

# RIBADEO HACK LAB

# LDAP INJECTION

## Attack and Defense Techniques

### Authors

Esteban Guillardoy [eguardoy@ribadeohacklab.com.ar]  
Facundo de Guzman [fdeguzman@ribadeohacklab.com.ar]  
Hernan Abbamonte [habbamonte@ribadeohacklab.com.ar]

LDAP (Lightweight Directory Access Protocol) is an application protocol that allows managing directory services. This protocol is used in several applications so it is important to know about the security involved around it. The objective of this article is not to provide an extensive explanation of the protocol itself but to show different attacks related to LDAP Injection and possible ways prevention techniques.

## Content

Content .....	2
Disclaimer.....	2
Introduction.....	2
LDAP Query - String Search Criteria.....	2
LDAP Injection.....	3
Login Bypass.....	3
Bind Method.....	3
Search Query.....	3
Information Disclosure .....	4
Charset Reduction .....	4
Privilege Escalation .....	5
Information Alteration .....	5
URL encoding & Unicode encoding.....	6
LDAP Injection vs. SQL Injection.....	6
Prevention Techniques .....	8
Tools .....	8
Reference and further reading.....	9
About the Authors.....	10

## Disclaimer

All the content of this article was designed with educational purposes only. The main purpose is to warn programmers about these attack techniques and possible countermeasures. RibadeoHackLab does not assume any responsibility for further uses the reader gives to the information provided on this paper.

## Introduction

A directory service is simply the software system that stores, organizes and provides access to information in a directory. Based on X.500 specification, the Directory is a collection of open systems cooperating to provide directory services. A directory user accesses the Directory through a client (or Directory User Agent (DUA)). The client, on behalf of the directory user, interacts with one or more servers (or Directory System Agents (DSA)). Clients interact with servers using a directory access protocol.[1]

LDAP provides access to distributed directory services that act in accordance with X.500 data and service models. These protocol elements are based on those described in the X.500 Directory Access Protocol (DAP). Nowadays, many applications use LDAP queries with different purposes. Usually, directory services store information like users, applications, files, printers and other resources accessible from the network. Furthermore, this technology is also expanding to single sign on and identity management applications. As LDAP defines a standard method for accessing and updating information in a directory, a person trying to gain

access to sensitive information stored on a directory will try to use an input-validation based attack known as LDAP Injection. This technique is based on entering a malformed input on a form that is used for building the LDAP query in order to change the semantic meaning of the query executed on the server. By doing this, it is possible for example, to bypass a login form or retrieve sensitive information from a directory with restricted access.

Some of the most well known LDAP implementations include OpenLDAP [2], Microsoft Active Directory [3], Novell eDirectory and IBM Tivoli Directory Server. Each of them may handle some LDAP search requests in a different way, yet regarding security, besides the LDAP server configuration, it is of capital importance all the applications making use of the LDAP server. These applications often receive some kind of user input that may be used to perform a request. If this user input is not correctly handled it could lead to security issues resulting in information disclosure, information alteration, etc. Commonly, LDAP injection attacks are performed against web apps, but of course you may find some other desktop applications making use of LDAP protocol.

## LDAP Query - String Search Criteria

LDAP Injection attacks are based on generating a user input that modifies the filtering criteria of the LDAP query. It is important to understand how these filters are formed.

RFC 4515 specifies the string representation of search filters which are syntactically correct on LDAP queries [4]. The Lightweight Directory Access Protocol (LDAP) defines a network representation of a search filter transmitted to an LDAP server. Some applications may find it useful to have a common way of representing these search filters in a human-readable form; LDAP URLs are an example of such application.

Search filters have the following form:

*Attribute Operator Value*

The string representation of an LDAP search filter is defined by the succeeding grammar, using the ABNF notation.

```
filter      = "(" filtercomp ")"
filtercomp = and / or / not / item
and         = "&" filterlist
or          = "|" filterlist
not         = "!" filter
filterlist = 1*filter
item        = simple / present / substring
            / extensible
simple       = attr filtertype value
filtertype = equal / approx / greater / less
```

```

equal      = "="
approx     = "~="
greater    = ">="
less       = "<="
present    = attr "="
substring  = attr "=" [initial] any [final]
initial    = value
any        = "*" *(value "*")
final      = value

```

As it is seen on the grammar, simple conditions can be combined using AND (&), OR (|) and NOT (!) operators, which must be between brackets.

