

Vulnerability Analysis of Web-based Applications

Summary. In the last few years, the popularity of web-based applications has grown tremendously. A number of factors have led an increasing number of organizations and individuals to rely on web-based applications to provide access to a variety of services. Today, web-based applications are routinely used in security-critical environments, such as medical, financial, and military systems.

Web-based systems are a composition of infrastructure components, such as web servers and databases, and of application-specific code, such as HTML-embedded scripts and server-side CGI programs. While the infrastructure components are usually developed by experienced programmers with solid security skills, the application-specific code is often developed under strict time constraints by programmers with little security training. As a result, vulnerable web-based applications are deployed and made available to the whole Internet, creating easily-exploitable entry points for the compromise of entire networks.

To ameliorate these security problems, it is necessary to develop tools and techniques to improve the security of web-based applications. The most effective approach would be to provide secure mechanisms that can be used by well-trained developers. Unfortunately, this is not always possible, and a second line of defense is represented by auditing the application code for possible security problems. This activity, often referred to as *web vulnerability analysis*, allows one to identify security problems in web-based applications at early stages of development and deployment.

Recently, a number of methodologies and tools have been proposed to support the assessment of the security of web-based applications. In this paper, we survey the current approaches to web vulnerability analysis and we propose a classification along two characterizing axes: detection model and analysis technique. We also present the most common attacks against web-based applications and discuss the effectiveness of certain analysis techniques in identifying specific classes of flaws.

1.1 Introduction

The World-Wide Web started in the mid 90's as a system to support hypertextual access to static information and has since then evolved into a full-fledged platform for the development of distributed applications. This has

been made possible by the introduction of a number of mechanisms that can be used to trigger the execution of code on both the client and the server side. These mechanisms are the basis to implement web-based applications.

As the use of web applications for critical services has increased, the number and sophistication of attacks against web application has grown as well. A series of characteristics of web-based applications make them a valuable target for an attacker. First, web applications are often designed to be widely accessible. Indeed, by design, they are almost always reachable through firewalls and a significant part of their functionality is available to anonymous users. Because of this, they are considered the most effective entry point for the compromise of computer networks. Second, web-based applications often interface with back-end components, such as mainframes and product databases, that might contain sensitive data, such as credit card information. Therefore, they become an attractive target for attackers who aim at gaining a financial profit. Third, the technology used to implement, test, and interact with web-based applications is inexpensive, well-known, and widely available. Therefore, attackers can easily develop tools that expose and automatically exploit vulnerabilities.

Other factors contribute to make web applications a preferred target for attackers. For example, some of the most popular languages used to develop web-based applications are currently easy enough to allow novices to start writing their own applications, but, at the same time, they do not provide a comprehensive, easy-to-use set of mechanisms that support the development of secure applications. This problem is particularly difficult to solve. In fact, while the infrastructure components, such as web servers and browsers, are usually developed by experienced programmers with solid security skills and reviewed by a large developer team, the application-specific code is often developed under strict time constraints by few programmers with little security training. As a consequence, vulnerable code is made available on the web.

This trend is confirmed by various statistics. In the first semester of 2005, Symantec cataloged 1,100 new vulnerabilities, which represent well over half of all new vulnerabilities, as affecting web-based applications. This is a 59% increase over the previous semester, and a 109% increase over the same period of the previous year [33].

An analysis of the reported vulnerabilities shows various types of problems. Web applications can be affected by flaws that are not web-specific and that have been commonly found also in traditional applications. Examples of such problems include broken authentication and authorization management, where account credentials and session tokens are not properly protected; improper handling of errors or exceptional conditions, which leads to the leaking of confidential information or to unexpected system behavior.

In addition to these well-known security problems, web-based applications are affected by a number of vulnerabilities that are specific of the web environment. Some vulnerabilities are due to architectural choices, such as the use of relational databases as back-ends for long-term storage, which lead

to vulnerabilities such as SQL injections [1, 30] and permanent Cross Site Scripting (XSS) [14]. Other causes of web-based vulnerabilities are the incorrect handling of trust relations between clients and servers, which might lead to XSS, and inconsistencies in web protocols implementations, which lead to request smuggling [20] and response splitting [15]. Another source of security problems are the unforeseen consequences of the use of special features provided by the languages used to implement web-based applications, such as the use of the `register_globals` option in the PHP language. In this paper, we will focus on vulnerabilities that are specific to the web environment.

Clearly, the abundance of vulnerabilities in web-based applications and their increasing popularity make a strong case for the need of techniques and tools for their security assessment. A number of approaches to secure web-based applications have been proposed in the recent past by both industry and academia. While most of these techniques reuse well-know ideas from the past, these ideas have to be extended to take into account the novel aspects of web-based application security. These approaches can be classified on the basis of when they can be applied in the life-cycle of a web-based application.

At the coding phase, new programming languages have been adopted that, among other things, take away from the programmer the burden of performing error-prone tasks, and, in addition, eliminate the ability to perform insecure operations commonly found in traditional languages, such as memory management and pointer arithmetic.

New testing tools and methodologies have been proposed that aim at identifying and removing flaws by exercising an instance of an application with unexpected, random, or faulty input. Testing-time approaches are appealing because, in general, they can be performed even on applications whose source code is not available. In addition, they are usually independent of the internals of the application under testing, and, therefore, they are reusable on different applications. Finally, they are characterized by the absence of false positives, i.e., flaws found through testing correspond to actual bugs in the application code. The main disadvantage of these testing approaches is their lack of completeness, that is, in general, they cannot guarantee to expose all vulnerabilities in a program.

Code reviews and security audits are part of the quality assurance phase. In particular, vulnerability analysis is the process of examining different characteristics of an application in order to evaluate whether there exists a sequence of events that are capable of driving the application into an insecure state. Thus, vulnerability analysis supports the identification and fixing of errors. In its simplest form, the analysis is performed by manually auditing the source code of an application. However, a number of more sophisticated and automatic approaches have been proposed in the last few years.

Finally, at deployment and operation time, an application can be protected through the use of web application firewalls. These applications examine the requests directed to a web server and determine if a request is to be considered an attack or not. The focus of web application firewalls is on preventing attacks

directed against a web-based application rather than identifying and fixing its errors. These security mechanisms usually do not require the understanding of an application internals or its modification.

This chapter reviews and discusses a number of techniques that can be used to perform the vulnerability analysis of web-based applications. Before delving into the details of these techniques, Sect. 1.2 presents the existing mechanisms for the execution of code in web-based applications. Then, Sect. 1.3 discusses the type of attacks that are common of web-based application. Finally, Sect. 1.4 provides a survey of the techniques used to perform vulnerability analysis of web-based applications.

1.2 Technologies

The technologies used to implement web-based applications have rapidly evolved since the appearance of the first simple mechanisms to create dynamic web pages. In this section, we will briefly present the most important steps in this evolution.

1.2.1 Common Gateway Interface

One of the first mechanisms that enabled the generation of dynamic content was the Common Gateway Interface (CGI) [23]. The CGI standard defines a mechanism that a server can use to interact with external applications. The CGI standard specifies the rules of this interaction but does not dictate the use of any particular technology for the implementation of the external applications. Therefore, CGI programs can be written virtually in any programming language and executed by virtually all web servers.

The original goal of the CGI invocation mechanism was to provide web-based access to legacy applications. In this case, a CGI program acts as a *gateway* between the web server and the legacy application, e.g., a database. More precisely, the life-cycle of a CGI-based program is as follows: whenever a request references a CGI program, the server creates a new process to execute the specified application. Then, the web server passes to the program the data associated with the user request. The CGI program executes and produces data, which is passed back to the server. The server, in turn, passes the data to the client. When the CGI program exits, the request is completed.

