Wireless Sensor Network, **2009, 1, 233-244**
doi:10.4236/wsn.2009.14030 Published Online November 2009 (http://www.scirp.org/journal/wsn).

LDAP Injection Techniques

Jose Maria ALONSO[1], Antonio GUZMAN[2], Marta BELTRAN[2], Rodolfo BORDON
[1]*Informatica* 64*, S. L., Madrid, Spain*
[2]*Rey Juan Carlos University, Madrid, Spain*
Email: *chema@informatica*64*.com*, {*antonio.guzman, marta.beltran* }*@urjc.es*
Received March 12, 2009; *revised July* 3, 2009; *accepted July* 5, 2009

Abstract

The increase in the number of databases accessed only by some applications has made code injection attacks an important threat to almost any current system. If one of these applications accepts inputs from a client and executes these inputs without first validating them, the attackers are free to execute their own queries and therefore, to extract, modify or delete the content of the database associated to the application. In this paper a deep analysis of the LDAP injection techniques is presented. Furthermore, a clear distinction between classic and blind injection techniques is made.

Keywords: Web Applications Security, Code Injection Techniques, LDAP

1. Introduction

The amount of data stored in organizational databases has increased very fast in last years due to the rapid advancement of information technologies. A high percentage of these data are sensitive, private and critical to the organizations, their clients and partners.

Therefore, the databases are usually installed behind internal firewalls, protected with intrusion detection mechanisms and accessed only by applications. To access a database, users have to connect to one of these applications and to submit queries trough them to the database. The threat to databases arises when these applications do not behave properly and construct these queries without sanitizing user inputs first.

Over a 50% of web applications vulnerabilities are input validation related [1], which allows the exploitation of code injection techniques.

These attacks have proliferated in recent years causing severe damages in several systems and applications. The SQL injection techniques are the most widely used and studied [2–5] but there are other injection techniques associated to other languages or protocols such as XPath [6,7] or LDAP (Light Directory Access Protocol) [8,9].

Preventing the consequences of these kinds of attacks, lies in studying the different code injection possibilities and in making them public and well known for all programmers and administrators [10–12]. In this paper the LDAP injection techniques are analyzed in depth, because all the web applications based on LDAP trees might be vulnerable to this kind of attacks.

The key to exploiting injection techniques with LDAP is to manipulate the filters used to search in the directory services. Using these techniques, an attacker may obtain direct access to the database underlying an LDAP tree, and thereby to important corporate information. This can be even more critical because the security of many applications and services are based on LDAP directories in current single sign-on environments [13,14].

Although the vulnerabilities that lead to these consequences are easy to understand and to solve, they persist due to the lack of information about these attacks and their effects.

Although the vulnerabilities that lead to these consequences are easy to understand and fix, they persist because of the lack of information about these attacks and their effects. Though previous references to the exploitation of this kind of vulnerability exist the presented techniques don't apply to the vast majority of modern LDAP service implementations. The main contribution of this paper is the presentation and deep analysis of new LDAP injection techniques which can be used to exploit these vulnerabilities. Furthermore, a real environment has been implemented to perform different experiments in typical LDAP scenarios and to evaluate the possible danger of this kind of attacks.

It is important to note that the use of filters to limit the information that is showed to a client sending an LDAP search to the server does not increase the security of the applications, because these filters does not prevent the use of blind code injection techniques, capable of exploiting injection techniques without having detailed error messages from the server. Therefore, both, the

classic and the blind code injection techniques will be studied in depth in this paper.

This paper is organized as follows: sections 2 and 3 explain the LDAP fundamentals needed to understand the techniques presented in the following sections. Section 4 presents the two typical environments where LDAP injection techniques can be used and exemplify these techniques with illustrative cases. Section 5 describes how BLIND LDAP Injection attacks can be done with more examples. In Sections 6 and 7, some recommendations for securing systems against this kind of attack are given and, finally, Section 7 presents conclusions and future work.

2. LDAP Overview

Lightweight Directory Access Protocol is a protocol for querying and modifying directory services running over TCP/IP [15,16]. The most widely used implementations of LDAP services are Microsoft ADAM (Active Directory Application Mode, [17]) and OpenLDAP [18]. LDAP directory services are software applications that store and organize information sharing certain common attributes; the information is structured based on a tree of directory entries, and the server provides powerful browsing and search capabilities, etcetera.

LDAP is object oriented, therefore every entry in an LDAP directory services is an instance of an object and must correspond to the rules fixed for the attributes of that object. Due to the hierarchical nature of LDAP directory services read-based queries are optimized to the detriment of write-based queries. LDAP is also based on the client/server model.

The most frequent operation is to search for directory entries using filters. Clients send queries to the server and the server responds with the directory entries matching these filters. LDAP filters are defined in the RFC 4515. The structure of these filters can be summarized as:

- Filter = (filtercomp)
- Filtercomp = and / or / not / item
- And = & filterlist
- Or = | filterlist
- Not = ! filter
- Filterlist = 1*filter
- Item= simple / present / substring
- Simple = attr filtertype assertionvalue
- Filtertype = ”=” /” =”/ ”¿=” / ”¡=”
- Present = attr = *
- Substring = attr ”=” [initial] * [final]
- Initial = assertionvalue

Final = assertionvalue All the filters must be in brackets, only a reduced set of logical (AND, OR and NOT) and relational (:, ~, =, ∗) operators is available to construct them. The special character “*” can be used to replace one or more characters in the construction of the filters. Apart from being logic operators, RFC 4256 allows the use of the following standalone symbols as two special constants:

