



Detection of injection exploitation by validating query, function and parameter's integrity

Abhishek Singh, Ramesh Mani, Anjan Venkatramani, Chih-Wei Chao

Table of Content

1.0 Introduction	2
2.0 Technical Details of OS Command Injection	3
2.1 Detection of OS Command Injection Exploitation	4
2.2 Introduction to Web Shells	5
2.2.1 Technical Details	5
2.2.3 Detection of Web shells	6
3.0 Technical Details of SQL Injection Attacks	7
3.2 Detection of SQL injection Exploitation	10
4.0 Technical Details of NoSQL Injection Exploitation	11
4.1 Detection of NoSQL Exploitation.	13
5.0 Technical Details of XPath Injection Exploitation	14
5.1 Detection of XPath Injection Exploitation	15
6.0 Technical details of LDAP injection Exploitation.	15
6.1 Detection of LDAP Exploitation	17
7.0 Conclusion	18
References:	19

1.0 Introduction

Web injection exploitation has ruled as the top web application vulnerability for a decade. Injection flaws [1] include SQL, NoSQL, OS command and LDAP injection techniques. Exploitation occurs when untrusted data is sent to an interpreter as part of a command or query. OS command injection is an exploitation technique in which the goal is the execution of arbitrary commands on the host operating system via a vulnerable application. OS command injection exploitation is operating system-independent, i.e. it can happen on Windows or Linux.

In the case of SQL injection exploitation, malicious SQL statements are inserted via the methods which accept a user's or external inputs such as GET, POST, HTTP headers, lambda functions, etc. These malicious SQL queries are executed and perform malicious activities such as dumping the database contents, etc. to a location determined by the attacker. Threat actor groups such as Axiom [2] and Magic Hound [3] have been observed using SQL injection to gain access to systems. Injection attacks such as SQL injection exploitation can lead not only to malicious access to the database but also to the installation of malicious code such as web shells [4]. Databases such as MongoDB support the use of JavaScript functions for query specifications and map/reduce operations. Since *MongoDB* databases (like other NoSQL databases) do not have strictly defined database schemas, using JavaScript for query syntax allows developers to write arbitrarily complex queries against disparate document structures. If these queries accept input from HTTP methods and headers such as GET, POST, Cookies, User-Agent, etc. then these JavaScript functions can be prone to NoSQL injection exploitation [5]. Besides SQL and NoSQL databases, threat actors can also construct a malicious XPath expression or XQuery to retrieve the data from an XML database.

Injection attacks such as SQL, NoSQL and OS command injection exploits add additional code, which leads to a change in the legitimate code of the application. This change in the legitimate code is reflected in the abstract syntax tree (AST), program dependency graph (PDG) and SQL parse tree. In the first section of this paper we will take examples of SQL, NoSQL and OS command injection exploits and show the changes in the AST, PDG and SQL parse tree due to the exploits. These changes in the code are the fundamental principle of the detection algorithms used to detect SQL, NoSQL and OS command injection, which are discussed in the subsequent sections.

2.0 Technical Details of OS Command Injection

Figure 1.0 shows the code vulnerable to OS Command Injection attack.

```
<?php
print("Please specify the name of the file to delete");
print("<p>");
$file=$_GET['filename'];
system("rm $file");
?>
```

Figure 1.0 Code Vulnerable to OS Command Injection Exploit.

The code takes the name of the file which is to be deleted from the value field of the method GET. In the subsequent line of code PHP program execution function `system()`, takes the value of the file name from the data field of GET method and invokes the bash command “`rm`” to delete the file.

Let's assume that the file shown in figure 1.0 is on the server with IP address `x.x.x.x` as `test.php`. “`hxxp://x.x.x.x:8000/test.php?filename=kral.php`” is an example of the legitimate input. It takes in input as “`kral.php`” from the data field of the GET and deletes the file “`kral.php`” from the folder. The dynamic call graph which captures the execution flow i.e. control and data dependency is shown in figure 2.0. This dynamic call graph captures the control and data flow not only when the code is executed by a PHP application but also when the OS commands are executed by the PHP application. The nodes in the graph denote the statement, predicates, OS events, and system calls. The edges denote the parent-child relationship.