The CGI specification defines different ways for a web server to communicate with a CGI program. At every request, the web server sets a number of environment variables that contain information about both the server (e.g., server name and version, or CGI specification supported) and the request (e.g., request method, request content type, and length). The request itself is passed to the CGI program through its standard input (if the request is issued using the POST HTTP method) or through an environment variable (if the request is issued using the GET HTTP method). When the CGI program has

```
#!/usr/bin/perl -w
use CGI;
use CGI::Session;

$cgi = new CGI;

my $user = $cgi->param("username");
my $pass = $cgi->param("password");

if (validate($user, $pass) == 1) {
    my $session = new CGI::Session("driver:File",
        undef, {Directory=>"/tmp"});
    $cookie = $cgi->cookie(CGISESSID => $session->id);
    print $cgi->header(-cookie=>$cookie);
    print $cgi->start_html("Login");
    print "Welcome!";
} else {
    print $cgi->header;
    print $cgi->start_html("Login");
    print "Invalid username or password!";
}

print $cgi->end_html;
```

Fig. 1.1. A sample CGI program.

finished serving the request, it sends back the results to the server through its standard output. The output can either consist of a document generated by the program or one or more directives to the server for retrieving the output.

The example in Fig. 1.1 shows a CGI program written in Perl. The program allows users to authenticate with the application to access services offered only to authenticated users. Users are expected to send their credentials as two request parameters, namely `username` and `password`. The program uses two Perl modules, `CGI` and `CGI::Session`, which provide a number of library functions to ease the tasks of parameter decoding and session management. The credential parameters are extracted from the request and validated through the `validate` function (which, for example, could lookup a database of registered users). If the credentials are found to be valid, a new session is started and a welcome message is shown to the user. Otherwise, the program returns to the user an error message. Note that this is just one of the components of a web-based application. For example, there might be other components that will provide access to information or services.

Developing web-based applications using the CGI mechanism to invoke server-side components has some advantages. First, as noted before, the CGI mechanism is language-independent, that is, each CGI-based component of

the application can be implemented using a different language. Second, CGI programs run in separate processes so problems in one program do not affect other components of the web applications or the web server.

The main disadvantage of using the CGI mechanism is that it requires that a new process be created and executed for each request, with significant impact on the performance and scalability of the web-based application. On many systems (notably UNIX), the creation of a new process is a costly operation. Furthermore, the use of a separate process for each request poses a limit to the maximum number of requests that can be satisfied at the same time, which is bounded by the maximum number of processes allowed by the OS. In addition, CGI programs run in a separate address space than the web server, and, therefore, they can only have a limited interaction with it. In particular, they cannot take advantage of its services (e.g., logging) leading to a duplication of functionalities between the server and the CGI programs.

In an attempt to overcome some of these problems, several extensions to the original CGI standard have been proposed. For example, the FastCGI mechanism creates a pool of resident processes that handle multiple requests without exiting and needing to be restarted [3]. Also, the FastCGI mechanism allows the web server and the external program to communicate through Unix-domain or TCP sockets rather than pipes, allowing developers to create more sophisticated architectures where server-side components are not required to run on the same host as the web server.

A different approach to overcome these limits consists in leveraging various functionalities of the server that are exposed through vendor-specific APIs. The most notable examples are the ISAPI extensions for Microsoft IIS and the API provided by Apache. External programs that use these server-specific APIs generally have low initialization cost and can perform more general functionalities than CGI-based programs. For example, they can rely on the web server to enforce access control or even hook into the server's request handling process.

However, server-specific APIs lack some of the benefits provided by the CGI mechanism. Writing a program that relies on server-specific APIs is generally more complex than writing a CGI program, because it requires some knowledge of the server's inner workings. In addition, the APIs are typically language-specific (i.e., they have C or C++ bindings) and vendor-specific, and, thus, not portable.

1.2.2 Embedded Web Application Frameworks

Today, the most common approach to web-based application implementation is a middle way between the original CGI mechanism and the use of server-specific APIs. More precisely, the web server is provided with extensions that implement frameworks for the development of web-based applications. At a minimum, these frameworks include an interpreter or compiler for the lan-

```

<?php
$username = $_GET["username"];
$password = $_GET["password"];

if (validate($username, $password)) {
    session_start();
    $_SESSION["user"] = $username;
?>
    <p>Welcome!</p>
<?php
} else {
?>
    <p>Invalid username or password!</p>
<?php
}
?>

```

Fig. 1.2. A sample PHP program.

guage used to encode the application's components and define rules that govern the interaction between the server and the application's components.

Frameworks vary greatly in the support provided to the application developer. At their most basic level, they simply parse request parameters and make them available to the application. Some frameworks also offer mechanisms to deal with HTTP-specific characteristics, e.g., cookies, authentication mechanisms, and connection handling. Most frameworks generally support other commonly-used features, such as access to back-end databases and sessions. More sophisticated environments are designed to support large-scale enterprise applications and include support for features such as transactions and complex authorization mechanisms.

Web application frameworks are available for a variety of programming languages. Most frameworks are built around scripting languages, such as PHP, Perl, and Python. These are high-level languages, which are generally interpreted, provide support for object-oriented programming and are loosely typed. These characteristics simplify the development of small components, which is a perfect match for web-based applications. In fact, these applications are usually characterized by a number of small server-side components that perform relatively simple tasks. Other significant choices are Java, used in the J2EE platform, and the languages compatible with the ASP.NET environment, such as Visual Basic, C# and JScript.

The example in Fig. 1.2 shows a PHP-based version of the login example shown in the Sect. 1.2.1. In PHP, the parameters of requests issued through the HTTP GET method are available in the predefined `$_GET` array. Also, PHP provides native support for sessions, and, therefore, it is extremely easy to keep

```

<%
String username = request.getParameter("username");
String password = request.getParameter("password");

if (validate(username, password)) {
    session.setAttribute("user", username);
}%>
    <p>Welcome!</p>
<%
} else {
}%>
    <p>Invalid username or password!</p>
<%
}
}%>

```

Fig. 1.3. A sample JSP program.

track of a user across different requests. In the example, users credentials are first checked using the `validate` function. If the validation is successful, a new session is started and a welcome message is printed; otherwise, an error message is sent back to the user. Note that in a PHP program it is possible to interleave PHP and HTML code.

Fig. 1.3 shows a similar program written using the JavaServer Pages (JSP) framework [32]. In both examples, the code of the `validate` function has been omitted for the sake of clarity.

1.3 Attacks

Web-based applications have fallen prey to a variety of different attacks that violate different security properties that should be enforced by the application. Note that here we are not concerned with attacks that might involve the infrastructure (for example, in terms of web server and databases) or the operation of the network (for example, in terms of routers and firewalls). Instead, we focus on attacks that try to induce a web-based application to behave in unforeseen (and unwanted) ways to disclose sensitive information or execute commands on behalf of the attacker.

Many web-based applications offer services that are available only to registered users, e.g., “premium” functionalities or personalized content. These services require that some *authentication* mechanism be in place to establish the identity of users. Errors in authentication code or logic can be exploited to bypass authentication or lock out legitimate users. For example, user credentials transferred in clear to the application can be stolen by eavesdropping

the network connection and weak authentication mechanisms can be broken by brute force or dictionary attacks [6].

Once a user has been authenticated, the application has to enforce the policy that establishes which resources are available to the user. Broken *authorization* can lead to elevation of privileges, disclosure of confidential data and data tempering. Authorization mechanisms are particularly critical when web-based applications handle sensitive data, such as financial or health information.

Web-based applications tend to be large, heterogeneous, complex pieces of software, whose *configuration* is far from being trivial. Configuration problems may affect both the infrastructure (e.g., the account under which the web server runs or the configuration of a back-end database) and the web application itself (e.g., where the application stores its temporary files). Configuration errors can allow an attacker to bypass otherwise effective authentication and authorization mechanisms. For example, improper configuration has been exploited to gain unauthorized access to administrative functionalities or retrieve sensitive information, such as secrets stored as plain-text in configuration files, such as database server passwords.

Attacks that exploit poorly-designed authentication, faulty authorization or configuration mechanisms are the cause of serious compromises. However, currently, most of attacks against web applications can be ascribed to one class of vulnerabilities: *improper input validation*. Most web application developers assume that they might receive from their users incorrect input, either as a result of an error or of malicious intent. Input validation is a defensive programming technique that makes sure that all user input is in the expected format and does not contain dangerous content. While simple in principle, performing correct and complete validation of all input data is a time-consuming task that requires notable expertise. Therefore, this type of flaw is all too common in current web-based applications.

The remaining of this section explores different types of attacks that take advantage of incorrect or missing input validation.