The special character "\*" matches one or more characters on a filter string.

A few examples of this notation

```

(cn=Babs Jensen)
(! (cn=Tim Howes))
(&(objectClass=Person)(|(sn=Jensen)(cn=Babs J*))
(o=univ*of*mich*)

```

## LDAP Injection

LDAP Injection attack is just another kind of injection attacks. Basically, the idea behind this technique is to take advantage of an application that is not handling input values correctly. This can be achieved by sending some carefully crafted data to generate a LDAP query of our choice. When the application uses this user supplied values to build a LDAP query without prior validation or sanitizing, the attacker may force the execution of a statement by altering the construction of the LDAP query. Notice that once the attacker alters the statement, by adding arbitrary code, the process will run with the same privileges of a valid query. This is a mayor security risk issue that must be eradicated. [5] [6] [7].

LDAP injection attacks are commonly used against web applications. They could also be applied to any application that has some kind of input used to perform LDAP queries.

Depending on the target application implementation one could try to achieve:

- Login bypass
- Information disclosure
- Priviledge escalation
- Information alteration

Along the article, all these items will be discussed in detail. Do notice that some of these attacks could be handled in a different way depending on the LDAP server implementation due to different search filter interpretation in each of them.

### Login Bypass

An LDAP repository is normally used to validate

credentials. Basically, two simple ways to implement an authentication using LDAP can be distinguished:

- to use "bind" function or method to connect to the LDAP server.
- using an LDAP search query against the LDAP repository checking username and password fields.

### Bind Method

This authentication method cannot be bypassed easily but, depending on the application logic, one could end up with an anonymous bind.

This is a sample code you could find in a web application using a bind method: [8]

```

<?php
$ldapuser = $_GET['username'];
$dappass = $_GET['password'];

$ldapconn =
ldap_connect("ldap.server.com")
or die("Could not connect to server");

if ($ldapconn) {
    $ldapbind = ldap_bind($ldapconn,
    $ldapuser, $dappass);
    if (! $ldapbind) {
        $ldapbind =
        ldap_bind($ldapconn);
    }
}
?>

```

This code tries to perform a bind using the username and password provided. If that is not successful it ends with an anonymous bind.

This could be useful because if LDAP server security is not correctly configured an anonymous connection could be enough to obtain information with the other LDAP injection techniques discussed later on this section.

### Search Query

This kind of authentication is similar to the one any programmer should use with a standard database storing username and password information. The application will run a query to determine if username and password hash are correct.

An LDAP search query to accomplish this could be something like this:

```
(&(Username=user)(Password=passwd))
```

If the username and password values are not checked before using them in a search like the one above, we could insert particular values to alter the final query.

For example, we could enter this text in the username field: "user)(&))(" and anything in the

password field just in case it validates for empty field. This will produce the following query:

```
(&(Username=user) (&)) ((Password=zz))
```

Note that this query will always be true even with invalid passwords.

We could try different variations of the example used here because the search query could be written using single or double quotes. Consequently, one could try with these inputs:

```
(')(Username='validUsername') (&)) (  
\')(Username=\'validUsername\') (&)) (  
) (Username="validUsername") (&)) (  
\") (Username=\'validUsername\') (&)) (  
)
```

In this case, the attribute named Username is guessed since it is a very common attribute name.

## Information Disclosure

It is important for an attacker to get familiar with the existing structure in a company. Every bit of information available can aid strangers on their quest to attack a potential target. If the developer of a web application is not careful enough, simple applications can be twisted to obtain critical data. Depending on the internal LDAP query an application is using an attacker could alter it resulting in another LDAP query with more information.

Supposing an application is using a filter with an OR condition like:

```
(|(objectClass=device)(name=parameter1))
```

If the parameter supplied was as following:

```
"test)(objectClass=*
```

the resulting query would be:

```
(|(objectClass=device)(name=test)(objectClass=*))
```

This is a totally valid query but it is showing all object classes and not just the devices.

The same can be achieved if the application uses an AND condition instead of OR.

The filters above have a valid syntax, but if the application is not checking the final filter the attacker could try to create more than one filter in a single string. If this is sent to the LDAP Server, depending on the implementation the server could parse the string and take only the first complete and valid filter ignoring the rest.