- (&) Absolute TRUE
- (|) Absolute FALSE

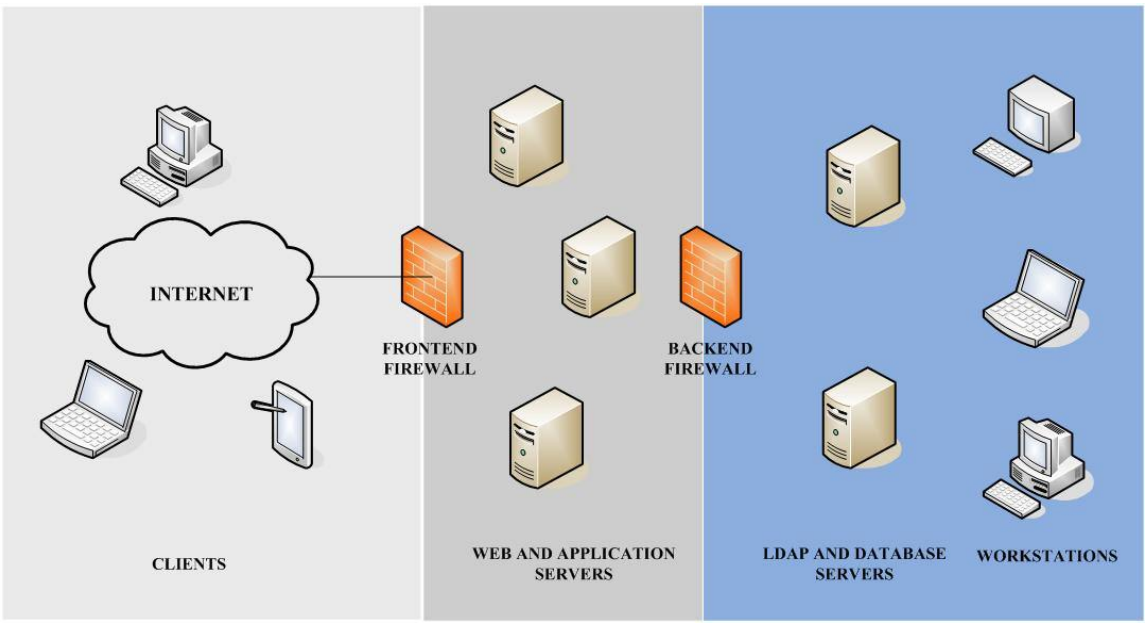

Figure 1. Typical LDAP scenario.

3. Common LDAP Environments

LDAP services are a key component for the daily operation in many companies and institutions. Directory Services such as Microsoft Active Directory, Novell E-Directory and RedHat Directory Services are based on the LDAP protocol. But there are other applications and services taking advantage of the LDAP services.

These applications and services used to require different directories (with separate authentication) to work. For example, a directory was required for the domain, a separate directory for mailboxes and distribution lists, and more directories for remote access, databases or web applications. New directories based on LDAP services are multi-purpose, working as centralized information repositories for user authentication and enabling single sign-on environments.

This new scenario increases the productivity by reducing the administration complexity and by improving security and fault tolerance. In almost every environment, the applications based on LDAP services use the directory for one of the following purposes:

- Access control (user/password pair verification, users certificates management).
- Privileges management.
- Resources management.

Due to the importance of the LDAP services for the corporate networks, the LDAP servers are usually placed in the backend with the rest of the database servers. Figure 1 shows the typical scenario deployed for corporate networks, and it is important to keep this scenario in mind in order to understand the implications of the injection techniques exposed in following sections.

4. LDAP Injection in Web Applications

LDAP injection attacks are based on similar techniques to SQL injection attacks. Therefore, the underlying concept is to take advantage of the parameters introduced by the user to generate the LDAP query. A secure Web application should sanitize the parameters introduced by the user before constructing and sending the query to the server. In a vulnerable environment these parameters are not properly filtered and the attacker can inject malicious code.

Taking into consideration the structure of the LDAP filters explained in section II and the implementations of the most widely used LDAP implementations, ADAM and OpenLDAP, the following conclusions can be drawn about the code injection. (The following filters are crafted using as value a non sanitized input from the user):

- **(attribute=value):** If the filter used to construct the query lacks a logic operator (OR or AND), an injection

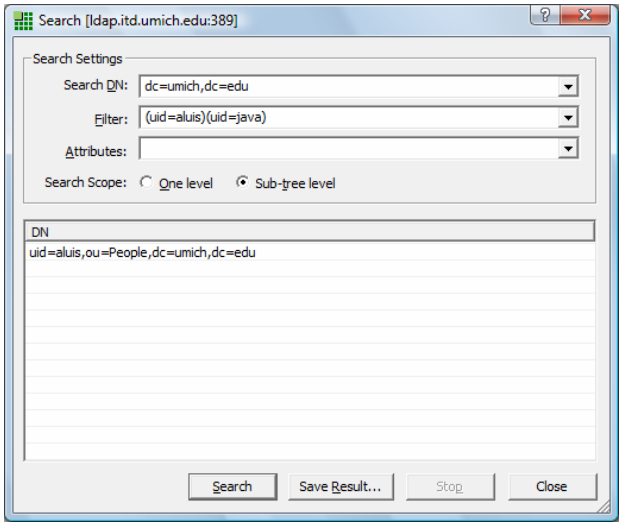

Figure 2. OpenLDAP processes only the first complete LDAP search filters. Data obtained with LDAP browser.

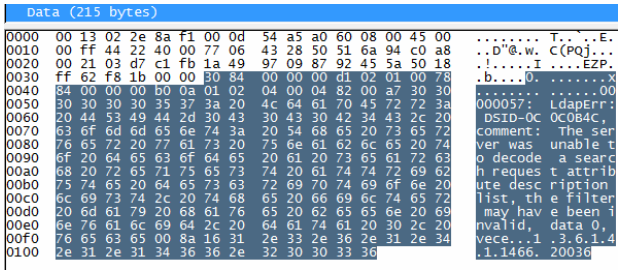

Figure 3. ADAM responses with a disconnection message in case of more than one filter are received in only one query. Data analyzed with wireshark.

like value) (injected filter will result in two filter: *(attribute*=value) (injected filter). In the OpenLDAP (Figure 2) implementations the second filter will be ignored, only the first one being executed.

In ADAM, a query with two filters isn't allowed (Figure 3). Therefore, the injection is useless.

(|(attribute=value) second filter)) or (& attribute value)(second filter)): If the filter used to construct the query has a logic operator (OR or AND), an injection like "value)(injected filter)" will result in the following filter: (&(*attribute*=***value*)(*injected filter***)) (*second filter*)). Though the filter is not even syntactically correct, OpenLDAP will start processing it left to right ignoring any character after the first filter is closed. Some LDAP Client web components will ignore the second filter, sending to ADAM and OpenLDAP only the first complete one, therefore allowing the injection (Figure 4).

Some application frameworks will check the filter for correctness before sending it to the LDAP server. Should this be the case, the filter has to be syntactically correct, which can be achieved with an injection like:

