Fall 12-7-2015

Detection of Lightweight Directory Access Protocol Query Injection Attacks in Web Applications

Pranahita Bulusu
Kennesaw State University

Detection of Lightweight Directory Access Protocol Query Injection Attacks in Web Applications

Master's Thesis

by

Pranahita Bulusu
MSCS Student
Department of Computer Science
Kennesaw State University, USA

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

December 2015

Detection of Lightweight Directory Access Protocol Query Injection Attacks in Web Applications

This thesis is approved for recommendation to the Graduate Council.

Hisham Haddad

Hisham M. Haddad
Thesis Co-Advisor

Hossain Shahriar

Hossain Shahriar
Thesis Co-Advisor

Ken Hogan

Ken Hoganson
Department Chair, Committee Member

Ying Xie
MSCS Director, Committee Member

Selena He
Assistant Professor, Committee Member

Kennesaw State UNIVERSITY
Graduate College

Thesis/Dissertation Defense Outcome

Name Pranahita Bulusu KSU ID 000526632

Email pbulusu@students.kennesaw.edu Phone Number 470-578-4389

Program MSCS

Title: Detection of Lightweight Directory Access Protocol Query Injection Attacks in Web Applications

Thesis/Dissertation Defense: ☑ Passed □ Failed Date 11/19/2015

All courses required for the degree have been completed satisfactorily ☑ YES □ NO

Signatures

Hisham Haddad Hossain Shahriar 11/30/2015
Thesis/Dissertation Chair/Major Professor Date

11/30/2015
Committee Member Date

11/30/2015
Committee Member Date

11/30/15
Committee Member Date

Committee Member Date

11/30/15
Program Director Date

11/30/15
Department Chair Date

11-30-15
Graduate Dean Date

Rev. 2/15/12

DEDICATION

This thesis is dedicated to my family,

I love you and thank you for always being there for me.

ACKNOWLEDGEMENTS

I would like to thank Dr. Hisham Haddad and Dr. Hossain Shahriar
for their support and encouragement through this entire process.
I am very thankful for this experience.

I would also like to thank my thesis committee members, Dr. Hoganson,
Dr. Xie and Dr. He for their insightful comments and valuable suggestions.

This research paper is made possible through the help and support from
everyone, including my professors, parents, my husband, family and friends.

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The Lightweight Directory Access Protocol (LDAP) is a common protocol used in organizations for Directory Service. LDAP is popular because of its features such as representation of data objects in hierarchical form, being open source and relying on TCP/IP, which is necessary for Internet access. However, with LDAP being used in a large number of web applications, different types of LDAP injection attacks are becoming common. The idea behind LDAP injection attacks is to take advantage of an application not validating inputs before being used as part of LDAP queries. An attacker can provide inputs that may result in alteration of intended LDAP query structure. LDAP injection attacks can lead to various types of security breaches including (i) Login Bypass, (ii) Information Disclosure, (iii) Privilege Escalation, and (iv) Information Alteration. Despite many research efforts focused on traditional SQL Injection attacks, most of the proposed techniques cannot be suitably applied for mitigating LDAP injection attacks due to syntactic and semantic differences between LDAP and SQL queries. Many implemented web applications remain vulnerable to LDAP injection attacks. In particular, there has been little attention for testing web applications to detect the presence of LDAP query injection attacks.

The aim of this thesis is two folds: First, study various types of LDAP injection attacks and vulnerabilities reported in the literature. The planned research is to critically examine and evaluate existing injection mitigation techniques using a set of open source applications reported to be vulnerable to LDAP query injection attacks. Second, propose an approach to detect LDAP injection attacks by generating test cases when developing secure web applications. In particular, the thesis focuses on specifying signatures for detecting LDAP injection attack types using Object Constraint Language (OCL) and evaluates the proposed approach using PHP web applications. We also measure the effectiveness of generated test cases using a metric named Mutation Score.

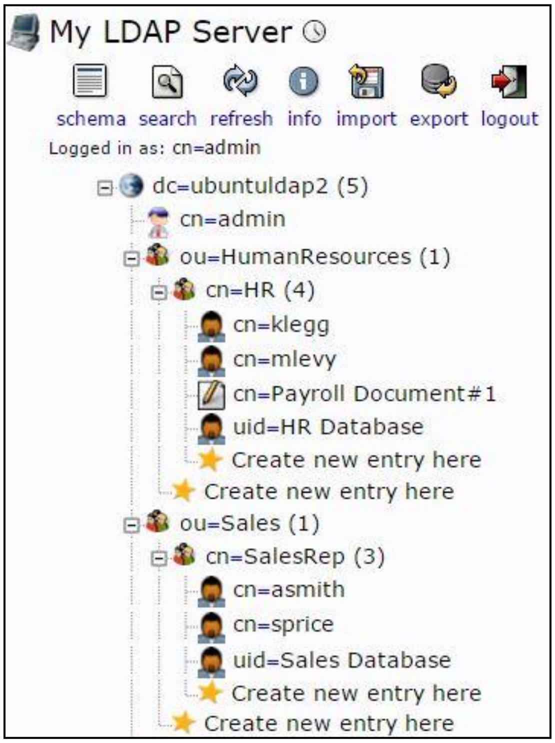
TABLE OF CONTENTS

CHAPTER 1

Motivation, Problem Statement and Contribution

1.1 Background

LDAP is a protocol used to access and maintain directory services. LDAP uses a client-server model for accessing directory information. The data required to form the Directory Information Tree (DIT) is stored in one or multiple LDAP servers [1]. The data models in directories enabled with LDAP are represented hierarchically to make the information easily accessible. Along with the hierarchical representation, LDAP also provides a standardized method of local and remote data access. Local access standards are provided by Relational Database Management Systems (RDBMS) systems such as SQL. However, remote access standards are usually proprietary. LDAP provides a method to move data to multiple locations without affecting any external access to the data. Such features and usability make LDAP unique and popular for its use in Directory services [2].



Logged in as: cn=admin

```
dc=ubuntuldap2 (5)
  cn=admin
  ou=HumanResources (1)
    cn=HR (4)
      cn=klegg
      cn=mlevy
      cn=Payroll Document#1
      uid=HR Database
      Create new entry here
    Create new entry here
  ou=Sales (1)
    cn=SalesRep (3)
      cn=asmith
      cn=sprice
      uid=Sales Database
      Create new entry here
    Create new entry here
```

Figure 1: Directory tree structure of LDAP server

Figure 1 shows an example of LDAP directory tree structure. The tree is subdivided into different Organizational Units *(ou)* along with common names for each of them *(cn)*. For example, the organizational unit of Human Resources has common name as HR. Different entries of each organizational unit are given under the common name such as the employees working in a particular department and any document relevant to the particular department.

The LDAP is used in a large number of web applications, and therefore, different types of injection attacks are common. The idea behind LDAP injection [3] attack is to take advantage of an application's vulnerability of not validating user inputs properly. A vulnerable application suffering from LDAP injection attacks can be exploited by providing carefully crafted input data containing parts of the LDAP query. After including the attacker's inputs, the intended structure of LDAP query gets altered. When the altered query is executed, many unwanted activities can take place leading to security breaches *(e.g., login bypass)*. A vulnerable application cannot differentiate a malicious query generated based on attacker's supplied inputs and legitimate query generated based on benign inputs. LDAP injection attacks, such as Login Bypass, can lead to various types of security breaches.

There are various possibilities of how LDAP injection attacks can be used to exploit a particular application. LDAP injection attacks allow attackers to disclose potentially sensitive information and manipulate certain data in the underlying database. As for example, in a popular event planner application, *Events Planner - SmarterMail 7.x (7.2.3925)* [4], LDAP injection vulnerability has been discovered where input type parameters can be provided to alter a disjunctive (OR) query to conjunctive (AND) query or vice versa. With this type of injection technique, an attacker can retrieve sensitive information. In many cases, administrators tend to configure LDAP server insecurely due to lack of knowledge. Thus, a simple injection technique could access user information or even change the password of the administrator. Many issues arise after LDAP being enabled because the applications were not tested with LDAP as the default protocol [5].

Different types of injection attacks such as SQL injections [6], LDAP injections [3] etc. have been prevalent among web applications over the last decade. In fact, query injection flaws remain as one to the top ranked security vulnerabilities according to Open Web Application Security Project (OWASP) report [7]. Among several common query injection attacks, much of the research efforts have been made to detect and prevent SQL injection attacks [8, 9, 10, 11, 12]. In contrast, LDAP injection attacks have not received enough attention.

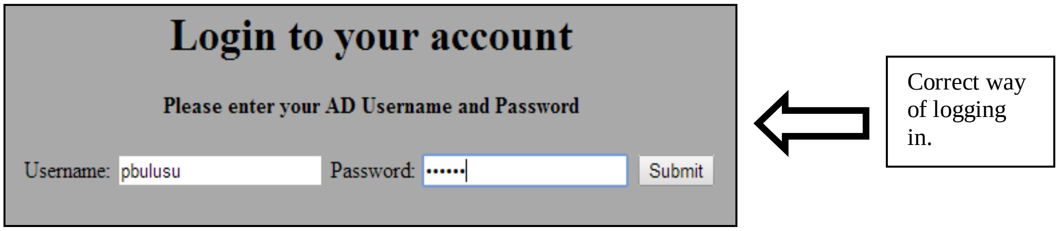
1.2 Motivation

LDAP injection attacks take place when an application does not validate user inputs properly. This vulnerability leads to exploitation of an application by providing carefully crafted data containing parts of the LDAP query. When the altered query is executed, it leads to different types of security breaches. Depending on the target application implementation, one could try to achieve any of the following types of attacks, including Login Bypass, Information Disclosure, Privilege Escalation, and Information Alteration. The attack types discussed in this Section have been gathered from the literature and technical reports [7, 13, 14, 15].

In this thesis, we have replicated these attacks with a prototype PHP web application employing backend LDAP server. Details of each of injection type are given below.

Login Bypass

First, we show an authorized access in Figures 2 and 3.



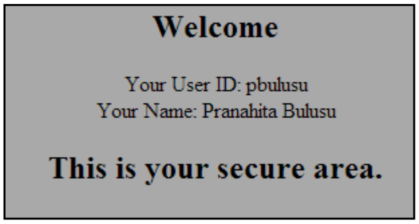
Login to your account

Please enter your AD Username and Password

Username: pbulusu Password: ••••••

Correct way of logging in.

Figure 2: A snapshot of a web application interface showing authorized access



Welcome

Your User ID: pbulusu
Your Name: Pranahita Bulusu

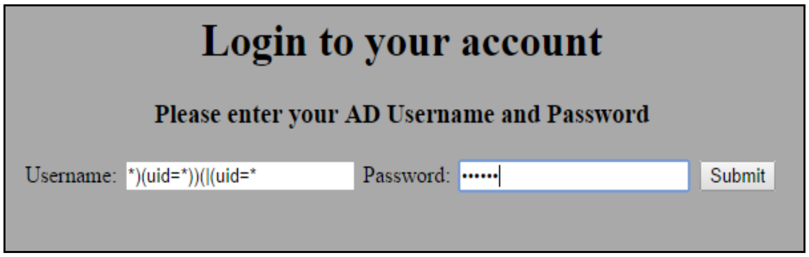
This is your secure area.

Figure 3: Resultant page after an authorized login

Figure 2 shows the login interface for a web application. The Figure shows an authorized access by providing the username as *pbulusu* and password as *123456* which leads to the response page shown in Figure 3. The search filter becomes

```
searchlogin = "(&(uid="pbulusu")(password="123456"))";
```

However, as shown in Figures 4 and 5, an attacker can login to the application bypassing the need of supplying valid username and password. If the provided username and password values are not validated before applying them to generate a query intended to perform a search operation, the query gets altered. An attacker commonly applies valid special characters supported by the LDAP query engine such as **&**, **|**, **(**, and **)**.

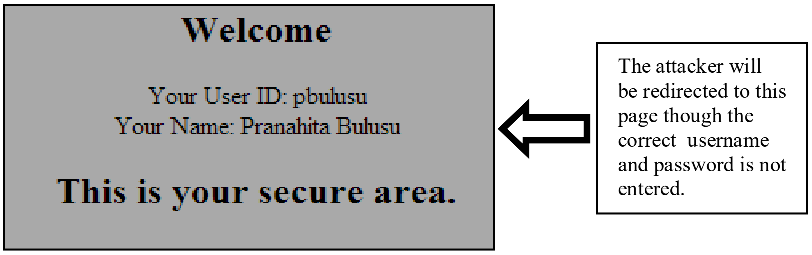


Login to your account

Please enter your AD Username and Password

Username: *)(uid=*))(|(uid=*   Password: ••••••

Figure 4: A snapshot of unauthorized login



Welcome

Your User ID: pbulusu
Your Name: Pranahita Bulusu

This is your secure area.

The attacker will be redirected to this page though the correct username and password is not entered.

Figure 5: Resultant page after an unauthorized login

Figure 4 shows an attempt of login bypass attack where an attacker provides the following string to bypass the login page [16]. The search filter becomes:

```
$searchlogin = "(&(uid=*)(uid=*))(|(uid=*)(password="abcdef"))";
```

When the application runs this query in the backend LDAP server, it returns all available records leading to the login for the attacker based on the first available username. Figure 5 shows a snapshot of the resultant page after the injection attack where the attacker gets access to the application with the same privilege of first username *pbulusu*. Note that the inputs supplied at the user interface are not validated by the server side application, hence leading to this security breach.

In a search query as stated above, if the username and password values are not checked, one could alter the dynamic query by inserting particular values. Special characters, such as `*`, `&`, `|`, `(`, and `)`, could be used to alter the final query’s purpose or intention. Though the correct user name and password may not be provided for a particular user, one can still get an access to the user account.

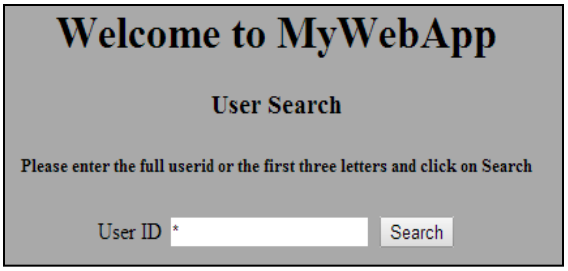
Information Disclosure

An attacker could alter a LDAP query thereby modifying it to another LDAP query with more information. This could be done depending on the internal LDAP query being used by the application. Figure 6 shows LDAP search operation code.

```
1. $conn = ldap_connect("servername");
2. $search_string = $_POST[‘search’];
3. $users = ldap_search ($conn,"uid=$search_string");
```

Figure 6: Code for LDAP search operation

Line 1 shows a user gets connected to the LDAP server. Line 2 shows the search string (`$search_string`) that the user has entered in a form (`$_POST[‘search’]`). Line 3 shows the search function, `ldap_search`, used to search for the provided input. An attacker enters `*` as the input. Therefore, line 3 becomes `$users = ldap_search($conn,"uid=*");`
Figure 7 shows the attacker uses only the `*` symbol instead of a valid user name, and gets access to the information of all employees in the system (Figure 8).



Welcome to MyWebApp

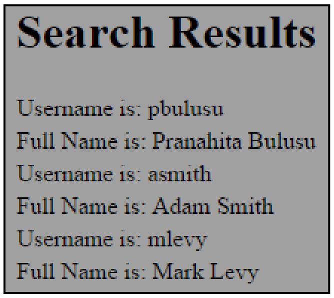
User Search

Please enter the full userid or the first three letters and click on Search

User ID *

Only the "*" symbol is being used instead of a valid user name in this case.

Figure 7: A snapshot of information disclosure attack



Search Results

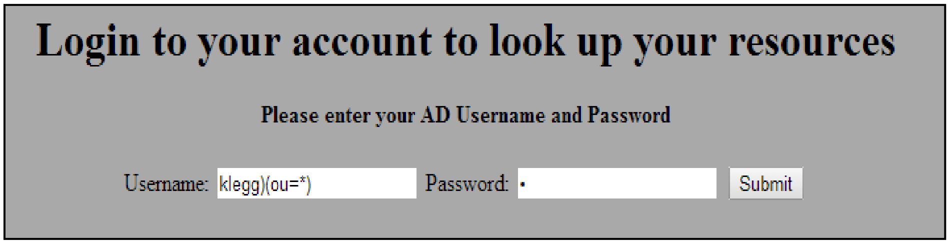
Username is: pbulusu
Full Name is: Pranahita Bulusu
Username is: asmith
Full Name is: Adam Smith
Username is: mlevy
Full Name is: Mark Levy

The attacker can get access to the entire Directory since the code is vulnerable.

Figure 8: The resultant page after information disclosure attack

Privilege Escalation

An attacker could alter a LDAP query modifying it to another query with the intention of gaining more privilege defined by the security level of the objects. This could be done depending on the internal LDAP query being used by the application. Figures 9 and 10 illustrate privilege escalation attack.

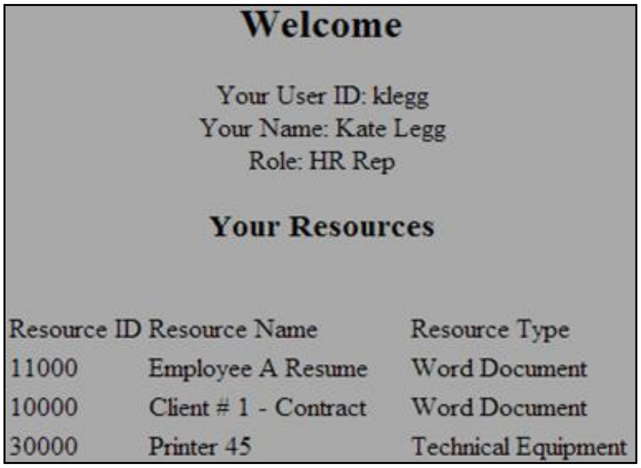


Login to your account to look up your resources

Please enter your AD Username and Password

Username: klegg)(ou=*) Password: •

Figure 9: A snapshot of privilege escalation attack



Welcome

Your User ID: klegg
Your Name: Kate Legg
Role: HR Rep

Your Resources

Resource ID	Resource Name	Resource Type
11000	Employee A Resume	Word Document
10000	Client # 1 - Contract	Word Document
30000	Printer 45	Technical Equipment

Figure 10: The resultant page after privilege escalation attack

In Figure 1, user `klegg` and `mlevy` are listed under `ou - HumanResources.` Each of them has different access privileges to various resources. In Figure 9, the user `mlevy` (attacker) is providing the user name as `klegg(ou=*` and the password as `*`. As a result, user `mlevy` will be able to gain access privileges of user `klegg` (victim). Now, `mlevy` has access to klegg's resources shown in Figure 10.

Information Alteration

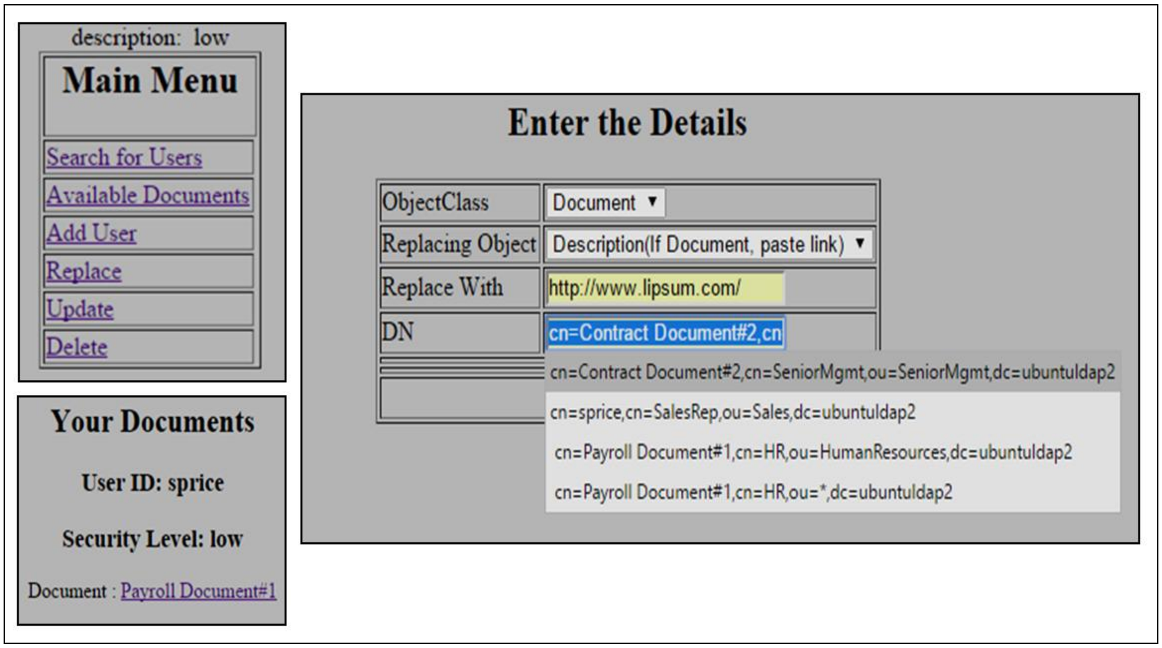
LDAP can be used for adding, modifying and deleting information along with search operations. Different applications that manage directory data in organizations are not necessarily connected to the directory server. Instead, Application Programming Interfaces (APIs) are used to interact via LDAP with the information stored in the directory. If the user provides inputs to an application through a form, an attacker may modify this information to generate an unexpected result such as modification or deletion of information [14]. Figure 11 shows code for information alteration.

```
$attr["cn"] = "klegg";
$dn = "uid=klegg,ou=*";
$result = ldap_modify($ldapconn,$dn, $attr);
```

Figure 11: Code for information alteration

In Figure 1, `klegg` belongs under organizational unit (`ou`) HumanResources. However, at Line 2, and distinguished name (`$dn`) is formed having `ou=*`. The alteration of `ou` is invoked by the method `ldap_modify()` at Line 3. As a result, `klegg` now belongs to the other `ou` (Sales).

