of data that has to be transferred.

This standard mechanism for passing parameters is straightforward, 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.

The problem arises when a developer expects to receive a single parameter and, therefore, invokes methods (such as `Request.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 occurrences. 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 several 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 may produce an anomalous behavior in the application that can be potentially exploited by an attacker. As often in security, unexpected behaviors are a usual source of weaknesses that could lead to HTTP Parameter Pollution attacks in this case.

HTTP Parameter Pollution

In a nutshell, HTTP Parameter Pollution allows to override or introduce new HTTP parameters by injecting query string delimiters. This attack occurs when a malicious parameter, preceded by an (encoded) query string delimiter, is appended into an existing parameter `P_host`. If `P_host` is not properly sanitized by the application and its value is later (decoded and) used, the attacker is able to inject one or more new parameters.

The RFC 3986 defines the ampersand (`&`) symbol being the standard query string delimiter, but some applications may user different symbols (`|,;`). In the basic form of parameter pollution the attacker encodes his delimiter using the percent-encoding method (`%FF`); then depending from the application, other encoding schema like the double-encoding one (`%25FF`) can be adopted, or may reveal being unnecessary (ref. the Google Blogger example).

As a consequence of an HPP flaw, the attacker may be able to override a parameter that is submitted by a user to the application or that is fetched from a database server by the backend logic. These two scenarios result in two flavours of HTTP Parameter Pollution: client-side and server-side.

HPP Client-Side

In a typical client-side scenario (ref. Figure 1), a malicious user is interested into distributing a malicious URL that triggers the HPP vulnerability and runs an unintended attack to his victims.

For example, consider a website for the election of two candidates. The application uses a parameter `poll_id` to look-up the candidates, to build the appropriate links for voting and to generate the election page.

URL: `http://host/election.jsp?poll_id=4568`
Link A: `<a href="vote.jsp?poll_id=4568&candidate=white">Vote for Mr. White</a>`
Link B: `<a href="vote.jsp?poll_id=4568&candidate=green">Vote for Mrs. Green</a>`

Now suppose that an attacker *Mallory* wants to subvert the results of the election by forcing their victims into unintentionally voting for Mrs. Green. *Mallory* creates and distributes the following *trigger* URL to their victims:

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

Note how Mallory polluted the `poll_id` parameter by injecting into it a new *candidate=green* parameter. By clicking on the *trigger* URL, the victims are redirected to the vulnerable election website that now offers a page containing two injected links (Link A1 and B1 contain the new candidate):

Link A1: `<a href=vote.jsp?poll_id=4568&candidate=green&candidate=white>Vote for Mr. White</a>`
Link B1: `<a href=vote.jsp?poll_id=4568&candidate=green&candidate=green>Vote for Mrs. Green</a>`

No matter which link users click on, the application (in this case the JSP script) will receive two `candidate` parameters. Furthermore, since the first parameter will always be set to `green`, the candidate Mrs. Green will always be voted.

It's important to note that in order to perform a successful attack the parameter precedence must be consistent with the position where the injected parameter is placed. In the example, if the developer used a standard `JSP`'s `Request.getParameter(„par")` function, only the first value (the injected one) is returned to the application, and the second value (the one carrying the user's actual vote) is discarded.

HPP Server-Side

While in a client-side attack the goal of the *bad guy* is to attack other users, in the server-side variant the

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

Language/ 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)