1.3.1 Interpreter Injection

Many dynamic languages include functions to dynamically compose and interpret code. For example, the PHP language provides the `eval` function, which accepts a string as a parameter and evaluates it as PHP code. If unchecked user input is used to compose the string to be evaluated, the application is vulnerable to arbitrary code execution.

For example, consider the following simple example of interpreter injection that was present in *Double Choco Latte* (version 0.9.4.3 and earlier), a PHP web-based application that provides basic project management functionality [2]. The attack URL is of the form:

```
http://[target]/[dcl-directory]/
    main.php?menuAction=htmlTickets.show;system(id);ob_start
```

The parts of the request URL containing the strings `menuAction=html-Tickets.show` and `ob_start` are required to avoid errors. The arbitrary code is the part between these two string, and, in the example above, corresponds to the part containing `system(id)`.

The vulnerability is contained in the following code snippet:

```
if ($g_oSec->ValidateMenuAction() == true)
{
    list($class, $method) = explode(".", $menuAction);
    $obj = CreateObject('dcl.' . $class);
    eval("\$obj->$method()");
}
else
{
    commonHeader();
    PrintPermissionDenied();
}
```

As can be seen from the code above, the `class` and `method` variables, obtained from the user's controlled `menuAction` variable, are never validated. Therefore, it is possible to insert a command to be executed in the string that represents the variable's value. After requesting the attack URL, the `eval` call becomes:

```
eval("$obj->show;system(id);ob_start()");
```

Thus, in addition to execute the `show` method on the `obj` object, the interpreter will also execute the command specified by the attacker. In this example, the `id` UNIX command will be executed and the information about the user ID under which the command is executed will be printed. Of course, arbitrary (and more malicious) commands can be executed by exploiting this flaw.

One difficulty in preventing interpreter injection attacks is that popular languages offer many attack vectors. In PHP, `eval` and `preg_replace` can be used to interpret PHP code. In addition, the functions `system`, `passthru`, `backticks`, and `shell_exec` pass a command to the system shell. Finally, `exec`, `pcntl_exec`, `popen`, and `proc_open` can be used to execute external programs.

Some languages offer natively *sanitization* primitives that ensure that malicious user input is properly removed before use. For example, in PHP, `escapeshellarg` and `escapeshellcmd` can be used to escape quotes or other special characters that might be inserted to trick an application into executing arbitrary commands. However, programmers must be aware of the problem, choose the proper sanitization function, and remember to invoke it on all possible code paths that lead to an invocation of a dangerous function. This requires substantial expertise, and might be foiled by subsequent reorganizations of the code.

1.3.2 Filename Injection

Most languages used in the development of web-based applications allow programmers to dynamically include files to either interpret their content or present them to the user. This feature is used, for example, to modularize an application by separating common functions into different files or to generate different page content depending on user's preferences, e.g., for internationalization purposes. If the choice of the file to be included can be manipulated by the user, a number of unintended consequences can follow. To worsen the situation, some languages, most notably PHP, even supports the inclusion of files from remote sites.

The following snippet of code illustrates a filename injection vulnerability in *txtForum*, an application to build forums [11]. In *txtForum*, pages are divided in parts, e.g., header, footer, forum view, etc., and can be customized by using different “skins,” which are different combination of colors, fonts, and other presentation parameters. For example, the code that defines the header is the following:

```
DEFINE("SKIN", "$skin");
...
function
t_header($h_title, $pre_skin='', $admin_bgcolor='') {
    ...
    include(SKIN.'/header.tpl');
}
```

During execution, each page is composed by simply invoking the functions that are responsible of creating the various parts, e.g., `t_header("page title")`. Unfortunately, the `skin` variable can be controlled by an attacker, who can set it to cause the inclusion and evaluation of arbitrary content. Because PHP allows for the inclusion of remote files, the code to be added to the application can be hosted on a site under the attacker's control. For example, requesting the `login.php` page and passing the parameter `skin` with value `http://[attacker-site]` leads to the execution of the code at `http://[attacker-site]/header.tpl`.

For this type of problem, PHP does not offer any sanitization methods natively. Therefore, appropriate, *ad hoc* checks must be put in place by the developers.

1.3.3 Cross-site Scripting

In Cross-site Scripting (XSS) attack, an attacker forces a client, typically a web browser, to execute attacker-supplied executable code, typically JavaScript code, which runs in the context of a trusted web site [14].

This attack allows the perpetrator to bypass the *same-origin* policy enforced by browsers when executing client-side code, typically JavaScript. The

same-origin policy states that scripts or documents loaded from one site cannot get or set the properties of documents from different sites (that is, from different “origins”). This prevents, for instance, a malicious web application from stealing sensitive information, such as cookies containing authentication information, associated with other applications running on different sites.

However, the same-origin policy can be circumvented, under certain conditions, when an application does not perform correct input validation. In these cases, the vulnerable application can be tricked into storing malicious code from an attacker and then presenting that malicious code to users, so it will be executed under the assumption that it originates from the vulnerable application rather than from the attacker.

There exist different forms of XSS attacks, depending on how malicious code is submitted to the vulnerable application and later echoed from the application to its users. In *non-persistent* (or *reflected*) attacks, the user is lured into visiting a specially-crafted link that points to the vulnerable application and embeds the malicious code (e.g., as the value of a parameter or the name of a resource). When the link is activated the vulnerable web application immediately reflects the code to the user (e.g., as part of an error message). The code is then executed in the context of the vulnerable site and has access to all the information associated with the attacked application, such as authentication cookie or session information.

In *persistent* (or *stored*) attacks, the malicious code is first stored by the vulnerable application, and then, at a later time, it is presented to its users. In this case, the security of a user is compromised each time he/she visits a page whose content is determined using the stored malicious code. Typical examples of vulnerable applications include guestbook applications or blog systems. If they allow users to submit entries containing scripting code, then they are vulnerable to persistent XSS attacks.

A third form of XSS attacks, called *DOM-based*, is also possible. In this case, the vulnerable application presents to the users an HTML page that uses data from parts of its Document Object Model (DOM) in insecure ways [16]. The DOM is a data structure that allows client-side scripting code to dynamically access and modify the content, structure, and style of HTML documents. Some of its properties are populated by the browser on the basis of the request parameters, rather than on the characteristics of the document itself. For example, the `document.URL` and `document.location` properties are set to the URL of the document by the browser. If an HTML page contains code that dynamically changes the appearance of the page using the content of `document.URL` (e.g., to show to the user the URL associated with the page), it is possible to use a maliciously crafted URL to execute malicious scripting code.

An example of code vulnerable to non-persistent XSS attacks could be found in the application *PHP Advanced Transfer Manager* (version 1.30 and earlier) [28]. The vulnerability is contained in the following snippet of code.

```
$font = $_GET['font'];
...
echo "<font face=\"$font\" color=\"$normalfontcolor\"
      size=\"1\">\n";
```

The variable `$font` is under the control of the attacker because it is extracted from the request parameters and it is used to create the web page returned to the user, without any sanitizing check. To exploit this vulnerability an attacker might request the following URL:

```
http://[target]/[path]/viewers/txt.php?font=
  \%22\%3E\%3Cscript\%3Ealert(document.cookie)\%3C/script\%3E
```

As a consequence, the vulnerable application will generate the following web page:

```
<font face=""><script>alert(document.cookie)</script>
```

When interpreted by the browser, the scripting code will be executed and it will show in a pop-up window the cookies associated with the current page. Clearly, a real attack would, for instance, send the cookies to the attacker.

1.3.4 SQL Injection

A web-based application has an SQL injection vulnerability when it uses unsanitized user data to compose queries that are later passed to a relational database for evaluation [1, 30]. This can lead to arbitrary queries being executed on the database with the privileges of the vulnerable application.

For example, consider the following code snippet:

```
$activate = $_GET["activate"];
$result = dbquery("SELECT * FROM new_users " .
                  "WHERE user_code='$activate'");
if ($result) {
    ...
}
```

The `dbquery` function is used to perform a query to a back-end database and return the results to the application. The query is dynamically composed by collating a static string with a user-provided parameter. In this case, the `activate` variable is set to the content of the homonymous request parameter. The intended use of the variable is to contain the user's personal code to dynamically compose the page's content. However, if an attacker submits a request where the `activate` parameter is set to the string `' OR 1=1 --` the query will return the content of the entire `new_users` table. If the result of the query is later used as the page content, this will expose personal information. Other attacks, such as the deletion of database tables or the addition of new users, are also possible.