Figures 12 and 13 illustrate an example of information alteration type of LDAP injection attack. In Figure 12, the user `sprice` of `ou` Sales Representative who has security level as 'low' accesses and replaces the Contract Document #2 which belongs to `ou` Senior Management with the security level 'high'. In applications which are vulnerable to LDAP injection attacks, an attacker can replace information successfully as shown in Figure 13.



description: low

Your Documents

User ID: sprice

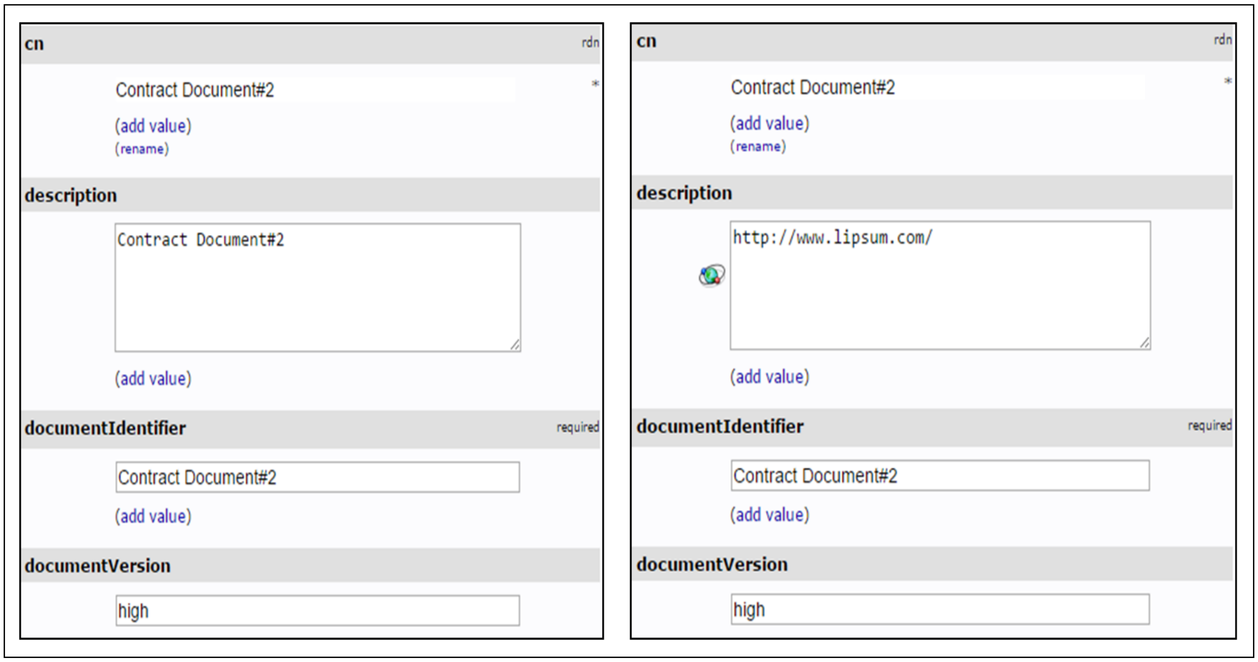
Security Level: low

Document : Payroll Document#1

ObjectClass	Document
Replacing Object	Description(If Document, paste link)
Replace With	http://www.lipsum.com/
DN	cn=Contract Document#2,cn

cn=Contract Document#2,cn=SeniorMgmt,ou=SeniorMgmt,dc=ubuntuldap2
cn=sprice,cn=SalesRep,ou=Sales,dc=ubuntuldap2
cn=Payroll Document#1,cn=HR,ou=HumanResources,dc=ubuntuldap2
cn=Payroll Document#1,cn=HR,ou=*,dc=ubuntuldap2

Figure 12: A snapshot of information alteration attack



cn: Contract Document#2
description: Contract Document#2
documentIdentifier: Contract Document#2
documentVersion: high

cn: Contract Document#2
description: http://www.lipsum.com/
documentIdentifier: Contract Document#2
documentVersion: high

Figure 13: Resultant changes after information alteration attack

1.3 Problem Statement

Protection of LDAP-enabled web applications involves significant effort for the administrators and developers. Though prevention approaches of LDAP injection attacks are available *(e.g., administrator and developer techniques)*, they cannot eliminate LDAP injection attacks completely. This work is an effort to research and propose a new approach to address LDAP injection attack types.

During our literature search, we identified a list of common limitations among past research efforts. These limitations are summarized as follows:

- Lack of exhaustive LDAP injection attack type detection coverage. In particular, most efforts are effective to detect only login bypass attacks. However, they are not suitable for mitigating privilege escalation attacks that may occur through legitimate login of a user in vulnerable applications.
- Little support to enable developers to securely implement web applications resistant to LDAP injection attacks. In particular, there is no effort of formally specifying attack

signatures which should be tested with suitable test inputs for the presence of LDAP injection vulnerabilities.

Given the obtained literature search results, we define the problem statement for this thesis as follows:

This research work addresses common limitations found in past research efforts by performing an in-depth study of LDAP injection attack types, provides an approach to detect the different types of LDAP injection attacks, and evaluates the proposed approach using PHP web applications known to be vulnerable to LDAP injection attacks.

1.4 Research Methodology

The research methodology comprised of an intensive literature review of over 40 articles consisting of information on LDAP injection attacks and mitigation techniques. The research methodology involves the following activities:

1) Conduct literature search on existing LDAP injection attack types and their prevention techniques.
2) Study and analyze collected information to understand how LDAP injection attacks are performed by the attackers and how they are executed at the server side.
3) Develop a prototype web application to replicate selected LDAP injection attacks to specify the signatures of LDAP injection attacks.
4) Develop a technique to detect LDAP injection attacks based on the identified attack signatures.
5) Evaluate the proposed detection technique against PHP web applications.

1.5 Contribution

The objectives of this thesis are to conduct an in-depth survey of various types of LDAP injection attacks; study LDAP injection attacks for various scenarios and understand their impact; develop a taxonomy of LDAP code injection attacks; select a set of tools to be studied to compare the suitability of LDAP injection detection; and develop a new technique to overcome existing limitations and detect LDAP injection attacks.

The work addresses the stated problem statement by performing the following tasks:

1) Conduct Literature Search and Develop Web Application
   1. Conduct literature survey on existing techniques and methodologies used to prevent LDAP injection attacks and critically examine the code of existing injection mitigation techniques.
   2. Compile the literature research results and document the findings for conference submission.
   3. Develop a web application to replicate selected LDAP injection attacks and to check the effect it has on the application.

2) Develop Detection Technique
   1. Apply Object Constraint Language-based (OCL) to specify signatures for LDAP injection attack types. With OCL, we capture the needed pre-conditions, post-conditions, and invariants that might get affected due to LDAP query injection attacks.
   2. Develop an algorithm to perform fault-injection on OCL constraints (pre-conditions and post-conditions).

3) Perform Evaluation and Dissemination
   1. Apply and evaluate the proposed technique with a developed PHP web application and one open source PHP web application reported to have LDAP injection vulnerability.
   2. Disseminate the work results through conference publications.

In the next Chapter, we discuss the literature search findings about the various mitigation techniques for LDAP injection attacks.

CHAPTER 2

Literature Review

2.1 Overview

Many research works have been done in the past to prevent LDAP injection attacks [17, 18, 19, 20, 21]. This Chapter presents a literature review of related work on LDAP injection attacks and highlights common Mitigation Techniques (MTs). This chapter also contrasts SQL Injection and LDAP Query Injection attacks.

2.2 Mitigation Techniques for LDAP Injection Attacks

Upon literature review of existing mitigation techniques for LDAP query injection attacks, we classify these techniques into six categories (MT1-MT6) described below.

MT1: Administrator techniques

Password Policy Schema – LDAP has an overlay called Password Policy (*ppolicy*) to prevent LDAP injection attacks. The default policy of *ppolicy* is that the user account gets locked for 24 hours after 10 failed access attempts so as to prevent unauthorized access [13].

LDAP Configuration – This is another approach in which access control is implemented on the data in the LDAP directory, especially during configuration of permissions on user objects, and also when the directory is used for single sign-on solution. The access level permitted to the users can be limited wherein they are not allowed to make any modifications thereby preventing LDAP injection attacks [13 and 15].

IP Firewall – Access can be restricted by using the IP firewall capability of the server system. This is either based on the clients' IP address and network interface, or only the network interface used to communicate with the client. The configuration of IP firewall are dependent on the type of IP firewall used [22].

MT2: Developer techniques

Incoming data Validation and Dynamic checks – This is another technique of prevention. All client supplied data should be thoroughly checked for any kind of malicious input. The best way of achieving this is to default-deny everything else other than letters and numbers. However, if symbols need to be used, they should be converted to HTML substitutes before usage [13].

Another prevention technique is to escape special characters. Most of the LDAP injection attacks are performed using special characters either in the 'Distinguished Name' field or in the 'Search' filter. Escaping these characters help prevent LDAP injection attacks.

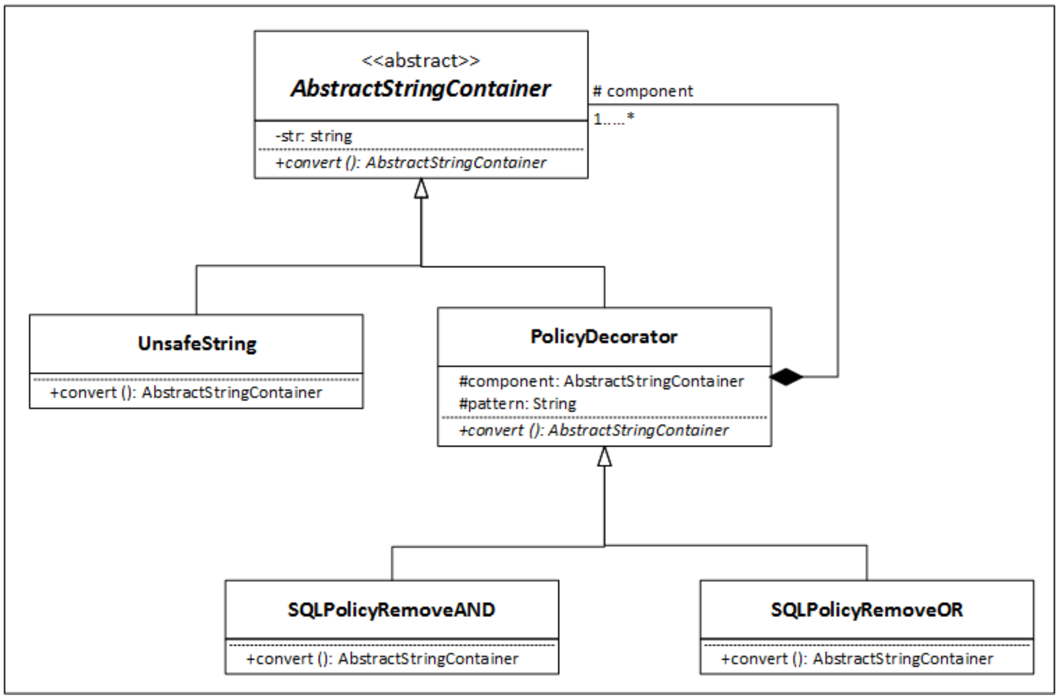
Characters such as &,!,|,=,<,>,,,+,-,",', and ;, should be escaped using \ before being used in a query; while characters such as (,),\,*,/ and **Null** used in the Search filter can be escaped using {\ASCII}, which are given as {\28},{\29},{\5c},{\2a},{\2f} and {\0} respectively. It is a good practice to include '\\' at the beginning of escaped character listings to prevent recursive replacements [23].

MT3: Program Transformation technique

The security of a system perimeter can be improved by using security oriented program transformations by introducing the components of authentication, authorization and input validation. These three components play a critical role in the security of any system as they form the basis for prevention of exploitation when applied effectively. When the code is developed for a particular system, the general approach is to design security from the base level and fix new vulnerabilities when a security threat is faced. Since it is not feasible to redesign the entire software whenever a security threat emerges, the vulnerability is fixed only at a certain set of points. In either way, the vulnerability is not fixed globally. Though the approach of security-oriented program transformations is compared to 'refactoring', it does not preserve the original behavior of the system; instead, it preserves the expected behavior and responds in accordance with the attacks.

The developer plays a key role in specifying the input validation policies as well as parameters for the program transformation mechanism. Once the input validation policy and the parameters are known, the implementation of program transformations becomes faster and more reliable.

The input validation task can either be achieved by adding a centralized perimeter filter thereby eliminating duplication of code or multiple validation policies can be applied to the input variable by using 'Decorated Filter'. Figure 14 shows how a SQL injection attack could be eliminated by using 'Decorated Filter'. However, the policies to be applied for the input validation of SQL and LDAP injection are different; hence the developer can either use a library of filters or customize it by using parameters.



<<abstract>>
AbstractStringContainer
-str: string
+convert (): AbstractStringContainer
# component
1.....*
UnsafeString
+convert (): AbstractStringContainer
PolicyDecorator
#component: AbstractStringContainer
#pattern: String
+convert (): AbstractStringContainer
SQLPolicyRemoveAND
+convert (): AbstractStringContainer
SQLPolicyRemoveOR
+convert (): AbstractStringContainer

Figure 14: Elimination of SQL injection attack using 'Decorated Filter' Transformation

In this technique of adding security on demand, when an unsafe input occurs, it is transformed to a safe input thereby preventing the attack. By using automated tools in this type of rectification policy, programmers can focus on policies instead of writing and implementing checks, which is a time consuming task. Program transformations are currently used to eliminate SQL injection attacks, Log injection attacks, XSS injection attacks, and Direct Static Code injection attacks by implementing policies and removing AND and OR statements. However, they can be similarly implemented to avoid LDAP injection attacks as well [17 and 18].

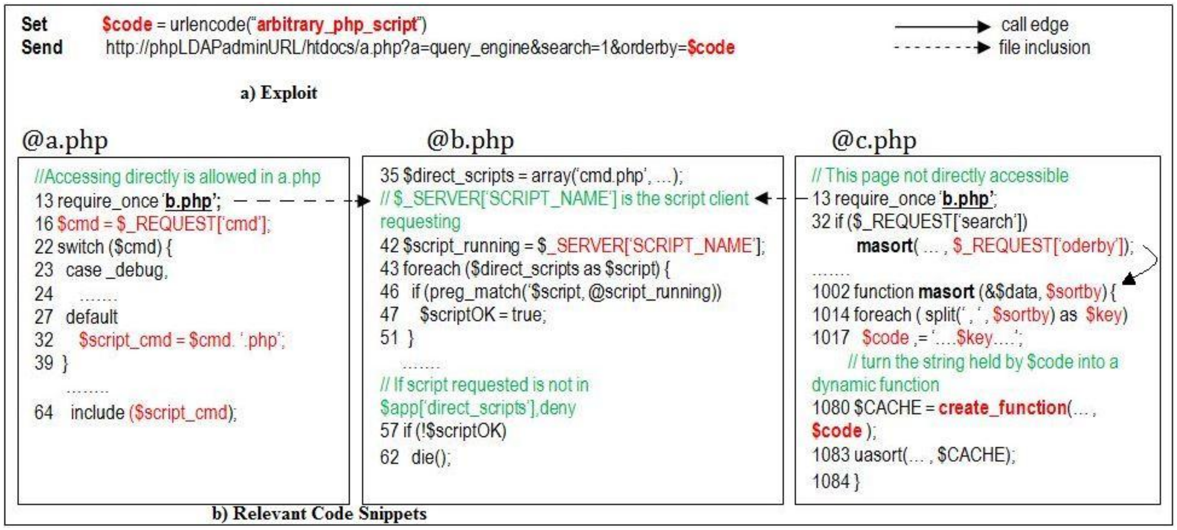
MT4: Application Security IDE technique

Application Security IDE (Integrated Development Environment) is an approach that works similar to spelling and grammar checks in word processor, indicating potential errors while developing a program thereby allowing developers to fix it and reducing the chances for a possible future security threat. Moreover, these warnings can also be used at a later stage to reduce time in software security audits. This particular approach has not been implemented to prevent LDAP injection attacks primarily. However, the mechanism of 'Interactive Code Refactoring' is used for 'Input Validation', which is one of the key factors in preventing LDAP injection attacks [19].

MT5: Remote Code Execution detection technique

Remote Code Execution (RCE) attacks are considered as one of the most prominent security threats for web applications in the recent times. The attacks caused with the help of RCE are to such an extent that it is the most widespread PHP Security issue since the mid of 2004 (Open Web Application Security Project - OWASP). These kind of attacks are similar to Cross Site Scripting but in a more sophisticated way as they require multiple rounds of communication between client and server. This approach has been applied to phpMyAdmin and phpLDAPadmin applications.

Figure 15 shows that a particular type of RCE vulnerability exists in phpLDAPadmin because malicious code can be provided and executed. The malicious code executes for access control and allows the attacker to perform privilege escalation. Using RCE detection prevents privilege escalation injection attacks [21].



```
Set     $code = urlencode("arbitrary_php_script")
Send    http://phpLDAPadminURL/htdocs/a.php?a=query_engine&search=1&orderby=$code
```

```
//Accessing directly is allowed in a.php
13 require_once 'b.php';
16 $cmd = $_REQUEST['cmd'];
22 switch ($cmd) {
23   case _debug,
24     .......
27   default
32     $script_cmd = $cmd. '.php';
39 }
   ........
64   include ($script_cmd);
```

```
35 $direct_scripts = array('cmd.php', ...);
// $_SERVER['SCRIPT_NAME'] is the script client
requesting
42 $script_running = $_SERVER['SCRIPT_NAME'];
43 foreach ($direct_scripts as $script) {
46   if (preg_match('$script, @script_running))
47     $scriptOK = true;
51 }
     .......
// If script requested is not in
$app['direct_scripts'],deny
57 if (!$scriptOK)
62   die();
```

```
// This page not directly accessible
13 require_once 'b.php';
32 if ($_REQUEST['search'])
       masort( ... , $_REQUEST['oderby']);
.......
1002 function masort (&$data, $sortby) {
1014 foreach ( split(' , ' , $sortby) as  $key)
1017   $code .= '....$key....';
       // turn the string held by $code into a
dynamic function
1080 $CACHE = create_function(... ,
$code );
1083 uasort(... , $CACHE);
1084 }
```

Figure 15: RCE in phpLDAPadmin v1.2.1.1 (Simplified)

MT6: Static Analysis technique

In this approach, a formal vulnerability signature can be used to detect a possible vulnerability. Object Constraint Language (OCL) is used to capture vulnerability signatures. Depending on whether the source of the vulnerability is related to input validation, output validation, processing or hosting, they are categorized so as to perform static/dynamic analysis. Using a formal vulnerability analysis definition can be beneficial to perform 'threat analysis' and 'vulnerability analysis' during the development stage as well as 'attack analysis' after deployment of the project. This can help find possible vulnerabilities and appropriate action can be taken. OCL based vulnerability signatures can be applied for prevention of SQLI, XSS, Improper Authentication, and Improper Authorization. Though this approach has not been directly used for prevention of LDAP injection attacks, it can be modified accordingly [20].

Table 1 shows the source and scope of implementation for each mitigation technique discussed above. Other types of mitigation techniques include those used to prevent SQL injection attacks. Among several common types of query injection attacks, much of the research efforts have focused on preventing SQL injection attacks [22, 24, 10, 11, 12]. In contrast, LDAP injection attacks have not received enough attention.

Table 1: Mitigation Techniques: Source and Scope of Implementation

	Source	Mitigation Technique	Scope of Implementation
MT1	Administrator techniques [15,22,25]	Password Policy Schema; Frequent changing of Password; LDAP Configuration; IP Firewall	LDAP injection
MT2	Developer techniques [13,14]	Data validation and dynamic checks; Outgoing data validation	LDAP injection
MT3	Prevention of Injection Attacks by Rectification policies [17]	Program Transformations	SQL injection; LDAP injection; Log injection; XSS injection; Direct Static Code injection
	Program transformation techniques [18]		
MT4	Application Security IDE technique [19]	Interactive Code Refactoring; Interactive Code Annotation; (still in implementation stage)	Input validation; Broken Access control; Cross-site Request Forgery
MT5	Remote Code Execution detection technique [20]	Static Analysis	Static Analysis
MT6	Static Analysis based technique [21]	OCL-based vulnerability signature approach	SQL injection; XSS injection; Improper Authorization; Improper Authentication

Table 2 provides a mapping of mitigation techniques to attack types. The symbols ✓and ✗ indicate whether a mitigation technique is applicable or not to prevent the corresponding attack types. The α symbol indicates that a mitigation technique has been applied to prevent other types of attacks (SQL injection), but not LDAP Injection.

Table 2: Mapping mitigation techniques to attack type

	Login Bypass	Information Disclosure	Privilege Escalation	Information Alteration
MT1	✓	✓	✓	✓
MT2	✓	✓	✓	✓
MT3	✓	✓	α	α
MT4	α	α	α	α
MT5	✗	✗	✗	✗
MT6	α	✗	✗	✗

As shown in Table 2, various mitigation techniques that have been developed and implemented focusing on injection attacks in general but not particularly on LDAP injection attacks types. Moreover, these techniques do not necessarily provide a complete solution to such injection attacks. They are implemented to focus on a particular scenario, or provide a solution with prior restrictions.

2.3 Comparison of SQL and LDAP Queries

Our work is motivated by the syntactical differences and usage between SQL and LDAP queries. Techniques such as proxy-based approach of preventing SQL injection attacks [9], static analysis [12], library class *PreparedStatements* in Java [26] can be used to prevent SQL query injection attacks but cannot be directly applied to prevent LDAP query injection attacks.

Given the syntactic differences between SQL and LDAP, further discussed in Chapter 3 (Section 3.5), the injection attack inputs and the subsequent consequences are also different. For example, a traditional tautology SQL injection attack may lead to deleting an entire table; whereas a tautology LDAP injection attack may lead to leaking privileged information from a node at a specific level in the directory tree [27, 28].

Table 3 illustrates the difference between LDAP and SQL query. A benign input is given in rows 1 and 2 and rows 3 and 4 consist of an attack input `(username` and `password)`. The second and third column shows the search query for LDAP `(ldapQuery)` and SQL `(sqlQuery)`, respectively. The second row shows the LDAP and SQL queries for the supplied benign input `(username` and `password)`. In particular, the LDAP query has two expressions `(uid="pbulusu", password="123456")`, the results of these expressions are combined logically with `AND` (&). The SQL query result is affected by the `WHERE` condition, where all columns from table `users` for the rows having username and password columns as `pbulusu` and `123456`, respectively, are returned.

Table 3: LDAP vs. SQL query

	LDAP Search Query	SQL Search Query
Benign Input	Username: pbulusu Password: 123456	Username: pbulusu Password: 123456
Query with benign input	ldapQuery="(&(uid="pbulusu")(password="123456"))";	sqlQuery = 'SELECT * FROM users WHERE username="pbulusu" AND password= "123456"';
Attack input	Username: *)(uid=*))(|(uid=* Password: ""	Username: "" Password: "" OR ("1"="1)
Query with attack input	ldapQuery="(&(uid=*)(uid=*))(|(uid=*)(userPassword=""))";	sqlQuery = 'SELECT * FROM users WHERE username="" AND password="" OR ("1"="1")';

The third and fourth rows of Table 3 show attack inputs and corresponding queries for LDAP and SQL, respectively. An attacker performs an LDAP query injection attack by providing the username as `'*)(uid=*))(|(uid=*'` and the password as blank. The resultant LDAP query becomes `ldapQuery="(&(uid=*)(uid=*))(|(uid=*)(userPassword=))"`, which would return all user IDs from the directory tree. In contrast, an attacker performs a SQL query injection attack by providing the username as `""` and the password as `"" OR ("1"="1)`. The resultant SQL query becomes `sqlQuery = 'SELECT * FROM users WHERE username="" AND password="" OR ("1"="1")'`. The `WHERE` condition will be evaluated as true, which would return all selected rows from table users. Hence, an attacker can gain unauthorized access.

Different types of injection attacks such as SQL injections [6] and LDAP injections [3] have been prevalent among web applications over the last decade. In fact, query injection flaws remain as one of the top ranked web security vulnerabilities according to Open Web Application Security Project (OWASP) report [7].

In the next Chapter, we present technology overview and discuss some more details of LDAP.

CHAPTER 3

Technology Overview

3.1 Overview

This Chapter presents a technology overview of the LDAP protocol. Some more details of LDAP such as formation of LDAP query, different nodes in LDAP such as Common Name *(cn)*, Distinguished Name *(dn)* etc. are introduced in this Chapter. It also includes an introduction to SQL and the basic differences between LDAP and SQL.

3.2 Technology Overview

The Lightweight Directory Access Protocol (LDAP) is a commonly used protocol in organizations for accessing information from directories. LDAP was developed by The Internet Engineering Task Force (IETF) in order to find a simpler version of the existing Directory Access Protocol X.500. The term 'Lightweight' in LDAP comes from the fact that there is a reduction in the number of protocol overheads in comparison with the X.500 [2]. LDAP gains its popularity because of features such as representation of data objects in hierarchical form, being open source and relying on TCP/IP, which is necessary for Internet access. LDAP uses the TCP/IP for its transport and network layers instead of Open Systems Interconnection (OSI Model) stack which was used in the X.500. Some duplicate functions from X.500 standard were eliminated in LDAP. It is significantly simpler and can be customized according to the needs of a particular organization [29].

LDAP is particularly used for 'Write-once-Read-many' kind of applications. Thus, LDAP is suitable for maintaining the contact information of all the employees in any organization which would remain the same for a large period of time. However, LDAP is not suitable for applications that require frequent content update such as online transaction processing and e-commerce. LDAP is highly suitable when the data being stored requires features such as cross-platform availability of data, access to data from a large number of computers or applications, few changes of data, and storage of data in a single record [30].

