Abusing Locality in Shared Web Hosting

Nick Nikiforakis, Wouter Joosen
DistriNet, Katholieke Universiteit Leuven
{nick.nikiforakis,wouter.joosen}
@cs.kuleuven.be

Martin Johns
SAP Research - Karlsruhe
martin.johns@sap.com

ABSTRACT

The increasing popularity of the World Wide Web has made more and more individuals and companies to identify the need of acquiring a Web presence. The most common way of acquiring such a presence is through Web hosting companies and the most popular hosting solution is shared Web hosting.

In this paper we investigate the workings of shared Web hosting and we point out the potential lack of session isolation between domains hosted on the same physical server. We present two novel server-side attacks against session storage which target the logic of a Web application instead of specific logged-in users. Due to the lack of isolation, an attacker with a domain under his control can force arbitrary sessions to co-located Web applications as well as inspect and edit the contents of their existing active sessions. Using these techniques, an attacker can circumvent authentication mechanisms, elevate his privileges, steal private information and conduct attacks that would be otherwise impossible. Finally, we test the applicability of our attacks against common open-source software and evaluate their effectiveness in the presence of generic server-side countermeasures.

Categories and Subject Descriptors

K.4.4 [**Electronic Commerce**]: Security; K.6.5 [**Security and Protection**]: Unauthorized access

General Terms

Security, Design, Experimentation

Keywords

Web applications, session identifiers, session storage, server-side attacks

1. INTRODUCTION

In 1993 CERN announced the release of the World Wide Web as a free product available to everyone. Today, the WWW has become synonymous to the whole Internet infrastructure since for most people, "Internet" is accessed through their browser. In this ever-expanding online world, more and more individuals and companies identify the need to have a Web presence, i.e, a website to provide services, sell goods and communicate news to their customers. Thus thousands of domain names are being registered every day and thousands of hosting plans from Web hosting providers are acquired. At the time of writing, there are more than 120 million active registered domains[1]. Each of these domain names, finally points to the IP address of a server, hosting the content of each individual website. With the exception of large cooperations, institutions and government services, the majority of websites are hosted on companies that provide Web hosting services.

Web hosting providers offer a variety of hosting products, ranging from shared solutions to fully dedicated servers. While competition has driven the prices of all hosting products down, the most economical solution was and still is "shared hosting". In shared Web hosting, the client is given access to a limited number of resources (e.g., a limited number of Gigabytes on disk and of SQL databases) which he can use to host his own website/Web application. The physical servers that provide the shared-hosting resources are shared amongst hundreds or even thousands of clients at the same time. Thus, the user doesn't suffer only from limited performance but there is also a potential for limited security since his Web applications are co-located with other scripts that can act maliciously by design or by omission. Due to this fact, several barriers are placed for each user in-order to ensure that a malicious or vulnerable application cannot interfere with the others [2].

In this paper we present two new attacks against the session management system of Web applications in shared hosting environments: *Session Snooping* and *Session Poisoning.* To be susceptible to these attacks, a Webserver's configuration has to meet certain criteria in respect to the way sessions are handled (see Sec. 3 for details). If these preconditions are fulfilled, an adversary who controls a Web application that is hosted on such a server, is able to attack all other applications that share this locality. More precisely, he is able to force arbitrary sessions to the vulnerable Web applications (Session Poisoning) or to inspect and modify their session values (Session Snooping). This way, an attacker can circumvent authentication procedures, steal private data, evade Web application firewalls, and use this attack as an intermediate step to launch other attacks such

[1] `http://www.domaintools.com/internet-statistics/`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EUROSEC'11, Salzburg, Austria.
Copyright 2011 ACM 978-1-4503-0613-3/11/04 ...$10.00.

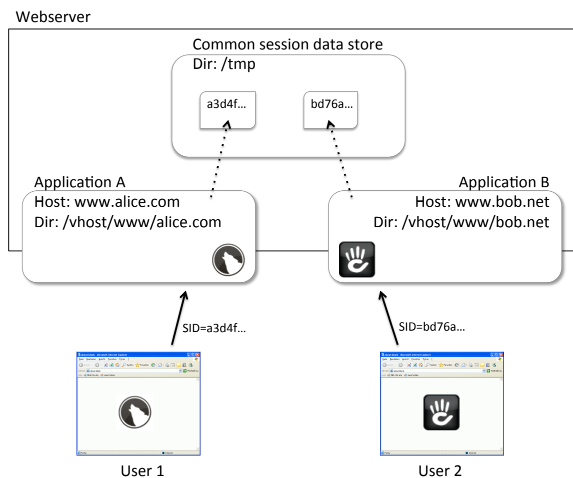

Figure 1: Common session store for shared hosting

as SQL injection and command execution. Unlike existing attacks against session identifiers which target authenticated users, our attacks are performed on the server and they circumvent a Web application's logic.