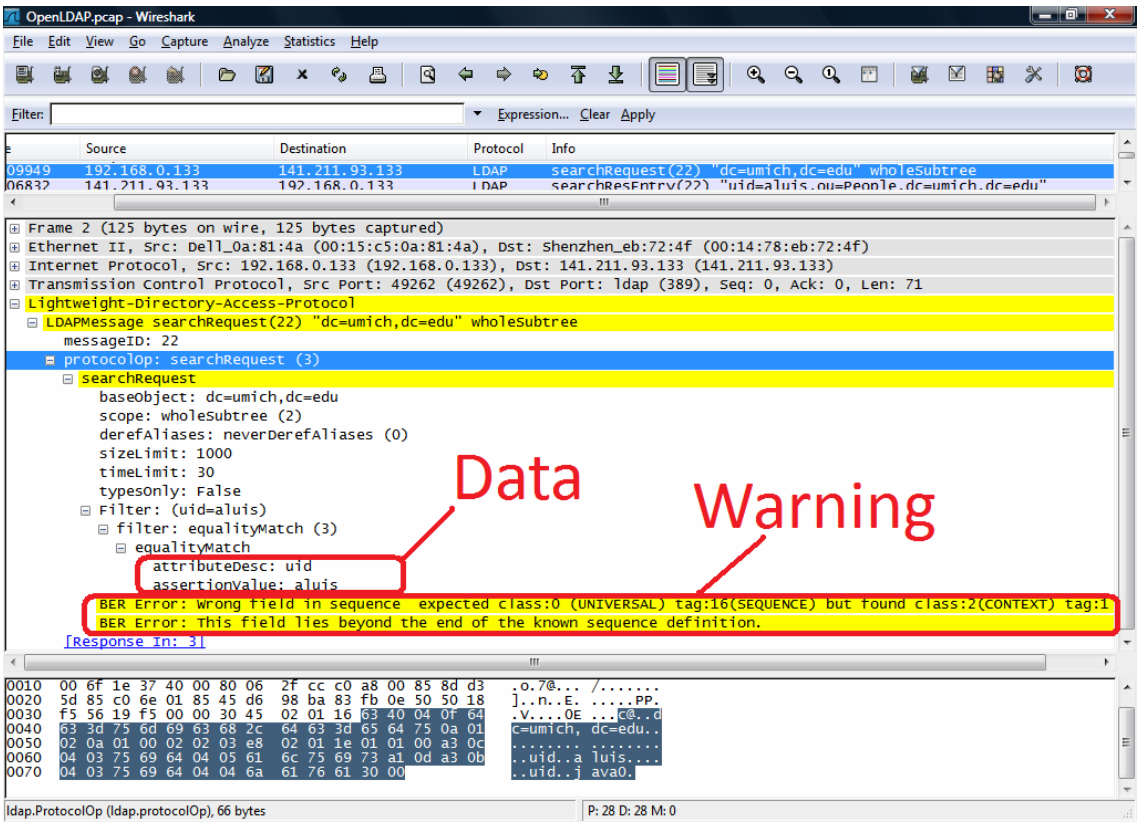

Figure 4. This is just because it tries to match the second filter to a list of attributes required. If this can be done then OpenLDAP only response with those attributes, else OpenLDAP will ignore the second filter responding with data obtained after first filter is executed and a warning message. Data analyzed with WireShark.

value)(*injected filter*))(&(1=0. This will result in two different filters, the second being ignored: (&(*attribute* = ***value)(injected filter)***)(&(1=0)(*second filter*)).

As the second filter is going to be ignored by the LDAP Server, some components won't allow an LDAP query with two filters. In these cases a special injection must be crafted in order to obtain a single-filter LDAP query. An injection like: value) (injected filter will result in the following filter: (& (attribute=value) (injected filter)(second filter)).

The typical test to know if an application is vulnerable to code injection consists of sending to the server a query that generates an invalid input. Therefore, if the server returns an error message, it is clear for the attacker that the server has executed his query and that he can exploit the code injection techniques. Taking into account the previous discussion, two kinds of environments can be distinguished: AND injection environments and OR injection environments.

4.1. AND LDAP Injection

In this case the application constructs the normal query to search in the LDAP directory with the "&" operator

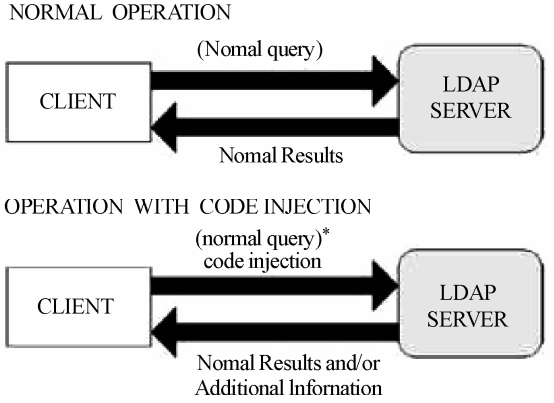

Figure 5. Injection LDAP.

and one or more parameters introduced by the user. For example:

(&(*parameter* 1= *value*1)(*parameter* 2= *value* 2))

Where *value* 1 and *value* 2 are the values used to perform the search in the LDAP directory. The attacker can inject code, maintaining a correct filter construction but using the query to achieve his own objectives.

1) *Example* 1: *Access Control Bypass:* A login page has two text box fields for entering user name and

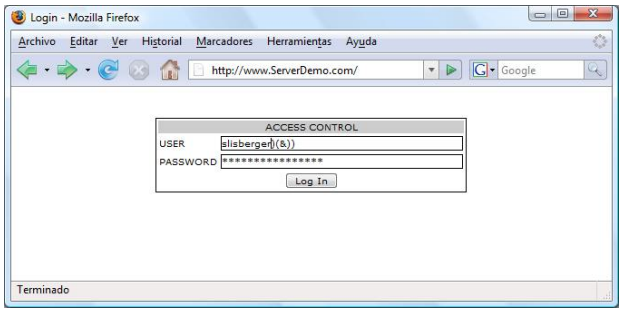

Figure 6. Loginpage with LDAP Injection.

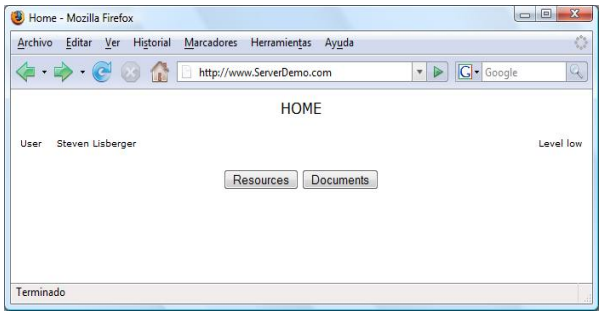

Figure 7. Home page shown to the attacker after avoiding the access control.

password (Figure 6). *Uname* and *Pwd* are the user inputs for USER and PASWORD. To verify the existence of the user/password pair supplied by a client, an LDAP search filter is constructed and sent to the LDAP server:

(&(*USER* = *Uname*) (*PASSWORD* = *Pwd*))

If an attacker enters a valid username, for example, *slisberger*, and injects the appropriate sequence following this name, the password check can be bypassed. Making *Uname*=***slisberger)(&))*** and introducing any string as the *Pwd* value, the following query is constructed and sent to the server:

(&(*USER* = *slisberger*)(&))(*PASSWORD* = *Pwd*))

Only the first filter is processed by the LDAP server, that is, only the query (&(USER=***slisberger)(&)***) is processed. This query is always true, so the attacker gains access to the system without having a valid password (Figure 7).

In case of being working with ADAM Microsoft implementation, this injection can be done just in order to obtain only one filter at the end:

USER=admin)(!(&(|PASSWORD=any))(&(USER= admin)(!(&(|)(PASSWORD = any))))

As can be seen, in this example, it is necessary to inject code in the user and password fields but it will work out not only with Microsoft implementations but with any other LDAP engine.