3.3 Formation of LDAP Query

The formation of an LDAP query consists of defining the LDAP Server details, connecting to the LDAP server and binding to the LDAP server. Depending on the final requirement, the query will consist of several operations such as search, update and delete. Figure 16 shows example code that uses PHP to connect and bind to the LDAP server.

```
// configuration
1. $ldapserver = '192.168.1.124';    -- LDAP Server
2. $ldaptree   = "dc=ubuntuldap2";   -- Defining DC (Domain Component)

// connection to ldap server
3. $ldapconn = ldap_connect($ldapserver) or die("Could not connect to LDAP
server.");
4. ldap_set_option($ldapconn, LDAP_OPT_PROTOCOL_VERSION, 3);

// verifying connection to ldap server
5. if($ldapconn){

// binding to ldap server
6.   $ldapbind = ldap_bind($ldapconn, "cn=admin,".$ldaptree, "admin") or die
("Please enter valid login credentials!");

// verify binding
7.   if ($ldapbind) {
8.     Perform multiple tasks here...(Such as Search/Update/Delete)
11.  }
12.}
```

Figure 16: Code for formation of LDAP query

In Figure 16, line 1 defines the IP address of the LDAP server. The LDAP tree with base domain component (`dc=ubuntuldap2`) is defined in line 2. Line 3 establishes a connection to the LDAP server through *ldap_connect(...)* method call. If the connection is unsuccessful, the script throws an error stating *'Could not connect to LDAP server'*. Line 4 sets the protocol version.

Once the LDAP connection is verified in line 5, the next step is to bind to the LDAP server as shown in line 6. Once the LDAP Binding is verified in line 7, the user can perform multiple tasks based on the requirement.

3.4 Different Nodes in LDAP

Common nodes in an LDAP directory are defined as *o* (*organization*), *dc* (*domain component*), *ou* (*organizational unit*), *dn* (*distinguished name*) and *cn* (*common name*) according to the X.500 Directory specification.

Data in LDAP is represented in a hierarchical tree structure called Data Information Tree (DIT). The tree structure of LDAP directory is similar to the top-down representation of UNIX file directories or Domain Name Server (DNS) trees. The top level in the LDAP tree structure is referred to as the base *dn* and breaks down into individual objects, each of which is called an entry. A simple representation of DIT is shown in Figure 17.

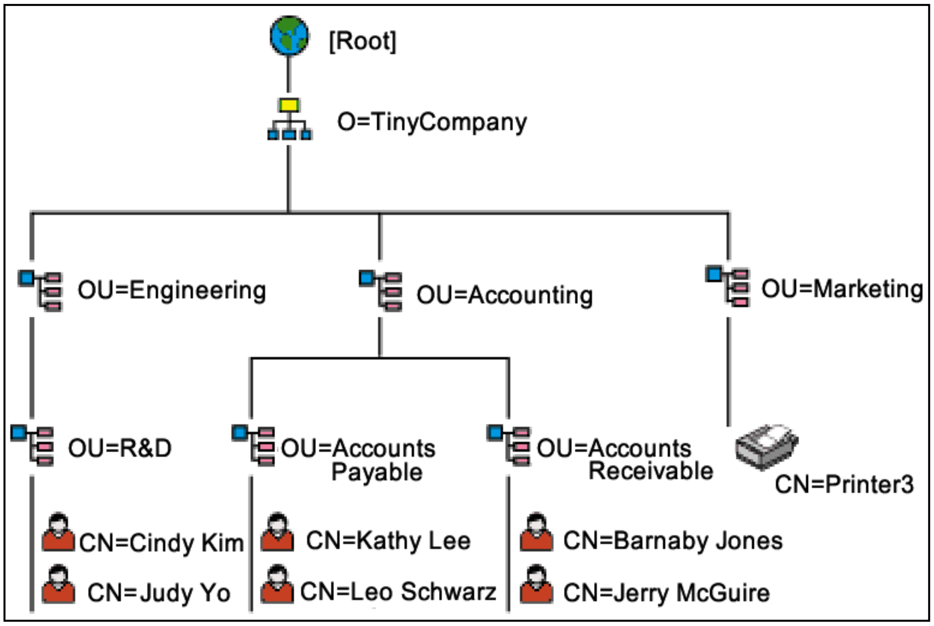


Figure 17: Diagrammatic representation of LDAP directory tree structure

Figure 17 shows the directory structure for a small company. The top-level node in TinyCompany has an Organization (*O*) attribute. TinyCompany comprises of the Engineering, Accounting, and Marketing departments which are represented with Organizational Unit (*OU*) entry *(e.g.,* OU=Engineering*)*. Accounting *OU* has two organizational units which are Accounts Payable and Accounts Receivable. For example, the common name (*CN*) Kathy Lee is under the *OUs* Accounts Payable and Accounting. It can thus be seen that, an entry can be composed of one or more *OU*s. The Marketing *OU* has one printer resource (CN=Printer3) [31].

3.5 SQL and LDAP

SQL is a database query language whereas LDAP is a protocol used for accessing directory service. There are significant differences between SQL and LDAP. First, LDAP is a protocol for accessing directory data over the network, where directory information is faster and easier to read. However, update and delete operations are expensive. On the other hand, SQL is a query language that supports transactional operations on relational databases requiring frequent read, write, update, and delete operations.

The data representation and organization between LDAP and SQL also remain largely dissimilar. For example, LDAP stores directory data in a tree structure having a set of nodes and edges; whereas SQL stores data in tables of a relational database. The LDAP tree may reside among multiple machines in a network, whereas traditional SQL database stores tabular data in one local machine.

At the syntax level, though there are some similarities between SQL and LDAP *(e.g.,* **=**, **>**, **<***)*. Operation wise, both support insertion, deletion, and viewing of data. However, there are dissimilarities in syntax. For example, SQL "AND" means the logical AND operation, whereas LDAP represents it as "&". Further, SQL supports a rich set of aggregate *(e.g., count, sum)* and join *(e.g., inner join, outer join)* operations for multiple queries. In contrast, LDAP has no such support.

In the next Chapter, we introduce fault-injection based testing, some related work on fault-injection based testing, and our proposed approach.

CHAPTER 4

OCL Fault-Injection Based Testing Approach

4.1 Overview

This Chapter introduces the approach fault-injection based testing and covers some relevant work on fault-injection based testing in Section 4.2. Next, we introduce Object Constraint Language (OCL) and our proposed approach of OCL fault-injection based testing in Sections 4.3 and 4.4, respectively.

4.2 Related Work on Fault-Injection Based Testing

Fault-based testing technique is intended to generate or assess test cases by anticipating errors in a system under test and deliberately inject faults in the system. This approach can demonstrate only the presence of specific faults (injected), but not the absence of faults during testing. The approach can identify effective test cases that can reveal specific faults we injected in the system [32].

Some relevant works on fault-injection based testing [33-38] are shown in Table 4. These works use this technique to test robustness of web application security scanner [33], effectiveness of intrusion detection systems intended to detect network protocol level attacks [34], robustness of router protocol implementation to tolerate malformed protocol data units against failure or crashes [35], effectiveness of input validation routines in web applications implemented in PHP [36], capability of handling malformed input PDF files by Java applications [37], robustness of embedded applications against skipping login mechanism bypassing [38], utility applications written in C/C++ [39], and evaluation of performance and security of web services [40].

Table 4: Comparison of related work

Tool / Work (Test Level)	Vulnerability covered	Source of test cases	Test case generation method	Target applications
Fonseca *et al.* [33]	SQLI and XSS	N/A	N/A	Web scanners
Vigna *et al.* [34]	Buffer Overflow (BOF), Format String Bug (FSB)	Attack templates	Inject fault in application and network layer	Intrusion detection systems
Tal *et al.* [35]	BOF	Protocol syntax	Inject faults	Network router algorithm implementation (daemons)
Kiezun *et al.* [36]	SQLI and XSS	Source code	Solve path constraints in applications and replace non malicious test cases with attack test case	Web applications in PHP
Ghosh *et al.* [37]	Testing of program crash, abnormal behavior through exceptions such as DocumentExcepti on, IOException etc.	Java byte code	Instrumenting class file using BCEL tool, tester needs to select a program line to replace with injected faulty line of code at Java opcode level	Java, tested on PDF generator application written in Java
Fouque *et al.* [38]	Bypassing password checking through buffer overflow exploit	N/A	Faults injected at the assembly code level by changing the return address of a function conducting access control check, with the next instruction as skipping access control	Applications running on embedded hardware, used for access control application such as password checking
Voas [39]	Buffer overflow	Source code	Replacing, adding or deleting source code, replacing implemented function call with perturbed function call intended to lead fault by altering parameter or return values	Utility applications such as FTP server (wu-ftpd)
Oliveira *et al.*[40]	Assess performance anomaly of web service frameworks	Web service	Performance of web services stacks are evaluated using a benchmark called WSTest with different SOAP object sizes and security of web services is evaluated using security testing tool called WSFAggressor	Web Service frameworks
Salas *et al.* [41]	SQLI	Incomplete or under-specified model	Obtain constraints from design documents, express constraints in Object Constraint Language (OCL), finally solving it with a constraint solver	Web applications design
Grela *et al.* [42]	Anomaly of Business Processes	Fault-injection in BPEL processes	Software Fault Injector for BPEL processes (SFIBP)	Business Process Execution Language (BPEL) processes
Aichernig *et al.*[32]	N/A	Model-based specifications	Mutation of OCL specifications	Triangle type determination program
Our approach	LDAP Injection	Source code and OCL	Derive pre-conditions and post-conditions from flow graph, and mutate pre-conditions and post-conditions	Web applications

A few approaches rely on application design level information to generate test cases. Salas *et al.* [41] generate OCL constraints from UML class diagrams and solve them to generate test cases; while Grela *et al.* [42] propose fault-injection in business processes expressed in Business Process Execution Language.

Our work is closely related to the test case generation method presented by Aichernig *et al.* [32]. In their work, the authors have presented a method of test case generation for pre-condition and post-condition specifications. They have generated Object Constraint Language (OCL) specifications for a triangle type determination program, whereas we have generated OCL specifications for PHP web applications relevant to LDAP query injection attacks. Our approach also consists of generating control flow charts to identify needed pre and post-conditions.

In contrast to earlier work, our proposed technique obtains control flow from source code, derives expected constraints and then applies faults in pre-conditions or post-conditions to generate test cases to reveal LDAP injection vulnerabilities. We express pre-conditions and post-conditions using OCL notation.

4.3 Object Constraint Language (OCL)

OCL is a language that complements Unified Modeling Language (UML) notations. OCL is used to describe and enhance rules that are applicable to UML. Detailed aspects of a system design can be precisely described using OCL which is not possible with UML alone. Thus, OCL is widely used in model-driven engineering (MDE) techniques as a default language to express model transformations, rules or code-generation templates. Use of OCL makes UML class diagrams more precise as OCL is used to specify invariants of class attributes and pre-conditions and post-conditions of class methods [43].

OCL helps in achieving automation of software development in UML. A combination of UML and OCL helps developers generating effective and coherent models. OCL plays a key role in Model Driven Architecture by enabling platform-specific models to communicate with platform-independent models [44].

A UML class diagram 'Person' and its respective attributes 'name', 'address' and 'birthdate' is shown in Figure 18. The generated OCL constraints are shown in Figure 19 [45].

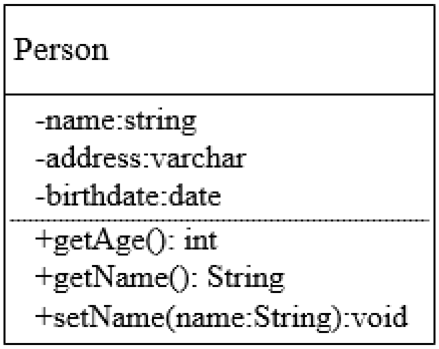


Figure 18: Example of UML class diagram

```
1. context Person
2. inv fields_nonnull: self.birthdate -> notEmpty() and
   self.name -> notEmpty()and self.address -> notEmpty()

3. context Person :: getAge():int
4. post positive_age : result >= 0

5. context Person :: setName(name:String): void
6. pre name_given: name -> notEmpty()
7. post name_set: self.name = name
```

Figure 19: OCL constraints for Class Person

Figure 19, line 2 defines an invariant for class 'Person' which indicate all three attributes to be 'non-empty'. Next, `getAge()` method (line 3) needs to satisfy the post-condition that age must be a positive integer number (line 4). The method `setName` (line 5) has a pre-condition in line 6 which checks if the name attribute is not empty. In line 7, once the pre-condition is satisfied, the given name is set to the name of the particular user.

4.4 Proposed OCL Fault-Injection Based Testing Approach

Ideally, we like to have a set of pre-conditions and post-conditions (expressed in OCL) to apply fault injection on these conditions as part of the test case generation process. However, we rarely have the design level information for a given implementation *(e.g., class diagrams and dependency)*. Given that, we follow a process (Figure 20) to capture set of conditions from program source code.

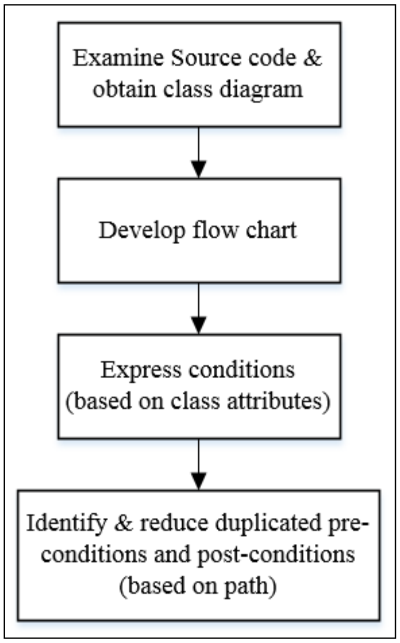

Examine Source code & obtain class diagram
Develop flow chart
Express conditions (based on class attributes)
Identify & reduce duplicated pre-conditions and post-conditions (based on path)

Figure 20: Generation of pre and post-conditions

<u>Process for Generation of pre and post-conditions</u>

Step 1. *Examine the source code, identify functionalities of applications related to LDAP query generation and invocation, and obtain class diagrams capturing the key class attributes that may contribute to be part of pre and post-conditions.*

Step 2. *Develop a flow chart for application functionality.*

Step 3. *Record needed pre and post-conditions required for successful completion of functionalities. The conditions are expressed in terms of class attributes captured in step 1.*

Step 4. *For each path in the flow chart, combine all pre and post-conditions by removing duplicate pre-conditions.*

Once the list of pre and post-conditions is identified, we apply the fault adequate test case generation algorithm discussed next.

<u>Algorithm: Fault adequate test case generator</u>

Let the input and output for the fault adequate test case generator be given as follows:

Input: D $\in$ {pre-conditions, post-conditions}

The intended design D is composed of valid pre-conditions and post-conditions, which when satisfied will prevent the occurrence of LDAP injection attacks.

Output: T = {test cases revealing LDAP injection attack types}

Here, T = {t_1, t_2, ...} where **t_k = <i_k, o_k>;** and i_k is the k^{th} input and o_k is the k^{th} output

Step 1: *Given that the intended design is D, generate D' from D*

where D' is the faulty design and

D' $\in$ {pre', post'},

where pre' and post' define faulty pre-conditions and post-conditions respectively.

pre' is generated by randomly replacing one of the logical operator and relational operator with other from the sets {OR, AND, NOT} and {$\leq$, $\geq$}

Step 2: *Define an input i such that i satisfies pre $\vee$ pre'.*

($\vee$ indicates logical OR)

Step 3: *Apply input i to D and D'.*

Observe outputs o and o' for D(i) and D'(i) respectively.

If D(i) $\neq$ D'(i), accept the input i and include it in the set T = T $\cup$ {<i, o>} (good test case)

If D(i) = D'(i), reject input i and move to step 2.

In the next Chapter, we discuss the evaluation of our proposed approach to a number of case studies.

CHAPTER 5

Case Studies and Evaluation

5.1 Overview

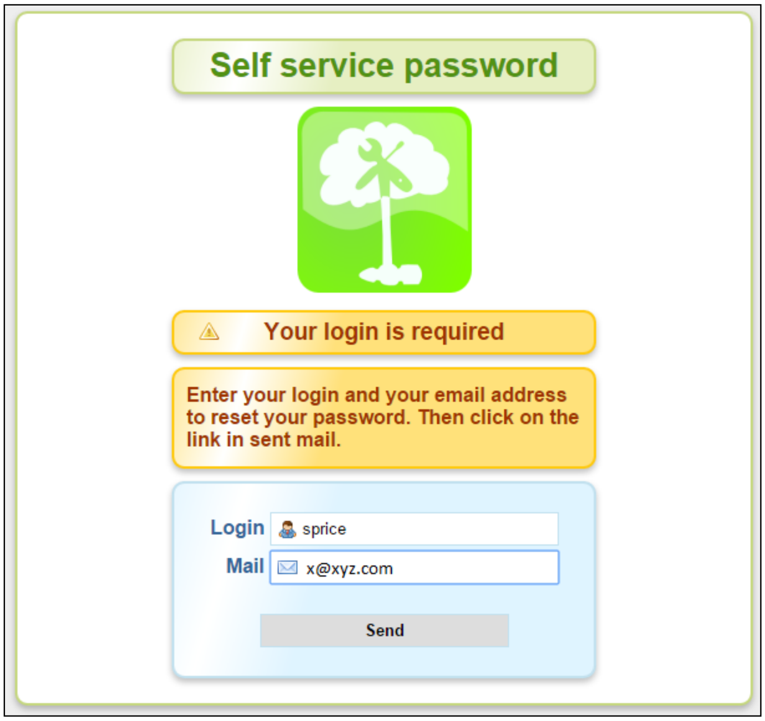
In this Chapter, we demonstrate example applications of test case generation based on OCL Fault Injection algorithm discussed in Chapter 4. We evaluate test case generation approach with an open source PHP LDAP web application Self Service Password [46] which has been reported to contain LDAP injection vulnerability in Open Source Vulnerability Database (OSVDB) [47]. Moreover, we demonstrate our approach for a developed web application having LDAP injection vulnerabilities. We deploy both applications in an Apache web server having phpLDAPadmin configured appropriately [48].

5.2 Case Study 1: Self Service Password (Login Bypass, Information Disclosure)

Self Service Password can be used to reset the password of an LDAP entity (*common entity* or *cn*). We have used the source code of Self Service Password related to password change functionalities to generate the pre and post-conditions. We find that there are three ways to change password: reset by security questions, reset by old and new password and reset by sending token in email. We evaluate our approach for reset by sending token in email option.

We first show the legitimate way of password reset in Figure 21. We assume that user *Sam Price* has a legitimate login ID as *sprice* with a valid email ID given as *x@xyz.com*. The reset password link will be sent to the email ID and the user can change his password by clicking on the link provided in the email.

Figure 22 shows that an attacker can access the password reset link by providing input containing wild card character (*sp**) in the login ID field and email as *x@xyz.com*. Figure 23 shows that providing the above inputs result in receiving a password reset link (with token information). This attack is an example of information disclosure and may further lead to login bypass type of LDAP injection attack.



Self service password

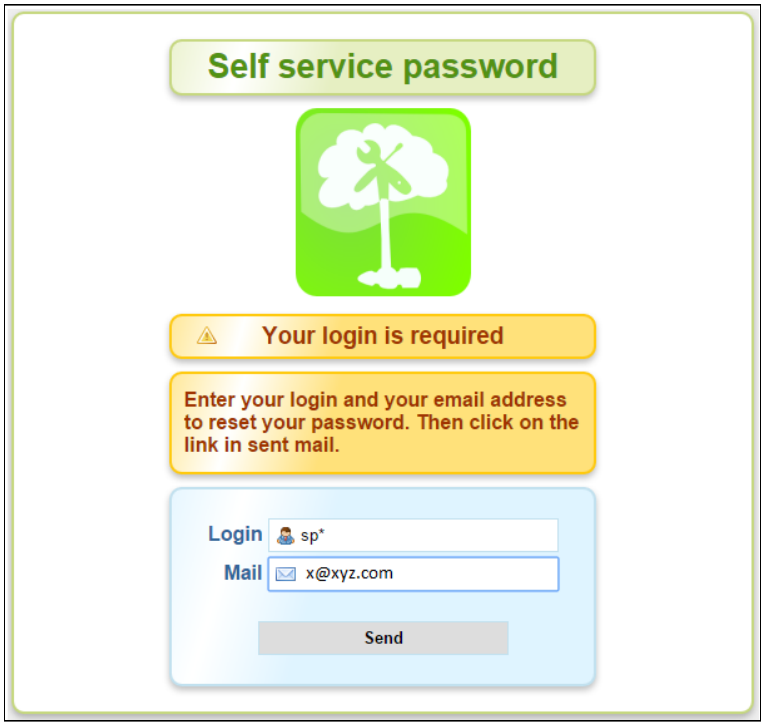
Your login is required

Enter your login and your email address to reset your password. Then click on the link in sent mail.

Login: sprice

Mail: x@xyz.com

Figure 21: Correct technique of changing password in Self Service Password



Self service password

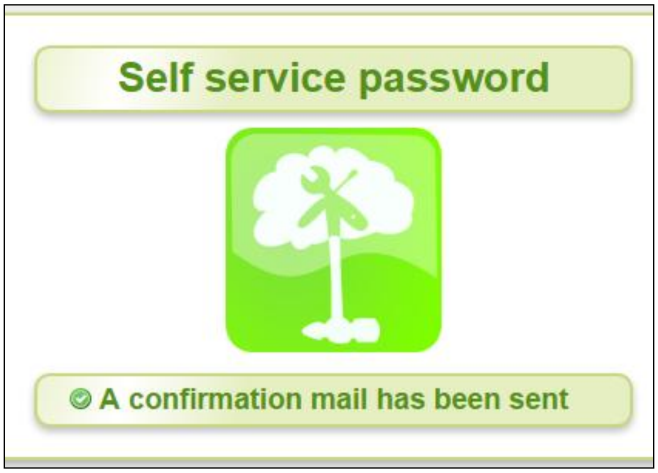
Your login is required

Enter your login and your email address to reset your password. Then click on the link in sent mail.

Login: sp*

Mail: x@xyz.com

Figure 22: Attacker using '' wildcard to access the system*



Self service password

A confirmation mail has been sent

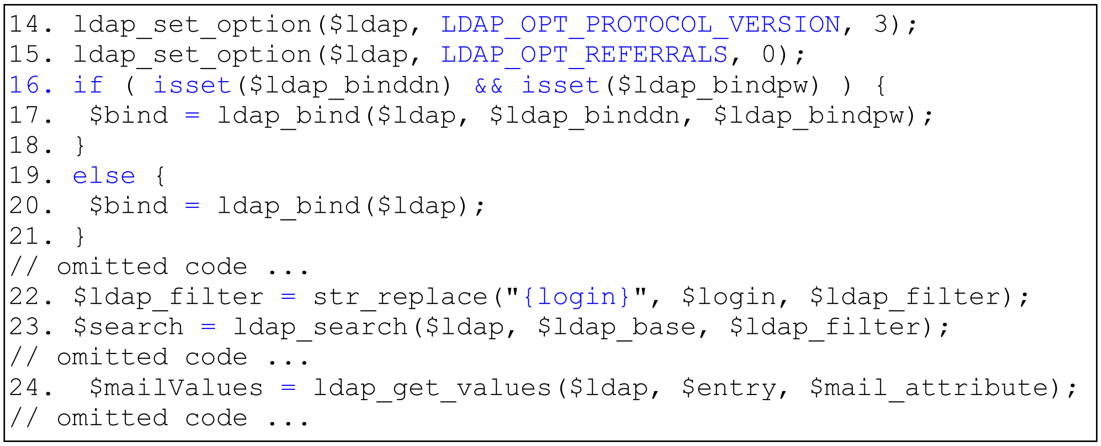
Figure 23: Confirmation email sent to the attacker

We now discuss the process for generation of pre and post-conditions (presented in Chapter 4). The PHP code for vulnerable version of Self Service Password application is shown in Figure 24.