For most parts of this paper, we focus on the ubiquitous PHP programming language to present and evaluate our findings; a sensible choice given the perceived dominance of PHP in the area of shared hosting offers. However, the discussed issues are not necessary unique to PHP. In fact, every Web application framework that exposes the necessary preconditions is potentially susceptible. To validate this assumption, we also briefly examine the session handling provided by `Mod_Python` and `Mod_Perl` and identify similar issues as with PHP (see Sec. 4.3).

The rest of our paper is structured as follows: In Section 2 we present background information concerning the workings of session identifiers on the server side. In Section 3 we describe the aforementioned server-side attacks against session identifiers and we explore their implications. We evaluate our attacks against open-source software in Section 4, we discuss the reasons that made these attacks possible in Section 5 and test generic server-side countermeasures in Section 6. In Section 7 we present the related work and we conclude our paper in Section 8.

2. BACKGROUND

In this section we provide the necessary background knowledge about session identifiers to understand the attacks presented in Section 3.

The most common protocol used today, the Hyper Text Transfer protocol and its secure version (HTTPS) are by design stateless. While there are several ways of enforcing basic access control, such as HTTP Authentication and client-side SSL certificates, none can provide the flexibility and fine-grained control of session identifiers. Thus session identifiers are the *de facto* modern mechanism used in virtually all non-static webpages.

The first time that a user visits a website/Web application, he is assigned a session identifier (typically consisting of a long pseudo-random alpha-numerical string). This identifier is communicated to the client using a variety of ways, the most common of which, is through `Cookie` headers. On the server-side, the Web application binds the identifier with information about the user (e.g., if he is logged in and his user privileges). On every subsequent request, the client provides the Web application with the identifier that was first entrusted to him. This way the application can recognize the user amongst many requests and respond appropriately. When sessions are used to recognize a logged-in user, the session identifier token is as sensitive as the username and password combination of the logged-in user.

While the client-side part of session management is straightforward and well-understood by the security community, the corresponding server-side is implementation-specific and hasn't received as much attention as its client-side counterpart. In PHP, when a new session identifier is requested by the Web application (e.g., through the `session_start()` function), PHP generates a pseudo-random identifier which it returns to the application. All the values that the application stores in the session id, are stored, by default, on a flat file on the disk of the Webserver. On standard installations of PHP and unless instructed otherwise by the Web programmer or the Web administrator, the file is stored in a temporary directory, (e.g. `/tmp`), and the name of the file is predictable and consists of the string `sess` followed by an underscore and the value of the session identifier. The values that are set by the Web application are stored in this file, in a manner which will enable PHP to re-construct them on subsequent requests.

At a later request of the same user, PHP uses the session identifier provided by the request to locate the file containing the previously-set session values and load them in the global session array (`$_SESSION`) that the Web application utilizes. The session identifier is actually read from the cookie value that the user's browser appended to the user's request. With PHP managing all the details of session identifiers, the Web application can track the user in time and use simple logic built around values of the global session array, to enforce the desired access control to its resources.

3. LOCALITY ABUSE

In Section 2 we provided an overview of how PHP handles session identifiers and how user-provided information (the value of the session Cookie) is used to determine the session file containing the appropriate stored information for the user's session. The two attacks that we are about to present are based on the following two weaknesses:

1. PHP's standard session mechanisms do not offer a way for a Web application to distinguish between sessions that were created by itself and sessions that were created by other Web applications located on the same physical server
2. The session store of PHP installations is by default a temporary folder which is shared among all the PHP scripts running on that physical machine, see Fig. 1.

The combination of the two above weaknesses enable an attacker to create session identifiers from his own Web application and force them as legitimate session identifiers to the other Web applications residing on the same physical server (Session Poisoning). It also allows the attacker's scripts to open session identifiers that were created by co-located Web applications, inspect the stored values and make arbitrary changes to them (Session Snooping). The specific details for these attacks are presented in the following two sections.