2) *Example* 2: *Elevation of Privileges:* For example, suppose that the following query lists all the documents visible for the users with a low security level (Figure 8):

(&(directory = documents)(security_level = low))

Where *documents* is the user entry for the first parameter and low is the value for the second (Figure 9). If the attacker wants to list all the documents visible for the high security level, he can use an injection like

*documents)(security level = *))(&(directory = documents*

resulting in the following filter:

(& (directory = documents)(security level = *))(&(directory = documents)(security level = low))

The LDAP server will only process the first filter ignoring the second one, therefore, only the following query will be processed: (&(directory = documents) (security_level=*)), while (& (directory = documents) (security_level = low)) will be ignored. As a result, a list with all the documents available for the users with all security levels will be displayed for the attacker although he doesn't have privileges to see them.

4.2. OR LDAP Injection

In this case the application constructs the normal query to search in the LDAP directory with the "|" operator and one or more parameters introduced by the user. For example:

(| (parameter 1= value1)(parameter2= value2))

Where *value1* and *value2* are the values used to perform the search in the LDAP directory. The attacker can inject

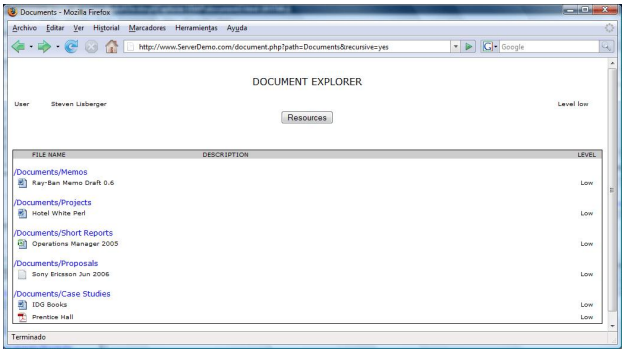

Figure 8. Low security documents.

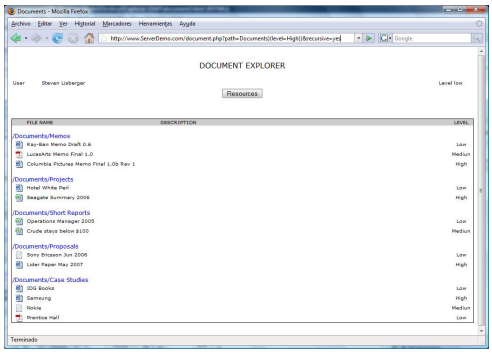

Figure 9. All security levels documents.

code, maintaining a correct filter construction but using the query to achieve his own objectives.

1) *Example* 1: *Information Disclosure:* Suppose a resources explorer allows users to know the resources available in the system (printers, scanners, storage systems, etc.). This is a typical OR LDAP Injection case, because the query used to show the available resources is:

(| (type = Rsc1)(type = Rsc2))

Rsc 1 and *Rsc 2* represent the different kinds of resources in the system. In Figure 10, *Rsc1=printer* and *Rsc 2=scanner* to show all the available printers and scanners in the system.

If the attacker enters *Rsc1=* ***printer)(uid=*)***, the following query is sent to the server:

*(|(type = printer)(uid = *))(type = scanner))*

The LDAP server responds with all the printer and user objects (Figure 11).

5. Blind Ldap Injection

Suppose that an attacker can infer from the server responses, although the application does not show error messages, the code injected in the LDAP filter generates a valid response (true result) or an error (false result). The attacker could use this behavior to ask the server true or false questions. These types of attacks are named "Blind Attacks". Blind LDAP Injection attacks are slower than classic ones but they can be easily implemented, since they are based on binary logic, and they let the attacker extract information from the LDAP Directory.

5.1. AND Blind LDAP Injection

Suppose a web application wants to list all available Epson printers from an LDP directory where error messages are not returned. The application sends the following LDAP search filter: *(& (object-Class=printer) (type= Epson*))* With this query, if there are any Epson printers available, icons are shown to the client, otherwise no icon is shown. If the attacker performs a Blind LDAP injection attack *injecting *) (objectClass = *)) (& (objectClass = void*, the web application will construct the following LDAP query:

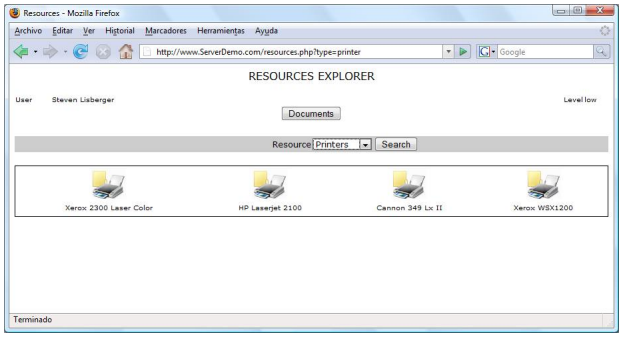

Figure 10. Resources available to the user from the resources consoles management.

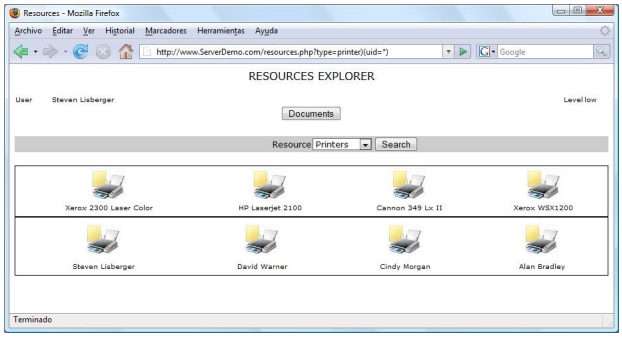

Figure 11. Resources available to the user from the resources consoles management.

*(&(objectClass = *)(objectClass=*))*

(&(objectClass=void)(type = Epson))*

Only the first complete LDAP filter will process:

*(&(objectClass = *)(objectClass = *))*

As a result, the printer icon must be shown to the client, because this query always obtains results: the filter *objectClass=** always returns an object. When an icon is shown the response is true, otherwise the response is false. From this point, it is easy to use blind injection techniques. For example, the following injections can be constructed:

*(&(objectClass=****)(objectClass=users))***

(&(objectClass=foo)*(type=Epson*))*

*(&(objectClass=****)(objectClass=resources))***

(&(objectClass=foo)*(type=Epson*))*

This set of code injections allows the attacker to infer the different objectClass values possible in the LDAP directory service. When the response web page contains at least one printer icon, the objectClass value exists (TRUE), on the other hand the objectClass value does not exist or there is no access to it, and so no icon, the objectclass value does not exist(FALSE). Blind LDAP injection techniques allow the attacker access to all information using TRUE/FALSE questions.