Process for Generation of pre and post-conditions

Step 1: The PHP code of the vulnerable version of Self Service Password is shown in Figure 24. Here, Lines 2 and 8 retrieve Email ID and Login ID. Lines 22-24 use the login and email to form LDAP query. Line 22 replaces the occurrence of the text '{login}' in variable `'$ldap_filter'` with the value that is assigned to the variable `'$login'` which is the user entered value in login field. Line 22 is not filtering out the meta characters which becomes the part of LDAP search and leads to LDAP injection attacks. Line 13 connects with LDAP server, and Line 17 executes the generated LDAP query.

```
...
1. if (isset($_POST["mail"]) and $_POST["mail"]){
2.   $mail = $_POST["mail"];
3. }
4. else {
5.   $result = "mailrequired";
6. }
7. if (isset($_REQUEST["login"]) and $_REQUEST["login"]) {
8.   $login = $_REQUEST["login"];
9. }
10.else {
11.  $result = "loginrequired";
12.}
// omitted code ...
13. $ldap = ldap_connect($ldap_url);
```



```
ldap_set_option($ldap, LDAP_OPT_PROTOCOL_VERSION, 3);
ldap_set_option($ldap, LDAP_OPT_REFERRALS, 0);
if ( isset($ldap_binddn) && isset($ldap_bindpw) ) {
 $bind = ldap_bind($ldap, $ldap_binddn, $ldap_bindpw);
}
else {
 $bind = ldap_bind($ldap);
}
// omitted code ...
$ldap_filter = str_replace("{login}", $login, $ldap_filter);
$search = ldap_search($ldap, $ldap_base, $ldap_filter);
// omitted code ...
 $mailValues = ldap_get_values($ldap, $entry, $mail_attribute);
// omitted code ...
```

Figure 24: PHP code for vulnerable version of Self Service Password application

Based on source code, we derive the class diagram as shown in Figure 25. Here, we show three classes each representing password change for three different ways. In particular, password reset by email has four attributes (*loginid*, *emailid*, *logincount*, *emailcount*) and three methods (*isValid()*, *isRegistered()* and *isTokenSent()*).

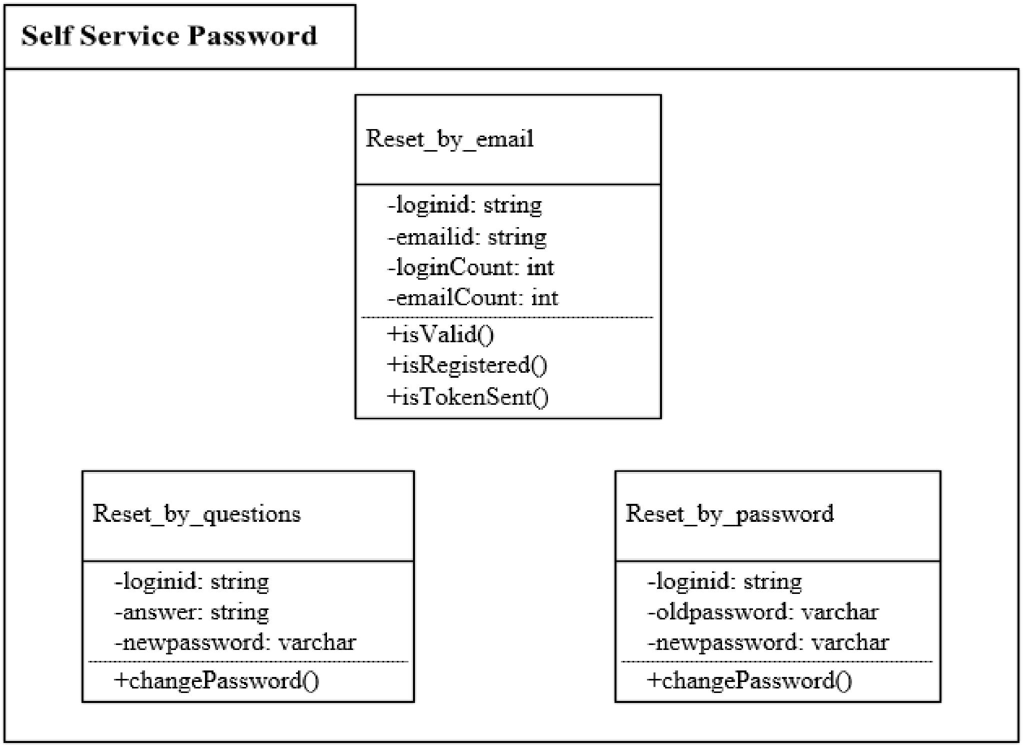


Figure 25: Class diagram for Self Service Password application

Step 2: We develop a flow chart as shown in Figure 26 for email token-based password reset functionality. Here, a rectangle means steps (input or output), an ellipse means the start or end state and a diamond is a decision making step where testing of conditions are performed.

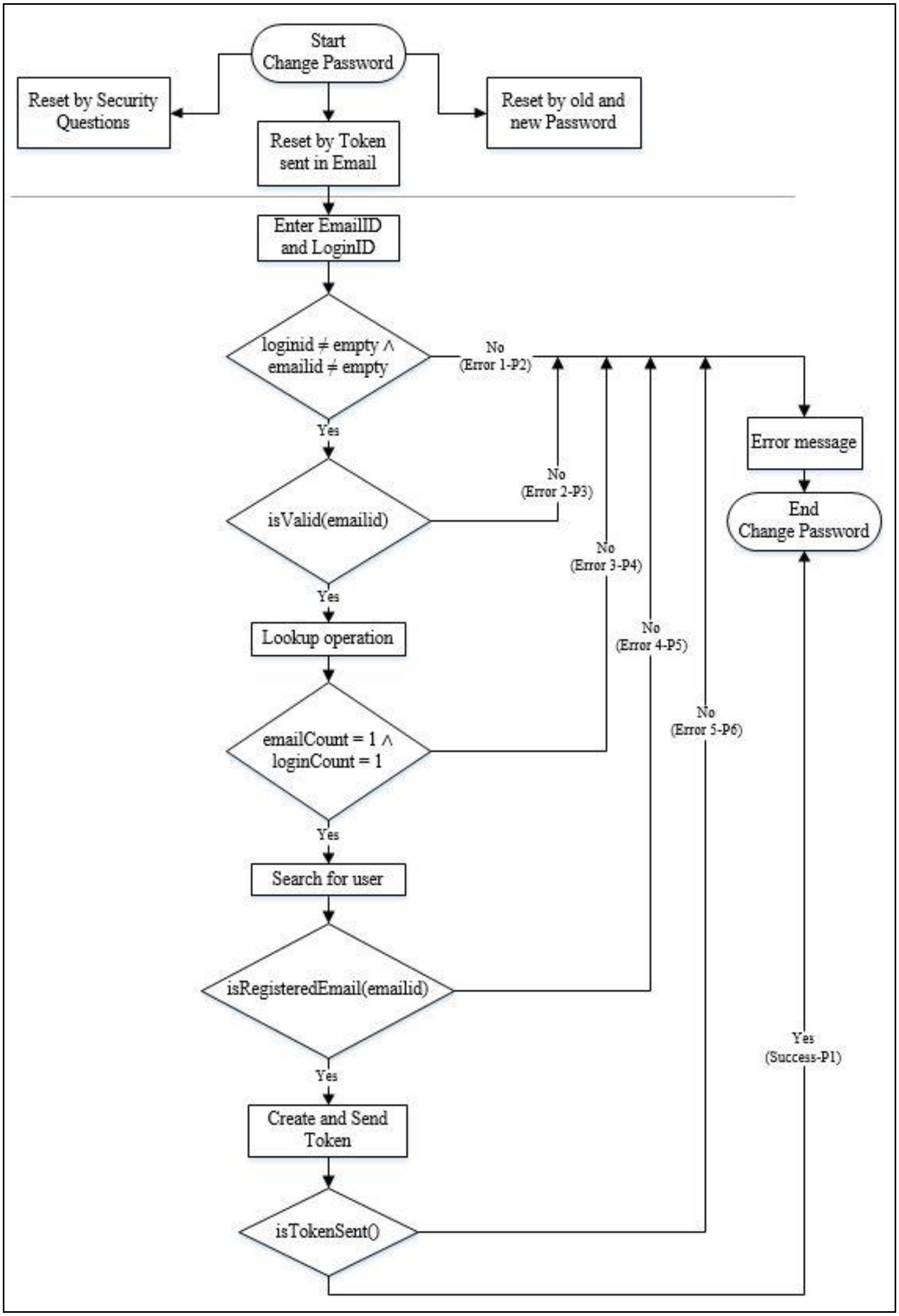


Figure 26: Flowchart for password change based on reset token sent via email

Step 3: Figure 26 shows various paths related to both successful and unsuccessful password change by token sent to email. For example, the path showing successful password change requires that Login ID and Email ID be not empty (*loginid ≠ empty* AND *emailid ≠ empty*), Email ID is valid syntactically (*isValid(emailid)*), Email and Login ID counts are one, and Email ID belongs to known registered email address (*isRegisteredEmail(emailid)*). The post-condition is receiving a token by Email (captured as *isTokenSent=TRUE*). Similarly, we can capture pre and post-conditions for other paths that would result in error message (total five paths). We can obtain a set of pre and post-conditions for all six paths (P1-P6).

Step 4: Table 5 shows the combined pre and post-conditions for each of the six paths based on Figure 26. There is no duplicate condition, so no reduction of conditions needs to be performed.

Table 5: Pre and post-conditions for Self Service Password

Path	pre-conditions	post-conditions
P1 (Success)	$(\text{loginid} \neq \text{empty} \wedge \text{emailid} \neq \text{empty}) \wedge$ $\text{isValid (email)} \wedge$ $(\text{emailCount} = 1 \wedge \text{loginCount} = 1) \wedge$ isRegisteredEmail (emailid)	isTokenSent()
P2 (Error1)	$!(\text{loginid} \neq \text{empty} \wedge \text{emailid} \neq \text{empty})$	!isTokenSent()
P3 (Error2)	$(\text{loginid} \neq \text{empty} \wedge \text{emailid} \neq \text{empty}) \wedge$!isValid(email)	!isTokenSent()
P4 (Error3)	$(\text{loginid} \neq \text{empty} \wedge \text{emailid} \neq \text{empty}) \wedge$ $\text{isValid(email)} \wedge$ $!(\text{emailCount} = 1 \wedge \text{loginCount} = 1)$	!isTokenSent()
P5 (Error4)	$(\text{loginid} \neq \text{empty} \wedge \text{emailid} \neq \text{empty}) \wedge$ $\text{isValid (email)} \wedge$ $(\text{emailCount} = 1 \wedge \text{loginCount} = 1) \wedge$!isRegisteredEmail (emailid)	!isTokenSent()
P6 (Error5)	$(\text{loginid} \neq \text{empty} \wedge \text{emailid} \neq \text{empty}) \wedge$ $\text{isValid (email)} \wedge$ $(\text{emailCount} = 1 \wedge \text{loginCount} = 1) \wedge$ $\text{!isRegisteredEmail (emailid)} \wedge$!isTokenSent()	N/A

Now, we apply fault injection for each of the obtained pre-conditions (which becomes part of *D*) to generate test cases. Below we illustrate the test case generation for path P1. Similarly, we can apply for paths P2 - P6. Next, we apply the three steps of Fault adequate test case generator Algorithm (discussed in Chapter 4).

Algorithm: Fault adequate test case generator

Step 1: From *D* we obtain *D′*. We generate altered pre-conditions (*pre′*). Table 6 shows five examples of altered pre-conditions (*pre′*) for P1 where we replaced $\wedge$ with $\vee$ randomly. This is not an exhaustive list of all possible pre′ but we show some examples for illustrative purposes. Each expression relates to two input variables (or test inputs) represent two fields: *loginid* and *emailid*. We assume that valid *emailid* is *x@xyz.com* and valid *loginid* is *sprice.* We also assume that * is an invalid character and is not permitted as any part of the inputs for this application. These set of valid inputs along with meta-characters will be combined to generate test cases that we discuss next.

Table 6: Altered pre-conditions for Self Service Password (P1)

Example	pre	pre′
1	(loginid $\neq$ empty $\wedge$ emailid $\neq$ empty) $\wedge$ isValid (email) $\wedge$ (emailCount = 1 $\wedge$ loginCount = 1) $\wedge$ isRegisteredEmail (emailid)	(loginid $\neq$ empty $\wedge$ emailid $\neq$ empty) $\wedge$ isValid (email) $\wedge$ (emailCount = 1 $\vee$ loginCount = 1) $\wedge$ isRegisteredEmail (emailid)
2	(loginid $\neq$ empty $\wedge$ emailid $\neq$ empty) $\wedge$ isValid (email) $\wedge$ (emailCount = 1 $\wedge$ loginCount = 1) $\wedge$ isRegisteredEmail (emailid)	(loginid $\neq$ empty $\wedge$ emailid $\neq$ empty) $\wedge$ isValid (email) $\wedge$ (emailCount = 1 $\vee$ loginCount = 1) $\vee$ isRegisteredEmail (emailid)
3	(loginid $\neq$ empty $\wedge$ emailid $\neq$ empty) $\wedge$ isValid (email) $\wedge$ (emailCount = 1 $\wedge$ loginCount = 1) $\wedge$ isRegisteredEmail (emailid)	(loginid $\neq$ empty $\wedge$ emailid $\neq$ empty) $\vee$ isValid (email) $\vee$ (emailCount = 1 $\vee$ loginCount = 1) $\vee$ isRegisteredEmail (emailid)
4	(loginid $\neq$ empty $\wedge$ emailid $\neq$ empty) $\wedge$ isValid (email) $\wedge$ (emailCount = 1 $\wedge$ loginCount = 1) $\wedge$ isRegisteredEmail (emailid)	(loginid $\neq$ empty $\wedge$ emailid $\neq$ empty) $\vee$ isValid (email) $\vee$ (emailCount = 1 $\vee$ (loginCount = 1) $\vee$ isRegisteredEmail (emailid)
5	(loginid $\neq$ empty $\wedge$ emailid $\neq$ empty) $\wedge$ isValid (email) $\wedge$ (emailCount = 1 $\wedge$ loginCount = 1) $\wedge$ isRegisteredEmail (emailid)	(loginid $\neq$ empty $\wedge$ emailid $\neq$ empty) $\wedge$ isValid (email) $\wedge$ (emailCount = 1 $\vee$(loginCount = 1) $\wedge$ isRegisteredEmail (emailid)

Step 2: The *loginid* in input i_1 is *sp** and *emailid* is a legitimate emailid given as *x@xyz.com*. For checking if generated test input i_1, from step 1 can satisfy $pre \vee pre'$, we do further analysis as shown in Table 7. Here, the second column shows a set of test inputs (*i*). Columns 3 and 4 in Table 7 show whether the test input *i* satisfies *pre* and *pre′*. Column 5 shows that i_1 satisfies $pre \vee pre'$ based on step 2 in the algorithm.

Step 3: Then, we apply each of the inputs i_1-i_5 in columns 6 and 7 to obtain output from original program (D) and altered program (D $'$) under o and o' , respectively. The last column indicates if the particular test case can be included in set T or not. A test input is added when o and $o'$are dissimilar.

Table 7: Generation of test cases with altered pre and post-conditions for Self Service Password (P1)

#	**Input**	**pre satisfied?**	**pre' satisfied?**	**pre ∨ pre'**	**Output D(i)**	**Output D'(i)**	**Include in set T?**
1	**i_1:** loginid = "sp*", emailid = "x@xyz.com"	**No**	**Yes**	**Yes**	**o_1:** isTokenSent = FALSE	**o'_1:** isTokenSent = TRUE	**Yes**
2	**i_2:** loginid = "sp*", emailid = "p@xyz.com"	**No**	**Yes**	**Yes**	**o_2:** isTokenSent = FALSE	**o'_2:** isTokenSent = TRUE	**Yes**
3	**i_3:** loginid = "sprice", emailid = "*"	**No**	**Yes**	**Yes**	**o_3:** isTokenSent = FALSE	**o'_3:** isTokenSent = TRUE	**Yes**
4	**i_4:** loginid = "sp*", emailid = "*"	**No**	**Yes**	**Yes**	**o_4:** isTokenSent = FALSE	**o'_4:** isTokenSent = TRUE	**Yes**
5	**i_5:** loginid = "sprice", emailid = "x@xyz.com"	**Yes**	**Yes**	**Yes**	**o_5:** isTokenSent = TRUE	**o'_5:** isTokenSent = TRUE	**No**

From Table 7, the first four test cases are added in test set T, and it is being enhanced $T = T \cup \{(i_1, o_1), (i_2, o_2), (i_3, o_3), (i_4, o_4)\}$ with vulnerability revealing test cases $\{< i_1, o_1>, < i_2, o_2>, < i_3, o_3>, < i_4, o_4>\}$.

We now discuss more on each of the test inputs. For example, we assume that the *loginid* in input i_1 is given as *sp** and the *emailid* is registered in the system. The *loginid* in input i_2 is given as *sp** and the *emailid* is not registered in the system. Input i_3 is given as a legitimate *loginid sprice* and the *emailid* is *. Input i_4 has *loginid* as *sp** and the *emailid* is *.

In Table 7, rows 1, 2, 3 and 4 show input i. Columns 3 and 4 show whether i satisfies *pre* and *pre'* (not satisfied for *pre*, satisfied for *pre'*). Column 5 shows that i satisfies $pre \vee pre'$ based on step 2 in the algorithm. Then, we apply input *i* in columns 6 and 7 to obtain output from *D* and *D'*. The last column indicates if the particular test case can be included in set *T* or not, based on conditions in step 4 (included for i_1 - i_4).

For the last input i_5, we assume that the *loginid* is *sprice* and *emailid* is *x@xyz.com*, which are both valid. Though the input satisfies both *pre* and *pre'*, the output after applying to *D* and *D'* become the same. Thus, it is not added in test set *T*.

Therefore, the final test set *T* after applying the algorithm results in selected test inputs as follows: *{<sp*, x@xyz.com>, <sp*, p@xyz.com>, <sprice, *>, <sp*, *>}*. As we apply these generated test inputs on the target application, we find that i_1, i_2, and i_4 have the capability of revealing login bypass injection attacks, and i_3 can be applied for discovering privilege escalation attack.

Note that all test cases may not be suitable for discovering vulnerabilities. Further, multiple test cases may reveal the same vulnerability (*e.g., both* i_1 *and* i_2 *can reveal login bypass attack*). Our approach enables developers to consider critical program input variables and program paths that can contribute to LDAP injection vulnerabilities. Thus, our approach can generate and select effective test cases revealing LDAP injection attacks based on altered program path constraints expressed in OCL.

5.3 Case study 2: Custom Web Application (Login Bypass)

In this Section, we demonstrate the test case generation for Login Bypass injection attack. First we show the process for generation of pre and post-conditions.

Process for Generation of pre and post-conditions

Step 1: We develop a class diagram as shown in Figure 27 capturing the key class attributes in the source code.

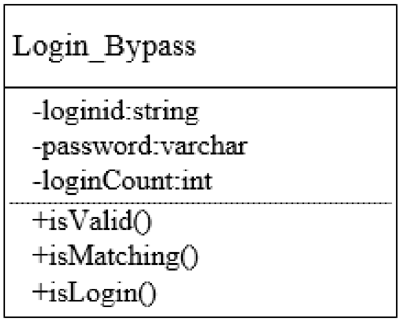


Figure 27: Class diagram for Custom application (Login Bypass)

Step 2: We develop a flow chart as shown in Figure 28 for login bypass LDAP injection attack type. Here, a rectangle means steps (input or output), an ellipse means the start or end state and a diamond is a decision making step where testing of conditions are performed.

Step 3: Figure 28 shows various paths related to both successful and unsuccessful login bypass operation. For example, the path showing successful password change requires that Login ID and Password be not empty (*loginid ≠ empty* AND *password ≠ empty*), Login ID is valid syntactically (*isValid(loginid)*), Login ID count is one, and Password matches to the Login ID of the particular user (*isMatching(password)*). The post-condition is user login (captured as *isLogin=TRUE*). Similarly, we can capture pre and post-conditions for other paths that would result in error message (total five paths). We can obtain a set of pre and post-conditions for all six paths (P1-P6).

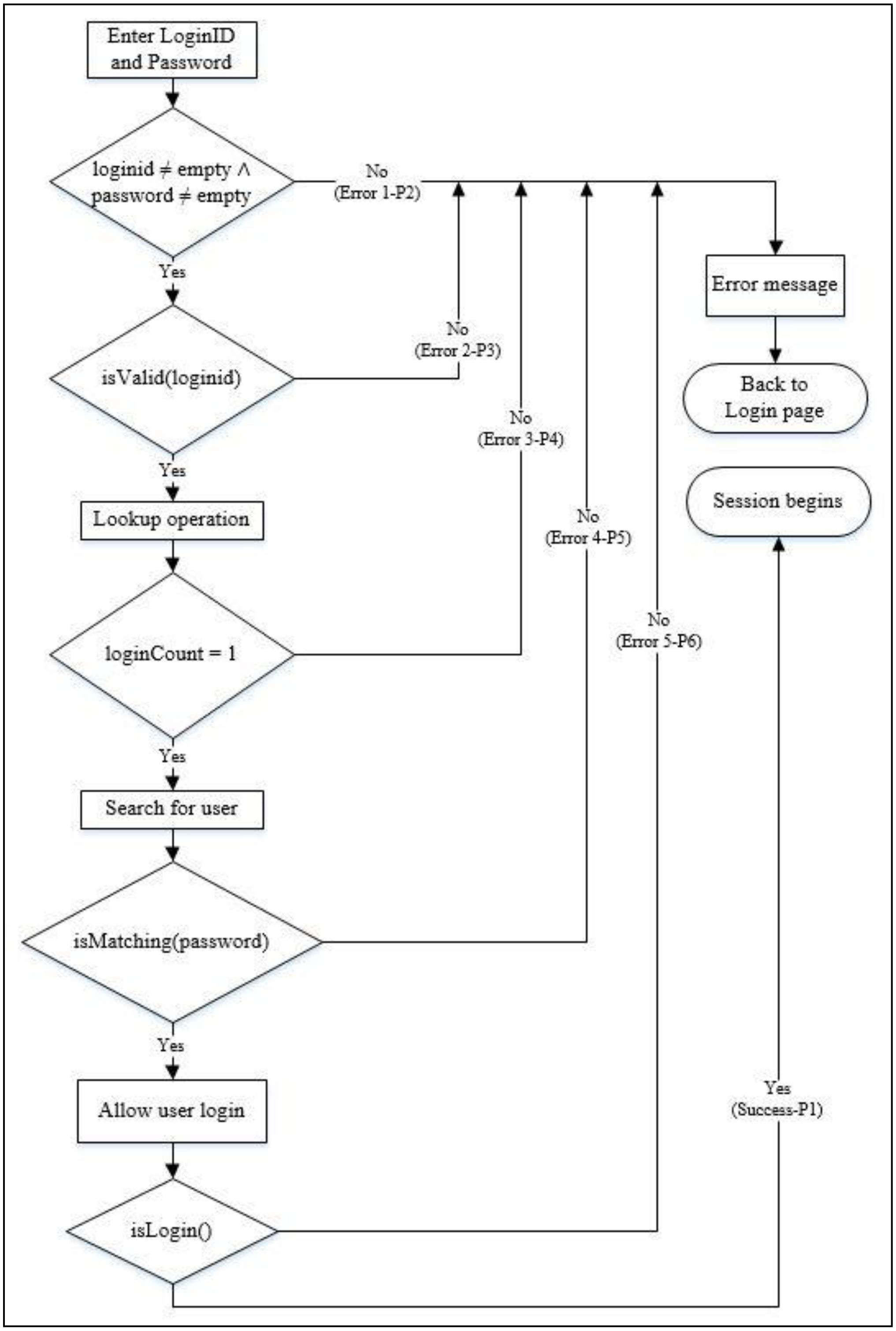


Figure 28: Flowchart for login bypass type of LDAP injection attack

Step 4: Table 8 shows the combined pre and post-conditions for each of the six paths based on Figure 28. There is no duplicate condition, so no reduction of conditions needs to be performed.