1.3.5 Session Hijacking

Most web applications use HTTP as their communication protocol [5]. HTTP is a stateless protocol, i.e., there is no built-in mechanism that allows an application to maintain state throughout a series of requests issued by the same user. However, virtually all non-trivial applications need a way to correlate the current request with the history of previous requests, i.e., they need a “session” view of their interaction with users. In e-commerce sites, for example, a user adds to a cart items he/she intends to buy and later proceeds to the checkout. Even though these operations are performed in separated requests, the application has to keep the state of the user’s cart through all requests, so that the cart can be displayed to the user at checkout-time.

Consequently, a number of mechanisms have been introduced to provide applications with the abstraction of sessions. Some languages provide session-like mechanisms at the language level, others rely on special libraries. In other cases, session management has to be implemented at the application level.

The session state can be maintained in different ways. It can be encoded in a document transmitted to the user in a way that will guarantee that the information is sent back as part of later requests. For example, HTML hidden form fields can be used for this purpose. These fields are not showed to the user, but when the user submits the form, the hidden variables are sent back to the application as part of the form’s data. In our e-commerce example, the application might keep the current sub-total of the transaction in a hidden field. When the user chooses the shipping method, the field is returned to the application and used to calculate the final total cost.

The state can be kept in cookies sent to a user’s browser and automatically resent by the browser to the application at subsequent visits. Cookies might contain the items currently inserted in a user’s cart. The application, during checkout, looks up the price of each item and presents the total cost to the user.

All the methods mentioned above require the client to cooperate with the application to store the session state. A different approach consists in storing the state of all sessions on the server. Therefore, each user is assigned a unique session ID, and this is the only information that is sent back and forth between the application and the user, e.g., by means of a cookie, or of a similar mechanism that rewrites all the URLs in the page adding the session identifier as a parameter. As a consequence, every future request will include the session identifier as a parameter. Then, whenever the user submits a request to the site, e.g., to add an item to the cart, the application receives the session ID, looks up the associated session in its repository, and updates the session’s data according to the request.

A number of attacks have been designed against session state management mechanisms. Approaches that require clients to keep the state assume that the client will not change the session state, for example by modifying the hidden field (or the cookie) storing the current sub-total to lower the price of

an item. Countermeasures include the use of cryptographic techniques to sign parameters and cookies to make them tamper-resistant.

A more general attack is *session fixation*. Session fixation forces a user's session ID to an explicit value of the attacker's choice [17]. The attack requires three steps. First, the attacker sets up a session with the target application and obtains a session ID. Then, the attacker lures the victim into accessing the target application using the fixed session ID. Finally, the attacker waits until the victim has successfully performed all the required authentication and authorization operations and then impersonates the victim by using the fixed session ID. Depending on the characteristics of the target web applications, different methods can be used to fix the session ID. In the simplest case, an attacker can simply lure the users into selecting a link that contains a request to the application with a parameter that specifies the session ID, such as ``.

1.3.6 Response Splitting

HTTP response splitting is an attack in which the attacker is able to set the value of an HTTP header field such that the resulting response stream is interpreted by the attack target as two responses rather than one [15]. Response splitting is an instance of a more general category of attacks that take advantage of discrepancies in parsing when two or more devices or entities process the data flow between a server and a client.

To perform response splitting the attacker must be able to inject data containing the header termination characters and the beginning of a second header. This is usually possible when user's data is used (unsanitized) to determine the value of an HTTP header. These conditions are commonly met in situations where web applications need to redirect users, e.g., after the login process. The redirection, in fact, is generally performed by sending to the user a response with appropriately-set `Location` or `Refresh` headers.

The following example shows part of a JSP page that is vulnerable to response splitting attack:

```
<%
response.sendRedirect("/by_lang.jsp?lang=" +
    request.getParameter("lang"));
%>
```

When the page is invoked, the request parameter `lang` is used to determine the redirect target. In the normal case, the user will pass a string representing the preferred language, say `en_US`. In this case, the JSP application generates a response containing the header:

```
Location: http://vulnerable.com/by\_lang.jsp?lang=en\_US.
```

However, consider the case where an attacker submits a request where `lang` is set to the following string:

```

dummy%0d%0a
Content-Length:%200
%0d%0a%0d%0a
HTTP/1.1%20200%200K%0d%0a
Content-Type:%20text/html%0d%0a
Content-Length:%2019%0d%0a%0d%0a
<html>New document</html>

```

The generated response will now contain multiple copies of the headers `Content-Length` and `Content-Type`, namely, those injected by the attacker and the ones inserted by the application. As a consequence, depending on implementation details, intermediate servers and clients may interpret the response as containing two documents: the original one and the document forged by the attacker.

Use cases of the attack most often mention web cache poisoning. In fact, if a caching proxy server interprets the response stream as containing two documents and associates the second one, forged by the attacker, with the original request, then an attacker would be able to insert in the cache of the proxy a page of his/her choice in association to a URL in the vulnerable application.

Recently, support to contrast response splitting has been introduced in some languages, most notably PHP. In the remaining cases, the programmer is responsible to properly sanitize data used to construct response headers.

1.4 Vulnerability Analysis

The term *vulnerability analysis* refers to the process of assessing the security of an application through auditing of either the application's code or the application's behavior for possible security problems. In this section, we survey current approaches to vulnerability analysis of web-based applications and classify them along two characterizing axes: *detection model* and *analysis technique*. We show how existing vulnerability analysis techniques are extended to address the specific characteristics of web application security, in terms of both technologies (as seen in Sect. 1.2) and types of attacks (as shown in Sect. 1.3).

The identification of vulnerabilities in web applications can be performed following one of two orthogonal detection approaches: the *negative* (or vulnerability-based) approach and the *positive* (or behavior-based) approach.

In the negative approach, the analysis process first builds abstract models of known vulnerabilities (e.g., by encoding expert knowledge) and then matches the models against web-based applications, to identify instances of the modeled vulnerabilities. In the positive approach, the analysis process first builds models of the “normal,” or expected, behavior of an application (for example, using machine-learning techniques) and then uses these models to

analyze the application behavior to identify any abnormality that might be caused by a security violation.

Regardless of whether a positive or negative detection approach is followed, there are two fundamental analysis techniques that can be used to analyze the security of web applications: *static analysis* and *dynamic analysis*.

Static analysis provides a set of pre-execution techniques for predicting dynamic properties of the analyzed program. One of the main advantages of static analysis is that it does not require the application to be deployed and executed. Since static analysis can take into account all possible inputs to the application by leveraging data abstraction techniques, it has the potential to be sound, that is, it will not produce any false negatives. In addition, static analysis techniques have no impact on the performance of the actual application because they are applied before execution. Unfortunately, a number of fundamental static analysis problems, such as *may alias* and *must alias*, are either undecidable or uncomputable. Consequently, the results obtained via static analysis are usually only a safe and computable approximation of actual application behavior. As a result, static analysis techniques usually are not complete and suffer from false positives, that is, these techniques often flag as vulnerable parts of an application that do not contain flaws.

Dynamic analysis, on the other hand, consists of a series of checks to detect vulnerabilities and prevent attacks at run-time. Since the analysis is done on a “live” application, it is less prone to false positives. However, it can suffer from false negatives, since only a subset of possible input values is usually processed by the application and not all vulnerable execution paths are exercised.

In practice, hybrid approaches, which mix both static and dynamic techniques, are frequently used to combine the strengths and minimize the limitations of the two approaches. Since many of the approaches described hereinafter are hybrid, in the context of this chapter, we will use the term *static techniques* to signify that the detection of vulnerabilities/attacks is done based on some information derived at pre-execution time and the term *dynamic techniques* when the detection is done based on dynamically-acquired data. We will use the *positive vs. negative* approach dichotomy as our main taxonomy when describing current research in security analysis of web-based applications.