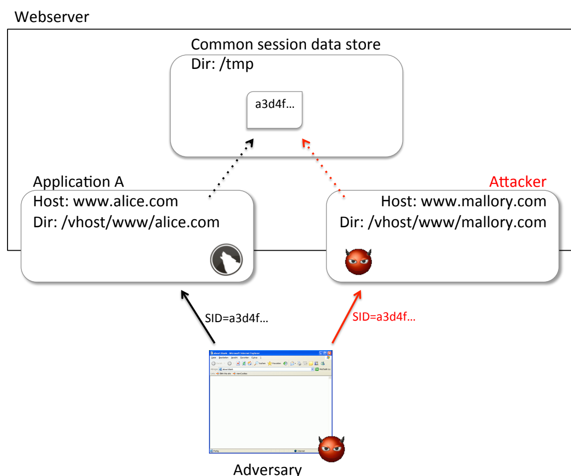

Figure 2: Attacking the common session store

Code Listing 1 Session Poisoning script

```
<?php
  session_start();
  ...
  $_SESSION['isadmin'] = true;
  $_SESSION['userid'] = 1;
  $_SESSION['user'] = "admin";
  ...
  print session_id();
?>
```

3.1 Session Poisoning

In a Session Poisoning attack, the attacker tries to circumvent the logic of a victim Web application by creating a session with arbitrary content and forcing it as a valid session onto the vulnerable Web application. For example, consider two domains, *alice.com* and *mallory.com* hosted on the same physical server and using a common session store. *Alice.com* has a Web administration interface where an administrator can login and perform privileged operations. When the administrator provides valid credentials, a boolean variable `isadmin` is created in his active session and it is set to `true`. This enables the Web application to track him between page-loads and to provide him administrative rights without the need for re-sending his credentials with each request. The malicious owner of *mallory.com* creates a session for *mallory.com* where he also sets a boolean variable named `isadmin` to `true`. In the next and final step, the user from *mallory.com* visits the Web administration interface at *alice.com* where he provides as a session-cookie the session created for *mallory.com*. Since both Web applications share a common session store, *alice.com* will locate and load the session file created by *mallory.com*, which will in-turn provide the attacker with administrative rights (see Fig. 2).

In the aforementioned process, the attacker can force the malicious session identifier, simply by adding an HTTP `Cookie` header with the value of the session identifier, as returned by *mallory.com*. The only real difficulty of the malicious user controlling *mallory.com* is to predict the names and contents of the session variables needed to grant him administrative rights. An attacker can compensate for this by using a large set of commonly-used variable names (see Listing 1). As long as the subset of variables needed exist in the set of variables the attacker creates, the vulnerable Web application will ignore the rest and take into account only the ones dictated by its access-control logic. Additionally, if the victim application is not a collection of custom scripts but is an open-source software product, the attacker can inspect the source-code of the particular application, discover the access-control variables and set them appropriately.

3.2 Session Snooping

The Session Snooping attack is essentially a reverse of the Session Poisoning attack where the attacker can inspect active sessions created by other Web applications and arbitrarily change their content. In this scenario, *alice.com* hosts a Web application that users can register, login and access different content based on their user privileges such as a forum or a Webmail platform. The malicious administrator of *mallory.com*, visits *alice.com* registers an account and logs-in. *Alice.com* populates a variable named `userid` in the attacker's session with the user's identifier which is used to provide him with the right content from the Web application's database. At this point, the attacker switches to *mallory.com* and instructs his malicious scripts to load the session corresponding to the session identifier provided by *alice.com*. The attacker can now inspect the variables set by *alice.com* and change the `userid` to that of another user using standard session functionality. The final step for the attacker, consists of simply visiting again the vulnerable Web application at *alice.com*. The application will reload the values stored in the session file corresponding to the attacker's session and thus "recognize" the attacker as a different user. The attacker has now full rights over the account of that user and depending on the specifics of the application, the attacker will be able to get access to a wide range of private data such as the user's home address, emails, credit card information and so on.

It is important to point out that changing a session identifier to a user-provided value is a legitimate operation in PHP and can be performed by the built-in `session_id()` function. Even if the function did not exist, Web frameworks still rely on the user to provide the "correct" session identifier, in which case the attacker could simply change the value of his HTTP `Cookie` header as in Section 3.1. Lastly, note that it is advantageous for an attacker to use the built-in session management functions to load, inspect and edit session variable values instead of opening the session file directly since potential protections that are implemented in the file-system level may be lifted when the data is accessed through PHP's session management system(see Section 6.2).