Table 8: Pre and post-conditions for Login Bypass

Path	pre-conditions	post-conditions
P1 (Success)	(loginid ≠ empty ∧ password ≠ empty) ∧ isValid (loginid) ∧ loginCount = 1 ∧ isMatching (password)	isLogin()
P2 (Error1)	!(loginid ≠ empty ∧ password ≠ empty)	!isLogin()
P3 (Error2)	(loginid ≠ empty ∧ password ≠ empty) ∧ !isValid(loginid)	!isLogin()
P4 (Error3)	(loginid ≠ empty ∧ password ≠ empty) ∧ isValid(loginid) ∧ !(loginCount = 1)	!isLogin()
P5 (Error4)	(loginid ≠ empty ∧ password ≠ empty) ∧ isValid (loginid) ∧ (loginCount = 1) ∧ !isMatching (password)	!isLogin()
P6 (Error5)	(loginid ≠ empty ∧ password ≠ empty) ∧ isValid (loginid) ∧ (loginCount = 1) ∧ !isMatching (password) ∧ !isLogin()	N/A

Now, we apply the three steps of Fault adequate test case generator Algorithm to illustrate the test case generation for path P1 (Table 8).

<u>Algorithm: Fault adequate test case generator</u>

Step 1: From *D* we obtain *D'*. We generate altered pre- conditions (*pre'*). Table 9 shows two examples of altered pre-conditions (*pre'*) for P1 where we replaced ∧ with ∨ randomly. Table 9 shows two examples (non-exhaustive) of *pre'* that we generate by randomly substituting AND with OR in *pre* (changes are shown in bold in the third column). Each expression relates to two input variables (or test inputs) represent two fields: *loginid* and *password*. We assume that valid *emailid* is *x@xyz.com* and valid *loginid* is *sprice.* We also assume that * is an invalid character and is not permitted as any part of the inputs for this application. These set of valid inputs along with meta-characters will be combined to generate test cases that we discuss next.

Table 9: Altered pre-conditions and test inputs for Login Bypass (P1)

Example	pre	pre′
1	(loginid ≠ empty ∧ password ≠ empty) ∧ isValid (loginid) ∧ loginCount = 1 ∧ isMatching (password)	(loginid ≠ empty ∧ password ≠ empty) ∨ isValid (loginid) ∨ loginCount = 1 ∨ isMatching (password)
2	(loginid ≠ empty ∧ password ≠ empty) ∧ isValid (loginid) ∧ loginCount = 1 ∧ isMatching (password)	(loginid ≠ empty ∧ password ≠ empty) ∧ isValid (loginid) ∧ loginCount = 1 ∨ isMatching (password)

Step 2: The *loginid* in input i_1 is given as *)(uid=*))(|(uid=*) and *password* is given as *abcdef*. For checking if generated test input i_1, from step 1 can satisfy $pre \lor pre'$, we do further analysis as shown in Table 10. Here, the second column shows a set of test inputs (i). Columns 3 and 4 in Table 10 show whether the test input i satisfies pre and pre'. Column 5 shows that i_1 satisfies $pre \lor pre'$ based on step 2 in the algorithm.

Step 3: Then, we apply each of the inputs i_1 - i_2 in columns 6 and 7 to obtain output from original program (D) and altered program (D') under o and o′, respectively. The last column indicates if the particular test case can be included in set T or not. A test input is added when o and $o'$are dissimilar. From Table 10, the first test case can be added in test set T, and it is being enhanced $T = T \cup \{i_1, o_1\}$ with vulnerability revealing test case $< i_1, o_1>$.

Table 10: Generation of test cases with altered pre and post-conditions for Login Bypass (P1)

#	Input	pre satisfied?	pre' satisfied?	pre ∨ pre'	Output D(i)	Output D'(i)	Include in set T?
1	**i_1:** loginid = "*)(uid=*))(|(uid=*)", password = "abcdef"	**No**	**Yes**	**Yes**	**o_1:** isLogin = TRUE	**o'_1:** isLogin = FALSE	**Yes**
2	**i_2:** loginid = "sprice", password = "prices"	**Yes**	**Yes**	**Yes**	**o_2:** isLogin = TRUE	**o'_2:** isLogin = TRUE	**No**

For example, we assume that the *loginid* in input i_2 is given as *sprice* and the *password* is a legitimate password as *prices*. Though the input satisfies both *pre* and *pre'*, the output after applying it on *D* and *D'* remains the same. Thus, test input i_2 is not added in test set *T*.

Therefore, the final test set *T* after applying the algorithm results in selected test input and output as: *{<*)(uid=*))(|(uid=*), abcdef>}*. As we apply this generated test inputs on the target application, we find that i_1 has the capability of revealing login bypass injection attack.

5.4 Case study 3: Custom Web Application (Privilege Escalation)

In this Section, we demonstrate the test case generation for Privilege Escalation type of LDAP injection attack. We apply the process for generation of pre and post-conditions below.

Process for Generation of pre and post-conditions

Step 1: We develop a class diagram as shown in Figure 29 capturing the key class attributes in the source code.

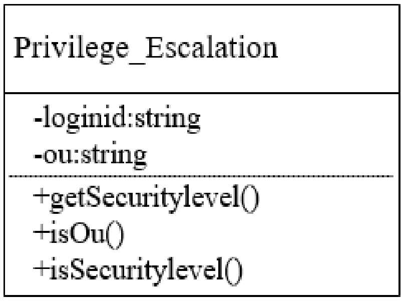


Figure 29: Class diagram for Custom application (Privilege Escalation)

Step 2: We develop a flow chart as shown in Figure 30 for privilege escalation LDAP injection attack type. Here, a rectangle means steps (input or output), an ellipse means the start or end state and a diamond is a decision making step where testing of conditions are performed.

Step 3: Figure 30 shows various paths related to both successful and unsuccessful privilege escalation scenarios.

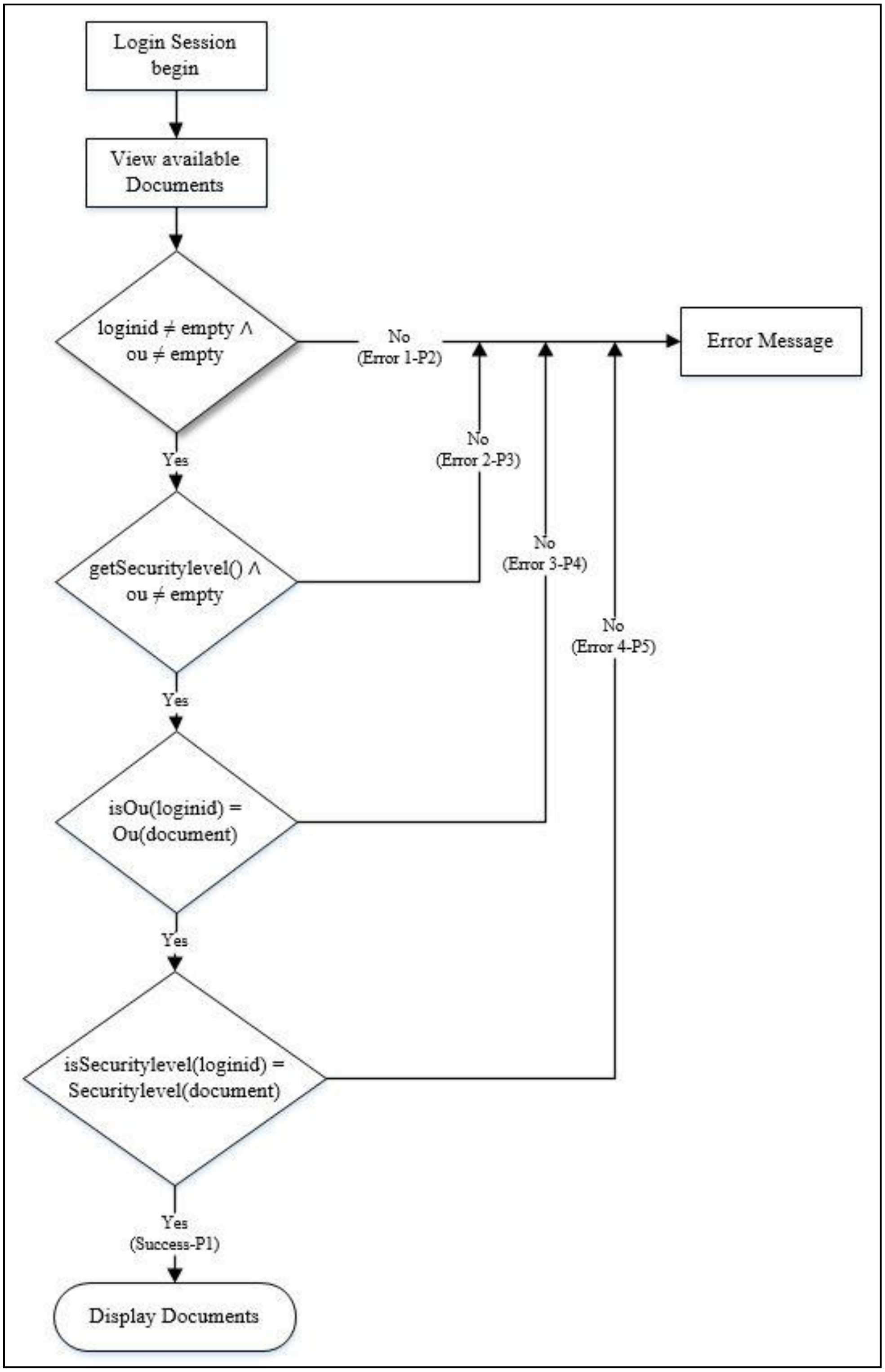


Figure 30: Flowchart for privilege escalation type of LDAP injection attack

Step 4: Table 11 shows the combined pre and post-conditions for each of the five paths based on Figure 30. There is no duplicate condition, so no reduction of conditions needs to be performed.

Table 11: Pre and post-conditions for Privilege Escalation

Path	pre-conditions	post-conditions
P1 (Success)	(loginid ≠ empty ∧ ou ≠ empty) ∧ (getSecuritylevel() ∧ ou ≠ empty) ∧ isOu(loginid) = Ou(document) ∧ isSecuritylevel(loginid) = Securitylevel(document)	Display Documents
P2 (Error1)	!(loginid ≠ empty ∧ ou ≠ empty)	Display Documents fail
P3 (Error2)	(loginid ≠ empty ∧ ou ≠ empty) ∧ !(getSecuritylevel() ∧ ou ≠ empty)	Display Documents fail
P4 (Error3)	(loginid ≠ empty ∧ ou ≠ empty) ∧ (getSecuritylevel() ∧ ou ≠ empty) ∧ !(isOu(loginid) = Ou(document))	Display Documents fail
P5 (Error4)	(loginid ≠ empty ∧ ou ≠ empty) ∧ (getSecuritylevel() ∧ ou ≠ empty) ∧ isOu(loginid) = Ou(document) ∧ !(isSecuritylevel(loginid) = Securitylevel(document))	N/A

Now, we apply the three steps of Fault adequate test case generator Algorithm for paths P1 and P3.

Algorithm: Fault adequate test case generator

Step 1: We generate altered pre-conditions (*pre'*). Table 12 shows three examples of altered pre-conditions (*pre'*) where the first and second rows correspond to P1 and the third row corresponds to P3. Here, we randomly replaced ∧ with ∨ (changes are shown in bold font in the third column). Each expression relates to two input variables (or test inputs) represent two fields: *loginid* and *ou*. We assume that valid *emailid* is *x@xyz.com* and valid *loginid* is *sprice.* We also assume that * is an invalid character for this application. These set of valid inputs along with meta-characters will be combined to generate test cases that we discuss next.

Table 12: Altered pre-conditions and test inputs for Privilege Escalation (P1 and P3)

Example	pre	pre′
1 (P1)	(loginid $\neq$ empty $\wedge$ ou $\neq$ empty) $\wedge$ (getSecuritylevel() $\wedge$ ou $\neq$ empty) $\wedge$ isOu(loginid) = Ou(document) $\wedge$ isSecuritylevel(loginid) = Securitylevel(document)	(loginid $\neq$ empty $\wedge$ ou $\neq$ empty) $\vee$ (getSecuritylevel() $\wedge$ ou $\neq$ empty) $\wedge$ isOu(loginid) = Ou(document) $\wedge$ isSecuritylevel(loginid) = Securitylevel(document)
2 (P1)	(loginid $\neq$ empty $\wedge$ ou $\neq$ empty) $\wedge$ (getSecuritylevel() $\wedge$ ou $\neq$ empty) $\wedge$ isOu(loginid) = Ou(document) $\wedge$ isSecuritylevel(loginid) = Securitylevel(document)	(loginid $\neq$ empty $\wedge$ ou $\neq$ empty) $\wedge$ (getSecuritylevel() $\wedge$ ou $\neq$ empty) $\vee$ isOu(loginid) = Ou(document) $\vee$ isSecuritylevel(loginid) = Securitylevel(document)
3 (P3)	(loginid $\neq$ empty $\wedge$ ou $\neq$ empty) $\wedge$!(getSecuritylevel() $\wedge$ ou $\neq$ empty)	(loginid $\neq$ empty $\wedge$ ou $\neq$ empty) $\vee$!(getSecuritylevel() $\wedge$ ou $\neq$ empty)

Step 2: The *loginid* in input i_1 is given as *sprice* and *ou* is given as *. For checking if generated test input i_1, from step 1 can satisfy $pre \vee pre'$, we do further analysis as shown in Table 13. Here, the second column shows a set of test inputs (*i*). Columns 3 and 4 in Table 13 show whether the test input *i* satisfies *pre* and *pre′*. Column 5 shows that i_1 satisfies $pre \vee pre'$ based on step 2 in the algorithm.

Step 3: We apply each of the inputs i_1 - i_3 in columns 6 and 7 (Table 13) to obtain output from original program (D) and altered program (D′) under *o* and o′, respectively. The last column indicates if the particular test case can be included in set T or not, based on conditions in step 4. A test input is added when *o* and *o′*are dissimilar. From Table 13, test cases 1 and 2 can be added in test set *T*, and it is being enhanced $T = T \cup \{(i_1, o_1), (i_2, o_2)\}$ with vulnerability revealing test cases $\{< i_1, o_1>, < i_2, o_2>\}$.

Let us consider the example shown in the first row (Table 13). The *loginid* is given as *sprice* and *ou* is given as *. Columns 3 and 4 show whether *i* satisfies *pre* and *pre'* (not satisfied for *pre*, satisfied for *pre'*). Column 5 shows that *i* satisfies $pre \vee pre'$ based on step 2 in the algorithm. Then, we apply input i_1 in columns 6 and 7 to obtain output from *D* and *D′*. The last column indicates if the particular test case can be included in set *T* or not, which indicates that i_1 can be included as a test case. Similarly, i_2 can be included in set *T*.

Table 13: Generation of test cases with altered pre and post-conditions for Privilege Escalation (P1 and P3)

#	Input	pre satisfied?	pre' satisfied?	pre $\vee$ pre'	Output D(i)	Output D'(i)	Include in set T?
1	**i_1:** loginid = " sprice ", ou = "*"	**No**	**Yes**	**Yes**	**o_1:** Display Documents = FALSE	**o'$_1$:** Display Documents fail = TRUE	**Yes**
2	**i_2:** loginid = " sp* ", ou = "*"	**No**	**Yes**	**Yes**	**o_2:** Display Documents = FALSE	**o'$_2$:** Display Documents fail = TRUE	**Yes**
3	**i_3:** loginid = "sprice" ou = "SalesRep"	**Yes**	**Yes**	**Yes**	**o_3:** Display Documents = TRUE	**o'$_3$:** Display Documents = TRUE	**No**

Let us consider the third example. We assume that *loginid* is *sprice* and *ou* is *SalesRep*. Though the input satisfies both *pre* and *pre'*, the output after applying on *D* and *D'* remains the same. Thus, it is not added in test set *T*.

Therefore, the final test set *T* after applying Fault adequate test case generator Algorithm results in test input and output as: *{<sprice,*>, <sp*,*>}*. As we apply this generated test inputs on the target application, we find that i_1 *and* i_2 have the capability of revealing privilege escalation injection attacks.

5.5 Case study 4: Custom Web Application (Information Alteration)

In this Section, we demonstrate the test case generation for Information Alteration type of LDAP injection attack. We apply the four steps of process for generation of pre and post-conditions below.

Process for Generation of pre and post-conditions

Step 1: We develop a class diagram as shown in Figure 31 capturing the key class attributes in the source code.

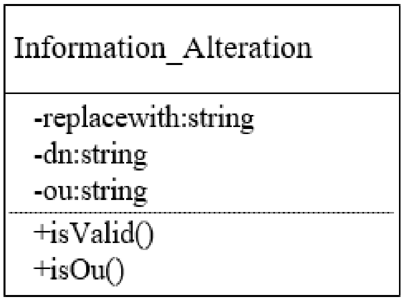


Figure 31: Class diagram for Custom application (Information Alteration)

Step 2: We develop a flow chart as shown in Figure 32 for information alteration LDAP injection attack type. Here, a rectangle means steps (input or output), an ellipse means the start or end state and a diamond is a decision making step where testing of conditions are performed.

Step 3: Figure 32 shows various paths related to both successful and unsuccessful information alteration scenarios.

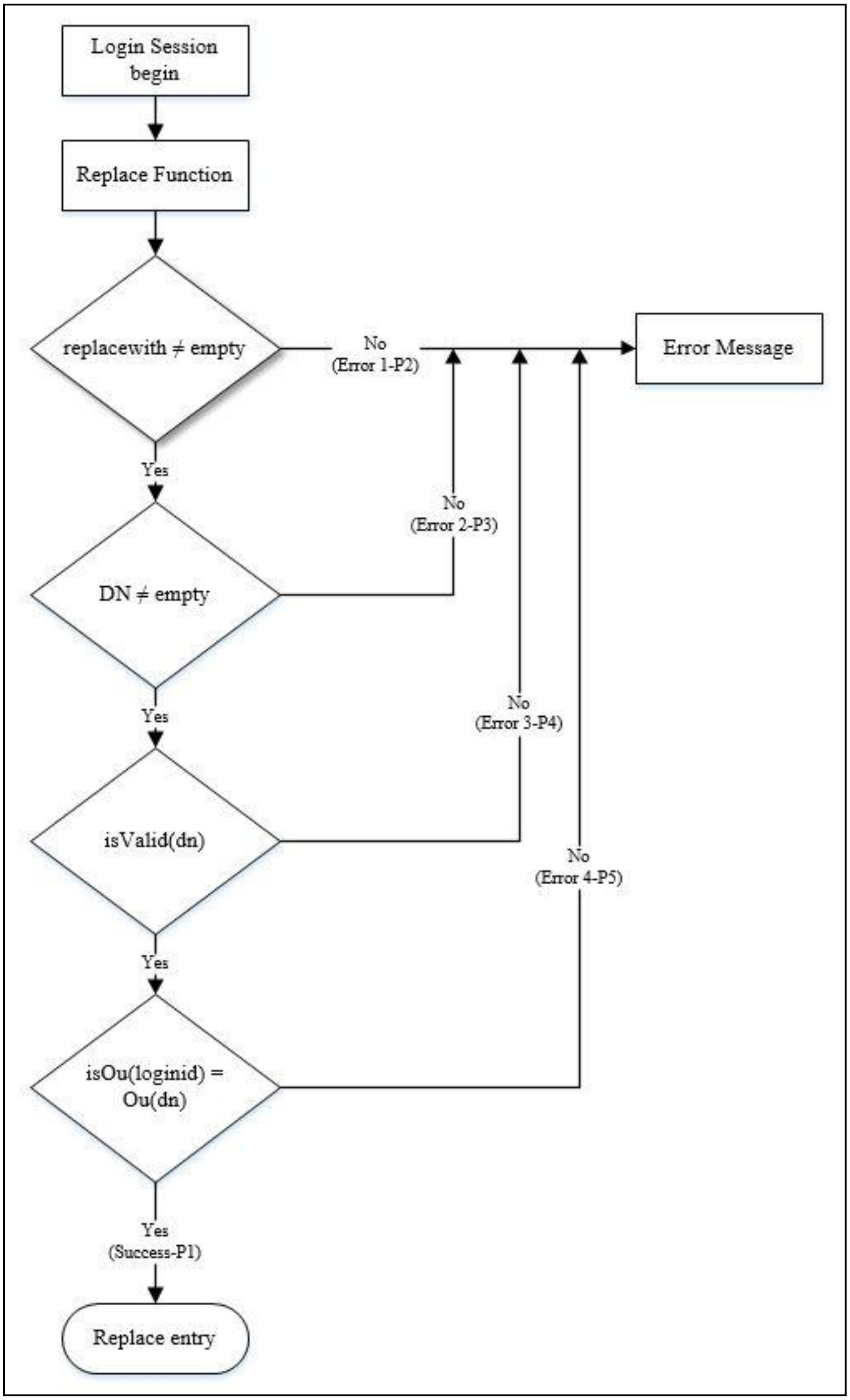


Figure 32: Flowchart for information alteration type of LDAP injection attack

Step 4: Table 14 shows the combined pre and post-conditions for each of the four paths based on Figure 32. There is no duplicate condition, so no reduction of conditions needs to be performed.

Table 14: Pre and post-conditions for Information Alteration

Path	pre-conditions	post-conditions
P1 (Success)	(replacewith ≠ empty ∧ dn ≠ empty) ∧ isValid(dn) ∧ (isOu(loginid) = Ou(dn))	Replace Entry
P2 (Error1)	!(replacewith ≠ empty)	Replace Entry fail
P3 (Error2)	!(dn ≠ empty)	Replace Entry fail
P4 (Error3)	!(isValid(dn))	Replace Entry fail
P5 (Error4)	!(isOu(loginid) = Ou(dn))	N/A

Now, we apply the three steps of Fault adequate test case generator Algorithm for path P1.

Algorithm: Fault adequate test case generator

Step 1: From *D* we obtain *D'*. We generate altered pre- conditions (*pre'*). Table 15 shows two examples of altered pre-conditions (*pre'*) for P1 where we replaced ∧ with ∨ randomly. This is not an exhaustive list of all possible pre′ but we show some examples for illustrative purposes. Each expression relates to two input variables (or test inputs) represent two fields: *replacewith* and *dn*. We assume that valid *dn* is *uid=sprice, cn=SalesRep, ou=Sales,dc=ubuntuldap2*. We also assume that * is an invalid character and is not permitted as any part of the inputs for this application. These set of valid inputs along with meta-characters will be combined to generate test cases that we discuss next. We discuss the input i_1 in which a document is being replaced.

Step 2: The *replacewith* in input i_1 is given as *randomlink* and *dn* is given as *uid=jreed, cn=SeniorMgmt, ou=SeniorMgmt, dc=ubuntuldap2*. For checking if generated test input i_1, from step 1 can satisfy $pre \vee pre'$, we do further analysis as shown in Table 16. Here, the second column shows a set of test inputs (*i*). Columns 3 and 4 in Table 16 show whether the test input *i* satisfies pre and pre'. Column 5 shows that i_1 satisfies $pre \vee pre'$.