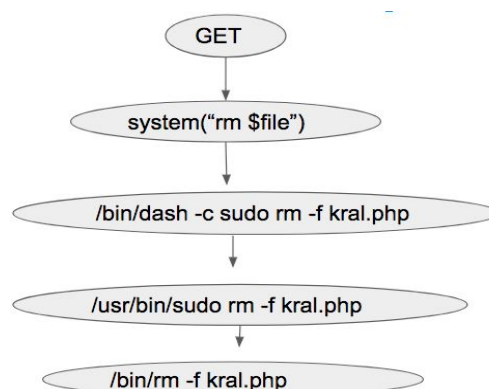


Figure 2.0 Dynamic Call Graph of the legitimate request.

"`http://x.x.x.x:8000/test.php?filename=kral.php;ls`" is an example of an OS command injection exploit. In the request, the threat actor has injected the OS command "`ls`" with the value of the GET method "`kral.php`".

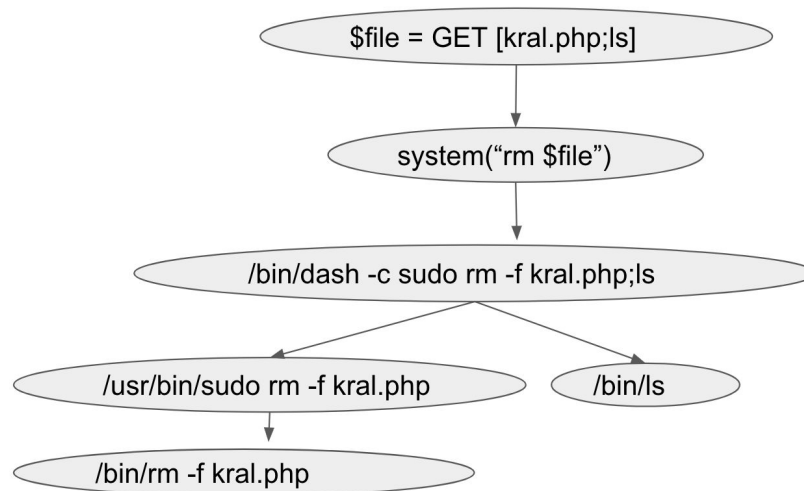


Figure 3.0 Dynamic Call graph having OS Command Injection Exploit.

Figure 3.0 shows the dynamic call graph when the exploit is sent to the vulnerable application. As can be seen, a new node having the value "`/bin/ls`", the injected OS command, is spawned. The value of the additional node can be traced up the graph and is the same as the command "`ls`" to the value field of the GET method.

2.1 Detection of OS Command Injection Exploitation

The algorithm to detect the OS command injection attack not only makes use of the application-level hooks but also captures system calls and other OS events to generate the dynamic call graph. As a part of the first step, the dynamic call graph traces the data passed to those methods that accepts user or external inputs such as GET, POST, Cookies, HTTP Headers etc. The subsequent child processes which can be OS events or invocation of the system calls spawned due to the execution of a program execution function such as `eval()`, `passthru()`, `shell_exec()`, `system()`, `proc_open()`, `expect_open()` or `ssh2_exec()` gets captured in the dynamic call graph.

Once the dynamic call graph has been generated, the algorithm uses the graph to check, whether the exact name of the command line argument of any of the child process of the program execution functions that are being spawned by the OS command is same as the value or appears in the part of the value passed to the methods that accepts user inputs such as GET, POST, Cookies & HTTP headers. If the condition is found to be true, an alert of OS command injection vulnerability is raised. The algorithm is independent both of the application which is being executed, and of the operating system on which the algorithm is running, hence it

helps to identify both known and zero-day OS command injection vulnerabilities. The algorithm can not only be used to detect the OS command injection vulnerabilities, but it is also effective to detect web shells. In the next subsection, we share the details of detection of the web shell.