This section is structured as follows. Sections 1.4.1 and 1.4.2 discuss negative and positive approaches, respectively. Each section is further divided into subsections covering static and dynamic techniques. Section 1.4.3 summarizes the challenges in the security analysis of web-based applications and proposes directions for future work.

1.4.1 Negative Approaches

In the context of the vulnerability analysis of web-based applications, we define as the *negative approaches* those approaches that use characteristics of known security vulnerabilities and their underlying causes to find security

flaws in web-based applications. More specifically, known vulnerabilities are first modeled, often implicitly, and then applications are checked for instances of such models. For example, one model for the *SQL Injection* vulnerability in PHP applications can be defined as “untrusted user input containing SQL commands is passed to an SQL database through a call to `mysql_query()`.”

The vast majority of negative approaches to web vulnerability analysis are based on the assumption that web-specific vulnerabilities are the result of insecure data flow in applications. That is, most models attempt to identify when untrusted user input propagates to security-critical functions without being properly checked and sanitized.

As a result, the analysis is often approached as a *taint propagation* problem. In taint-based analysis, data originated from the user input is marked as tainted and its propagation throughout the program is traced (either statically or dynamically) to check whether it can reach security-critical program points.

When taint propagation analysis is used, models of known vulnerabilities are often built implicitly and are simply expressed in the form of the analysis performed. For example, the models are often expressed by specifying the following two classes of objects:

1. a set of possible sources of untrusted input (such as variables or function calls);
2. a set of functions, often called *sinks*, whose input parameters have to be checked for malicious values.

To track the flow of data from sources in (1) to sinks in (2), the type system of the given language is extended with at least two new types: **tainted** and **untainted**. In addition, the analysis has to provide a mechanism to represent transitions from tainted to untainted, and vice-versa. Usually, such transitions are identified using a set of technique-specific heuristics. For example, tainted data can become untainted if it is passed to some known sanitization routine. However, modeling sanitization is a very complex task, and, therefore, some approaches simply extend the language with additional type operations, such as `untaint()` and require programmers to explicitly execute these operations to untaint the data.

In the following two sub-sections, we explore in greater details how negative approaches are applied, both statically and dynamically, to the vulnerability analysis of web-based applications.

Static Techniques

All of the works described in this section use standard *static analysis* techniques to identify vulnerabilities in web-based applications. Despite the fact that many of the static analysis problems have been proven to be undecidable, or at least uncomputable, this type of analysis is still an attractive approach

for a number reasons. In particular, static analysis can be applied to applications before the deployment phase, and, unlike dynamic analysis, static analysis usually does not require modification of the deployment environment, which might introduce overhead and also pose a threat to the stability of the application. Therefore, static analysis is especially suitable for the web applications domain, where the deployment of vulnerable applications or the execution in an unstable environment can result in a substantial business cost.

As a result, there is much recent work that explores the application of static analysis techniques to the domain of web-based applications. The current focus of the researchers in this field is mostly on the analysis of applications written in PHP [10, 12, 13, 37] and Java [9, 21]. This phenomenon can be explained by the growing popularity of both languages. For example, the popularity of PHP has grown tremendously over the last five years, making PHP one of the most commonly used languages on the Web. According to the Netcraft Survey [24], about 21,000,000 sites were using PHP in March of 2006 compared to about 1,500,000 sites in March of 2000. In the monthly Security Space Reports [29], PHP has constantly been rated as the most popular Apache module over the last years. In the Programming Community Index report published monthly by TIOBE Software [34], Java and PHP are consistently rated in the top five most popular programming languages around the world.

A tool named WebSSARI [10] is one of the first works that applies taint propagation analysis to finding security vulnerabilities in PHP. WebSSARI targets three specific types of vulnerabilities: cross-site scripting, SQL injection, and general script injection. The tool uses flow-sensitive, intra-procedural analysis based on a lattice model and tpestate. In particular, the PHP language is extended with two *type-qualifiers*, namely `tainted` and `untainted`, and the tool keeps track of the type-state of variables. The tool uses three user-provided files, called *prelude files*: a file with preconditions to all sensitive functions (i.e., the sinks), a file with postconditions for known sanitization functions, and a file specifying all possible sources of untrusted input. In order to untaint the tainted data, the data has to be processed by a sanitization routine or cast to a safe type. When the tool determines that tainted data reaches sensitive functions, it automatically inserts *run-time guards*, or sanitization routines.

The WebSSARI tool is not publicly available and the paper does not provide enough implementation details to draw definitive conclusions about the tool's behavior. However, from the information available, one can deduce that WebSSARI has at least the following weaknesses. First of all, the analysis performed seems to be intra-procedural only. Secondly, to remain sound, all dynamic variables, arrays, and other complex data structures, which are commonly used in scripting languages, are considered tainted. This should greatly reduce the precision of the analysis. Also, WebSSARI provides only a limited support for identifying and modeling sanitization routines: sanitization done through the use of regular expressions is not supported.

A more recent work by Xie and Aiken [37] uses intra-block, intra-procedural, and inter-procedural analysis to find SQL injection vulnerabilities in PHP code. This approach uses *symbolic execution* to model the effect of statements inside the basic blocks of intra-procedural Control Flow Graphs (CFGs). The resulting *block summary* is then used for intra-procedural analysis, where a standard reachability analysis is used to obtain a *function summary*. Along with other information, each block summary contains a set of locations that were tainted in the given block. The block summaries are composed to generate the function summary, which contains the pre- and post-conditions of the function. The preconditions for the function contain a derived set of memory locations that have to be sanitized before function invocation, while the postconditions contain the set of parameters and global variables that are sanitized inside the function. To model the effects of sanitization routines, the approach uses a programmer-provided set of possible sanitization routines, considers certain forms of casting as a sanitization process, and, in addition, it keeps a database of sanitizing regular expressions, whose effects are specified by the programmer.

The approach proposed by Xie and Aiken has a number of advantages comparing to WebSSARI. First of all, it is able to give a more precise analysis due to the use of inter-procedural analysis. Secondly, their analysis technique is able to derive preconditions for some functions automatically. Also, the Xie and Aiken approach provides support for arrays, commonly-used data structures in PHP, in the presence of simple aliases. However, they only simulate a subset of PHP constructs that they believe is relevant to SQL injection vulnerabilities. In addition, there seems to be no support for the object-oriented features of PHP, and the modeling of the effects of many sanitization routines still depends on manual specification.

One of the most recent works on applying taint-propagation analysis for security assessment of applications written in Java is the work by Livshits and Lam [21]. They apply a scalable and precise points-to analysis to discover a number of web-specific vulnerabilities, such as SQL injection, cross-site scripting, and HTTP response splitting. The proposed approach uses a context-sensitive (but flow-insensitive) Java points-to analysis based on binary decision diagrams (BDDs) developed by Whaley and Lam [36]. The analysis is performed on the bytecode-level image of the program and a program query language (PQL) is used to describe the vulnerabilities to be identified.

The main problem with the Livshits and Lam’s approach is the fact that each vulnerability that can be detected by their tool has to be manually described in PQL. Therefore, previously unknown vulnerabilities cannot be detected and the detection of known vulnerabilities is only as good as their specification.

Even though static analysis has a number of desirable characteristics that make it suitable for web vulnerability analysis, it also has a number of both inherent and domain-specific challenges that have to be met to be able to apply it to real-world applications in an effective way. First of all, static anal-

ysis heavily depends on language-specific parsers that are built based on a language grammar. While this is not generally a problem for general-purpose languages, such as Java and C, grammars for some scripting languages, like PHP, might not be explicitly defined or might need some workarounds to be able to generate valid parsers.

More importantly, many web applications are written in dynamic scripting languages that facilitate the use of complex data structures, such as arrays and hash structures using non-literal indices. Moreover, the problems associated with alias analysis and the analysis of object-oriented code are exacerbated in scripting languages, which provide support for dynamic typing, dynamic code inclusion, arbitrary code evaluation (for example, `eval()` in PHP), and dynamic variable naming (for example, `$$` in PHP). Some of these challenges are described in greater details in the recent research work of Jovanovic et al. who developed a static analysis tool for PHP, called Pixy [12], and implemented new precise alias analysis algorithms [13] targeting the specifics of the PHP language.