Table 15: Altered pre-conditions and test inputs for Information Alteration (P1)

Example	pre	pre′
1	(replacewith $\neq$ empty $\wedge$ dn $\neq$ empty) $\wedge$ isValid(dn) $\wedge$ (isOu(loginid) = Ou(dn))	(replacewith $\neq$ empty $\wedge$ dn $\neq$ empty) $\vee$ isValid(dn) $\wedge$ (isOu(loginid) = Ou(dn))
2	(replacewith $\neq$ empty $\wedge$ dn $\neq$ empty) $\wedge$ isValid(dn) $\wedge$ (isOu(loginid) = Ou(dn))	(replacewith $\neq$ empty $\vee$ dn $\neq$ empty) $\vee$ isValid(dn) $\wedge$ (isOu(loginid) = Ou(dn))

Step 3: We apply each of the inputs i_1 and i_2 in columns 6 and 7 to obtain output from original program (D) and altered program (D′) under o and o′, respectively. The last column indicates if the particular test case can be included in set *T* or not. A test input is added when o and o' are dissimilar.

From Table 16, the first test case can be added to test set *T*, and it is being enhanced $T = T \cup \{i_1, o_1\}$ with vulnerability revealing test case $< i_1, o_1>$.

Table 16: Generation of test cases with altered pre and post-conditions for Information Alteration (P1)

#	**Input**	**pre satisfied?**	**pre' satisfied?**	**pre $\vee$ pre'**	**Output D(i)**	**Output D'(i)**	**Include in set T?**
1	**i_1:** replacewith = "randomlink" dn = "uid=jreed,cn=SeniorMgmt,ou= SeniorMgmt,dc=ubuntuldap2"	**No**	**Yes**	**Yes**	**o_4:** ReplaceEntry = TRUE	**o'_4:** ReplaceEntry = FALSE	**Yes**
2	**i_2:** replacewith = "correctlink" dn = "uid=jreed,cn=SeniorMgmt,ou= SeniorMgmt,dc=ubuntuldap2"	**Yes**	**Yes**	**Yes**	**o_5:** isLogin = TRUE	**o'_5:** isLogin = TRUE	**No**

For example, we assume that the *replacewith* in input i_2 is given as *correctlink* and the *dn* is the *dn* of the user logged in. Though the input satisfies both *pre* and *pre'*, the output after applying it on *D* and *D'* remains same. Thus, it is not added to test set *T*.

Therefore, the final test set *T* after applying the algorithm results in test input and output as: *{<replacewith="randomlink";dn=uid=jreed,cn=SeniorMgmt,ou=SeniorMgmt,dc=ubuntuldap2>}*. As we apply this generated test inputs on the target application, we find that i_1 has the capability of revealing information alteration injection attack.

Thus, our approach enables developers to generate effective test cases based on OCL fault injection approach to detect LDAP injection vulnerabilities.

The next Chapter demonstrates tool implementation in which we have automated the process of random OR replacement.

CHAPTER 6

Tool Implementation

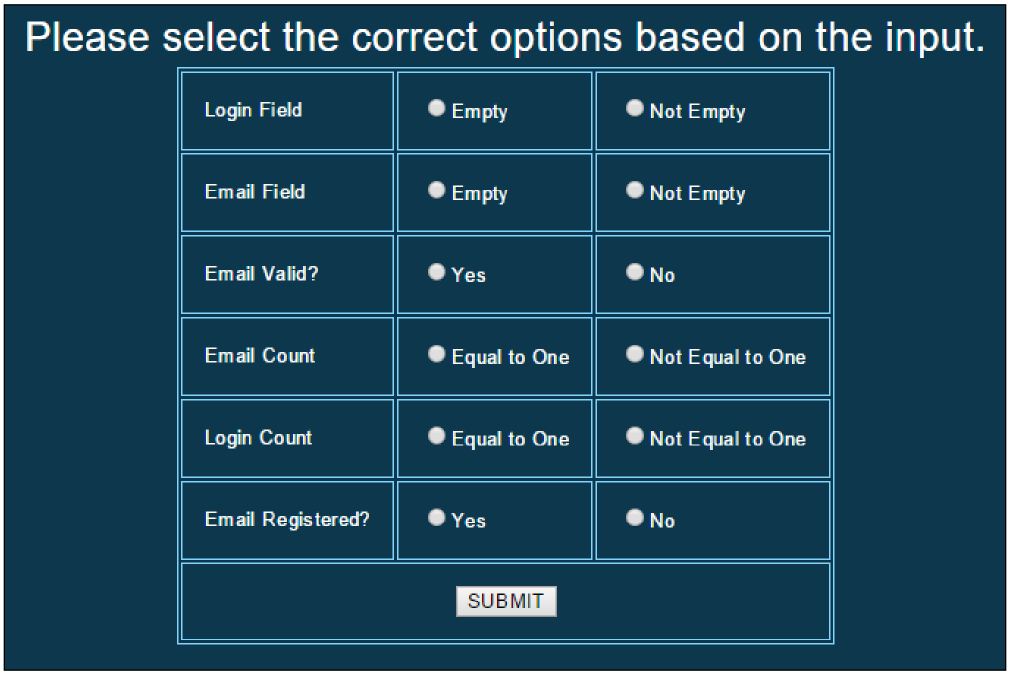
6.1 Overview

In this Chapter, we demonstrate a tool which can be used for the selection of test cases. The test cases mentioned earlier in Chapter 5 were generated manually. We demonstrate implementation of tool by using Self Service Password and Login Bypass case studies (mentioned earlier in Chapter 5, Section 5.2 and 5.3) as examples. We apply a common measure to assess the quality of generated test cases called Mutation Score (MS) [49]. It is the ratio between the number of test cases included in the test set *T* to the total number of test cases generated. The Mutation Score is affected by the combination of user inputs. When the input values include high number of erroneous (invalid) inputs, the algorithm tends to generate high number of test cases to be included in the test set *T*. For example, when the user inputs are all valid, the number of generated test cases that can be included in the test set *T* are either zero or a minimum number possible. This leads to a low mutation score. On the other hand, when the inputs are all invalid, it is most likely that all or a high number of generated test cases are included in the test set *T*, leading to a high mutation score.

6.2 Tool implementation for Self Service Password

Example 1:

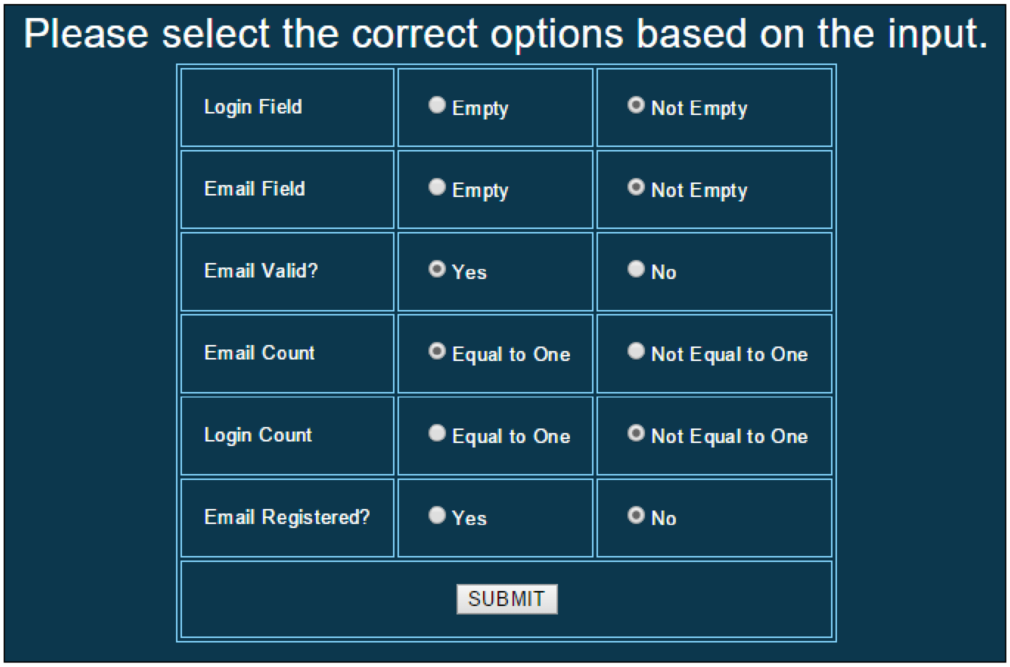
First, we have the questions based on the application of Self Service Password. We defined six attributes based on the application implementation to represent pre-conditions. Depending upon the options selected at this point, the tool randomly substitutes AND with OR and provides us all the possible combinations for this substitution. Developers are required to generate initial test cases that can satisfy the generated constraints.



Please select the correct options based on the input.

Login Field	○ Empty	○ Not Empty
Email Field	○ Empty	○ Not Empty
Email Valid?	○ Yes	○ No
Email Count	○ Equal to One	○ Not Equal to One
Login Count	○ Equal to One	○ Not Equal to One
Email Registered?	○ Yes	○ No

Figure 33: Questions based on Self Service Password application



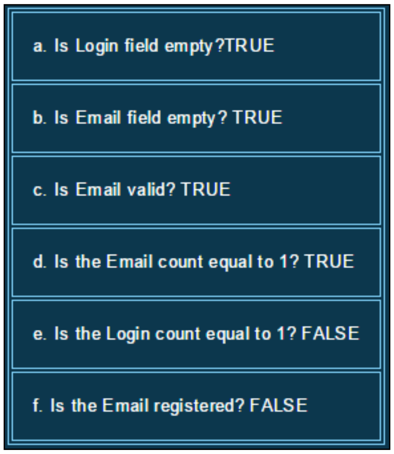
Please select the correct options based on the input.

Login Field	○ Empty	● Not Empty
Email Field	○ Empty	● Not Empty
Email Valid?	● Yes	○ No
Email Count	● Equal to One	○ Not Equal to One
Login Count	○ Equal to One	● Not Equal to One
Email Registered?	○ Yes	● No

Figure 34: Selected options for Self Service Password application (Example 1)

For example, the options are selected as shown in Figure 34. For the given selection an example input can be Login: *sp**; Email: *x@xyz.com*.

Boolean values are assigned to each entity (field) as shown in Figure 35. These values are based on the selection from the previous page. Also, our pre-condition for the Self Service password application is $[(a \wedge b) \wedge c \wedge (d \wedge e) \wedge f]$ which should be satisfied for a successful operation.



a. Is Login field empty?TRUE

b. Is Email field empty? TRUE

c. Is Email valid? TRUE

d. Is the Email count equal to 1? TRUE

e. Is the Login count equal to 1? FALSE

f. Is the Email registered? FALSE

Figure 35: Assigned Boolean values based on selection (Example 1)

Once we have the Boolean values assigned depending on the given selection, the options available for logical OR Replacements are shown in Figure 36.

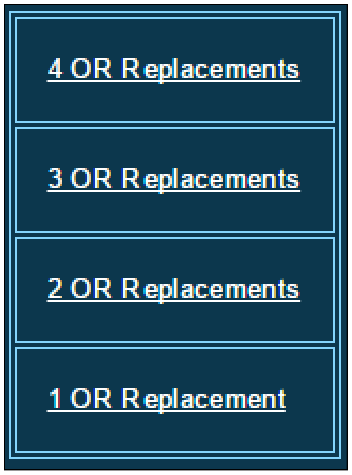
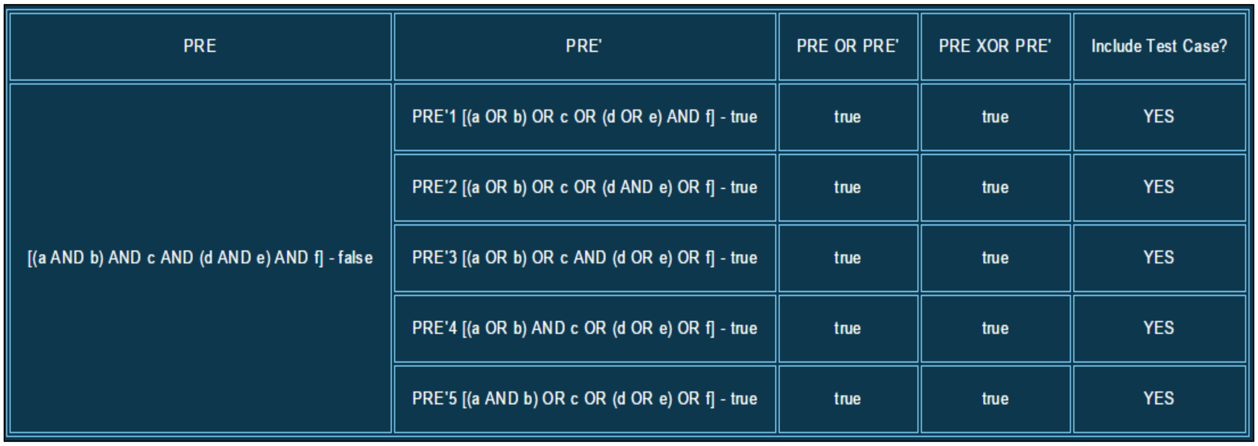


Figure 36: Available OR replacement options

Let us check the selection of test cases for each OR replacement option.

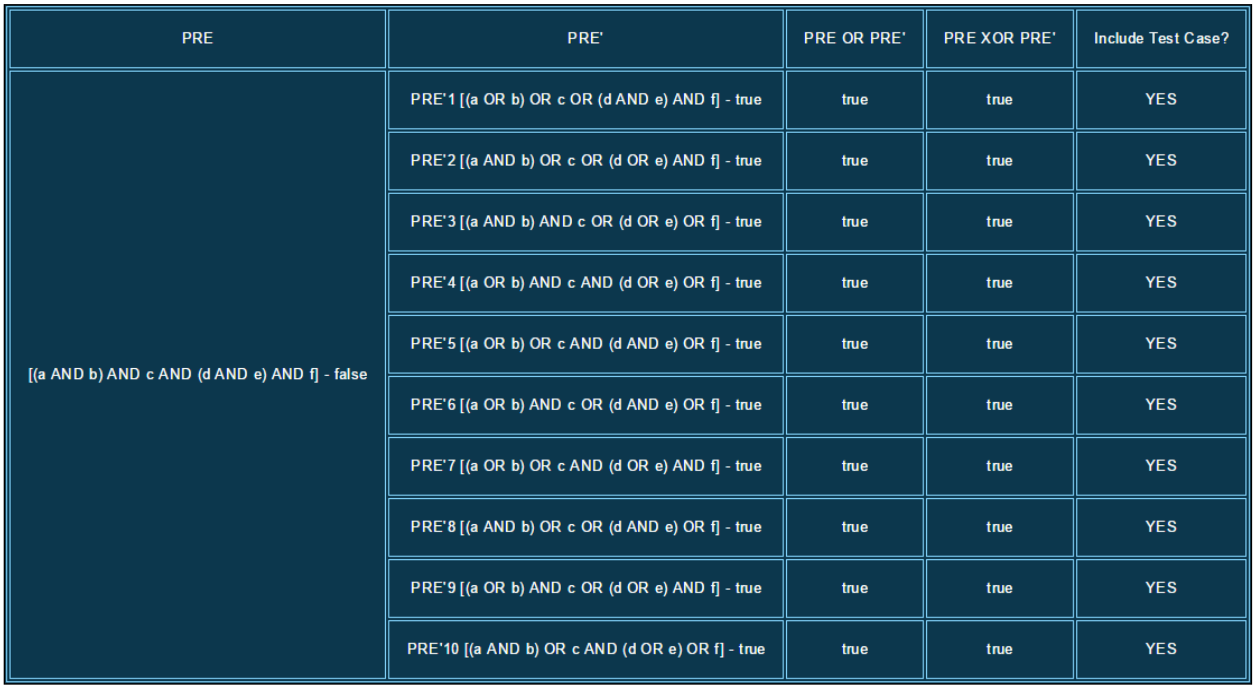
4 OR Replacement Result Set:



PRE	PRE'	PRE OR PRE'	PRE XOR PRE'	Include Test Case?
[(a AND b) AND c AND (d AND e) AND f] - false	PRE'1 [(a OR b) OR c OR (d OR e) AND f] - true	true	true	YES
	PRE'2 [(a OR b) OR c OR (d AND e) OR f] - true	true	true	YES
	PRE'3 [(a OR b) OR c AND (d OR e) OR f] - true	true	true	YES
	PRE'4 [(a OR b) AND c OR (d OR e) OR f] - true	true	true	YES
	PRE'5 [(a AND b) OR c OR (d OR e) OR f] - true	true	true	YES

Based on the last column of the 4 OR Replacement screenshot, all test cases are relevant to detect an attack, therefore are included in the test set *T*. This leads to a Mutation Score (MS) of 5/5 or 100%.

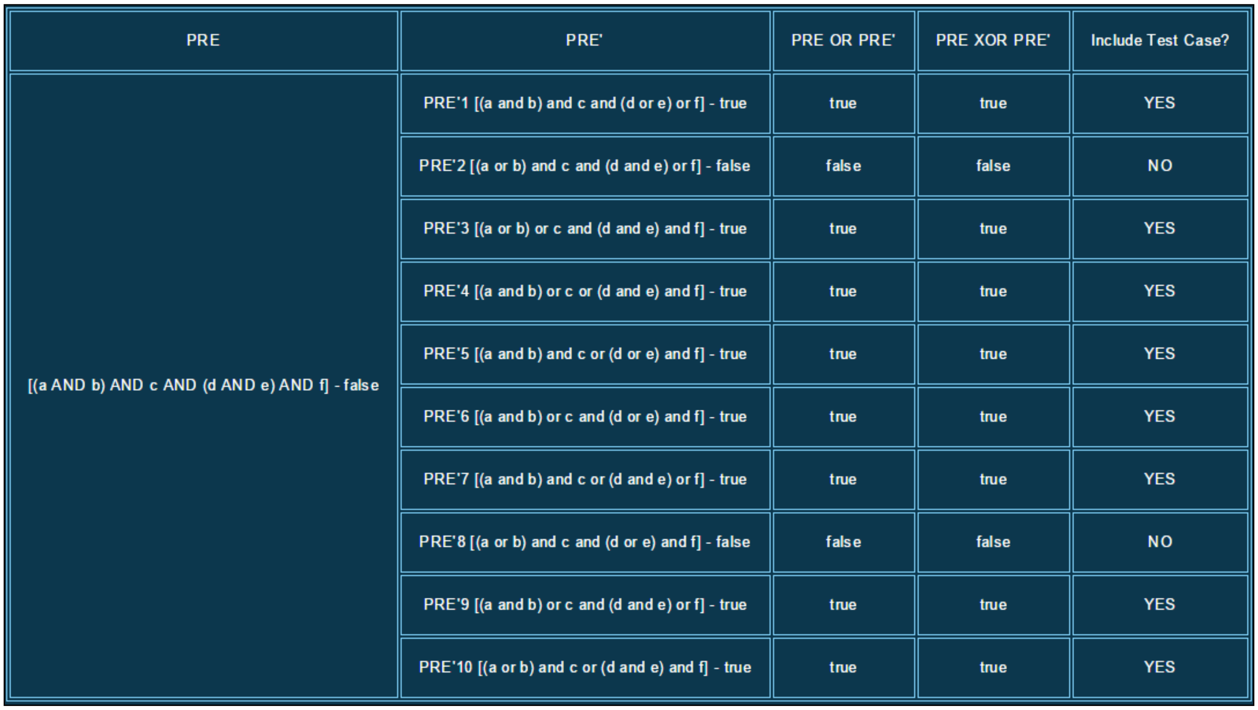
3 OR Replacement Result Set:



PRE	PRE'	PRE OR PRE'	PRE XOR PRE'	Include Test Case?
[(a AND b) AND c AND (d AND e) AND f] - false	PRE'1 [(a OR b) OR c OR (d AND e) AND f] - true	true	true	YES
	PRE'2 [(a AND b) OR c OR (d OR e) AND f] - true	true	true	YES
	PRE'3 [(a AND b) AND c OR (d OR e) OR f] - true	true	true	YES
	PRE'4 [(a OR b) AND c AND (d OR e) OR f] - true	true	true	YES
	PRE'5 [(a OR b) OR c AND (d AND e) OR f] - true	true	true	YES
	PRE'6 [(a OR b) AND c OR (d AND e) OR f] - true	true	true	YES
	PRE'7 [(a OR b) OR c AND (d OR e) AND f] - true	true	true	YES
	PRE'8 [(a AND b) OR c OR (d AND e) OR f] - true	true	true	YES
	PRE'9 [(a OR b) AND c OR (d OR e) AND f] - true	true	true	YES
	PRE'10 [(a AND b) OR c AND (d OR e) OR f] - true	true	true	YES

Based on the last column of the 3 OR Replacement screenshot, all test cases are relevant to detect an attack, therefore are included in the test set *T*. This leads to a Mutation Score (MS) of 10/10 or 100%.

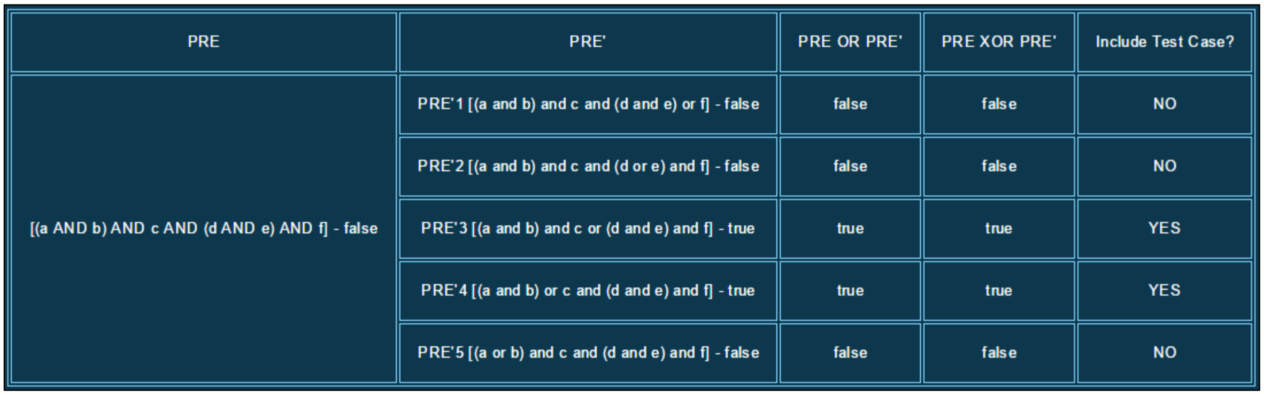
2 OR Replacement Result Set:



PRE	PRE'	PRE OR PRE'	PRE XOR PRE'	Include Test Case?
[(a AND b) AND c AND (d AND e) AND f] - false	PRE'1 [(a and b) and c and (d or e) or f] - true	true	true	YES
	PRE'2 [(a or b) and c and (d and e) or f] - false	false	false	NO
	PRE'3 [(a or b) or c and (d and e) and f] - true	true	true	YES
	PRE'4 [(a and b) or c or (d and e) and f] - true	true	true	YES
	PRE'5 [(a and b) and c or (d or e) and f] - true	true	true	YES
	PRE'6 [(a and b) or c and (d or e) and f] - true	true	true	YES
	PRE'7 [(a and b) and c or (d and e) or f] - true	true	true	YES
	PRE'8 [(a or b) and c and (d or e) and f] - false	false	false	NO
	PRE'9 [(a and b) or c and (d and e) or f] - true	true	true	YES
	PRE'10 [(a or b) and c or (d and e) and f] - true	true	true	YES

Based on the last column of the 2 OR Replacement screenshot, eight test cases are relevant to detect an attack, therefore are included in the test set *T*. This leads to a Mutation Score (MS) of 8/10 or 80%.