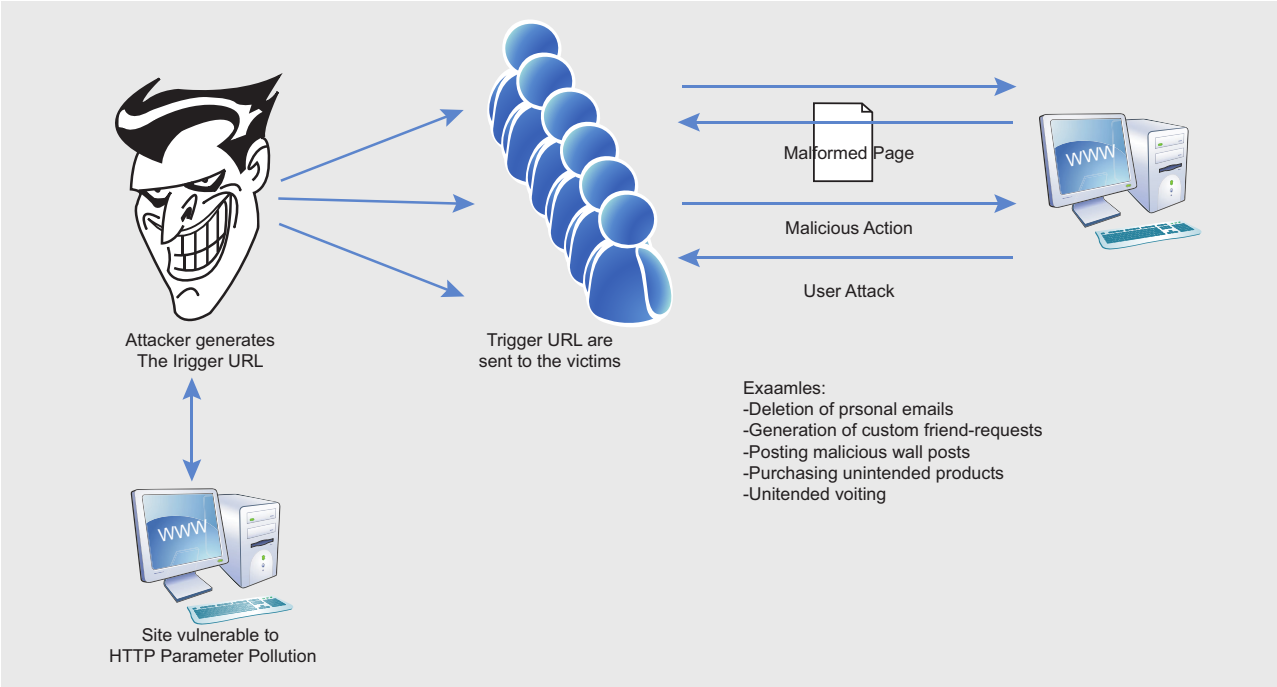

Figure 1. *A standard client-side HPP attack*

attacker leverages a vulnerable web application to access protected data or perform actions that either not permitted or not supposed to be executed.

The following example describes how an attacker can exploit a HPP flaw to alter a database query in a useful way.

Let's consider the application `printEmploys` that accepts a single parameter called department to specify for which department the client has requested the list of users. The value of this parameter is decoded and used by the backend to execute a query (`select`) on the database. A second parameter called `what` is hardcoded in the backend code and specifies which resource should be retrieved (the list of users).

URL: `printEmploys?department=engineering`
Back-end: `dbconnect.asp?what=users&department=engineering`
Database: `select users from table where department=engineering`

If the `department` parameter is not sanitized, an attacker may be able to introduce a second `what` parameter by encoding his value using the standard percent-encoding method. As the following snippet details, the backend is developed using the ASP technology and the values of the two parameters with the same name (`what`) are concatenated via comma. This will result in a query that selects for the list of users and their associated passwords.

Malicious URL: `printEmploys?department=engineering%26what%3Dpasswd`
Back-end: `dbconnect.asp?what=users&department=engineering&what=passwd`
Database: `select users,passwd from table where department=engineering`

Obviously, the real impact of this attack scenario depends on how the application treats query results.