2.2 Introduction to Web Shells

Web shells are one of the critical malware which has extensively been used by threat actors. Web applications exploits or configuration weakness including Cross-Site Scripting, SQL injection, Vulnerability in applications/services, File processing Vulnerabilities, Remote File Inclusion (RFI) and Local File Include (LFI) vulnerabilities [14] can be used to deliver web shells. Web shells act as a remote access tool for a threat actor. Once a Web Shell has been installed on a web server, it will allow a threat actor to launch the next stage of attack such as execute commands, upload malware. APT32, APT34, Deep Panda, Dragonfly, OilRigs are some of the threat actors who have made use of Web shells for persistence and privilege escalations [13]. The web shell can be PHP, ASP, JSP, Perl, Ruby, Python applications. In this document, we have taken the PHP based Web shell, explained the execution flow of code and then discussed the solution to detect it. The same methodology can be applied to the non-PHP based web shells as well.

2.2.1 Technical Details

Figure 2.1 shows the code of pws.php web shell. As shown in figure 2.1, the code accepts the input from the threat actor via the data field of the `POST` method. The data field then acts as an input to the `passthru()` PHP function which executes the input from a threat actor

```
<?php
if ($_POST['cmd']){
    $cmd = $_POST['cmd'];
    passthru($cmd);
}
?>
```

Figure 2.1 showing code of PHP WebShell.

Similar to the `POST` method, web shells have also made use of the `GET` method. `<?php eval($_GET['c']);>` is the code of another web shell, which will take the input from threat actor via the data field of the `GET` method and executes the input by making a call to `eval()`. `Eval()`, `passthru()`, `shell_exec()`, `system()` are some of the PHP API which have been used by web shells to execute malicious commands sent via the data field of `GET` and `POST` methods. Besides the execution of malicious commands, Web Shells are also

used for uploading files to the web server, sending SQL queries to the Database servers or used as a relay to communicate with the host in the internal networks.

2.2.3 Detection of Web shells

The algorithm to detect web shell leverages application level hooks to trace the data passed to the methods such as `GET`, `POST`. This tracing of data is used to generate a program dependency graph (PDG) which captures the data and control dependencies amongst the statements and predicates. Once the PDG is created, it is used to trace if the data passed to the methods such as `GET`, `POST` are the exact direct parameters to the program execution functions such as `eval()`, `passthru()`, `shell_exec()`, `system()`, `proc_open()`. If the condition is found to be true, then the process is marked as malicious.

The algorithm provides the following inherent advantages.

- It is independent of the first stage of exploitation or delivery vector which leads to the installation of the web shells.
- It is independent of the later stages of exploitation such as defensive evasion, credential access, discovery, lateral movement, collection, exfiltration, command, and control phases.
- It is independent of the signatures of the past web shells [15].

These factors make it a recommended solution to prevent web shells.

3.0 Technical Details of SQL Injection Attacks

Figure 4.0 is an example of code that is vulnerable to SQL injection attack. The code accepts the `username` from the HTTP request in the variable `user`. This variable `user` then acts as an input to the SQL query `sqlstmt`. In the subsequent part of the code, the query gets executed by the function `Connection.Execute()`.

```
user = Request.Form("username")
Set Connection = Server.CreateObject("ADODB.Connection")
Connection.Open "DSN=testdsn;UID=xxx"
sqlstmt = "SELECT Host FROM mysql.user WHERE User ='" + user + "'"
Set rs = Connection.Execute(sqlstmt)
```

Figure 4.0 Code Vulnerable to SQL Injection

"SELECT Host FROM mysql.user WHERE User = 'abhi';", is an example of the legitimate SQL query executed by the application. The legitimate query sent to the database is defined in the application and the value of the variable `user` as 'abhi' is taken from the value field of the methods which accept user input such as the GET, POST, etc.. Figure 5.0 shows the PostgreSQL parse tree[7] of the normalized query.


```

<PgQuery:0x0000559530ca0280
@aliases=nil,
@cte_names=nil,
@query="SELECT Host FROM mysql.user WHERE User=$1;",
@tables=nil,
@tree=
[{"RawStmt"=>
  {"stmt"=>
    {"SelectStmt"=>
      {"targetList"=>
        [{"ResTarget"=>
          {"val"=>
            {"ColumnRef"=>
              {"fields"=>[{"String"=>{"str"=>"host"}]}, "location"=>7}},
            {"location"=>7}}],
          {"fromClause"=>
            [{"RangeVar"=>
              {"schemaname"=>"mysql",
               "relname"=>"user",
               "inh"=>true,
               "relpersistence"=>"p",
               "location"=>17}}],
            {"whereClause"=>
              {"A_Expr"=>
                {"kind"=>0,
                 "name"=>[{"String"=>{"str"=>""}}],
                 "lexpr"=>
                  {"SQLValueFunction"=>{"op"=>11, "typmod"=>-1, "location"=>34}},
                 "rexpr"=>{"ParamRef"=>{"number"=>1, "location"=>39}},
                 "location"=>38}},
                {"op"=>0}},
              {"stmt len"=>41}}],
          {"warnings"=>[]}>
<PgQuery:0x0000559530bba6a0