5.2. OR Blind LDAP Injection

In this case, the logic used to infer the desired information is the opposite, due to the presence of the OR logical operator. Following with the same example, the injection in an OR environment should be:

*(|(objectClass=****void)(objectClass=void))***

(&(objectClass=void)*(type=Epson*))*

This LDAP query obtains no objects from the LDAP directory service, therefore the printer icon is not shown to the client (FALSE). If any icon is shown in the response web page then, it is a TRUE *response. Thus, an attacker could inject the following LDAP filters for gathering information:*

(|(objectClass=void)(objectClass=users))

(&(objectClass=void)(type=Epson))*

(|(objectClass=void)(objectClass=resources))
(&(objectClass=void)(type=Epson))*

5.3. Exploitation Example

In this section, an LDAP environment has been implemented to show the use of the injection techniques explained above and also to describe the possible effects of the exploitation of these vulnerabilities and the important impact of these attacks in current systems security. In this example the page printerstatus.php receives a parameter *idprinter* to construct the following LDAP search filter:

(&(idprinter=Value1)(objectclass=printer))

1) *Discovering Attributes:* Blind LDAP Injection techniques can be used to obtain sensitive information from the LDAP directory services by taking advantage of the AND operator at the beginning of the LDAP search filter built into the web application. For example, given the attributes defined for the printer object shown in Figure 12 and the response web page of this LDAP query in Figure 13 for *Value* 1=*HPLaserJet* 2100, an attribute discovering attack can be performed by making these following LDAP injections:

*(&(idprinter=**HPLaserJet2100)(ipaddress=*)**)*
(objectclass=printer))
*(&(idprinter=**HPLaserJet2100)(department=*)**)*
(objectclass=printer))
*(&(idprinter=**HPLaserJet2100)(department=*)**)*
(objectclass=printer))

Obviously, the attacker can infer from these results which attributes exist and which do not. In the first case, the information about the printer is not given by the application because the attribute ipaddress does not exist or it is not accessible (FALSE), as is shown in Figure 14.

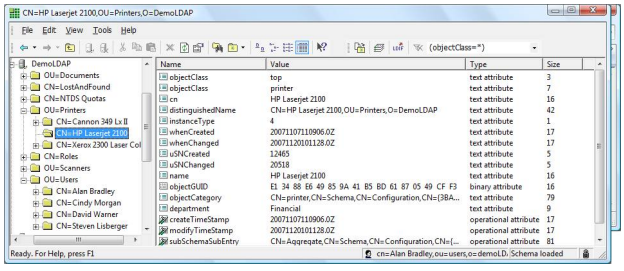

Figure 12. Attributes defined for the printer object.

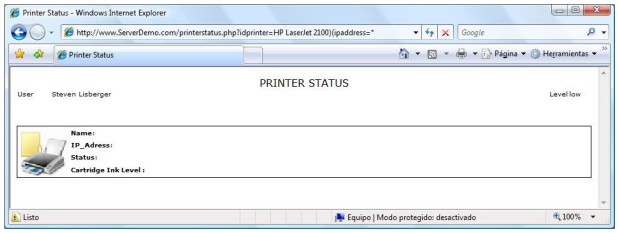

Figure 13. Normal behavior of the application.

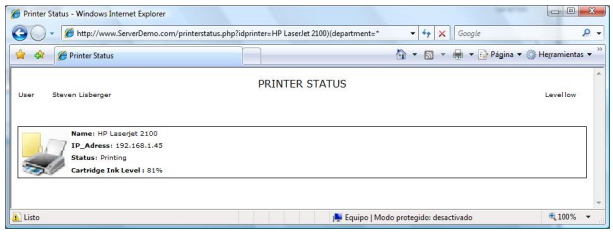

Figure 14. Response web page when the attribute does not exist.

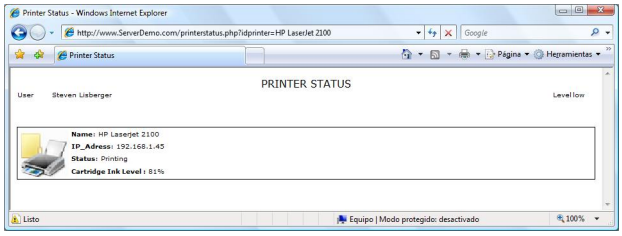

Figure 15. Response web page when the attribute exists.

On the other hand, in the second case, the response web page shows the printer status and therefore, the attribute department exists in the LDAP directory and it is possible access to it (Figure 15). Furthermore, with blind LDAP injection attacks the values of some of these attributes can be obtained. For example, suppose that the attacker wants to know the value of the department attribute: he can use booleanization and charset reduction techniques, explained in the next sections, to infer it.

2) *Booleanization:* An attacker can extract the value from attributes using alphabetic or numeric search. The crux of the idea is to transform a complex value (e.g. a string or a date) into a list of TRUE/FALSE questions. This mechanism, usually called booleanization, is summarized in Fgure 16 and can be applied in many different ways.

Suppose that the attacker wants to know the value of the department attribute. The process would be the following:

*(&(idprinter=**HPLaserJet2100)(department=a*)**)*
(objectclass=printer))
*(&(idprinter=**HPLaserJet2100)(department=f*)**)*
(objectclass=printer))

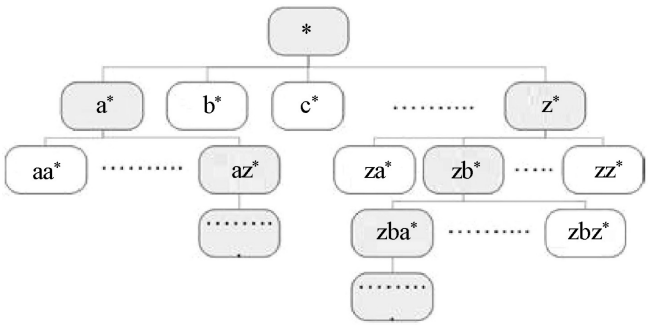

Figure 16. Booleanization.

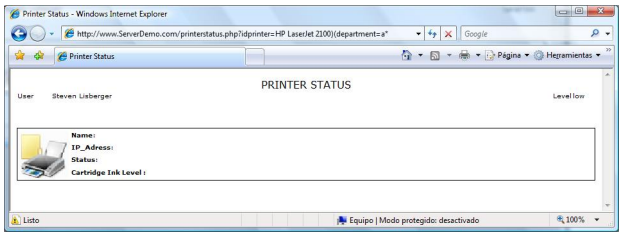

Figure 17. FALSE. Value does not start with 'a'.