3.3 Resulting malicious capabilities

Session snooping can be used as an intermediate step which will help the attacker to launch new attacks against a Web application that would otherwise be impossible. We believe it is a common programming practice to trust the values of set variables when the variables are not directly affected by user input. Through Session Snooping, any variable stored in the global session array (`$_SESSION`) can be arbitrarily modified by an attacker. Thus, while a login script may be thoroughly checked for SQL injection attacks, the values that are set by the programmer's script inside session identifiers may be used directly in SQL queries without

Code Listing 2 SQL Injection through Session Snooping

```
<?php
...
if (isset($_SESSION['userid'])){
    $result = db_request("SELECT * from users
           where id = " . $_SESSION['userid']);
    $row = mysql_fetch_array($result);
    ...
}
?>
```

further sanitization. Code listing 2 shows a snippet of PHP code that is responsible of getting the personal details of a logged-in user based on his user identifier. This action is very common in Web applications which provide a user the option to view and edit his personal profile.

Traditionally the programmer has no reason to sanitize the `$_SESSION['userid']` value, since it is only set by the database as a result of successful user login (not shown). However, through Session Snooping, an attacker can change the value of `userid` to an SQL statement and thus perform an SQL injection attack, that was previously not possible. In addition to SQL injection, Session Snooping can be used to modify the values of session variables that affect local file operations, remote file includes, database settings and so on.

Even more interestingly, the attack vectors are not present in external HTTP(S) traffic towards the vulnerable Web application. That is because, the attacker modifies the contents of a session file (through traffic directed at his domain) which is at a later request internally loaded into the running PHP environment of the victim Web application. This means that Web application firewalls, both rule-based and behaviour-based [10, 13], that check the contents of headers will be unable to stop common Web attacks. For the same reason, the Webserver logs of the victim domain will contain no information about the actual attack, making post-exploitation forensic investigations much harder.

3.4 Attack variants

In the above sections we assumed that the adversary is controlling a Web application which is co-hosted on the same server as the attacked application. Such circumstances are not a necessary precondition for the attacks to be feasible. In this section we list alternative scenarios that enable the adversary to exploit common session data stores:

Co-located identical applications: Consider the following scenario: On a shared hosting server a given Web application (e.g., a specific CMS) is installed by two different hosting customers. Based on the observations above, a legitimate user on one of these installations (e.g., an user with administrative rights) can force the second application to use the session data store which was originally filled by the first application. Depending on the application's logic, this might result in gaining access to the second application, potentially even with administrative rights.

PHP code injection: Applications written in PHP frequently expose code injection vulnerabilities. Such vulnerabilities enable an attacker to execute PHP commands with the rights of the vulnerable applications. Hence, he can conduct the attacks described above against all co-located Web applications, even if these applications do not expose any vulnerabilities of their own.

3.5 Finding co-located applications

A practical issue for an attacker, is that of finding which websites/Web applications are located on the same physical server as his own. Depending on the configuration details of each Webserver this can be done in multiple ways with varying degrees of success. If a Webserver permits it, the attacker can use his scripts to recursively list directories towards the root of the file-system and record the names of his co-located applications. In cases where the Webserver is running scripts with the permissions of their owners (see Section 6.1) the owner of each listed file can also identify URIs of other applications. Lastly, an attacker can utilize online services[2] which, given a URI, report other URIs hosted on the same physical server based on matching IP addresses and other heuristics.

4. PRACTICAL EVALUATION

4.1 Common Session Stores

In Sections 2 and 3 we discussed about the temporary directories in which PHP saves, by default, the files corresponding to active sessions and how this behavior can be abused to perform server-side attacks against Web applications. We decided to perform a simple experiment to discover what percentage of PHP websites keep the default, and unsafe, session save path (`session.save_path` in PHP) and what percentage change it to per-domain value.

PHP provides a function named `phpinfo()` which prints a detailed report of all its configuration parameters. Among others, the save path of sessions is included in the generated report. This function is normally used for debugging and testing purposes however in many cases the PHP files that make use of `phpinfo()` are forgotten on the Webserver. Using Google, we located *phpinfo* pages on websites and crawled through the first 500 of them, recording the stated session path. Surprisingly, we discovered that 89.71% of all sites used a default path (e.g., `/tmp`,`/var/lib/php4` and `C:\PHP\sessiondata`). This means that for 9 out of 10 tested websites, a per-domain session path is not forced by each Webserver and that the programmer of each Web application did not change the default session configuration.