```

Figure 5.0 PostgreSQL parsetree of the valid normalized Query

In the case of an SQL injection exploitation, a threat actor will insert a malicious SQL query along with a value into the methods that accept user or external inputs (such as GET, POST). This malicious query will get executed by the application along with the legitimate query. Explaining it with the example, if the threat actor injects malicious SQL query *"UNION SELECT username, Password FROM Users"*, by appending it to the value of the variable user *"abhi"*, the query that gets executed by the application becomes *"SELECT Host FROM mysql.user WHERE user= 'abhi' UNION SELECT username, Password FROM Users;"*

A PostgreSQL parse tree of the normalized SQL query with the injected SQL query *"UNION SELECT username, Password FROM Users"* is shown in Figure 6.0. As can be seen in Figure 6.0 an additional node is created in the PostgreSQL parse tree as a result of the inserted SQL command.

```

"SELECT Host FROM mysql.user WHERE user=$1 UNION SELECT Username, Password FROM Users;",
@tables=nil,
@tree=
[{"RawStmt"=>
  {"stmt"=>
    {"SelectStmt"=>
      {"op"=>1,
       "larg"=>
        {"SelectStmt"=>
          {"targetList"=>
            [{"ResTarget"=>
              {"val"=>
                {"ColumnRef"=>
                  {"fields"=>[{"String"=>{"str"=>"host"}]},
                  "location"=>7}},
                "location"=>7}}],
          "fromClause"=>
            [{"RangeVar"=>
              {"schemaname"=>"mysql",
               "relname"=>"user",
               "inh"=>true,
               "relpersistence"=>"p",
               "location"=>17}}],
          "whereClause"=>
            {"A Expr"=>
              {"kind"=>0,
               "name"=>[{"String"=>{"str"=>"="}}],
               "lexpr"=>
                {"SQLValueFunction"=>
                  {"op"=>11, "typmod"=>-1, "location"=>34}},
               "rexpr"=>{"ParamRef"=>{"number"=>1, "location"=>39}},
               "location"=>38}},
              "op"=>0}},
            "rarg"=>
              {"SelectStmt"=>
                {"targetList"=>
                  [{"ResTarget"=>
                    {"val"=>
                      {"ColumnRef"=>
                        {"fields"=>[{"String"=>{"str"=>"username"}]},
                        "location"=>55}},
                      "location"=>55}},
                    {"ResTarget"=>
                      {"val"=>
                        {"ColumnRef"=>
                          {"fields"=>[{"String"=>{"str"=>"password"}]},
                          "location"=>65}},
                        "location"=>65}}],
                  "fromClause"=>
                    [{"RangeVar"=>
                      {"relname"=>"users",
                       "inh"=>true,
                       "relpersistence"=>"p",
                       "location"=>79}}],
                  "op"=>0}}}],
                "stmt_len"=>84}}],
@warnings=[]>

```

Figure 6.0 PostgreSQL parse tree with the Injected SQL command

SQL injection exploits will introduce an additional SQL query to the database. This injected SQL query will be reflected as an additional nodes in the query parse tree of the legitimate normalized SQL query.