Other solutions and workarounds to these challenges include different techniques, such as abstraction of language features or simplification of the analysis, and result in different levels of precision of the analysis. For example, the WebSSARI tool chooses to ignore all complex language structures by simply considering them tainted. The tool proposed by Xie and Aiken models only a subset of PHP language that is believed to be relevant to the targeted class of vulnerabilities. Lam and Livshits, on the other hand, apply scalable points-to analysis to the full Java language, but choose to abstract away from flow sensitivity.

Precise evaluation of sanitization routines becomes even more difficult for applications written in scripting languages. Dynamic languages features stimulate programmers to extensively use regular expression and dynamic type casting to sanitize user data. Unfortunately, it is not possible to simply consider the process of matching a regular expression against tainted data as a form of sanitization, if the analysis has to be sound. To increase the precision of the analysis, it is necessary to provide a more detailed characterization of the filtering performed by the regular expression matching process.

One of the main drawbacks of static analysis in general is its susceptibility to false positives caused by inevitable analysis imprecisions. Researches only started exploring the benefits of applying traditional static analysis techniques, such as symbolic execution and points-to analysis, to the domain of web-based applications. However, the first efforts in this direction clearly show that web-based applications have their domain-specific additional complexities, which require novel static analysis techniques.

Dynamic Techniques

The dynamic negative approach technique is also based on taint analysis. As for the static case, untrusted sources, sensitive sinks, and the ways in

which tainting propagates need to be modeled. However, instead of running the analysis on the source code of an application, either the interpreter or the program itself are first extended/instrumented to collect the right information and then the tainted data is tracked and analyzed as the application executes.

Perl's *Taint mode* [27] is one of the best-known example of dynamic taint propagation mechanism. When the Perl interpreter is invoked with the `-T` option, it makes sure that no data obtained from the outside environment (such as user input, environment variables, calls to specific functions, etc.) can be used in security critical functions (commands that invoke sub-shell, modify files, etc.). Even though this mode can be considered too conservative because it can taint data that might not be tainted in reality¹, it is a valuable security protection against several of the attacks described in Sect. 1.3.

Unsurprisingly, approaches similar to Perl taint mode have been applied to other languages as well. For example, Nguyen-Tuong et al. [25] propose modification of the PHP interpreter to dynamically track tainted data in PHP programs. Halder et al. [8] apply a similar approach to the Java Virtual Machine (JVM).

The approach followed by Nguyen-Tuong et al. modifies the standard PHP interpreter to identify data originated from untrusted sources in order to prevent command injection and cross-site scripting attacks. In the modified interpreter, strings are tainted at the granularity of the single character and tainting is propagated across assignments, compositions, and function calls. Also, the source of taintedness, such as the parameters of a GET method and the cookies associated with a request, is kept associated with each tainted string. Such precision comes at a price, and even though the authors report a low average overhead, the overhead of run-time taint tracking sometimes reaches 77%. Besides the possible high overhead, the proposed solution has the additional disadvantage that the only way to untaint a tainted string is to explicitly call a newly-defined `untaint` routine, which requires manual modification of legacy code. In addition, deciding when and where to untaint a string is an error-prone activity that requires security expertise.

The approach proposed by Halder et al. implements a taint propagation framework for an arbitrary JVM by using Java bytecode instrumentation. In the framework, system classes like `java.lang.String` and `java.lang.StringBuffer` are instrumented to propagate taintedness. This instrumentation has to be done off-line because no modification of system classes is allowed at run- or load-time by the JVM. All other classes are instrumented at loading time. Tainted data can be untainted by passing the data to one of the methods of the `java.lang.String` class that performs some kind of checking or matching operations.

The dynamic approach to the taint propagation problem has some advantages over static analysis. First of all, a modified interpreter can be transpar-

¹ For example, Perl considers any sub-expression of tainted expressions to be tainted as well.

ently applied to all deployed applications. Even more important, no complex analysis framework for features such as alias analysis is required, because all the required information is available as the result of program execution.

However, there are some inherent disadvantages of this approach as well. As noted earlier, the analysis is only performed on executed paths and does not give any guarantee about paths not covered during a given execution. This is not a problem if the modified interpreter is used in production versions of the application, but provides no guarantees of security if the dynamic analysis framework is used in test versions only.

Another problem associated with the use of dynamic techniques is the possible impact on application functionality. More precisely, dynamic checks might result in the termination or blocking of a dangerous statement, which, in turn, might have the side-effect of halting the application or blocking it in the middle of a transaction. Also, any error in the modifications performed on the interpreter or in the instrumentation code can have an impact on application stability and might not be acceptable in some production systems.

More importantly, despite the fact that dynamic analysis has the potential of being more precise, it can still suffer from both false positives and false negatives. If taint propagation is done in an overly conservative way, safe data can still be considered tainted and lead to a high false positive rate. Imprecisions in the modeling of untainting operations, on the other hand, can lead to false negatives. Unfortunately, in either case, the increased precision comes at the price of increased overhead and worse run-time performance.

Summary

As we have shown, many known classes of web-specific vulnerabilities are the result of improper or insufficient input validation and can be tackled as a taint propagation problem. Taint propagation analysis can be done either statically or dynamically, and, depending on the approach taken, it has both strengths and weaknesses. In particular, if it is done statically, the precision of the analysis highly depends on the ability of dealing with the complexities of dynamic features. Precise evaluation of sanitization routines is especially important, and none of the current approaches is able to deal with this aspect effectively. If taint propagation analysis is done dynamically, on the other hand, issues of analysis completeness, application stability and performance arise.

Regardless of the approach taken, taint propagation analysis depends on the correct identification of the sets of untrusted sources and sensitive sinks. Any error in identifying these sets can lead to incorrect results. Currently, there is no known fully automated way to derive these sets, and at least some sources and sinks have to be specified manually. The other challenge, which is common to all taint propagation based approaches, is how to safely untaint previously tainted data to decrease the number of false positives. In many

cases, this becomes a problem of precise sanitization identification, evaluation, or modeling.

However, taint propagation analysis is not the only possible negative approach to vulnerability analysis of web-based applications. For example, Minamide [22] proposes another approach to static detection of cross-site scripting attacks in PHP applications. The PHP string analyzer developed by Minamide approximates the output of PHP applications and constructs a context-free grammar for the output language. This grammar is then statically checked against user-provided description of unsafe strings.

For example, a user can describe the cross-site scripting vulnerability as the regular expression “.*<script>.*”. In this case, if the `script` tag is contained in the output language of an application, the application will be marked as vulnerable to cross-site scripting. As originally presented by Minamide, this approach cannot be applied to check for cross-site scripting vulnerabilities in real-world applications, because of its high false positive rate. For example, all applications that are designed to generate JavaScript code would be considered vulnerable. Since cross-site scripting is in the class of vulnerabilities caused by improper handling of user input, some mechanism to identify user input in program-produced output is needed.

In general, negative approaches rely on the knowledge of causes and manifestations of different types of vulnerabilities. Their main disadvantage is that the analyzers developed for a particular set of vulnerabilities might not be able to recognize previously-unknown classes of vulnerabilities. Nonetheless, currently, this is the most adopted approach because many vulnerabilities, both known and newly discovered, are caused by the same type of problems, such as insufficient input validation. As a result, the same analysis techniques can be effectively applied to detect a wide range of vulnerabilities.

1.4.2 Positive Approaches

In the *positive approaches* to the identification of vulnerabilities in web-based applications, the analysis is based on inferred or derived models of the “normal” application behavior. These models are then used, usually at run-time, to verify if the dynamic application behavior conforms to the established models, in the assumption that: 1) deviations are manifestations of attacks or vulnerabilities; and 2) attacks create an anomalous manifestation.

Models are built either statically, using some form of analysis done at pre-execution time, or dynamically, as a result of analysis of dynamic application behavior. Detection of attacks (or vulnerabilities) is almost always done at run-time, and, thus, most approaches are not purely static or dynamic in the traditional sense, but should be considered hybrid. In the context of this section, we will classify the approaches as *static* if models are built prior to program execution, and as *dynamic* otherwise.