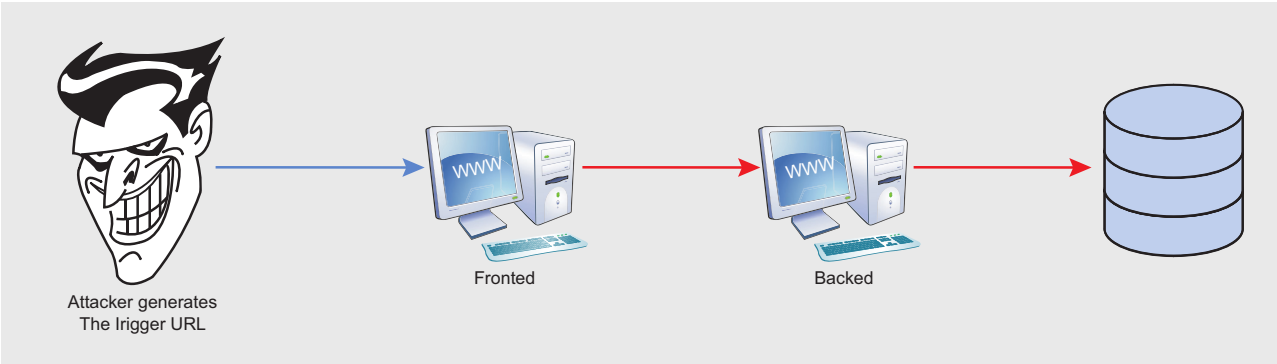

Figure 2. *The server-side attack*

In order to show the risk introduced by HPP attacks, let's also consider the following code snapshot of an e-banking application:

```
void private executeBackendRequest(HTTPRequest request){
String amount=request.getParameter(„amount”);
String beneficiary=request.getParameter(„recipient”);
HttpRequest(„http://backendServer.com/actions”,”POST”,
”action=transfer&amount=”+amount+”&recipient=”+beneficiary );
}
```

In this example, a frontend resource provides proxy functionalities to an internal application resource. As mindful readers have probably noticed, a malicious user could potentially pollute the `recipient` parameter with the `%26action% 3ddeposit` attack payload.

As a result, the backend request will look like:

```
action=transfer&amount=1000&recipient=Mat&action=deposit
```

Assuming that request is processed by a web framework which considers the last occurrence of multiple parameters (e.g. PHP), the overall action would allow an unauthorized deposit operation of 1000$.

Other Uses

In a previous example, we described how an attacker can pollute a link by injecting a new parameter into an existing parameter of a GET request. HPP attacks can also be used to override parameters between different input channels (e.g., GET, POST, or HEADER fields). In the next snippet, the attacker forces the client to build a request that can potentially override the value of the `id=1` POST parameter. This is achieved by injecting the `id=6` in the vulnerable parameter via GET.

URL: `foo?vulnerable-parameter=foo%26id%3d6`

FORM:

```
<form action=buy?vulnerable-parameter=foo&id=6
 <input type=”text” name=”id” />
 <input type=”submit” value=”Submit” />
</form>
```

Request:

```
POST  /buy?vulnerable_parameter=foo&id=6
Host: site.com

id=1
```

For instance in Apache Tomcat, the previous request would result in a selection of the item with id number 6. In fact, such web application environment considers the first occurrence of multiple parameters selecting GET parameters first.

For web application frameworks that concatenate multiple values of the same parameter (e.g. ASP), an attacker could use HPP to launch traditional web attacks (e.g. SQL Injection) bypassing web application firewalls (WAFs). As a matter of fact, HPP can be successfully