3.2 Detection of SQL injection Exploitation

The algorithm to detect the SQL injection attack makes use of the application-level hooks to construct a program dependency graph (PDG). The PDG captures the flow of data and control from methods which accept external or user input such as GET, POST, Cookies etc. to the functions which execute the SQL query such as `mysql_query()`, `mysql_db_query()`, `mysql_unbuffered_query()`, `pg_execute()`, `pg_query()`, `pg_query_params()`, `pg_prepare()`, `pg_send_query()`, and `pg_send_query_params()`. Once the PDG is generated, it is used to identify the legitimate SQL queries which are the sink for the value from the methods which accepts user's or external input such as GET, POST etc. These SQL queries are the parameters for the SQL query execution functions and are stored as the parse tree. Any injected malicious SQL query by a threat actor to the legitimate SQL query will get reflected by the additional node in the parse tree of the normalized legitimate SQL query. For every database access, the parse tree of the executing query is compared with the parse tree of the legitimate query. If there is any additional node in the parse tree of the executing SQL query, then it gets matched with the value or with the part of the value passed to the methods that accept user or external input such as GET, POST etc. In the case of a match, an alert for SQL injection is raised.

4.0 Technical Details of NoSQL Injection Exploitation

A NoSQL database provides the ability to run JavaScript in the database engine to perform complicated queries or transactions. Figure 7.0 shows an example of a JavaScript query. The query accepts external input in the variable `search_year` and then searches in the database for all the publications that match the values of the search year.

```
var query = {  
  $where: function()  
  {  
    var search_year = input_value;  
    return this.publicstionYear = search_year;  
  }  
}
```

Figure 7.0 JavaScript query.

A legitimate GET request will appear in the format `http://server.com/app.php?year=2015`. The input '2015' will act as an input value for the variable `search_year`, and the query will return the publications which were published in the year 2015. An example of a NoSQL injection attacks is `http://server.com/app.php?year=2015;while(true){}`. The exploit injects the NoSQL code `"while(true){}"`. This injected code will be executed by the database, and it will cause the database to enter into an infinite loop leading, leading to a denial of service attack. If we construct an abstract syntax tree[6] of the legitimate javascript function, shown in figure 7.0 and abstract syntax tree of the function with injected NoSQL code and compare them, as can be seen in figure 8.0, NoSQL injection leads to the addition of new node.

```
    },  
    {  
      "type": "WhileStatement",  
      "test": {  
        "type": "Literal",  
        "value": true,  
        "raw": "true"  
      },  
      "body": {  
        "type": "BlockStatement",  
        "body": []  
      }  
    }  
  ],  
}
```

Figure 8.0 Addition in the AST of the legitimate JS function

Besides using JavaScript to draft queries, JSON format is also used to send queries to the NoSQL databases and is prone to NoSQL injection exploitation. Figure 9.0 shows an example

of the vulnerable code[10] which crafts a query in JSON format. The code accepts the value of variables `username` and `passwords` from the POST method, and the query is sent to the database by invoking `findOne` function.

```
app.post('/user', function (req, res) {
    var query = {
        username: req.body.username,
        password: req.body.password
    }

    db.collection('users').findOne(query, function (err, user) {
        console.log(user);
    });
})
```

Figure 9.0 code vulnerable to NoSQL injection Exploitation.

If a user enters “admin” as the value for the variable `username` and “password” as the value for variable `password`, the JSON query which is sent to the database will be as shown in the figure 10.0

```
{
  "username": "admin",
  "password": "password"
}
```

Figure 10.0 JSON query sent to the database.

AST of the legitimate JSON query is shown in Figure 11.0. AST of the legitimate JSON query for this example will always have two members.

```
AST Dump (all):
object (prolog: "{", epilg: "\n") [1,1]
├── member (epilog: ",\n") [2,9]
│   ├── string (body: "\"username\"", value: "username", epilg: "\"") [2,9]
│   └── string (body: "\"admin\"", value: "admin") [2,21]
└── member [3,9]
    ├── string (body: "\"password\"", value: "password", epilg: "\"") [3,9]
    └── string (body: "\"password\"", value: "password") [3,21]
```

Figure 11.0 AST of the legitimate JSON Query.

Taking an example of a malicious query, a threat actor can enter `{"$gt": ""}` as the value for the variables `username` and `password`, which will lead to the query shown in Figure 12.0

```
{
  "username": {"$gt": ""},
  "password": {"$gt": ""}
}
```

Figure 12.0 Malicious query due to Injection.