For example, if the application internally uses a

filter like:

```
(&(attr1=userValue)(objectClass=device))
```

And the userValue is set to

```
test)(objectClass=*)) (&(1=1
```

it will generate a final filter like:

```
(&(attr1=test)(objectClass=*)) (&(1=1)(objectClass=device))
```

This string has 2 filters and each one of them by separate is valid. The LDAP server would then interpret the first filter (which is the one with the objectClass injected condition) and ignore the second one.

When performing this kind of attacks you can always try with some common LDAP attribute names like objectClass, objectCategory, etc.

## Charset Reduction

The objective of this technique is to enable the attacker to determine valid characters that form the value of a given object property. The purpose is to take advantage of the LDAP query wildcards to construct queries with them and random characters. Each time a query guess is run, if the query is successful (meaning that some information is retrieved) a part of the property value will be revealed to the attacker. After a finite number of successful guesses, an attacker will be in a position to guess the complete value (or at least to iterate between the character matches to find the correct order).

Supposing the target is

'http://ribadeohacklab.com.ar/people\_search.aspx'. By looking at the search page it was possible to determine that the LDAP objects being query have a 'last\_name', 'name', 'address', 'telephone' and a hidden 'zone' property (that was disclosed using one of the above techniques). By default the application is meant to give person details only from the 'public' zone. How could this limit be bypassed?

The following query is successful:

```
http://ribadeohacklab.com.ar/people_search.aspx?name=John)(zone=public)
```

Assuming that a 'John' is also part of a different zone we need to find a reasonable amount of characters to make a guess about a zone name. First thing to do is try to guess the first character of a different zone. Using the '\*' wildcard one could try to see if a zone begins with the character 'b':

```
http://ribadeohacklab.com.ar/people_search.aspx?name=Peter)(zone=b*)
```

This doesn't retrieve any results. After several attempts the following query:

```
http://ribadeohacklab.com.ar/people_search.aspx?name=Peter)(zone=m*)
```

Shows the following results:

```
name: John
last_name: Doe
address: Fake Street 123
telephone: 1234-12345
```

At this moment there are several choices. One could try to find the next character (like 'mo\*' if a vowel is present in the zone (like 'm\*i\*'), etc. After some trial and error attempts the desired result is achieved:

```
http://ribadeohacklab.com.ar/people_search.aspx?name=Peter)(zone=main)
```

It would be easy to use the value just found to gain further insight about the information stored.

This technique may look as a brute force approach, but the great advantage here is that every query will give the attacker a partial knowledge of the successful value string. An automated attack would be able to guess values without too much difficulty and if the attacker is clever, he could minimize the amount of queries needed to find a given value. For example, it would be possible to use a dictionary of words of a particular domain (like people names) to make a decision tree and then use it to run a wordlist attack using the wildcards.

## Privilege Escalation

To clarify, when speaking of a privilege elevation attack through LDAP injection, it is meant a change of privilege in the authentication structure represented by a schema stored in a LDAP database. In this particular case, the objects should have some kind of property that determines the access or security level required to work with them. Taking for example a product order repository located in the 'Sales' server, where not all users are able to see all the product orders, if the default query is:

```
(&(category=latest)(clearance=none)
```

only the following would be seen:

```
http://sales.ourdomain/orders.php?category=latest
```

```
Order A, Amount = 1000, Salesman = "John Doe"
```

```
Order C, Amount = 700, Salesman = "Jane Doe"
```

```
Order E, ...
```

Just by looking at the result set, it is plausible that something may be missing. So finding a higher 'clearance' level (just using a '\*' wildcard or by 'Charset Reduction', see *supra*) would be enough to access the missing information.

In the current example, the higher clearance level found is 'confidential' so if the application is vulnerable to injection, it is easy enough to use it in order to gain access to the remaining product orders.

Therefore:

```
http://sales.ourdomain/orders.php?category=latest)(clearance=confidential)
```

or

```
http://sales.ourdomain/orders.php?category=latest)(clearance=*)
```

show the following results:

```
Order A, Amount = 1000, Salesman = "John Doe"
```

```
Order C, Amount = 5000000, Salesman = "Joe Doakes"
```

```
Order B, Amount = 700, Salesman = "Jane Doe"
```

```
Order B, Amount = 1000000, Salesman = "Jannine Dee"
```

```
Order D, ...
```

Even with such a rough example the security risk of disclosing personal information of the top tier salesmen of this company is clear.

## Information Alteration

LDAP not only allows performing search operations, but also adding, modifying and deleting information.