Static Techniques

Static models of expected application behavior are usually derived either automatically, by means of traditional static analysis techniques, or analytically, by deducing a set of rules that must hold during program execution. Usually, models are not concerned with all aspects of application behavior, but instead they focus on specific application properties that are relative to specific types of attacks/vulnerabilities.

A good example of the static, positive approach is the work of Halfond and Orso, whose tool is called AMNESIA [9]. AMNESIA is particularly concerned with detecting and preventing SQL injection attacks for Java-based applications. During the static analysis part, the tool builds a conservative model of expected SQL queries. Then, at run-time, dynamically-generated queries are checked against the derived model to identify instances that violate the intended structure of a query. AMNESIA uses the Java String Analysis (JSA) [4], a static analysis technique, to build an automata-based model of the set of legitimate strings that a program can produce at given points in the code. AMNESIA also leverages the approach proposed by Gould, Su, and Devanbu [7] to statically check type correctness of dynamically-generated SQL queries.

More precisely, Halfond and Orso define an SQL injection as the attack in which the logic or semantics of a legitimate SQL statement is changed due to malicious injection of new SQL keywords or operators. Thus, to detect such attacks, the semantics of dynamically-generated queries must be checked against a derived model that represents the intended semantics of the query.

AMNESIA builds a Non-Deterministic Finite Automata (NFA) model of possible string expressions for each program point where SQL queries are generated. The derived character-level NFA is then simplified through string abstraction. The resulting model represents the structure of the legitimate SQL query and consists of SQL tokens intermixed with a place-holder, which is used to denote any string other than SQL tokens. To detect SQL injection attacks at run-time, the web-based application is instrumented with calls to a monitor that checks if the queries generated at run-time respect the abstract query structure derived statically.

The approach proposed by Halfond and Orso is based on the following two assumptions. First of all, they assume that the source code of the program contains enough information to build models of legitimate queries. It can be argued that this is usually the case with most applications. The second assumption, stated also by the authors, is that the SQL injection attack must violate the derived model in order to be detected. This is generally a safe assumption given that models are able to distinguish between SQL tokens and other strings. However, AMNESIA will generate false positives if an application allows user input to contain SQL keywords. The authors argue that this does not represent a real problem because usually only database-administration tools perform such queries.

The work by Su and Wassermann [31] is another example of positive approach that targets *injection attacks*, such as XSS, XPath injection, and shell injection attacks. However, the current implementation, called *SqlCheck* is designed to detect SQL injection attacks only. The approach works by tracking substrings from user input through program execution. The tracking is implemented by augmenting the string with special characters, which mark the start and the end of each substring. Then, dynamically-generated queries are intercepted and checked by a modified SQL parser. Using the meta-information provided by the substring markers, the parser is able to determine if the query syntax is modified by the substring derived from user input, and, in that case, it blocks the query.

Unlike in AMNESIA, in *SqlCheck* the model of application-specific legitimate SQL queries is built somewhat implicitly at pre-execution time and is expressed in the form of an *augmented grammar*. The observation made by the authors is that any non-malicious SQL query should have a node whose descendants comprise the entire input substring. These syntactically-correct queries are modeled by introducing additional rules into the augmented SQL grammar. For example, if characters \lll and \ggg are used to mark the start and the end of user input strings and the augmented SQL grammar has a production rule `value ::= \lll id \ggg` , then an entire user input substring covered by the subtree of the `value` node is considered non-malicious even if it contains SQL keywords. Thus, SQL grammar productions are used to implicitly specify which non-terminals are allowed to be roots of user input substrings.

The approach proposed by Su and Wassermann has one advantage over other approaches that have been shown so far. Since it works with the output language grammar (i.e., the SQL grammar), it does not require any analysis of the application source code, and, therefore, the tool can be potentially applied to applications written in different languages. However, the approach requires that the application code marks user input with meta-characters, which have to be inserted into the application either manually by the programmers or automatically as a result of some form of static analysis. In addition, from the published research, it is not clear whether or not the augmented SQL grammar has to be redefined for each tested application based on knowledge of the type of queries generated by that application.

One disadvantage that both the approaches described above have in common is the fact that detection of attacks or vulnerabilities can only be done at run-time. As a result, any error in model construction can result in undesired side effects, such as undetected application compromises or the blocking of valid queries.

To the best of our knowledge, in the web applications domain, the positive approach so far has only been applied to the detection of SQL injection attacks. However, this approach has the potential of being applied to a wider range of attacks resulting from insecure input handling by an application, such as cross-site scripting and interpreter injection attacks. More important, unlike the taint propagation analysis approaches described in Sect. 1.4.1, positive

approaches have the potential for being used to detect attacks that exploit logical errors in applications, such as attacks exploring insufficient authentication, authorization, and session management mechanisms.

Dynamic Techniques

Positive approaches based on dynamic information build models of expected behavior of an application by analyzing the application's execution profiles associated with attack-free input. In other words, the application's behavior is monitored during normal operation, and then the profiles are derived on the basis of the collected meta-information such as log files or system call traces. After the models have been established, the run-time behavior of an application is compared to the established models, to identify discrepancies that might represent evidence of malicious activity.

Traditionally, this approach has been applied to the area of learning-based anomaly detection systems. An example of the application of this approach to web-based application is represented by the work of Kruegel and Vigna [18]. In this case, an anomaly detection system utilizes a number of statistical models to identify anomalous events in a set of web requests that use parameters to pass values to the server-side components of a web-based application.

The anomaly detection system operates on the URLs extracted from successful web requests. The set of URLs is further partitioned into subsets corresponding to each component of the web-based application (for example, each PHP file). The anomaly detector processes each subset of queries independently, associating models with each of the parameters used to pass input values to a specific component of the web-based application.

The anomaly detection models are a set of procedures used to evaluate a certain feature of a request parameter, and operate in one of two modes, learning or detection. In the learning phase, models build a profile of the "normal" characteristics of a given feature of a parameter (e.g., the normal length of values for a parameter), setting a dynamic detection threshold for the parameter. During the detection phase, models return an anomaly score for each observed example of a parameter value. This is simply a probability on the interval $[0, 1]$ indicating how probable the observed value is in relation to the established profile for that parameter (note that a score close to zero indicates a highly anomalous value). For example, there are models that characterize the normal length and the expected character distribution of string parameters, models that derive the structure of path-like parameters, and models that infer if a parameter takes only a value out of a limited set of constants [19].

Since there are generally multiple models associated with each parameter passed to a web application, a final anomaly score for an observed parameter value during the detection phase is calculated as 1 minus the weighted sum of the individual model scores. If the weighted anomaly score is greater than the detection threshold determined during the learning phase for that parameter,

the anomaly detector considers the entire request anomalous and raises an alert.

The advantage of this approach is that, in principle, it does not require any human interaction. The system is able to automatically learn the profiles that describe the normal usage of an application and then it is able to determine abnormal use of a server-side component. In addition, by following a positive approach, this technique is able to detect both known and unknown attacks. Finally, by operating on the requests sent to the server, this approach is completely language independent and therefore can be applied, without modification, to web-based application developed with any technology.

The main disadvantage of this approach is shared by all the anomaly detection systems. These systems rely on two assumptions, namely that an anomaly is evidence of malicious behavior and that malicious behavior will generate an anomaly. Neither assumption is always valid. When the first assumption is violated, the system generates a false positive, that is, a normal request is blocked or identified as malicious. When the second assumption is violated, the system generates a false negative, that is, it fails to detect an attack.

Summary

Positive approaches have the advantage that, by specifying the normal, expected state of a web-based application, they can usually detect an attack whether it is part of the threat model or not. On the other hand, the concept of normality is difficult to define for certain classes of applications, and the creation of models that correctly characterize the behavior of an application still requires the use of *ad hoc* heuristics and manual work. Therefore, web vulnerability analysis systems based on the positive approach are not as popular as the ones based on the negative approach.

Another problem of these systems is that they are in general vulnerable to mimicry attacks [35]. These are attacks where a vulnerability is exploited in a way that makes its manifestation similar to what is considered to be normal usage in order to avoid detection. To counter these attacks, the models should be tightly “fit” to the application. Unfortunately, tighter models are more prone to produce false positives, and determining the right detection threshold to optimize detection and minimize errors still requires manual intervention and substantial expertise.