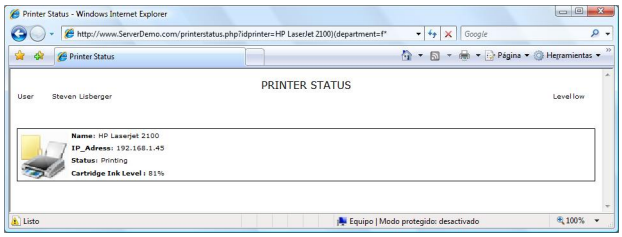

Figure 18. TRUE. Value starts with 'f'.

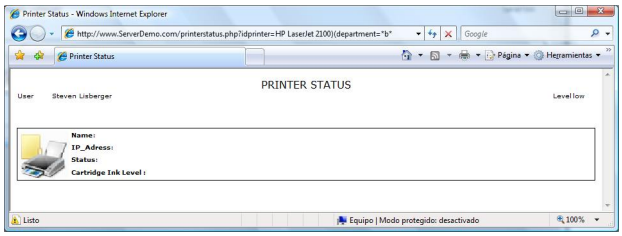

Figure 19. FALSE. Value doesn't start with 'fa'.

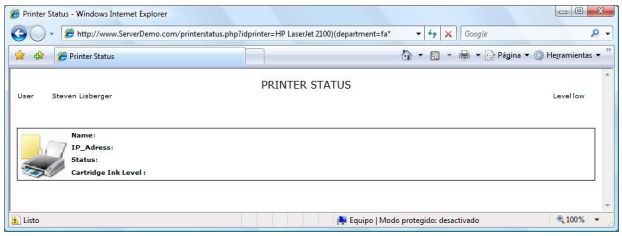

Figure 20. TRUE. Value starts with 'fi'.

*(&(idprinter=**HPLaserJet2100)(department=fa*)***
(objectclass=printer))
*(&(idprinter=**HPLaserJet2100)(department=fi*)***
(objectclass=printer))

As shown in Figure 12, the department value in this example is financial. The first try with the character "a" does not obtain any printer information (Figure 17) therefore, the first character is not an "a". After testing with the rest of the characters, the only one that obtains the normal behavior from the application is "f" (Figure18).

Regarding the second character, the only one that results in the normal operation of the application is 'i' (Figure 20) and so on. Following the process, the department value can be obtained. This algorithm can be also used for numeric values. In order to perform this, the booleanization process should use 'greater than or equal to' (≥) and 'less than or equal to' (≤) operators.

3) *Charset Reduction:* An attacker can use charset reduction to decrease the number of requests needed for obtain the information. In order to accomplish this, he uses wildcards to test if the given character is present *anywhere* in the value, e.g.:

*(&(idprinter=**HPLaserJet2100)(department=*n*)***
(objectclass=printer))

The Figure 21 shows the response web page when the character 'b' is tested: no results are sent from the LDAP directory service so no letter 'b' is present, but in Figure 22 a normal response web page is shown, meaning that the character 'n' is in the department value. Through this process, the set of characters comprising the department value can be obtained. Once the charset reduction is done, only the characters discovered will be used in the booleanization process, thereby decreasing the number of requests needed.

All these techniques can be easily performed with automated tools in order to extract all the information. Just as a proof of concept we developed LDAP Injector showed in Figure 23.

6. A Practical Proposal to Discover LDAP Vulnerabilities in Web Applications

In this section a practical proposal is described to recognize bugs in web applications vulnerable to LDAP injection attacks. This proposal is as general as needed to work with any LDAP directory the application might is using. It is based in black box techniques meaning no knowledge about the source code is needed. The core of this practical approach consists in try out different LDAP injections against every parameter and then to analyze the web application responses in order to recognize the vulnerability.

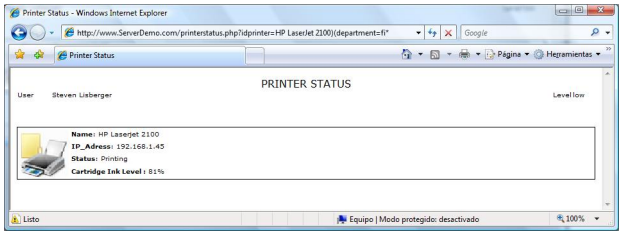

Figure 21. FALSE. Character 'b' is not in the department value.

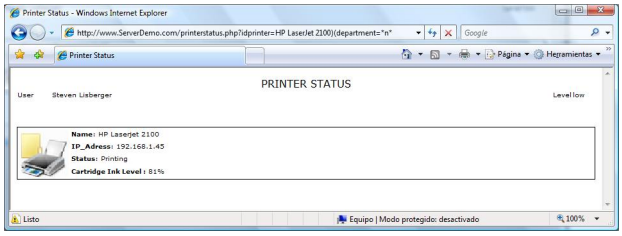

Figure 22. TRUE. Character 'n' is in the department value.

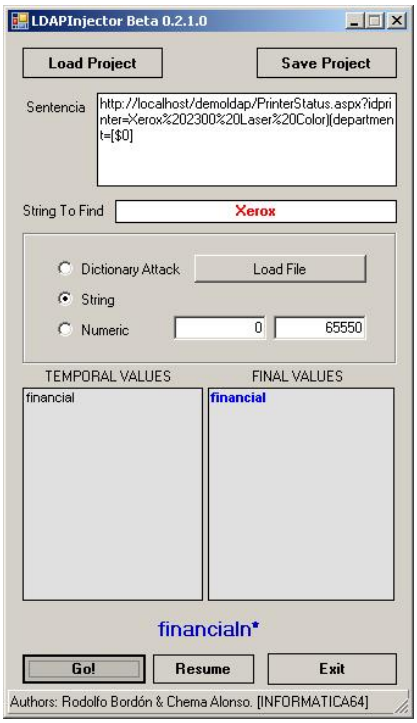

Figure 23. LDAP Injector performing a booleanization attack.

6.1. Definitions

Before start to describe the method some definitions are required to understand the basic principles in which relay on. These are the following:

Expected values. Set of characters forming the system's expecting input. These values generate a correct and normal result and behavior in the web application. These results are not an empty set of records. This means that after introduce an expected value web application retrieve any data from the LDAP directory.

Empty LDAP query (LDAP(Void)). LDAP query executed using expected values. This means no LDAP injection has been done.

Injection string (ILDAP). Set of characters not in the expected values. It is possible that the system is ready for any input character but it is assumed that LDAP special characters are those involved in LDAP Injection queries. Injections can be classified in:

- *Positive behavior change injection (ILDAP+).* It is an injection string to produce different number of retrieved records from the LDAP directory. It means the generated object list changes.
- *Negative behavior change injection (ILDAP-).* It is an injection string to produce fewer objects than the original one.

Zero behavior change injection (ILDAP0). It is an injection string to produce no change in the response object lists generated by the LDAP directory.