It is not uncommon to find organizations with different applications for managing directory data without having to connect to the directory server. These applications use APIs to interact via LDAP with the information stored in the directory. If an application gets user inputs via a form in order to alter some information on the directory, the attacker may modify this data to find out the way to

generate an unexpected result, like modifying or deleting more information than the expected.

For example, PHP allows to modify data on a directory by simply using a LDAP library function, `ldap_modify()` [8]. This function is defined as:

```
bool ldap_modify ( resource $li ,
string $dn , array $entry );
```

where `$li` represents an LDAP link identifier, returned by `ldap_connect()` function, `$dn` is the distinguished name of the entry to be modified and `$entry` is the information to be modified.

```
<?php
    $attr["cn"] = "ToModify";
    $dn = "uid=Ribadeo,ou=People,dc=foo";
    $result = ldap_modify($ldapconn, $dn,
$attr);
    if (TRUE === $result) {
        echo "Entry was modified.";
    }
    else {
        echo "Entry could not be modified.";
    }
?>
```

If the application receives `$attr` and `$dn` as parameter, and the attacker enters `"uid=Ribadeo,ou=People,dc=*" as the $dn value, and if the input is not sanitized, all CN entries under the branch will be modified with the "ToModify" value. The same attack technique can be used on any function receiving the distinguished name as a user input provided value, like PHP function ldap_mod_replace(), ldap_mod_del() or ldap_delete().`

## URL encoding & Unicode encoding

Like with any other web application attack, one can always try the injections using URL encoding, [9] [10] and Unicode encoding. [11] Sometimes the web server along with the web app may incorrectly interpret the characters provided. For example, in a path traversal attack some kind of encoding is frequently used. An attacker will try to put `"..\\" in the url to go to another directory, and this may be achieved using valid and/or invalid encoding like http://example/..%255c..%255c..%255c boot.ini`

The LDAP techniques mentioned here also heavily rely on the treatment given to the user input, and even if the application is performing some kind of check against it, using some character encoding the attacker may bypass this and get what he/she is looking for.

With the LDAP search syntax in mind, we can always try to use some kind of encoding on characters like (, ), &, |, !, =, ~, \*, ', ".

## LDAP Injection vs. SQL Injection

Most applications nowadays use databases to store information. IT professionals have a deep knowledge of SQL not only because it is commonly used, but due to the fact that SQL is a declarative programming language in which you simply describe what the program should do but not how to accomplish it. Despite LDAP searches share characteristics of a declarative language, it is not as widely known by IT professionals as SQL is.

Sometimes, in order to avoid working with LDAP searches directly, some steps are performed to delegate query logic on a relational model instead of using a directory. Particularly, Windows Active Directory can be queried using SQL syntax by using Microsoft OLE DB Provider for Microsoft Active Directory Service. [12] This gives ADO applications the possibility to connect to heterogeneous directory services through ADSI, by creating a read-only connection to the directory service.

A common practice on Microsoft environments is to use this OLE DB Provider with SQL Server. In this case our application will be connecting to a SQL Server RDBMS and querying a relational model via SQL, but this relational structure will be obtaining its data from a Directory Service. In order to do so, a linked server against the AD server must be created. A linked server enables SQL Server to execute commands against OLE DB data sources on remote servers, without taking into account the type of technology of the remote server (an OLE DB provider must be available).

To create a linked server against Windows 2000 Directory Service `sp_addlinkedserver` system stored procedure has to be used with `ADSDSOObject` as the 'provider\_name' parameter and `adsdatasource` as the 'data\_source' parameter.

```
EXEC sp_addlinkedserver 'ADSI', 'Active
Directory Services 2.5', 'ADSDSOObject',
'adsdatasource'
```

Once the linked server is configured, the directory can be queried. The Microsoft OLE DB Provider for Microsoft Directory Services supports two command dialects, LDAP and SQL, to query the Directory Service. The `OPENQUERY` function [13] can be used to send a command to the Directory Service and consume its results in a `SELECT` statement. It executes the specified pass-through query on the given linked server which is an OLE DB data source. The `OPENQUERY` function can be referenced in the `FROM` clause of a query as if it

was a table. For example:

```
SELECT [Name], SN [Last Name], ST State
FROM OPENQUERY( ADSI,
'SELECT Name, SN, ST
FROM ''LDAP://ADserver/DC=ribadeohacklab
OU=Sales,DC=sales,DC=ribadeohacklab,
DC=com,DC=ar''
WHERE objectCategory = ''Person'' AND
objectClass = ''contact''')
```

A common practice is to create a view (a view is a virtual table that consists of columns from one or more tables which are the result of a stored select statement) based on the result of the select statement against the directory (via OPENQUERY), and then make our applications query this view (via common SQL syntax) in order to validate data from the directory.