While our experiment is certainly not an exhaustive one, we believe that it still demonstrates that keeping the default session save path is the common case. At this point, one might think that finding a *phpinfo* page at a specific website is already enough evidence that security is not taken seriously and thus can't be used as a global metric towards the save path of sessions. We argue that this is not the case. Since *phpinfo* pages can be generated simply by calling the `phpinfo()` function, their mere presence can't be used to measure the security of a Webserver installation. On the other hand, the various configuration parameters presented by the *phpinfo* pages, give a more complete picture of the security mechanisms and configurations set in place by the Web administrators and in our case attest to the insecure practice of common session stores.

4.2 Effectiveness

In this section we present a security evaluation of the attacks described in Section 3 by testing them against popu-

[2] `http://www.yougetsignal.com/tools/web-sites-on-web-server/`

lar open-source Web applications. It is widely accepted that open-source software is more secure that proprietary software since many developers can review the code and report vulnerabilities. In our case however, open-source Web applications are an easier target since an attacker knows exactly which variables to set to which values through a Session Poisoning attack. For our evaluation, we decided to investigate open-source Content Management Systems.

Content Management Systems (CMSs) are software products that allow users to create and manage completely dynamic websites without the need of directly writing code. A recent study showed that more than 24.1% of the top 1 million websites (as reported by Alexa.com) use a CMS [6]. Due to their popularity, an attack against a CMS automatically means an attack against millions of its installations, much like attacks against Operating Systems or popular software.

Specifically, we investigated the session handling techniques of 10 popular CMSs to discover if they were vulnerable to Session snooping/Session poisoning attacks. From these ten CMSs (Joomla, WordPress, PHPNuke, Concrete5, Drupal, WolfCMS, ImpressCMS, Mambo, B2Evo and DotClear), 9 of them were using sessions (WordPress doesn't natively use sessions) and 2 out of these 9 were using the default PHP session management functionality (Concrete5 and WolfCMS) which makes them vulnerable when installed in a shared hosting environment. Using the described session attacks an attacker can login as an administrator into these two CMSs without knowing the administrator password.

4.3 Further affected technologies

In addition to our practical experiments with PHP, we examined `mod_python` and `mod_perl` to evaluate whether the observed issues are specific to PHP or if similar problems can be also encountered in other frameworks. We chose these two technologies as they are also frequently offered in shared hosting scenarios, probably because they are free and work well with the popular Apache Webserver.

`Mod_perl` does not provide native session handling. Instead, specialized Perl libraries are provided for this task. A common choice is `Apache::Session`. Very similar to PHP, the file-based mechanism of `Apache::Session` defaults to a common, server-global temporary directory. The filename of the session file is identical to the session ID value and the library's API-call to retrieve the session data only takes the session ID as the only argument.

`Mod_python`'s session handling mechanism utilizes in its default configuration a global session storage file called `mp_see.dbm.db` which holds the data values for all currently active sessions. This file is global for all hosted applications, regardless of applications's domains. Retrieval of session data is handled transparently for the application programmer and apparently is solely based on the value of the `pysid` cookie.

In our practical experiments both frameworks proved to be susceptible to the attacks described in Section 3.

5. DISCUSSION

To understand the circumstances that have led to these vulnerabilities, one has to recall the history of HTTP: In its original form and early versions HTTP had neither a session concept nor native capabilities for shared hosting. While the latter was added to HTTP 1.1 via the `Host`–header, the former was left to the programmers, which resorted to session identifiers (see Sec. 2). In consequence, we have a case of ill fitted separation of concerns: The Webserver is responsible to route the requests to the correct executable (deducted from the `Host`-header and the requested URI), while the programmer's duty is to implement the session handling. But as witnessed above, in shared hosting scenarios, these two functionalities should not be handled separately as there is a functional dependency between a given session and its `Host` or its `Host`s in cases in which the Web application spans more than one (sub-)domain.

This problem is not easily addressed in a general fashion. In most scenarios the Webserver, that handles the `Host` separation, the application framework, that provides the session handling, and the actual Web application are three distinct units, all created and maintained separately. However, without support from the application, neither the framework nor the server can assess to which set of domains (i.e, `Host`-header values) a given session should apply, rendering default isolation mechanism susceptible to potential weaknesses.