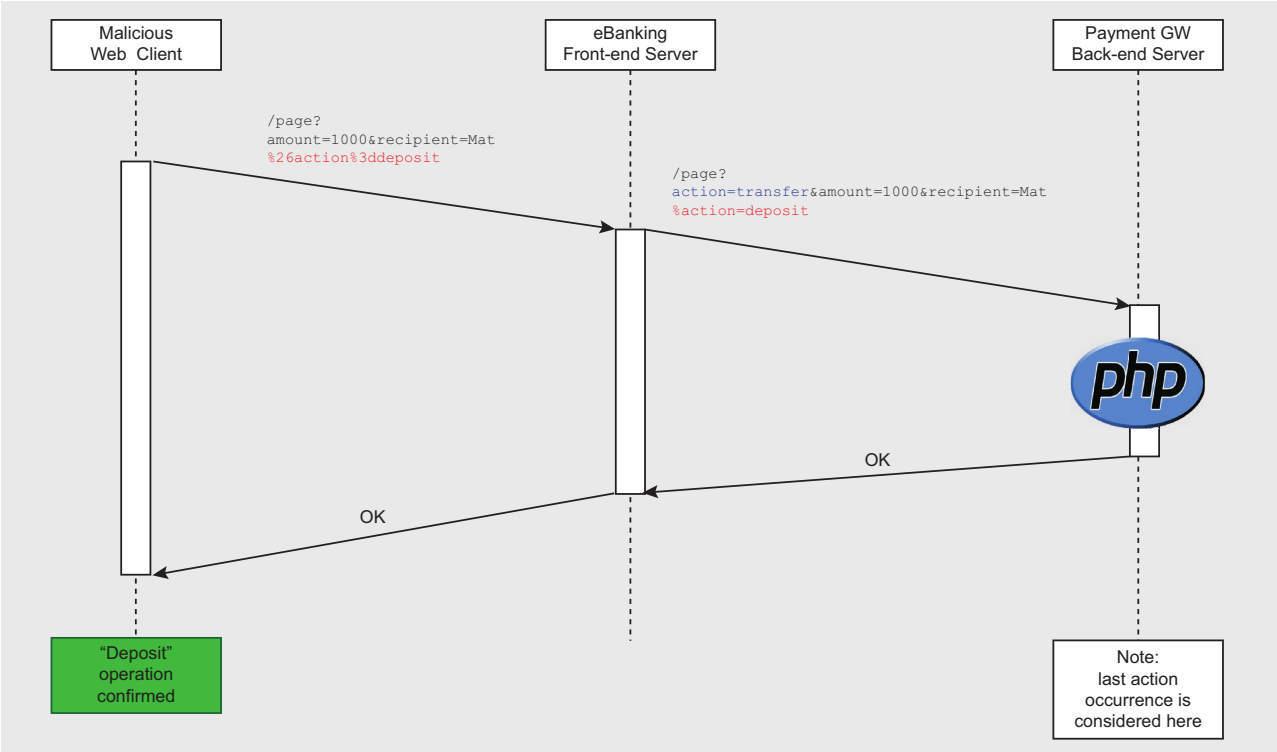

Figure 3. *An example of HPP server-side vulnerability affecting an ebanking application*

used to split malicious payloads and avoid signature-based detection.

In ASP, the two query strings `var=foo&var=bar` and `var=foo,bar` are equivalent as the second one can be obtained by the server-side concatenation of parameters.

The following shows how an attacker can setup a SQL Injection attack by splitting his query into multiple parameters with the same name.

Standard SQLi: `show_user.aspx?id=5;select+1,2,3+from+users+where+id=1--`
SQLi over HPP: `show_user.aspx?id=5;select+1&id=2&id=3+from+users+where+id=1--`

Lavakumar Kuppan published an alternative version of this example where *commas* introduced by the concatenation are stripped out with inline comments (only on Microsoft SQL Server).

Standard SQLi: `show_user.aspx?id=5+union+select+*+from+users--`
SQLi over HPP: `show_user.aspx?id=5/*&id=*/union/*&id=*/select+*/*&id=*/from+users--`

HPP in real-life

So far, we have illustrated several hypothetical situations where HPP can be used to override existing hard-coded HTTP parameters, modify the application behaviors and potentially exploit uncontrollable variables. Unfortunately, HPP vulnerabilities exist in real web applications too.

A dangerous utilization of HPP consists into bypassing the token protection mechanism used to prevent *Cross-Site Request Forgery* (CSRF) vulnerabilities. In this case, a unique token is generated by the application and inserted in all links to sensitive URLs. When the application receives a request, it verifies the 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.

The following URLs show how the CSRF protection adopted by Yahoo Mail! has been bypassed (in year 2009) via HPP. This client-side attack allowed an attacker to trick his victims into deleting all their personal e-mails. Note that the secret token is present in the `.rand` parameter.

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

Tricking a victim into clicking on the above link allowed an attacker to initiate a delete operation without the knowledge of the anti-CSRF token. In the victim's page, the previous request would result in the link:

```
showMessage?sort=date&order=down&startMid=0%26cmd%3Dfmgt.
emptytrash&DEL=1&DelFID=Inbox&cmd=fmgt.delete&.rand=107
                    6957714
```

A second click on any link in the webmail would irremediably empty the user's trash.

Interested readers can obtain further details watching a demonstration video (*http://www.youtube.com/watch?v=-O1y7Zy3jfc*) of the proof-of-concept attack.

Another HPP vulnerability turned out to affect Apple Cups, the well-known printing system used in Mac OS X and many UNIX systems. Exploiting HPP, an attacker could easily trigger a Cross-Site Scripting vulnerability.

```
http://127.0.0.1:631/admin/?kerberos=onmouseover=alert
                (1)&kerberos
```

The application validation checkpoint could be bypassed by adding an extra `kerberos` argument having a valid string as content (e.g. empty string). As the validation checkpoint would only consider the second occurrence, the first `kerberos` parameter in vulnerable versions is not properly sanitized before being used