This practice reduces our LDAP injection problem to a SQL injection one. At this point, one can apply all well known SQL injection and Blind SQL injection techniques. It is important to be aware of this kind of technology because deciding to use this option due to the ease of use, may introduce security risks.

Another common practice utilized to connect to an Active Directory repository is to use the same OLE DB provider for Active Directory Service, [14] without the SQL Server integration but with ADO objects. [15] Here is some Python sample code:

```
import win32com.client
def ADQuery(user,passwd,filters):
    #some constants for ADSI flags
    ADS_SECURE_AUTHENTICATION = 0x1
    ADS_SERVER_BIND = 0x200

    objConn = win32com.client.Dispatch("ADODB.Connection")
    COMCmd = win32com.client.Dispatch("ADODB.Command")

    objConn.ConnectionString = "Provider=ADsDSOObject;User Id=" + \
        user + ";Password="+ passwd + \
        ";Encrypt Password=True;ADSI Flag=" + \
        str(ADS_SECURE_AUTHENTICATION + ADS_SERVER_BIND)

    objConn.Open()

    COMCmd.ActiveConnection = objConn
    COMCmd.Properties("Page Size").Value = 500
    COMCmd.Properties("Searchscope").Value = 2
    COMCmd.Properties("Timeout").Value = 10

    COMCmd.CommandText = "SELECT displayName,sAMAccountName \
        FROM 'LDAP://SERVER/DC=DOMAINNAME' \
        WHERE objectCategory=%s\" % filters

    objRecordSet = COMCmd.Execute()[0]
    return objRecordSet
```

In the code, the connection string and the final query are created with some user input. This could allow for example, an alteration of the ADSI Flags used in the connection or some other type of connection string attack. [16]

If the password value entered was "s3cr3t;x" then the final and effective connection string would be:

```
Provider=ADsDSOObject;User
ID=someUser;Password=s3cr3t;Encrypt
Password=False;Extended
Properties="xxx;Encrypt
Password=True";Mode=Read;Bind
Flags=0;ADSIFlag=513
```

This means that the property that is located after the password parameter was changed by moving it to the "Extended Properties" and a default value appeared. So, depending on the implemented code one could even change ADSI flags or add extended properties that were not set by default.

Most importantly, the final query can be changed just because the "filters" parameter is not validated. Basically, this code converts a LDAP injection into a SQL injection.

As previously mentioned, this provider allows to use SQL syntax and also the LDAP search syntax so, depending on the application code an attack using any of the LDAP techniques mentioned before could also be performed.

Something interesting about this provider is that, since it has a particular syntax in which not only filters but also attributes and search scope are specified in the search string, [15] an attacker may extend the "information disclosure" technique.

## Prevention Techniques

LDAP Injection is just another type of Injection Attacks. As we have already discussed in this article, these kinds of attacks occur when an application (web or desktop application) sends to the LDAP interpreter user-supplied data inside the filter options of the statement. When an attacker supplies specially crafted data, the possibility to create, read, delete or modify arbitrary data gets unlocked. The most effective mitigation mechanism is to assume that all user inputs are potentially malicious. Assuming that, the following is clear: "user inputs must always be sanitized on server side (in order to avoid client side data manipulation) before passing the parameter to the LDAP interpreter".

This sanitizing procedure can be done in two different ways. The easiest one consists in detecting a possible injection attack by analyzing the parameter looking for certain known patterns attacks, aided by different programming techniques, like regular expressions. This technique has the main disadvantage of Type I statistical errors, also known as false positive cases. By applying this mechanism we might be excluding valid user inputs, mistaking them as invalid parameters.

A more sophisticated approach may include trying to modify the received user input to adapt it into a harmless one. This way, sanitizing the input would reduce the false positive cases.