6. EXISTING COUNTERMEASURES

Due to the popularity of shared Web hosting, much effort has been put into making shared hosting installations more secure. In this section, we will overview the most common server-side attack countermeasures that are deployed on Webservers and how they stand up against the server-side session attacks described in Section 3.

6.1 suEXEC and suPHP

In default Apache and PHP installations, all PHP scripts execute under a single user, the Webserver user, regardless of their path on the disk or of the owner of the file.

Two of the most popular solutions to this insecure behavior are suEXEC [3] and suPHP [4]. While the two solutions are implemented differently the end goal of their developers is the same; to have each script run under the permissions of their owners and not under the permissions of the calling Webserver. This means that the scripts of user `A` have different permissions of user `B` and thus each is protected from the other. Unfortunately, the extra security comes at a cost. Even though we couldn't locate official comparisons, individual users have benchmarked their installations and have found both solutions to incur a performance penalty ranging between 2,500% to 3,600% [7]. We believe that making a Webserver 36 times slower comes into antithesis with hosting companies' profit strategy which consists of placing as many users as possible on a single physical server. Even if we assume however that a hosting company deploys such a solution, a server-side session attack is still possible.

More specifically, Session Poisoning can still be executed on shared hosting sites that use a common session store. The only modification to the methodology presented in Sec. 3.1 is for the attacker to change the session file permissions, just before forcing it onto other Web applications. By making the file world-readable, the victim Web application is "allowed" to open the attacker's poisoned session file. On the other hand, Session Snooping is no longer possible since the attacker's scripts cannot open the session files of other Web applications and thus can't change their content.

---

[3]**suEXEC** http://httpd.apache.org/docs/current/suexec.html
[4]**suPHP** http://www.suphp.org

6.2 Suhosin patch

Suhosin[5] is a server-side protection mechanism which can, among others, protect session files by symmetrically encrypting their content. Thus, if an attacker opens a session file on a Webserver with the Suhosin mechanism, he can no longer inspect the session contents or intelligibly edit the values. The encryption/decryption key of each session consists of a global key (by default common to all scripts running on the same server) combined with one or more of the following values: the remote IP address of the current visitor, the `Document Root` of the executing script and the User-Agent of the visitor's browser. The active combination of the above values differs between installations and depends upon the installation framework and custom choices of Web administrators.

In our described attacks, the only fields that could differ between an attacker's encryption key and the encryption key of all other Web applications situated on the same server are the `Document Root` field and the global key. The remote IP address and the User-Agent of the attacker's browser will be the same towards all Web applications. This effectively means, that unless the `Document Root` encryption option is active and/or a Web application has explicitly modified the global Suhosin key, an attacker will be able to correctly encrypt and decrypt the session files of a given Web application and thus circumvent the protection provided by Suhosin. Note also that the encryption and decryption happens automatically by the PHP framework when the appropriate session management functions are called, thus the attacker doesn't need to manually perform these tasks.

6.3 PHP Safe Mode

The, since June 2009 deprecated, PHP Safe Mode[6] could be configured to mitigate the attack via restricting the `session_start()` method. However, reliance on Safe Mode features is actively discouraged and Safe Mode will be removed completely in the next major version of PHP, hence, on modern set-ups this option may not exist at all.

7. RELATED WORK

To the best of our knowledge, this paper is the first one to describe server-side data attacks against session identifiers. A similar technique is described in [12], where an attacker can browse the common session store of a shared host and use the data to perform session hijacking or expose private information. Note however, that the described attack targets the authenticated clients of a Web application, as opposed to the server-side attacks described in Sec. 3 which target the logic of the Web application itself. Additionally, a removal of read-permissions from the common session store by the administrator would stop the described attack (since directory listing is no longer feasible) but would have no impact on Session Snooping and Session Poisoning since these attacks do not rely on a list-able session store.

The client-side of session identifiers has, in general, been a common target of Web application attacks. The most standard way of hijacking a session identifier is through a Cross-site scripting attack. Cross-site scripting [15] (XSS) is a type of code injection attack, where the victim executes Javascript code on behalf of the attacker. XSS are among the top attacks conducted today against Web applications [11]. Apart from manually finding and exploiting them, Kieyzun et al. [8] have also shown that it is possible to automate the procedure.