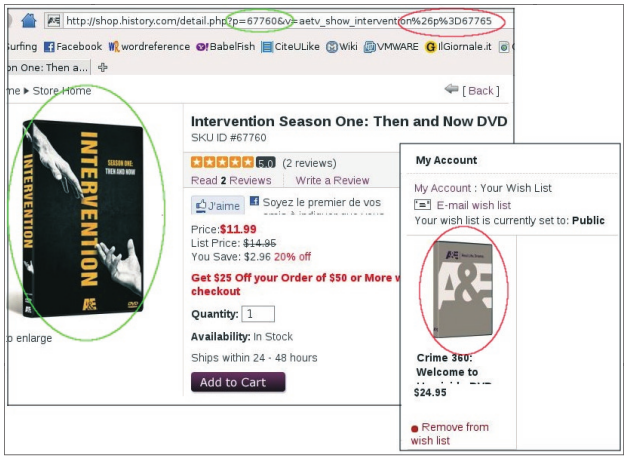

Figure 4. *An example of content pollution using HPP*

Figure 5. *Sharing components on the web*

to generate dynamic HTML content. Successful exploitation would result in Javascript code execution under the context of the hosting web site.

An even more critical vulnerability has been recently patched by Google in its popular blogging platform. An HPP bug in Blogger allowed malicious user to take ownership of the victim's blog. In detail, this authentication flaw exploited how different application layers consider multiple occurrences, treating either the first or the second occurrence. *Nir Goldshlager* discovered that the following request (note the two blogid parameters) would result in the attacker added as an author on the victim's blog:

```
POST /add-authors.do HTTP/1.1

security_token=attackertoken&blogID=attackerblo
gidvalue&blogID=victimblogidvalue&authorsList=goldsh
lager19test%40gmail.com(attacker email)&ok=Invite
```

The flaw resided in the authentication mechanism used by Blogger. The validation was performed on the first parameter, whereas the actual operation used the second occurrence. From that, the attacker could easily elevate his privileges from author to admin thanks to the following request:

```
POST /team-member-modify.do HTTP/1.1

security_token=attackertoken&blogID=attackerownblogid&
blogID=victimblogidvalue&memberID=attackermemberid&is
Admin=true&ok=Grant+admin+privileges
```

A demonstration video for this vulnerability can be watched on YouTube (*http://www.youtube.com/watch?v=AdIWl0gkynk*).

Then, HPP can also be used to create phishing traps or trick the user into performing involuntary actions through UI-Redressing or content pollution attacks.

A simple example is briefly illustrated in Figure 4.

In this case, tampering the product code (p parameter) an aggressor may be able to create misleading situations within an e-commerce site. The online customer see the description of product #67760, however it's actually buying product #67765.

Finally, one of the author discovered that the sharing functionality offered by Facebook was prone to parameter pollution client-side attacks. We are talking of the well known *like* and *share* buttons commonly found on blogs and news (ref. Figure 5). This attack is possible when the website (client) that uses the sharer API does not properly sanitizes the values sent to Facebook (server). In this situation, the attacker can abuse of a vulnerable page to inject a second reference parameter (called `u`) into the description parameter to overwrite the URL to share.

For example, given a vulnerable page of *www.vulnerable.com*, the following URL: *http://www.vulnerable.com/shareurl.jsp?shareurl=http://www.vulnerable.com/news.html&description=The+news+page+of+vulnerable.com%26u%3Dattackerurl* returns a page polluted with the Facebook injected link (the precedence is on the second parameter): `http://www.facebook.com/sharer.php?u=http://www.vulnerable.com/news.html&r=The+news+page+of+vulnerable.com&u=attackerurl`.

Facebook fixed this issue by checking that the URL to share is sent once to the component.

Automated Detection with PAPAS

Marco Balduzzi and colleagues developed a tool called PAramater Pollution Analysis System (PAPAS) to automate the detection of HPP flaws in Web Applications. PAPAS consists of four main components: A browser, a crawler, and two scanners. Figure 1 illustrates the global architecture of PAPAS.

The first component is an instrumented vesion of Firefox. The *browser* is responsible for fetching webpages, rendering their content, and extracting all