In order to improve the effectiveness of this measure, it is advised to make a double check, both on client and server side. By checking the input format on the client side application usability is improved, due to the fact that the user is prevented from getting explicit core application errors with a user friendly message. This first level of filtering should consider most common mistakes. However, a server side user input filtering or modification is mandatory. At this level, one has to make sure that the parameter received has the structure that is supposed to have. For example, if a user name is expected, it should only contain alphanumeric characters and perhaps other kind of special characters like underscore, but it would be really strange to find a bracket, an ampersand or an equal symbol. This can be checked by using a regular expression like "`^[A-Za-z0-9_-]+`". If we are

using PHP, a similar code can be used:

```
<?php
$user=$_GET['username'];
$UsrcRegex = "/(^[A-Za-z0-9_-]+)$/";

if preg_match($UsrcRegex,$user){

$dn = "o=My Company, c=US";
$filter="(|(sn=$username*)
        (givenname=$username*))";
$sr=ldap_search($ds, $dn, $filter);
}
else {
    print "Invalid UserName";
}
?>
```

As it was discussed before -URL encoding & Unicode encoding -, any programmer must know that some type of character encoding could be used in parameters and this has to be validated as well. For example, if the application is using APIs like MultiByteToWideChar or WideCharToMultiByte to translate Unicode characters, some code review may be needed since their incorrect usage could also lead to security issues. [17]

Another concept that must be taken into account are the error formats. Errors should give the attacker as little information as possible. This is extremely important because if attackers can reach any kind of conclusion based on error messages, this is helping them to make the attack easier. For example, if the attacker sends an invalid input in a form, by getting an error message that is returned by the server after the execution, it is easy to realize that the LDAP queries are executed without prior validation, what makes the application eligible for a possible exploit target.

As a general conclusion, we can say the best way to avoid this kind of injection attacks is to always mistrust from the parameters obtained from user input and always validate them before using to build a query.

## Tools

As shown, there are different techniques and trying all of them by hand could be very time consuming. Fortunately, there are some tools that automate LDAP injection attacks and help you find vulnerabilities. This article does not intend to list all of the existing tools, so here are briefly mentioned some of them.

### W3AF

This is a well known web attack and audit framework completely developed in Python. You can download it from <http://w3af.sourceforge.net>. This framework has a plugin named LDAPi which

can perform LDAP injections against a web application. By modifying the LDAPi.py plugin the user can add new strings to test on the injection attack.

### LDAP Injector

This is a tool developed by Informatica64 which can be downloaded from

[http://www.informatica64.com/foca/download/ldapInjector\\_0\\_2\\_1\\_0.zip](http://www.informatica64.com/foca/download/ldapInjector_0_2_1_0.zip)

The tool has a GUI that will let the user perform dictionary based attacks replacing values and analyzing responses and will also perform an attack by reducing the valid charset and then applying boolean analysis to find valid values.

This blog post (in Spanish) shows an example on how to use the tool: <http://elladodelmal.blogspot.com/2009/04/ldap-injector.html>

### JBroFuzz

This is a web app fuzzer you can download from OWASP at

[http://www.owasp.org/index.php/Category:OWASP\\_JBroFuzz](http://www.owasp.org/index.php/Category:OWASP_JBroFuzz)

This tool was developed in Java and has multiplatform support. It has a GUI with different fuzzing options with some graphing features to report results.

It has several fuzzers grouped by categories, and there's one for LDAP injections.

### Wapiti

Wapiti is a command line web app vulnerability scanner also developed in Python. You can find it at <http://wapiti.sourceforge.net>

It performs scans looking for scripts and forms where it can inject data. Once it gets this list, it acts like a fuzzer, injecting payloads to see if a script is vulnerable. There are some config files containing different payloads that can be customized.

### wsScanner and Web2Fuzz

wsScanner is a toolkit for Web Services scanning and vulnerability detection and Web2Fuzz is a web app fuzzing tool both developed by Blueinfy Solutions. You can obtain them from <http://blueinfy.com/tools.html>

These tools have a GUI and share some functionality. They allow to define the fuzzing load to use while scanning. This allows the user to define custom LDAP injection payloads and see the result.

Web2Fuzz tool also let the user choose different character encoding options to apply to the payloads.

### Wfuzz

This tool is a web bruteforce scanner developed in Python by Edge-Security. You can download it from <http://www.edge-security.com/wfuzz.php>

It performs different kind of injections attacks including some basic LDAP injection.

This application has some text files storing injection attacks and they can be customized by adding more injection patterns.