1 OR Replacement Result Set:

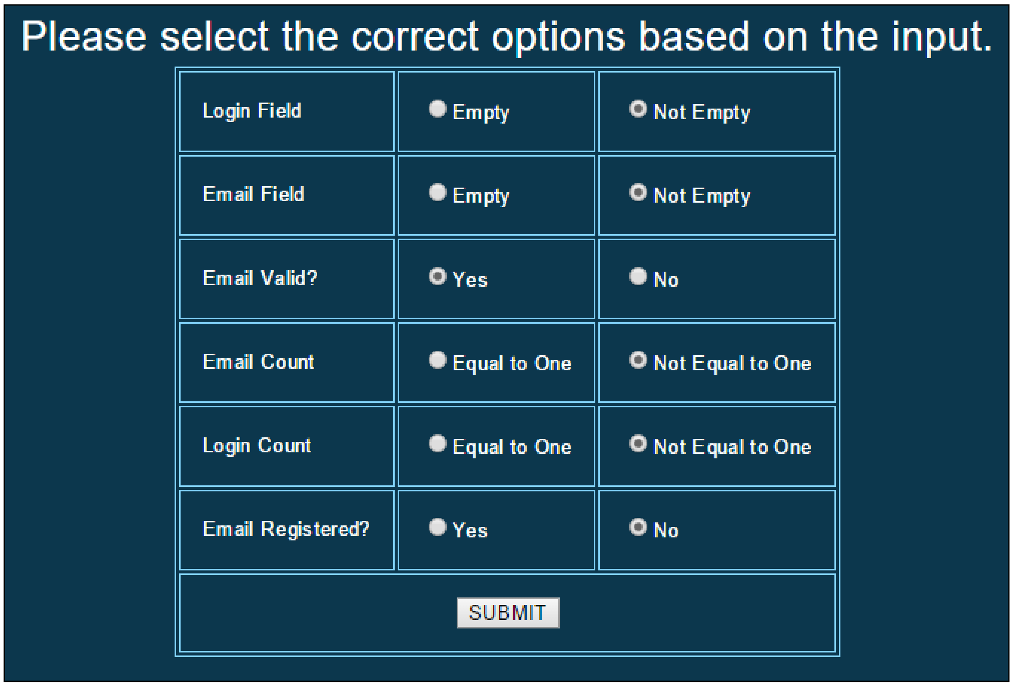


PRE	PRE'	PRE OR PRE'	PRE XOR PRE'	Include Test Case?
[(a AND b) AND c AND (d AND e) AND f] - false	PRE'1 [(a and b) and c and (d and e) or f] - false	false	false	NO
	PRE'2 [(a and b) and c and (d or e) and f] - false	false	false	NO
	PRE'3 [(a and b) and c or (d and e) and f] - true	true	true	YES
	PRE'4 [(a and b) or c and (d and e) and f] - true	true	true	YES
	PRE'5 [(a or b) and c and (d and e) and f] - false	false	false	NO

Based on the last column of the 1 OR Replacement screenshot, two test cases are relevant to detect an attack, therefore are included in the test set *T*. This leads to a Mutation Score (MS) of 2/5 or 40%.

<u>Example 2</u>:

First, we have the questions based on the application of Self Service Password. For example, the options are selected as shown in Figure 37.

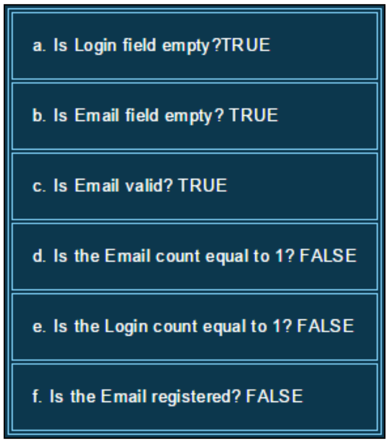


Please select the correct options based on the input.

Login Field	○ Empty	● Not Empty
Email Field	○ Empty	● Not Empty
Email Valid?	● Yes	○ No
Email Count	○ Equal to One	● Not Equal to One
Login Count	○ Equal to One	● Not Equal to One
Email Registered?	○ Yes	● No

Figure 37: Selected options for Self Service Password application (Example 2)

Boolean values are assigned to each entity (field) as shown in Figure 38. These values are based on the selection from the previous page. Also, our pre-condition for the Self Service password application is [(a ⋀ b) ⋀ c ⋀ (d ⋀ e) ⋀ f] which should be satisfied for a successful operation.



a. Is Login field empty?TRUE

b. Is Email field empty? TRUE

c. Is Email valid? TRUE

d. Is the Email count equal to 1? FALSE

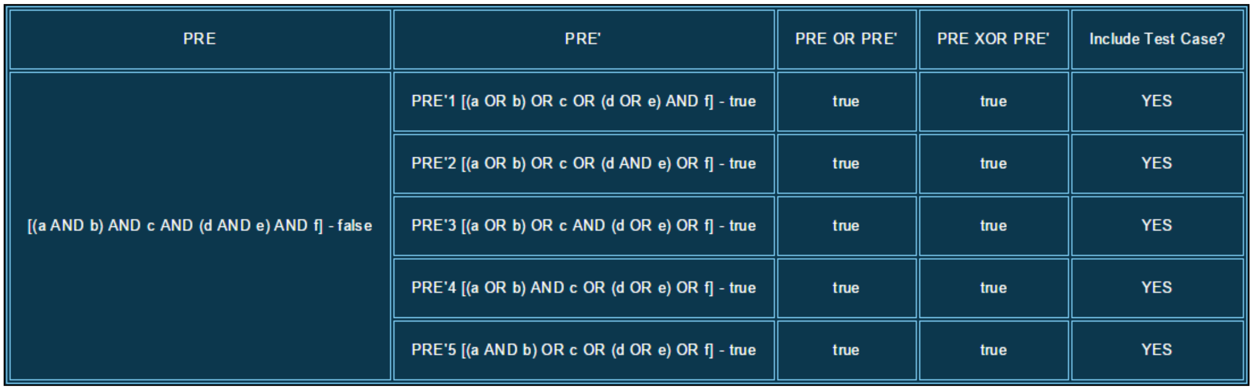
e. Is the Login count equal to 1? FALSE

f. Is the Email registered? FALSE

Figure 38: Assigned Boolean values based on selection (Example 2)

Let us check the selection of test cases for each OR replacement option.

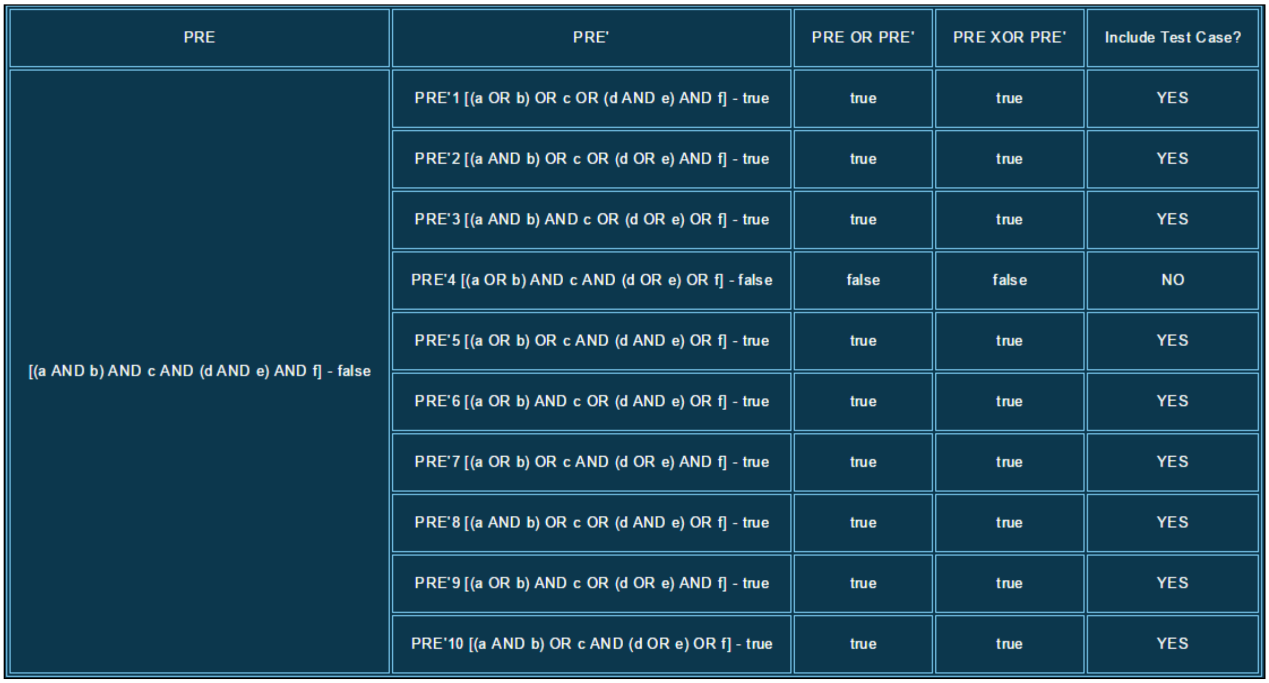
4 OR Replacement Result Set:



PRE	PRE'	PRE OR PRE'	PRE XOR PRE'	Include Test Case?
[(a AND b) AND c AND (d AND e) AND f] - false	PRE'1 [(a OR b) OR c OR (d OR e) AND f] - true	true	true	YES
	PRE'2 [(a OR b) OR c OR (d AND e) OR f] - true	true	true	YES
	PRE'3 [(a OR b) OR c AND (d OR e) OR f] - true	true	true	YES
	PRE'4 [(a OR b) AND c OR (d OR e) OR f] - true	true	true	YES
	PRE'5 [(a AND b) OR c OR (d OR e) OR f] - true	true	true	YES

Based on the last column of the 4 OR Replacement screenshot, all test cases are relevant to detect an attack, therefore are included in the test set *T*. This leads to a Mutation Score (MS) of 5/5 or 100%.

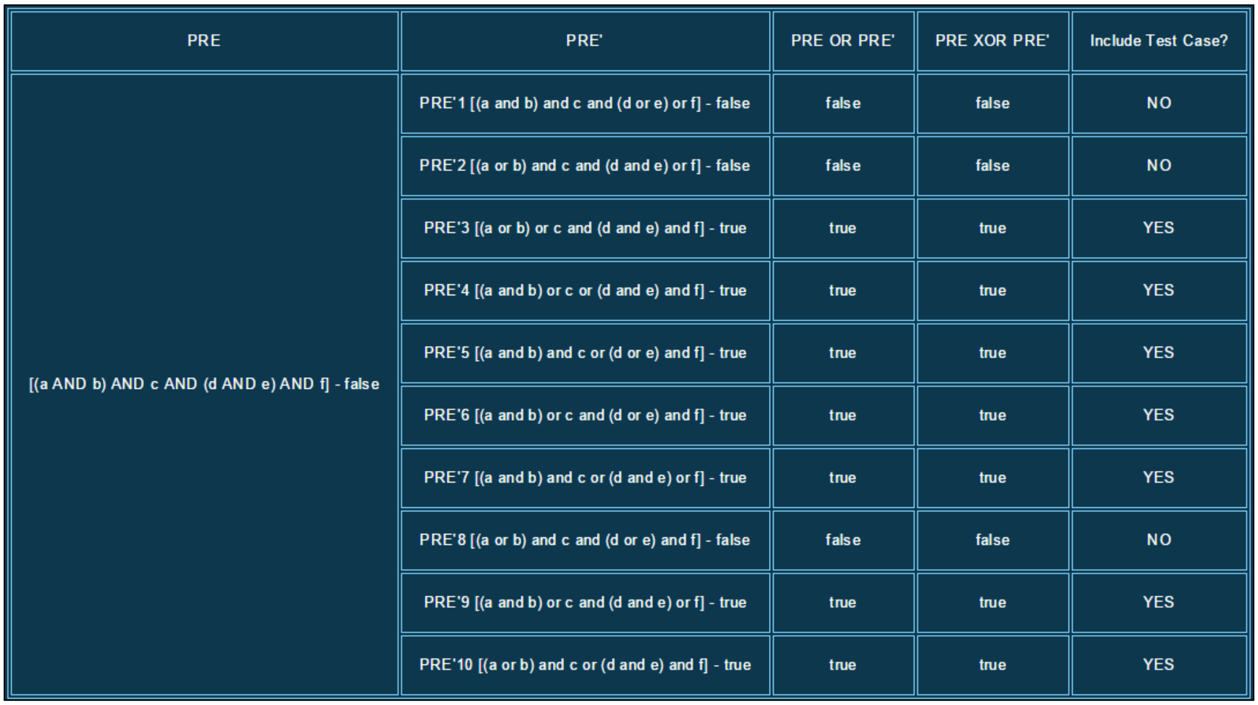
3 OR Replacement Result Set:



PRE	PRE'	PRE OR PRE'	PRE XOR PRE'	Include Test Case?
[(a AND b) AND c AND (d AND e) AND f] - false	PRE'1 [(a OR b) OR c OR (d AND e) AND f] - true	true	true	YES
	PRE'2 [(a AND b) OR c OR (d OR e) AND f] - true	true	true	YES
	PRE'3 [(a AND b) AND c OR (d OR e) OR f] - true	true	true	YES
	PRE'4 [(a OR b) AND c AND (d OR e) OR f] - false	false	false	NO
	PRE'5 [(a OR b) OR c AND (d AND e) OR f] - true	true	true	YES
	PRE'6 [(a OR b) AND c OR (d AND e) OR f] - true	true	true	YES
	PRE'7 [(a OR b) OR c AND (d OR e) AND f] - true	true	true	YES
	PRE'8 [(a AND b) OR c OR (d AND e) OR f] - true	true	true	YES
	PRE'9 [(a OR b) AND c OR (d OR e) AND f] - true	true	true	YES
	PRE'10 [(a AND b) OR c AND (d OR e) OR f] - true	true	true	YES

Based on the last column of the 3 OR Replacement screenshot, nine test cases are relevant to detect an attack, therefore are included in the test set *T*. This leads to a Mutation Score (MS) of 9/10 or 90%.

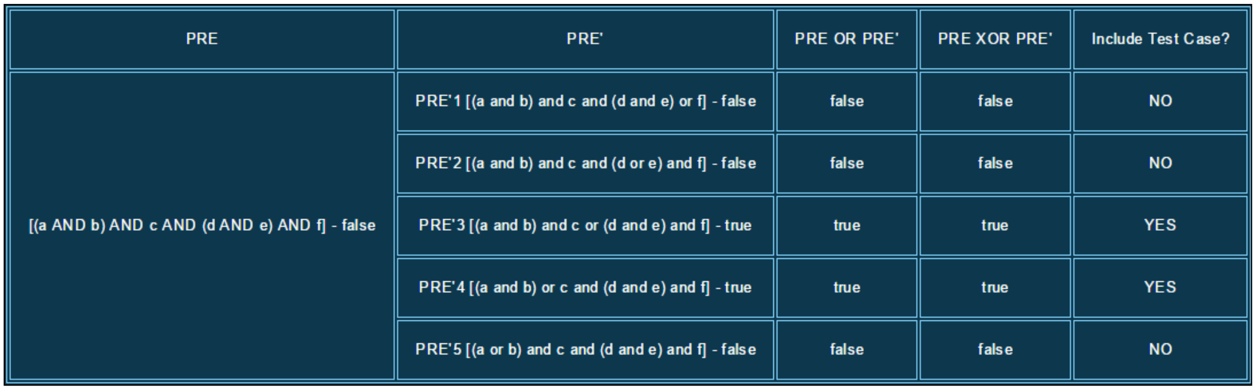
2 OR Replacement Result Set:



PRE	PRE'	PRE OR PRE'	PRE XOR PRE'	Include Test Case?
[(a AND b) AND c AND (d AND e) AND f] - false	PRE'1 [(a and b) and c and (d or e) or f] - false	false	false	NO
	PRE'2 [(a or b) and c and (d and e) or f] - false	false	false	NO
	PRE'3 [(a or b) or c and (d and e) and f] - true	true	true	YES
	PRE'4 [(a and b) or c or (d and e) and f] - true	true	true	YES
	PRE'5 [(a and b) and c or (d or e) and f] - true	true	true	YES
	PRE'6 [(a and b) or c and (d or e) and f] - true	true	true	YES
	PRE'7 [(a and b) and c or (d and e) or f] - true	true	true	YES
	PRE'8 [(a or b) and c and (d or e) and f] - false	false	false	NO
	PRE'9 [(a and b) or c and (d and e) or f] - true	true	true	YES
	PRE'10 [(a or b) and c or (d and e) and f] - true	true	true	YES

Based on the last column of the 2 OR Replacement screenshot, seven test cases are relevant to detect an attack, therefore are included in the test set *T*. This leads to a Mutation Score (MS) of 7/10 or 70%.

1 OR Replacement Result Set:

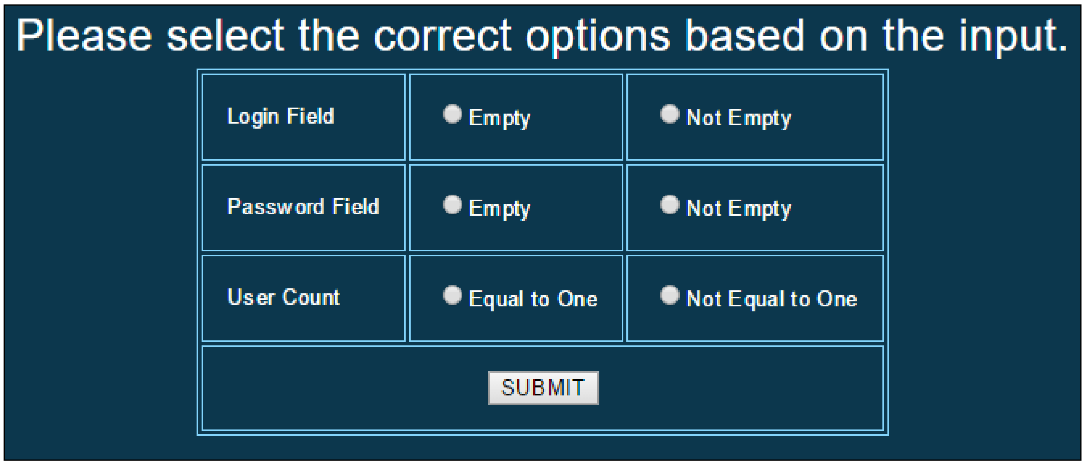


PRE	PRE'	PRE OR PRE'	PRE XOR PRE'	Include Test Case?
[(a AND b) AND c AND (d AND e) AND f] - false	PRE'1 [(a and b) and c and (d and e) or f] - false	false	false	NO
	PRE'2 [(a and b) and c and (d or e) and f] - false	false	false	NO
	PRE'3 [(a and b) and c or (d and e) and f] - true	true	true	YES
	PRE'4 [(a and b) or c and (d and e) and f] - true	true	true	YES
	PRE'5 [(a or b) and c and (d and e) and f] - false	false	false	NO

Based on the last column of the 1 OR Replacement screenshot, two test cases are relevant to detect an attack, therefore are included in the test set *T*. This leads to a Mutation Score (MS) of 2/5 or 40%.

6.3 Tool implementation for Custom web application (Login Bypass)

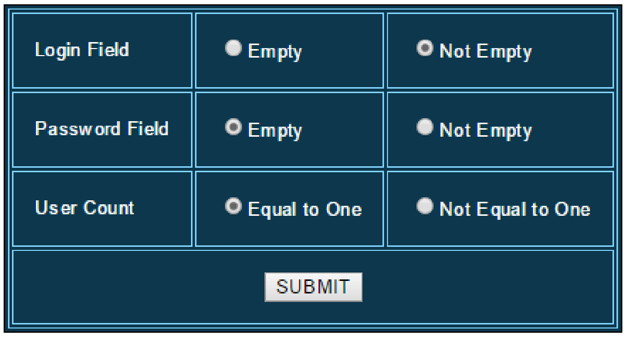
First, we have the questions based on custom web application for Login Bypass. We defined three attributes based on the implementation to represent pre-conditions. Depending upon the options selected at this point, the tool helps us randomly substituting AND with OR and gives us all the possible combinations for this substitution.



Please select the correct options based on the input.

Login Field	○ Empty	○ Not Empty
Password Field	○ Empty	○ Not Empty
User Count	○ Equal to One	○ Not Equal to One
SUBMIT		

Figure 39: Questions based on custom web application for Login Bypass

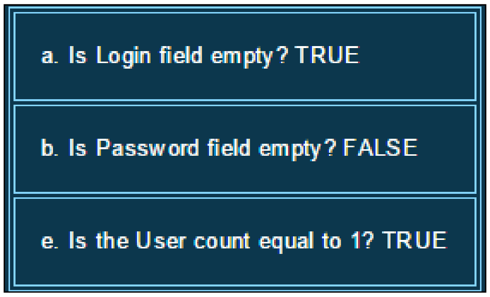


Login Field	○ Empty	● Not Empty
Password Field	● Empty	○ Not Empty
User Count	● Equal to One	○ Not Equal to One
SUBMIT		

Figure 40: Selected options for Self Service Password application

For example, the options are selected as shown in Figure 40. For the given selection an example input can be Login: *sprice*; Password: *(empty)*

Boolean values are assigned to each entity (field) as shown in Figure 41. These values are based on the selection from the previous page. Also, our pre-condition for custom application for login bypass is [(a $\wedge$ b) $\wedge$ c] which should be satisfied for a successful operation.



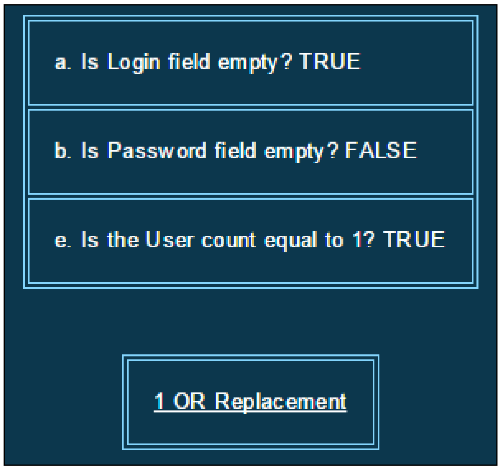
a. Is Login field empty? TRUE

b. Is Password field empty? FALSE

e. Is the User count equal to 1? TRUE

Figure 41: Assigned Boolean values based on selection

Once we have the Boolean values assigned depending on the given selection, the options available for logical OR Replacements are shown in Figure 42.



a. Is Login field empty? TRUE

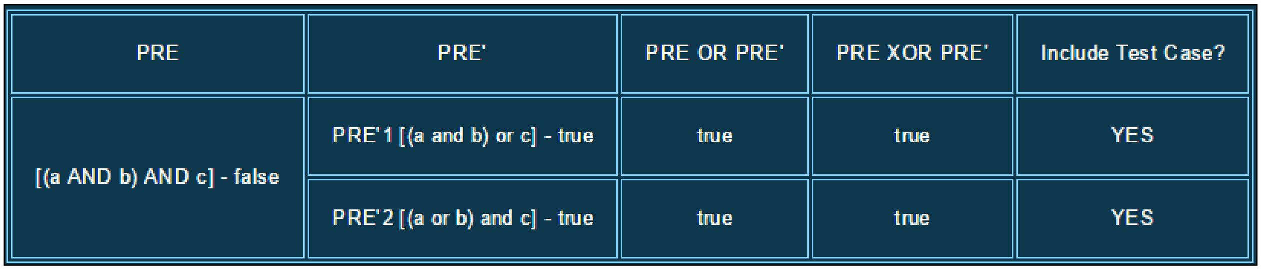
b. Is Password field empty? FALSE

e. Is the User count equal to 1? TRUE

Figure 42: Available OR replacement option

Let us check the selection of test cases for OR replacement option.