Finally, all known positive approaches require some form of run-time monitoring of the application behavior, and, therefore, are likely to introduce some form of overhead.

1.4.3 Challenges and Solutions

Web-based applications are complex systems, and while in the previous sections we have shown a number of approaches that attempt to make this class

of applications more secure, there are still a number of open problems, which will likely be the focus of research in the next few years.

A first general consideration is that there is no approach or technique that can be considered “the silver bullet,” under all conditions and cases. One challenge is, thus, that of combining the strengths from the various techniques and approaches that we have described so far.

Another general consideration is that there is already a corpus of work on vulnerability analysis techniques for traditional applications that can be extended to web-based applications. While some of the existing techniques can be applied to the web domain with little effort, some characteristics of web-based applications make the adaptation process difficult. For example, web-based applications implement shared, persistent state in a number of ways, such as cookies, back-end databases, etc. Modeling this state is not trivial when applying “traditional” vulnerability analysis techniques that were mostly developed for the analysis of structured languages such as C and C++.

In addition, some web-based applications have a complex interaction model and are assembled as a composition of various, heterogeneous modules, written in different languages. One challenge is thus to develop analysis techniques that are able to take into account the interaction between all the different technologies used in a web-based application. Consider for example a web-based application, in which a module, written in PHP, stores a value obtained from a user in a back-end database. This value is then retrieved by a module written in Python and used, without any sanitization, to perform a sensitive operation. In this case, the vulnerability analysis process should be able to analyze PHP, Python, and SQL code to identify the path that can bring the user-defined value to be used in a sensitive operation. Unfortunately, currently there are no techniques that are able to perform this type of analysis.

Another group of challenges is specific to the different techniques and approaches. For example, in the case of static analysis, it is necessary to include new techniques to perform more precise analysis in the context of dynamic languages. These new techniques should support object-oriented code, dynamic features of languages (e.g., \$\$ in PHP), complex data structures, etc.

Another major challenge is represented by the correct modeling of sanitization. So far, the only way to characterize sanitization in an application has been through simple heuristics. For example, if tainted data is passed to string manipulation functions or to functions that return an integer value, the data is considered “safe”. This approach is too naïve and it might lead to attacks that are able to exploit “blind spots” in the sanitization routines. Therefore, it is important to provide techniques and tools to better model sanitization operations and to assess whether a sanitization operation is appropriate for the task at hand (e.g., the sanitization necessary to prevent SQL injection is different from the sanitization required to avoid XSS attacks.)

Another set of challenges is represented by novel, web-specific attack techniques. In fact, while vulnerabilities caused by improper input validation are starting to be well-known, well-studied, and effectively detected, new vulnera-

bilities begin to surface. For example, attacks that tend to violate the intended logic of a web application cannot be easily expressed in terms of tainting. Consider, for example, a web-based application that implements an e-commerce site. A login process allows a registered user to access a catalog with links to sensitive documents. The developer assumed that the only way to access these documents is through the catalog page, which is presented to the user after the login process. Unfortunately, there is no automatic mechanism that prevents a de-registered user to simply provide the address of a sensitive document and completely bypass the authentication procedure. In this case, the attacker has not violated the logic of a web-application component. It has simply violated the implicit workflow of the application. Modeling and protecting from this types of attacks is still an open problem.

Finally, a set of challenges in the field is posed by the need to compare results between different approaches. Currently, there is no standard, accepted dataset usable as base-line for evaluation. While there exists some effort to build “standard” applications with known sets of vulnerabilities (e.g., Web-Goat [26]), there is still no consensus inside the security community on which applications to use for testing and how. As a consequence, every tool is evaluated on a different set of applications and a fair comparison of different approaches is not possible.

As web-based applications will become the prevalent way to provide services and distribute information on the Internet, the challenges described above will have to be addressed to support the development of secure applications based on web technologies.

References

1. C. Anley. Advanced SQL Injection in SQL Server Applications. Technical report, Next Generation Security Software, Ltd, 2002.
2. J. Bercegay. Double Choco Latte Vulnerabilities. http://www.gulftech.org/?node=research&article_id=00066-04082005, April 2005.
3. M. Brown. FastCGI Specification. Technical report, Open Market, Inc., 1996.
4. A. Christensen, A. Møller, and M. Schwartzbach. Precise Analysis of String Expressions. In *Proceedings of the 10th International Static Analysis Symposium (SAS'03)*, pages 1–18, May 2003.
5. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.
6. K. Fu, E. Sit, K. Smith, and N. Feamster. Dos and Don'ts of Client Authentication on the Web. In *Proceedings of the USENIX Security Symposium*, Washington, DC, August 2001.
7. C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *Proceedings of the 26th International Conference of Software Engineering (ICSE'04)*, pages 645–654, September 2004.
8. V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 303–311, December 2005.
9. W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the International Conference on Automated Software Engineering (ASE'05)*, pages 174–183, November 2005.
10. Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 12th International World Wide Web Conference (WWW'04)*, pages 40–52, May 2004.
11. N. Jovanovic. txtForum: Script Injection Vulnerability. <http://www.seclab.tuwien.ac.at/advisories/TUVSA-0603-004.txt>, March 2006.
12. N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2006.
13. N. Jovanovic, C. Kruegel, and E. Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *Proceedings of the ACM SIGPLAN*

- Workshop on Programming Languages and Analysis for Security (PLAS'06)*, June 2006.
14. A. Klein. Cross Site Scripting Explained. Technical report, Sanctum Inc., 2002.
 15. A. Klein. “Divide and Conquer”. HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics. Technical report, Sanctum, Inc., 2004.
 16. A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. Technical report, Web Application Security Consortium, 2005.
 17. M. Kolšek. Session Fixation Vulnerability in Web-based Applications. Technical report, ACROS Security, 2002.
 18. C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communication Security (CCS'03)*, pages 251–261, October 2003.
 19. C. Kruegel, G. Vigna, and W. Robertson. A Multi-model Approach to the Detection of Web-based Attacks. *Computer Networks*, 48(5):717–738, August 2005.
 20. C. Linhart, A. Klein, R. Heled, and S. Orrin. HTTP Request Smuggling. Technical report, Watchfire Corporation, 2005.
 21. V. Livshits and M. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium (USENIX'05)*, pages 271–286, August 2005.
 22. Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *Proceedings of the 14th International World Wide Web Conference (WWW'05)*, pages 432–441, May 2005.
 23. NCSA Software Development Group. The Common Gateway Interface. <http://hoo.hoo.ncsa.uiuc.edu/cgi/>.
 24. Netcraft. PHP Usage Stats. <http://www.php.net/usage.php>, April 2006.
 25. A. Nguyen-Tuong, S. Guarnieri, D. Greene, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 20th International Information Security Conference (SEC'05)*, pages 372–382, May 2005.
 26. OWASP. WebGoat. <http://www.wasp.org/software/webgoat.html>, 2006.
 27. Perl. Perl security. <http://perldoc.perl.org/perlsec.html>.
 28. rgod. PHP Advanced Transfer Manager v1.30 underlying system disclosure / remote command execution / cross site scripting. <http://retrogod.altervista.org/phpatm130.html>, 2005.
 29. Security Space. Apache Module Report. <http://www.securityspace.com/survey/data/man.200603/apachemods.html>, April 2006.
 30. K. Spett. Blind SQL Injection. Technical report, SPI Dynamics, 2003.
 31. Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the 33rd Annual Symposium on Principles of Programming Languages (POPL'06)*, pages 372–382, 2006.
 32. Sun. JavaServer Pages. <http://java.sun.com/products/jsp/>.
 33. Symantec Inc. Symantec Internet Security Threat Report: Vol. VIII. Technical report, Symantec Inc., September 2005.
 34. TIOBE Software. TIOBE Programming Community Index for April 2006. http://www.tiobe.com/index.htm?tiobe_index, April 2006.
 35. D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 255–264, Washington DC, November 2002.

36. J. Whaley and M. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'04)*, pages 131–144, June 2004.
37. Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th USENIX Security Symposium (USENIX'06)*, August 2006.