Other common attacks against session identifiers include Cross-site Request Forgery (CSRF) [14], session fixation [9] and session sidejacking [4]. Using CSRF, a malicious website is able to generate arbitrary requests towards trusted websites taking advantage of the fact that the visiting user is authenticated to the trusted websites and his session cookies will be appended automatically by his browser. In session fixation attacks, an attacker can force a victim to use a session identifier that is already known to him. This attack can be mainly performed on websites that accept session information as a `GET` parameter. Session sidejacking, is essentially session hijacking performed through side channels such as packet sniffing on open wireless or hubbed networks.

In addition to attacks against session identifiers, the complexities and subtleties of the HTTP protocol have also given rise to other, less common attacks against Web applications. Carettoni and Di Paola [5] were among the first ones to notice that there is no formal definition of parameter precedence for cases when a URI contains two or more parameters with the same name. The lack of a formal definition has led to implementation-specific behaviors which can be exploited by attackers, to inject new values in existing URIs. The new parameter values can, among others, overwrite hard-coded parameters and corrupt both the client-side as well as the server-side logic of a Web application. This attack, namely HTTP Parameter Pollution (HPP), was recently verified by Balduzzi et al. [1] who discovered that 29.88% from 5,000 tested websites were vulnerable to it. Researchers have also discovered that different inputs can lead Web applications to different code paths which in turn respond in different timings. These differences are measurable and can lead to the exposal of private information [3].

8. CONCLUSION

Shared hosting is the most popular type of Web hosting mainly due to its low-monthly costs and easy administration. At the same time however, Web applications stored on shared hosting plans, suffer from limited performance and, more importantly, limited security.

In this paper we presented two new attacks that abuse the lack of proper isolation between session identifiers of different Web applications hosted on the same physical server. We experimentally demonstrated that most websites use a standard and global session store which an attacker in control of a domain hosted on a shared hosting server can abuse. The resulting attacks, allow a malicious user to circumvent authentication mechanisms of vulnerable Web applications, elevate his privileges, steal private information and conduct attacks that would be otherwise impossible. Finally, we tested our attacks against popular Content Management Systems and showed that they are effective even when generic server-side countermeasures are deployed.

Acknowledgements:.

We thank the anonymous reviewers for their helpful comments. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, the IBBT, the Research Fund K.U.Leuven and by the EU Project WebSand (FP7-256964).

[5] **Suhosin** http://www.hardened-php.net/suhosin

[6] **SafeMode** http://php.net/manual/features.safe-mode.php

9. REFERENCES

[1] M. Balduzzi, C. T. Gimenez, D. Balzarotti, and E. Kirda. Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications. In *NDSS'11*, 2011.

[2] T. Ballad and W. Ballad. *Securing PHP Web Applications*. Addison-Wesley Professional, 2008.

[3] A. Bortz and D. Boneh. Exposing private information by timing web applications. WWW '07. ACM, 2007.

[4] E. Butler. Firesheep. `http://codebutler.com/firesheep`.

[5] L. Carettoni and S. D. Paola. HTTP Parameter Pollution. In *OWASP AppSec Europe*, 2009.

[6] Water and Stone: Open Source CMS Market Share Report, 2010.

[7] S. Herbert. Using suPHP To Secure A Shared Server. `http://blog.stuartherbert.com/php/2008/01/18/using-suphp-to-secure-a-shared-server/`.

[8] A. Kieyzun, P. Guo, K. Jayaraman, and M. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *ICSE'09*, May 2009.

[9] M. Kolsek. Session Fixation Vulnerability in Web-based Applications. `http://www.acrossecurity.com/papers/session_fixation.pdf`.

[10] T. Krueger, C. Gehl, K. Rieck, and P. Laskov. Tokdoc: a self-healing web application firewall. SAC '10. ACM, 2010.

[11] OWASP Top 10 Web Application Security Risks. `http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

[12] PHP Security Consortium - PHP Security Guide: Shared Hosts. `http://phpsec.org/projects/guide/5.html`.

[13] W. Robertson, G. Vigna, C. Kruegel, and R. A. Kemmerer. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *NDSS'06*, 2006.

[14] C. Shiflett. Cross-Site Request Forgeries. `http://shiflett.org/articles/cross-site-request-forgeries`.

[15] The Cross-site Scripting FAQ. `http://www.cgisecurity.com/xss-faq.html`.