1 OR Replacement Result Set:



PRE	PRE'	PRE OR PRE'	PRE XOR PRE'	Include Test Case?
[(a AND b) AND c] - false	PRE'1 [(a and b) or c] - true	true	true	YES
	PRE'2 [(a or b) and c] - true	true	true	YES

Based on the last column of the 1 OR Replacement screenshot, all test cases are relevant to detect an attack, therefore are included in the test set *T*. This leads to a Mutation Score (MS) of 2/2 or 100%.

Thus, the developed tool helps in selection of test cases which might be vulnerable to injection attacks, thereby allowing developers to develop secure web applications.

In the next Chapter, we present the dissemination of our research results.

CHAPTER 7

Dissemination of Research Results

This chapter shows dissemination of thesis results as poster presentations and conference papers. Below we list the title, abstract, and venue for each dissemination.

Detection of Lightweight Directory Access Protocol Query Injection attacks in Web Applications

Pranahita Bulusu, Hossain Shahriar and Hisham Haddad.

Poster presentation. *Kennesaw State University - Computer Science Student Expo 2015,* Marietta, GA, USA, December 2015

Abstract

LDAP is a directory access protocol commonly used in web applications to provide lookup information and enforcing authentication mechanism. However, poorly implemented web applications suffer from LDAP injection vulnerabilities that might lead to security breaches such as login bypassing, privilege escalation, information disclosure, and information alteration. Testing for the presence of LDAP injection attacks can help to discover vulnerabilities early and fix implementation. Towards this direction, generating effective test cases is important and requires systematic approach. This paper proposes fault injection-based testing of LDAP injection attacks based on program implementation. We extract design level information and constraints (in the form of pre-conditions and post-conditions) highlighting behaviors that should be maintained throughout application runtime. We express the constraints using a popular modeling language called OCL. We randomly alter the captured pre-conditions and post-conditions and solve them to generate suitable test cases that may have the capability to check for the presence of LDAP injection vulnerabilities. We proposed needed algorithms to implement our test case generation approach. We did an initial case study for an open source PHP application. The analysis shows that our approach can generate effective test cases to discover LDAP injection vulnerabilities.

OCL Fault Injection-Based Testing of LDAP Query Injection Attacks

Pranahita Bulusu, Hossain Shahriar and Hisham Haddad.

Conference paper submission in progress.

Abstract

LDAP is a directory access protocol commonly used in web applications to provide lookup information and enforcing authentication mechanism. However, poorly implemented web applications suffer from LDAP injection vulnerabilities that might lead to security breaches such as login bypassing, privilege escalation, information disclosure, and information alteration. Testing for the presence of LDAP injection attacks can help to discover vulnerabilities early and fix implementation. Towards this direction, generating effective test cases is important and requires systematic approach. This paper proposes fault injection-based testing of LDAP injection attacks based on program implementation. We extract design level information and constraints (in the form of pre-conditions and post-conditions) highlighting behaviors that should be maintained throughout application runtime. We express the constraints using a popular modeling language called OCL. We randomly alter the captured pre-conditions and post-conditions and solve them to generate suitable test cases that may have the capability to check for the presence of LDAP injection vulnerabilities. We proposed needed algorithms to implement our test case generation approach. We did an initial case study for an open source PHP application. The analysis shows that our approach can generate effective test cases to discover LDAP injection vulnerabilities.

Classification of Lightweight Directory Access Protocol Query Injection Attacks and Mitigation Techniques

Pranahita Bulusu, Hossain Shahriar and Hisham Haddad.

Conference Proceedings. *Proceedings of 2015 International Conference on Collaboration Technologies and Systems (CTS 2015),* Atlanta, GA, USA, June 2015, IEEE CS Press, pp. 337-344, ISBN: 978-1-4673-7648-8/15

Abstract

The Lightweight Directory Access Protocol (LDAP) is used in a large number of web applications, and therefore, different types of LDAP injection attacks are becoming common. These injection attacks take advantage of an application not validating inputs before being used as part of LDAP queries. An attacker can provide inputs that may result in the alteration of intended LDAP query structure. The attacks can lead to various types of security breaches including Login Bypassing, Information Disclosure, Privilege Escalation, and Information Alteration. Despite many research efforts to prevent LDAP injection attacks, many web applications remain vulnerable to such attacks. In particular, there has been little attention given to implement and test secure web applications that can mitigate LDAP query injection attacks. More attention has been given to prevent Structured Query Language (SQL) injection attacks but these mitigation techniques cannot be directly applied in order to prevent LDAP injection attacks. This work provides analysis and classification of various types of LDAP injection attacks and mitigation techniques used to prevent them, and it highlights the differences between SQL and LDAP injection attacks.

OCL Fault-Injection Based Testing of LDAP Query Injection Attacks

Pranahita Bulusu, Hossain Shahriar and Hisham Haddad.

Poster presentation. *Kennesaw State University* - *Computer Science Student Expo 2015,* Kennesaw, GA, USA, April 2015

Abstract

LDAP is a popular protocol for Directory Service. Data objects are represented in hierarchical form. Web applications relying on LDAP-based data object storing and retrieval may suffer from injection attacks due to lack or improper input validation. Common LDAP injection attacks include (i) Login Bypassing, (ii) Information Disclosure, (iii) Privilege Escalation, and (iv) Information Alteration.

Lightweight Directory Access Protocol Query Injection Attacks in Web Applications

Pranahita Bulusu, Hossain Shahriar and Hisham Haddad.

Poster presentation. *Kennesaw State University* - *Computer Science Student Expo 2014*, Kennesaw, GA, USA, December 2014

Abstract

LDAP is a popular protocol for Directory Service. Data objects are represented in hierarchical form. Web applications relying on LDAP-based data object storing and retrieval may suffer from injection attacks due to lack or improper input validation. Common LDAP injection attacks include (i) Login Bypassing, (ii) Information Disclosure, (iii) Privilege Escalation, and (iv) Information Alteration.

CHAPTER 8

Conclusions and Future Work

8.1 Conclusions

LDAP code injection vulnerability can be exploited to perform security breaches in web applications such as login bypassing and privilege escalation. Among well-known code injection attacks, LDAP injection has been least addressed and they do not share much similarities with other types of injection attacks *(e.g., SQL Injection)* [6]. We proposed OCL fault injection based testing approach to generate LDAP injection vulnerability revealing test cases. We extracted design level information and constraints (in the form of pre and post-conditions) highlighting behaviors that should be maintained throughout application runtime. We expressed the constraints using a popular modeling language called OCL. We evaluated our approach with two PHP web applications.

From the extensive survey we have done (Chapter 2), we find that most literature works are intended for other common code injection attacks, but not specifically for LDAP. We find that LDAP and SQL query have dissimilarities and attack inputs and contexts are also dissimilar. Further, few literature works have mostly focused on login bypass type of attacks, leaving the other three attacks types (information disclosure, privilege escalation, information alteration) unaddressed.

We have proposed two algorithms in Chapter 4. The first algorithm addresses the generation of pre and post-conditions from application source code where design level information is missing. The outcome of the algorithm is a set of combined pre-conditions for various program paths. The second algorithm then alters the pre-conditions with the goal of generating and selecting effective test cases that can detect the presence of LDAP query injection vulnerabilities.

Chapter 5 illustrates the application of the algorithms for two PHP web applications, including one having reported vulnerabilities (login bypass and information disclosure) based on OSVDB. We built a custom web application to validate our approach for login bypass, privilege escalation and information alteration type attacks. The evaluation indicates that our approach can generate suitable test cases having specific inputs capable of revealing LDAP injection vulnerabilities.

Chapter 6 discusses the implementation of a tool to automate the generation of altered pre conditions so that developers can assess if a given input is vulnerable to LDAP injection attacks. The tools can support the replacement of one logical operator with another, and in multiple locations of a given pre-condition. Developers can integrate our proposed OCL fault injection based approach to detect LDAP query injection attacks.

The proposed approach is targeted for application developers to be able to generate test sets with high mutation score so that included test cases can detect LDAP injection attacks with high probability.

8.2 Future Work

Future work includes applying OCL fault-injection based testing approach to more web applications and to test other types of code injection vulnerabilities. We plan to generalize our implemented tool's input taking mechanism so that users can specify their input fields and constraints to generate pre-conditions. We like to automate the generation of initial test input and alter pre-conditions for relational operators. Currently, our approach does not automatically extract class design level information from source code. We plan to develop or employ suitable tools for extracting design level information.

Appendix A: Source Code

Index.php

```
<center>
      <h1>Log in to LDAP Server</h1>
      <form action="login_action.php" method="post">
      <input type="text" name="userid"/>
      <input type="password" name="password"/>
      <input type="submit" name="submit" />
      </form>
</center>
```

Login_action.php

```
<?php
set_time_limit(30);
error_reporting(E_ALL);
ini_set('error_reporting', E_ALL);
ini_set('display_errors',1);

// config
$ldapserver = '192.168.1.124';
$ldapuser      = $_POST['userid'];
$ldappass     = $_POST['password'];
$ldaptree    = "dc=ubuntuldap2";


if (empty($ldapuser) || empty($ldappass)) {
      echo "Please enter the login credentials. <a 
href='javascript:history.back()'>Back</a>";
} else {
// connect
$ldapconn = ldap_connect($ldapserver) or die("Could not connect to LDAP 
server.");
ldap_set_option($ldapconn, LDAP_OPT_PROTOCOL_VERSION, 3);


if($ldapconn) {
    // binding to ldap server
    $ldapbind = ldap_bind($ldapconn, "cn=admin,".$ldaptree, "admin") or 
die ("Please enter valid login credentials. <a 
href='javascript:history.back()'>Back </a>");
    // verify binding
    if ($ldapbind) {
```

```
$search = "(&(&(uid=".$ldapuser.")(userPassword=".$ldappass.")))";
$sr=ldap_search($ldapconn, $ldaptree, $search);
$info = ldap_get_entries($ldapconn, $sr);
if ($info["count"] > 0) {
            echo "<center>";
            echo "You are logged in as: ".$ldapuser." <a
href='logout.php'>Logout</a><br>";
            echo "<h2>Your details</h2>";
            $ii=0;
                  for ($i=0; $ii<$info[$i]["count"]; $ii++){
                      $data = $info[$i][$ii];
                      echo
$data.":&nbsp;&nbsp;".$info[$i][$data][0]."<br>";
                  }

                  $ou = $info[0]["ou"][0];
                  $sec_level = $info[0]["description"][0];


                  echo "<table border=1>";
                  echo "<th colspan=2><h2>Main Menu</h2></th>";
                  echo "<tr><td><a
href='searchusers.php?ou=".$ou."'>Search for Users</a></td></tr>";
                  echo "<tr><td><a
href='documents.php?uid=".$ldapuser."&ou=".$ou."&securitylevel=".$sec_lev
el."'>Available Documents</a></td></tr>";
                  echo "<tr><td><a
href='add.php?uid=".$ldapuser."&ou=".$ou."'>Add User</a></td></tr>";
                  echo "<tr><td><a
href='replace.php?uid=".$ldapuser."&ou=".$ou."'>Replace</a></td></tr>";
                  echo "<tr><td><a
href='modify.php?uid=".$ldapuser."&ou=".$ou."'>Update</a></td></tr>";
                  echo "<tr><td><a
href='delete.php?uid=".$ldapuser."&ou=".$ou."'>Delete</a></td></tr>";
                  echo "</table>";
                  echo "</center>";

            } else {
            echo "<center>";
            echo "Please check your ID and Password";
            echo "</center>";
            }
      }
        }
// all done? clean up
ldap_close($ldapconn);
}
?>
```

Replace.php

```
<?php
set_time_limit(30);
error_reporting(E_ALL);
ini_set('error_reporting', E_ALL);
ini_set('display_errors',1);
// config
$ldapserver = '192.168.1.124';
$ldaptree    = "dc=ubuntuldap2";
// connect
$ldapconn = ldap_connect($ldapserver) or die("Could not connect to LDAP
server.");
ldap_set_option($ldapconn, LDAP_OPT_PROTOCOL_VERSION, 3);
if($ldapconn) {
    // binding to ldap server
    $ldapbind = ldap_bind($ldapconn, "cn=admin,dc=ubuntuldap2", "admin")
or die ("Please enter valid login credentials. <a
href='javascript:history.back()'>Back </a>");
    // verify binding
    if ($ldapbind) {
        echo "<center>";
      echo "<h2>Enter the Details</h2>";
            echo "<form action=replace_action.php method=post>";
            echo "<table border=1>";
            echo "<tr><td>ObjectClass</td><td><select
name=objectclass><option value=user>User</option><option
value=document>Document</option></select></td></tr>";
            echo "<tr><td>Replacing Object</td><td><select
name=replaceobj><option value=givenname>givenName</option><option
value=description>Description(If Document, paste
link)</option></select></td></tr>";
            echo "<tr><td>Replace With</td><td><input type=text
name=replace></td></tr>";
            echo "<tr><td>DN</td><td><input type=text
name=dn></td></tr>";
            echo "<tr><td colspan=2 align=center><input type=hidden
name=ou value=".$_GET['ou']."></td></tr>";
            echo "<tr><td colspan=2 align=center><input type=hidden
name=uid value=".$_GET['uid']."></td></tr>";
            echo "<tr><td colspan=2 align=center><input type=submit
name=submit value=Insert></td></tr>";
            echo "</table>";
            echo "</form>";
      echo "</center>";
      }
// all done? clean up
ldap_close($ldapconn);
}
?>
```

Replace_action.php

```
<?php
set_time_limit(30);
error_reporting(E_ALL);
ini_set('error_reporting', E_ALL);
ini_set('display_errors',1);

// config
$ldapserver = '192.168.1.124';
$ldaptree    = "dc=ubuntuldap2";

// connect
$ldapconn = ldap_connect($ldapserver) or die("Could not connect to LDAP
server.");
ldap_set_option($ldapconn, LDAP_OPT_PROTOCOL_VERSION, 3);
ldap_set_option(NULL, LDAP_OPT_DEBUG_LEVEL, 7);
ldap_set_option($ldapconn, LDAP_OPT_REFERRALS, 0);


if($ldapconn) {
    // binding to ldap server
    $ldapbind = ldap_bind($ldapconn, "cn=admin,dc=ubuntuldap2", "admin")
or die ("Please enter valid login credentials. <a
href='javascript:history.back()'>Back </a>");
    // verify binding
    if ($ldapbind) {
        $oc = $_POST['objectclass'];
            $uid = $_POST['uid'];
            $dn = $_POST['dn'];
            $oc = $_POST['objectclass'];
            $replaceobj = $_POST['replaceobj'];
            $replace = $_POST['replace'];
            $attr["$replaceobj"] = $replace;
            $result = ldap_mod_replace($ldapconn,$dn, $attr);
            if (TRUE === $result) {
                   echo "Entry was replaced.";
                   }
                   else {
                        echo "Entry cannot be replaced.";
                   }
      }
// all done? clean up
ldap_close($ldapconn);
}
?>
```

References

[1] Introduction to OpenLDAP Directory Services, Accessed from http://www.openldap.org/doc/admin24/intro.html

[2] LDAP open source guide, Accessed from http://www.zytrax.com/books/ldap/ch2/index.html#history

[3] Open Web Application Security Project (OWASP), LDAP Injection, https://www.owasp.org/index.php/LDAP_injection

[4] David Hoyt, *LDAP Injection Vulnerability in SmarterMail,* http://www.exploit-db.com/exploits/15189/

[5] Vulnerable Applications for LDAP Injection – http://security.stackexchange.com/questions/23032/vuln-web-app-which-includes-ldap-injection

[6] Open Web Application Security Project (OWASP), SQL Injection, https://www.owasp.org/index.php/SQL_Injection

[7] Open Web Application Security Project (OWASP) Top Ten 2013, https://www.owasp.org/index.php/Top_10_2013-Top_10

[8] M. Kumar and L. Indu, "Detection and Prevention of SQL Injection attack," *Proceedings of the International Journal of Computer Science and Information Technologies, Vol. 5 (1) (IJCSIT),* P. B. College of Engineering, Sriperumbudur, pages 374–377

[9] A. Liu, Yi Yuan, D. Wijesekera, and A. Stavrou, "SQLProb: A Proxy-based Architecture towards Preventing SQL Injection Attacks," *Proceedings of the ACM/SIGAPP Symposium On Applied Computing (SAC),* Honolulu, Hawaii, USA, March 2009, pages 2054–2061

[10] H. Shahriar and M. Zulkernine, "MUSIC: Mutation-based SQL injection vulnerability checking," *Proceedings of IEEE Eighth International Conference on Quality Software (QSIC),* London, UK, August 2008, pages 77–86

[11] H. Shahriar and M. Zulkernine, "Information-Theoretic Detection of SQL Injection Attacks," *Proceedings of IEEE 14th International Symposium on High-Assurance Systems Engineering (HASE),* Omaha, NE, USA, October 2012, pages 40–47

[12] G. Wassermann and Z. Su, "An Analysis Framework for Security in Web Applications," *Proceedings of the FSE Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004)*, 2004,pages 70–78

[13] N. DuPaul, LDAP Injection Guide, *Veracode,* https://www.veracode.com/ldap-injection?mkt_tok=3RkMMJWWfF9wsRoiu6rfLqzsmxzEJ8zx7eUtWbHr08Yy0EZ5VunJEUWy3YYCWoEnZ9mMBAQZC813xR5ZGe%2BReQ%3D%3D

[14] E. Guillardoy, F. Guzman, and H. Abbamonte, "LDAP injection Attack and Defense Techniques", *HITB Magazine,* http://magazine.hitb.org/issues/HITB-Ezine-Issue-001.pdf

[15] S. Faust, "LDAP Injection: Are Your Applications Vulnerable?", *SPI Labs,* http://www.networkdls.com/articles/ldapinjection.pdf

[16] Testing for LDAP Injection, OWASP – https://www.owasp.org/index.php/Testing_for_LDAP_Injection_(OTG-INPVAL-006)

[17] M. Hafiz and R. Johnson, "Improving Perimeter Security with Security-oriented Program Transformations," *Proceedings of the IEEE Software Engineering for Secure Systems (SESS),* Vancouver, Canada, May 2009, pages 61–67

[18] M. Hafiz, "Security oriented program transformations (Or how to add security on demand)," *OOPSLA '08: Companion to the 23rd annual ACM Special Interest Group on Programming Languages (SIGPLAN)conference on Object oriented programming, systems, languages, and applications,* New York, NY, USA, 2004

[19] J. Xie, B. Chu, H. Lipford, and J. Melton, "ASIDE: IDE Support for Web Application Security," *ACM Annual Computer Security Applications Conference (ACSAC)*, Orlando, Florida USA, December 2011, pages 267–276

[20] Y. Zheng and X. Zhang, "Path Sensitive Static Analysis of Web Applications for Remote Code Execution Vulnerability Detection," *IEEE International Conference on Software Engineering (ICSE),* San Francisco, CA, USA, 2013, pages 652–661

[21] M. Almorsy, J. Grundy, and A. Ibrahim, "Supporting Automated Vulnerability Analysis Using Formalized Vulnerability Signatures," *ACM Automated Software Engineering (ASE) conference*, Essen, Germany, September 2012, pages 100–109

[22] OpenLDAP Security Considerations – http://www.openldap.org/doc/admin24/security.html

[23] Escaping special characters in LDAP search filters, Accessed from http://blog.dzhuvinov.com/?p=585

[24] M. Kumar and L. Indu, "Detection and Prevention of SQL Injection attack," International Journal of Computer Science and Information Technologies (IJCSIT), , Vol. 5 (1) pages 374–377

[25] D. Venturin, Prevention of brute force attacks in LDAP, January 2013 – http://blog.squadrainformatica.com/blog/2013/01/ldap-how-to-prevent-brute force-attacks

[26] Oracle Document, Interface PreparedStatement, http://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html

[27] SQL Injection example, http://security.stackexchange.com/questions/34655/is-there-any-sql-injection-for-this-php-login-example

[28] IBM Knowledge Center, Injection Attacks, http://pic.dhe.ibm.com/infocenter/sprotect/v2r8m0/index.jsp?topic=%2Fcom.ibm.ips.doc%2Fconcepts%2Fwap_injection_attacks.htm

[29] Introduction to Lightweight Directory Access Protocol (LDAP), Article ID: 196455, http://support.microsoft.com/kb/196455

[30] An introduction to LDAP, Accessed from http://ldapman.org/articles/intro_to_ldap.html#security

[31] LDAP Connect User's Guide, Novell documentation, Accessed from http://www.novell.com/documentation/extend5/Docs/help/Composer/books/LDAPWelcome.html

[32] B.K. Aichernig, P.A. Pari Salas, "Test Case Generation by OCL Mutation and Constraint Solving," *Proeedings of IEEE 5th International Conference on Quality Software (QSIC)*, 2005

[33] J. Fonseca, M. Vieira, and H. Madeira, "Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks," *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, Australia, December 2007, pages 365–372

[34] G. Vigna, W. Robertson, D. Balzarotti, "Testing Network-based Intrusion Detection Signature Using Mutant Exploits," *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*, Washington DC, USA, October 2004, pages 21–30

[35] O. Tal, S. Knight, and T. Dean, "Syntax-based Vulnerabilities Testing of Frame-based Network Protocols," *Proceedings of the 2nd Annual Conference on Privacy, Security and Trust*, Fredericton, October 2004, pages 155–160

[36] A. Kieżun, P. Guo, K. Jayaraman, and M. Ernst, "Automatic creation of SQL injection and cross-site scripting attacks, " *MIT Computer Science and Artificial Intelligence Laboratory technical report*, MIT-CSAIL-TR-2008-054, Cambridge, MA, USA, September 2008

[37] S. Ghosh and J. Kelly, "Byte code fault injection for Java Software," *Journal of System and Software*, Vol. 81, Issue 11, November 2008, pages 2034–2043

[38] P. Fouque, D. Leresteux, and F. Valette, “Using Faults for Buffer Overflow Effects,” *Proeedings of ACM Symposium of Applied Computing (SAC)*, Riva Del Grada, Italy, March 2012, pages 1638–1639

[39] J. Voas, “Assessing Survivality using Software Fault Injection System,” *Technical Report (ADP 010875) from Defense Technical Information Center*, 2000

[40] R. A. Oliveira, N. Laranjeiro, M. Vieira, “Characterizing the Performance of Web Service Frameworks under Security Attacks,” *Proceedings of the ACM/SIGAPP Symposium On Applied Computing (SAC),* Salamanca, Spain, April 2015, pages 1711–1718

[41] P. Salas, Krishnan, K.J Ross, “Model-Based Security Vulnerability Testing,” *Proceedings of Australian Software Engineering Conference*, Australia, 2007, pages 284–296

[42] D. Grela, K. Sapiecha, J Strug, “A Fault Injection Based Approach to Assessment of Quality of Test Sets for BPEL proceeses,” *Proceedings of the International Conference on Evaluation of Novel Approaches of Software Engineering (ENASE)*, France, July 2015, pages 81–93

[43] J. Cabot and M. Gogolla, “Object Constraint Language (OCL): a Definitive Guide,” *Formal Methods for Model-Driven Engineering*, Volume 7320. Lecture Notes in Computer Science, 2012, pages 58–90

[44] J. Warmer and A. Kleppe. The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley, 2003

[45] M. Stock, “Automatic Generation of Junit Test-Harnesses, MSc Thesis,” ETH Zurich, March 2007, Accessed from http://ms.stradax.net/Publications/junit-testgen/junit-testgen.pdf

[46] Self Service Password web application, Accessed from http://tools.ltb-project.org/issues/391

[47] Self Service Password Unspecified LDAP Query Injection, Accessed from http://osvdb.org/show/osvdb/86564

[48] Web interface phpLDAPadmin to manage LDAP directory, Accessed from http://sourceforge.net/projects/phpldapadmin/

[49] S. Nica, "On the Improvement of the Mutation Score Using Distinguishing Test Cases," *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2011, pages 423–426