Injected LDAP query LDAP (ILDAP)). It is an LDAP query in which an injection string has been introduced. This injection should generate a syntax error or not. It this injection should not result in a syntax error then it is called Valid Injection (VI), otherwise it's called (Not Valid Injection). It is important to notice the use of *should* verb. This is because the injection should be correct in an injectable environment but security mechanisms in a web application could make it Not Valid. All NVI are also ILDAP-because no one object will be retrieved.

Minimum Valid Injection (MVI). It is an injection string which introduces no logic operators. It means the injections are constructed using the minimum number of parenthesis and operators without change the logic.

Complex Valid Injection (CVI). It is an injection string which introduces changes in the logic. The query should has a correct syntax and add new logic. Complex injections are necessaries to evaluate if the parameter is vulnerable to blind LDAP injections. In order to find out the correct syntax, the simplest Complex Valid Injection should be construct and this will only be possible if the web application is using and AND or an OR query, just as seen in the first part of this article. Table VI-A shows some examples.

Not Complex Valid Injection (NCVI). It is a correct injection with no syntax errors which injects new logic but changing the object list to retrieve none objects. It is a key to construct Boolean logic in Blind LDAP injection attacks. Table VI-A shows some examples.

Res(void). Object list retrieved after sending LDAP(void) to the web application. This is the result set sent from the LDAP engine to the web applications after the LDAP query is executed. Res(void) is constructed by the objects retrieved when no injection has been done and hence it is the normal result set.

Res(ILDAP). Object list retrieved after injecting and ILDAP. This result set obtained might has more or less objects than Resultset(void) depending on the ILAP. In each case will be known as Res(ILDAP+) or Res (ILDAP-). The results set obtained depend on several environmental aspects such as the normal query, the container in which is sent through, if it is recursive query, etc. If A is supposed to be injection string it will be an *ILDAP0 if RES(void)= RES(A),* it will be an *ILDAP+ if RES(void) < RES(A) and a ILDAP− if RES(void) > RES(A).*

HTMLRES (ILDAP). It will represent the response page obtained after sending the ILDAP to the web application. It is the data which methodology has to work with because is the info that web application sends back to the client as response to the test tried out. As it is working in a web environment this will be, normally, an HTML page.

6.2. Creating Valid LDAP Injections

Using as reference terminology defined in the previous section two rules can be settled up as:

1) If it is possible to construct a MVI for a parameter then it will be vulnerable against LDAP Injection attacks.

2) If it is possible to construct a CVI with AND/OR logic operators for a parameter then it will be vulnerable against Blind LDAP injection attacks.

As a general rule, in a black box pen testing audit, MVI should be constructed to test the parameter strength against LDAP Injection attacks. This is just because a Blind LDAP injection attack only can be conducted in parameters previously vulnerable against LDAP injection attacks. Let's suppose a web application retrieving a GET parameter as following:

http://www.myweb.com/prog.php?id=1.

It will be used to query an LDAP directory to obtain objects from a container matching filters as in this example:

(login_operator(atributte1=value1) (atributte2=value2)) or (atributte=value)

The query above will be known as LDAP(void) and the goal is to find out an MVI which guarantee no more records will be obtained.

As there is not a universal MVI which works in all the cases will be necessary to try out different ILDAPs. One ready for OR queries, another ready for AND ones and the last prepared to work in simple filters, it means with only one comparison and no one logic operator. In order to do this will be necessary to use as reference RES(void) supposing this is a normal behavior in the web application and that RES(void) is not null. This is mandatory in order to accomplish Res(void) > Res(ILDAP −).

Taking into consideration that:

- *Res(void) >=0 [not null].*
- *Res(void)= RES(LDAP 0).*
- *MV I are LDAP 0.*
- *Res(void) > RES (LDAP −).*
- *NCV I are LDAP −.*

Therefore is possible to conclude that if HTMLRES (void) = HTMLRES(MVI) and HTMLRES(VOID) != HTMLRES(NCVI) then the parameter is vulnerable against LDAP Injection attacks. The first condition proves LDAP directory is responding to LDAP injected queries correctly and second one proves which this is true, and not a web application behavior, by generating an empty object list and obtaining a different web application behavior.

It important to keep in mind that in blind environments, it means in web application in which data is never printed in the response web page or in the error messages, to extract all the data is necessary to find out not only a MVI which complaints the Vulnerable Rule but a CVI.

Vulnerable Rule against Blind LDAP Injection attacks: If HTMLRES (void) = HTMLRES(CVI) and HTMLRES (void)!= HTMLRES(NCVI) then the parameter is vulnerable against LDAP Injection attacks.

So, at the end, to find out if a parameter is vulnerable to LDAP Injection attacks or Blind LDAP Injection attacks, it is mandatory to recognize a response (HTMLRES) as a normal behavior or a response as a behavior when an LDAP or an empty object list has been retrieved. The first behavior will be referenced as a TRUE behavior and the other will be referenced as a FALSE behavior, allowing both to construct a binary logic.

6.3. Web Responses Analysis

Once a valid injection is constructed, it is necessary to analyze the response given by the web application in order to define the logic that is behind the booleanization. There are several behaviors that the system might has when it receives an injection. In fact these behaviors correspond to the treatment of errors implemented in the web server. The methodology has to deal with all the possibilities to be able to propose an effective criterion. This criterion determines if the response given by de the system for an CVI is a true response or a false one. The most important kinds of system responses when it faces an CVI are the following:

Table 1. Some examples of complex valid injections (Cvi).

Example	Original LDAP Query	injection String	Results
1 2 3	(attribute=value) (&(attribute1=value1)(attribute2=value2)) (|(attribute1=value1)(attribute2=value2))	Id=value)(Id=valu e1)(Id=value1)(|	(attribute=value)() (&(attribute1=value1)(&)(attribute2=value2)) (|(attribute1=value1)(|)(attribute2=value2))

Table 2. Some examples of not complex valid injections (Ncvi)).

	Original LDAP Query	injection String	Results
1 2 3	(attribute=value) (&(attribute1=value1)(attribute2=value2)) (|(attribute1=value1)(attribute2=value2))	Id=value** Id=value1)(| Id=value1)(&	(attribute=value**)() (&(attribute1= value1)(|)(attribute2=value2)) (|(attribute1= value1)(&)(attribute2=value2))

Web server error. These responses are predefined in the server configurations (p. e. http code 500).

Generic error. These responses are programmed by the application designer.

Correct results webpage. The response contains the expected values.

Last webpage displayed. The web application has implemented the errors treatment as a mechanism that proceed to send the last webpage displayed when an error occur.

For the first three alternatives it is easy to design a function to analyze the server response. Different techniques can be developed. For this work the following techniques have been evaluated:

HASH file signatures evaluation. Two different sets have to be define in order to classify which responses are false and which are true. This technique does not work with websites with dynamic content in its pages.

HTML tree evaluation. To deal with the problem exposed in the last point the focus of the evaluation is fixed on the tree structure of the HTML document not on the contents. This technique presents some limitations with websites where the error treatment maintains the same HTML structure that the normal documents.