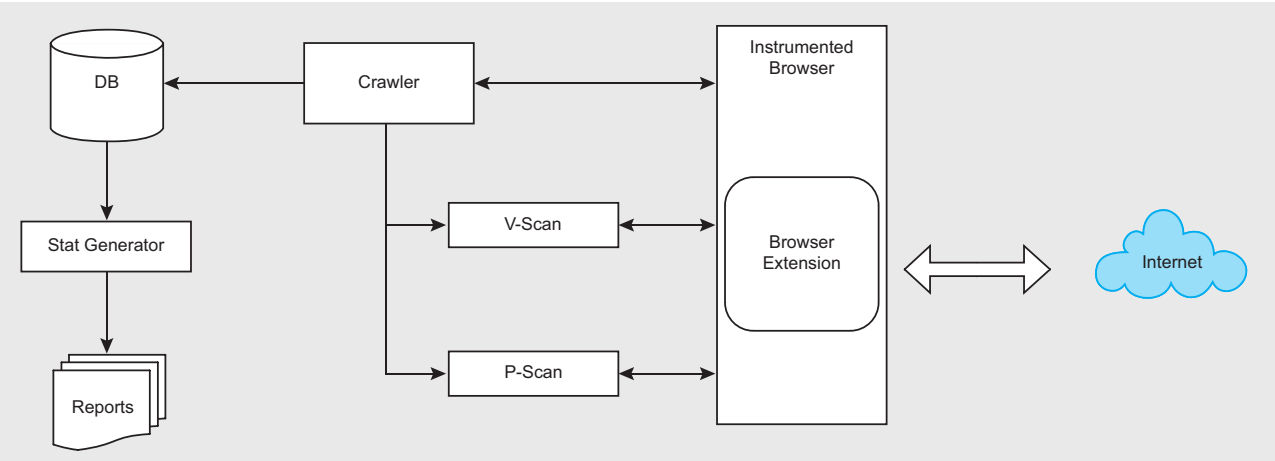

Figure 6. *PAPAS global architecture*

the links and form URLs contained in the page. The big benefit of having adopted a real browser instead of a custom HTTP/HTML client to render the pages, is that the browser provides *for free* an engine for handling client-side scripts (e.g. Javascript) and complex dynamic applications (IFRAMEs, events).

That is, when the crawler issues a new page to be tested, the browser in PAPAS first waits until the target page is loaded; after the browser parses the DOM, executes any client-side scripts, and loads additional resources; then a browser extension extracts the content, the list of links, and the forms in the page. This extension has been developed using the standard technology offered by the Mozilla development environment: a mix of Javascript and *XML User Interface Language* (XUL) and XPConnect to access Firefox's XPCOM components. These components are used for invoking GET and POST requests and for communicating with the scanning components.

The *crawler* communicates with the browser through a bidirectional TCP/IP 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. The crawler automatically fills forms with guesses data, for example random alphanumeric values of 8 characters are inserted into password fields and a default email address is inserted into fields with the name email, e-mail, or mail. When the authenticated area of a site wants to be tested, 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 logout page (e.g., by excluding links that include the `cmd=logout` parameter).

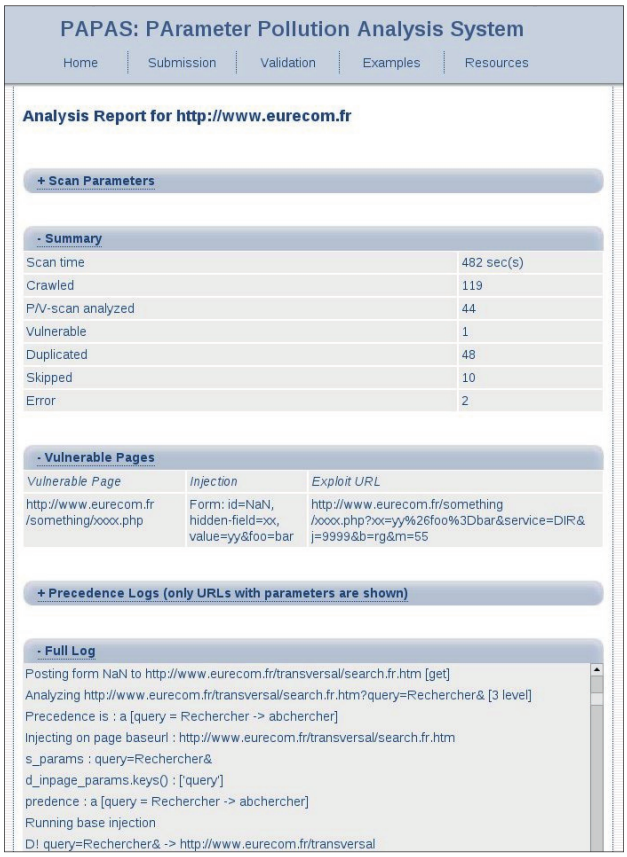

Figure 7. *The online version of PAPAS*

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) tests the page to determine if it is vulnerable to HPP attacks.