## References and further reading

[1] **Understanding LDAP - Design and Implementation - IBM RedBook**

<http://www.redbooks.ibm.com/redbooks/pdfs/sg244986.pdf>

[2] **OpenLDAP**

<http://www.openldap.org/>

[3] **Active Directory LDAP Compliance**

<http://www.microsoft.com/windowsserver2003/techinfo/overview/ldapcomp.msp>

[4] **RFC 4515 - String Representation of Search Filters**

<http://www.ietf.org/rfc/rfc4515.txt>

[5] **LDAP Injection and Blind LDAP Injection - Black Hat 08 Conference - Alonso - Parada**

<http://www.blackhat.com/presentations/bh-europe-08/Alonso-Parada/Whitepaper/bh-eu-08-alonso-parada-WP.pdf>

[6] **Web Application Security Consortium - LDAP Injection**

[http://www.webappsec.org/projects/threat/classess/ldap\\_injection.shtml](http://www.webappsec.org/projects/threat/classess/ldap_injection.shtml)

[7] **OWASP LDAP Injection**

[http://www.owasp.org/index.php/LDAP\\_injection](http://www.owasp.org/index.php/LDAP_injection)

[8] **PHP - LDAP Manual**

<http://php.net/manual/en/book.ldap.php>

[9] **HTML URL Encoding Reference**

[http://www.w3schools.com/TAGS/ref\\_urlencode.asp](http://www.w3schools.com/TAGS/ref_urlencode.asp)

[10] **Percent-encoding**

<http://en.wikipedia.org/wiki/Percent-encoding>

[11] **Unicode / UTF-8 encoded directory traversal**

[http://en.wikipedia.org/wiki/Directory\\_traversal#Unicode\\_UTF-8\\_encoded\\_directory\\_traversal](http://en.wikipedia.org/wiki/Directory_traversal#Unicode_UTF-8_encoded_directory_traversal)

[12] **OLE DB Provider for Microsoft Directory Services**

<http://msdn.microsoft.com/en-us/library/ms190803.aspx>

[13] **OPENQUERY**

<http://msdn.microsoft.com/en-us/library/aa276848%28SQL.80%29.aspx>

[14] **Microsoft OLE DB Provider for Microsoft Active Directory Service**

<http://msdn.microsoft.com/en->

[us/library/ms681571\(VS.85\).aspx](http://us/library/ms681571(VS.85).aspx)

**[15] How To Use ADO to Access Objects Through an ADSI LDAP Provider**

<http://support.microsoft.com/kb/187529>

**[16] Connection String attacks (spanish)**

<http://elladodelmal.blogspot.com/2009/09/conexion-string-attacks-i-de-vi.html>

**[17] Security of MultiByteToWideChar and WideCharToMultiByte**

<http://blogs.msdn.com/esiu/archive/2008/11/06/in-security-of-multibytetowidechar-and-widechartomultibyte-part-1.aspx>

<http://blogs.msdn.com/esiu/archive/2008/11/14/in-security-of-multibytetowidechar-and-widechartomultibyte-part-2.aspx>

For further reference links used for this article go to <http://www.ribadeohacklab.com.ar/articles/ldap-injection-hitb>

## About the Authors

RibadeoHackLab was formed by a group of technology enthusiasts on June 2009. Its main purpose is to publish investigations and findings related to information security. Its current members are Esteban Guillardoy, Facundo de Guzman and Hernan Abbamonte.

Esteban was born in 1982 and is about to graduate as Informatics Engineer from Universidad de Buenos Aires. He is an experienced consultant on security and operations monitoring, working as lead technical consultant in the Technology Team in Tango/04 Computing Group (<http://www.tango04.com>), conducting and developing different projects.

Facundo was born in 1982 and is an advanced student of Information Systems Engineering at Universidad Tecnologica Nacional. He works as technical consultant on Technology Team in Tango/04 Computing Group, leading and developing projects on infrastructure and security monitoring.

Hernan was born in 1985 and he holds a degree as Information Systems Engineer from Universidad Tecnologica Nacional. Currently he is doing a Master Course on Information Security at Universidad de Buenos Aires. He works as technical consultant on Technology Team in Tango/04 Computing Group, leading and developing monitoring projects on different technologies.

The group has a variety of interest, including, reverse engineering, security software development, penetration testing, python programming, operating systems security and database security.

For further information you can visit us at <http://www.ribadeohacklab.com.ar>