Key words searching. This technique is oriented to define two distinguishing patterns: one for the false responses and another one for the true ones.

However, when error treatment mechanism uses the last webpage displayed to deal with a not expected input there is not any technique to define an effective error function at least for the time of being.

6.4. The Analysis of the Vulnerability of Web Application Parameter

In response to the descriptions given in sections above, it is possible to propose the steps that are necessary to determine the weakness of a parameter defined for a web application when it is faced an injection attack.

1) To find out the application's input parameters.
2) To try to construct an IMV.
3) If one IMV exists then
   a) To try to construct at least one ICV
   b) If this valid ICV exists with the AND or OR operators, then the parameter is vulnerable to Blind Injection attacks. At this point, it is necessary to determine the error treatment mechanism implemented in order to propose an efficient error function. If this mechanism is based on the last response given, today, the parameter can be considered as secure.
   c) If is not possible construct a valid ICV the parameter can be consider as secure.
4) If no IMV exits then the parameter can be consider as secure.

7. Securing Applications against Blind LDAP Injection & LDAP Injection Attacks

The attacks presented in the previous sections are performed on the application layer, therefore firewalls and intrusion detection mechanisms on the network layer have no effect on preventing any of these LDAP injections. However, general security recommendations for LDAP directory services can mitigate these vulnerabilities or minimize their impact by applying minimum exposure point and minimum privileges principles.

Mechanisms used to prevent code injection techniques include defensive programming, sophisticated input validation, dynamic checks and static source code analysis. The work on mitigating LDAP injections must involve similar techniques.

It has been demonstrated in the previous sections that LDAP injection attacks are performed by including special characters in the parameters sent from the client to the server. It is clear therefore that it is very important to check and sanitize the variables used to construct the LDAP filters before sending the queries to the server.

However, developer communities are not widely aware of this kind of injections because there is no so much information about LDAP Injection and Blind LDAP Injection techniques, hence developers don't sanitize correctly their queries against LDAP directories. A quick search for "LDAP" in websites hosting open source projects retrieves a lot of projects with LDAP Injection vulnerabilities. On the other hand, static code analysis tools are not ready yet to discover LDAP injection vulnerabilities in the code. So it is easy, for a developer not strongly formed in security best practices, to create a vulnerable code just relaying in security post-analysis. Microsoft Code Analysis, a tool forming part of Microsoft Visual Studio Team System or Microsoft FXCop, two of the most used code analysis tools don't have any rule to detect LDAP injection vulnerabilities.

In order to sanitize correctly web application inputs which are going to be used in LDAP search filters, developers must only pay attention to ten special characters: |, &, (,), *, <, >, =, ~, !. If the developer sanitizes in a secure way the input to forbid those characters LDAP Injection attacks won't work.

8. Conclusions and Future Work

LDAP services facilitate access to networks information organizing it in a hierarchical database that allows authorized users and applications to find information related to people, resources and applications.

This protocol is simple to install, maintain, replicate and use, and it can be highly distributed. And it allows an easy implementation of the widely used single sign-on

environments. Therefore, given the increasing need for information in current systems, it is an essential service in almost all networks.

LDAP injection techniques are an important threat for these environments, specially, for the control access and privileges and resources management.

These attacks modify the correct LDAP queries, altering their behavior for the attacker benefit. And the consequences of these attacks can be very severe.

Our work is unique in providing a rigorous analysis of LDAP injection techniques and in showing representative examples of the possible effects of these attacks.

Even more, recommendations to secure applications against these techniques have been proposed. It has been showed that filtering the error messages produced by the server only fortifies the system but does not secure it against blind injection techniques. A more in depth protection is needed to avoid this kind of injection vulnerabilities too. It has been demonstrated with the presented examples, that it is essential to filter the client inputs used to construct the LDAP queries before sending them to the server. And that the AND and OR filter constructions should be avoided.

Finally, a very interesting line for future research is working on analyzing injection techniques with other protocols used to access databases and directories. And to study the possible utilization of mechanisms booleanization techniques such as character displaying or charset reduction in other environments.

9. References

[1] S. Barnum and G. McGraw, "Knowledge for software security," IEEE Security and Privacy Magazine, Vol. 3, No. 2, pp. 74–78, 2005.

[2] E. Bertino, A. Kamra, and J. Early, "Profiling database application to detect SQL injection attacks," in Proceedings of the IEEE International Performance, Computing, and Communications Conference, pp. 449–458. 2007.

[3] X. Fug, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao, "A static analysis framework for detecting SQL injection vulnerabilities," in Proceedings of the 31st Annual International Computer Software and Applications Conference, pp. 87–96, 2007.

[4] E. Merlo, D. Letarte, and G. Antoniol, "SQL-injection security evolution analysis in PHP," in Proceedings of the 9th IEEE International Workshop on Web Site Evolution, pp. 45–49, 2007.

[5] S. Thomas and L. Williams, "Using automated fix generation to secure SQL statements," in Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems, pp. 9–19, 2007.

[6] "XPath 1.0 specification," 1999, http://www.w3.org/TR/xpath.

[7] "XPath 2.0 specification," 2007, http://www.w3.org/TR/xpath20/.

[8] "RFC 1777: Lightweight Directory Access Protocol v2," 1995, http://www.faqs.org/rfcs/rfc1777.html.

[9] "RFC 2251: Lightweight Directory Access Protocol v3," 1997, http://www.faqs.org/rfcs/rfc2251.html.

[10] T. Holz, S. Marechal, and F. Raynal, "New threats and attacks on the world wide web," IEEE Security and Privacy Magazine, Vol. 4, No. 2, 2006.

[11] G. Hermosillo, R. Gomez, L. Seinturier, and L. Duchien, "AProSec: An aspect for programming secure web applications," in Proceedings of the Second International Conference on Availability, Reliability and Security, pp. 1026–1033, 2007.

[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, pp. 6–15, 2006.

[13] E. Jamhour, "Distributed security management using LDAP directories," in Proceedings of the XXI Internatinal Conference of the Chilean Computer Science Society, pp. 144–153, 2001

[14] R. Sari and S. Hidayat, "Integrating web server applications with LDAP authentication: Case study on human resources information system of ui," in Proceedings of the International Symposium on Communications and Information Technologies, pp. 307–312, 2006.

[15] M. Wahl, T. Howes, and S. Kille, "Lightweight Directory Access Protocol (v3)," 1997, http://www.ietf.org/rfc/rfc2251.

[16] V. Koutsonikola and A. Vakali, "LDAP: Framework, practices, and trends," IEEE Internet Computing, Vol. 8, No. 5, pp. 66–72, 2004.

[17] M. Russinovich and D. Solomon, Microsoft Windows Internals, Microsoft Press, 2004.

[18] "OpenLDAP main page," http://www.openldap.org.