The *Precedence Scanner* 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. 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`. Example:

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 (e.g. banners) that may still vary even when the page is accessed with exactly the same parameters. To solve this problem, the P-Scan component first pre-process the page by trying to eliminate content that does not depend on the parameter values, and then uses mathematical heuristics to compute the similarity among pages (more details in the references).

The *Vulnerability Scanner* tests the page to determine if some parameters can be injected by HPP. For every page that V-Scan receives from the crawler, it tries to inject a URL-encoded version of an innocuous parameter (a `nonce`) into each existing parameter of the query string and possibly of the page. 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, given a parameter `par1=val1`, V-Scan

On the 'Net

- *http://www.iseclab.org/people/embyte/slides/bh_series.pdf – Slides on HTTP Parameter Pollution*
- *http://papas.iseclab.org/ – The PAPAS service*
- *http://www.iseclab.org/people/embyte/slides/BHEU2011/whitepaper-bhEU2011.pdf – The Blackhat white paper on Parameter Pollution*

injects build a request as `par1=val1%26foo%3Dbar` and then checks if the nonce `&foo=bar` has been used to build a link or a form in the answer page. This technique works well for parameters that are reused by the application under the same or different name.

The scanner supports *three* different operational modes: fast mode, extensive mode and assisted mode. The fast mode aims to rapidly test a site for potential vulnerable parameters and enables the standard tests. In the extensive mode, the scanner pickup each single parameter from the page's content and injects the nonce. The extensive mode is lower but tries to detect injections in parameters that were not used in the original page.

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. As a result, the assisted mode can be used by security professionals to conduct semi-automated assessment of web applications, or to test websites that require a particular user authentication.

In the current version of PAPAS the vulnerability scanner only looks for client-side vulnerabilities. In fact, testing for server-side attacks using a black-box approach (that one used by PAPAS) is more difficult than testing for client-side attacks as comparing requests and answers is not sufficient.

Free-to-use online service

Marco has recently deployed an online version of PAPAS, shows in figure 6, that allows website maintainers to scan their sites for HTTP Parameter Pollution vulnerabilities. Web developers and analysts can submit their sites to PAPAS for being tested, without any additional fee. PAPAS uses a challenge-response mechanism based on tokens (called *PAPAS.txt*) to prove that the submission comes from the developer/maintainer of the site. Once the site has been validated, an automated engine queues the submission and contacts the user when the scan is completed and the HTML report is available.

The submission is customizable and settings such as scanning depths, delaying times between requests, precedence and vulnerability scanning engines, extensive operational mode and URL to exclude are all configurable by the analyst.

Countermeasures

Being aware of this new class of vulnerabilities gives you a competitive advantage over the attackers. If you went through the entire article, it should be already clear how malicious users can abuse your application and what is the root cause of this security problem.

As for every input validation vulnerability, filtering is the main countermeasure to avoid that malicious payloads can be injected in the application. In case of HPP, the query string delimiter and its encoded versions are the *dangerous* characters to be properly filtered.

Input validation and output encoding allow to protect our applications against HPP attacks.

As a final remark, we would like to suggest to the reader a few general recommendations:

- Encode query string delimiters using URL encoding
- Consider parameter precedence whenever implementing your business logic
- Perform proper channel (GET/POST/Header fields) validation
- Use strict regexp in URL rewriting

Lastly, it's important to know our systems and the technology used in our application environments so that data validation and encoding can be applied for the right context.

MARCO BALDUZZI

Marco `embyte` Balduzzi, MSc. in Computer Engineering, has been involved in IT-Security for more than 8 years with international experiences in both industrial and academic fields. He has worked as security consultant and engineer for different companies before joining a Ph.D. program in EURECOM (iSecLab group). He attended well-known and high-profile conferences all over (BlackHat, OWASP AppSec, NDSS) and in former times was an active member of open-source projects and Italian hacking groups.

LUCA CARETTONI

Luca `ikki` Carettoni, MSc. in Computer Engineering with a major specialization in web application and Java security. He has published several research papers, vulnerability advisories and articles on computer security. In short, he breaks things for a living.

STEFANO DI PAOLA

Stefano Di Paola is the CTO and a cofounder of Minded Security, where he is responsible for the Research and Development Lab. Prior to founding Minded Security, Stefano was a freelance security consultant, working for several private and public companies. Stefano is recognized as one of the top application security researchers.