`$gt` will select the documents which are greater than `"`. This statement will always be true, and the threat actor will successfully be able to authenticate. If we construct the AST of the modified JSON query due to the NoSQL injection exploit, as shown in figure 13.0 we can see that additional nodes are introduced.

```
object (prolog: "{\n      ", epilog: "}") [1,1]
├── member (epilog: ",\n      ") [2,9]
│   ├── string (body: "\"username\"", value: "username", epilog: ": ") [2,9]
│   └── object (prolog: "{", epilog: "}") [2,21]
│       └── member [2,22]
│           ├── string (body: "\"$gt\"", value: "$gt", epilog: ": ") [2,22]
│           └── string (body: "\"\"", value: "") [2,28]
└── member [3,9]
    ├── string (body: "\"password\"", value: "password", epilog: ": ") [3,9]
    └── object (prolog: "{", epilog: "}\n") [3,22]
        └── member [3,23]
            ├── string (body: "\"$gt\"", value: "$gt", epilog: ": ") [3,23]
            └── string (body: "\"\"", value: "") [3,30]
```

Figure 13.0 AST of the NoSQL Exploit JSON query

NoSQL databases accept queries in the form of JSON or Javascript functions. NoSQL injection exploits will introduce additional JavaScript code or a JSON query. This injected code or JSON query will show up as additional nodes in the abstract syntax tree of the legitimated normalized Javascript code or JSON query.

4.1 Detection of NoSQL Exploitation.

The algorithm used to detect a NoSQL injection attack makes use of the application-level hooks to monitor functions such as `mapReduce()`, `find()`, `findOne()`, `findID()` and the `where` operator. These functions accept queries in the form of NoSQL javascript functions or as JSON arguments which get executed in the NoSQL database. Once the functions are hooked in an executing application, arguments of these functions are then used to identify the JavaScript code or queries in JSON which gets executed by the database. A program dependency graph is used to identify the JS functions or JSON query which accepts user or external input from methods such as GET, POST, etc. If there is a flow of data from methods which accepts user inputs such as GET, POST. etc. to these javascript functions or JSON query, then abstract syntax tree (AST) of the legitimate JS function or JSON query is computed. In the case of a successful NoSQL injection exploitation, AST of the legitimate Javascript code or JSON query will change. NoSQL exploitation will introduce additional nodes in the legitimate AST. For every database access, AST of the executing JS function or JSON query is compared with the legitimate JS function or JSON query. If there is any additional node in the AST of executing JS function or JSON query it is compared with the value or with the part of the value passed to the

methods such as GET, POST, Cookies, User-Agent, etc.. In the case of a match, alert for NoSQL injection is raised.

5.0 Technical Details of XPath Injection Exploitation

XPath injection [5] exploitation is similar to the SQL injection exploitation with the difference being that, in the case of SQL injection exploitation, malicious SQL queries are sent to the database and, in the case of XPath injection exploits, malicious XPath expressions are sent to the XML databases which store the data. These malicious XPath expressions can then be used to modify or delete the data stored in the XML database.

```
<users>
  <user>
    <login>admin</login>
    <password>password</password>
    <home_dir>/home/admin</home_dir>
  </user>
  <user>
    <login>admin1</login>
    <password>password1</password>
    <home_dir>/home/admin1</home_dir>
  </user>
</users>
```

Figure 14.0 Showing the XML storing Login and Password

Figure 14.0 shows an example of the XML file storing the login and password detail. Figure 15.0 shows an example of code[5] vulnerable to the XML path injection vulnerability. The code accepts the user input username and password from the GET method and passes the variable `xlogin` to `XPathExpression`. The code then creates a new instance of the `Document` which, in the subsequent code `xlogin.evaluate(d)`, is used to validate the username and password stored in the XML file with the username and password in the variable `xlogin`. In the case of a valid username and password, authentication succeeds.

```
XPath xpath = XPathFactory.newInstance().newXPath();
XPathExpression xlogin = xpath.compile("//users/user[login/text()=' " +
login.getUserName() + "' and password/text() = '" + login.getPassword()
+ "']/home_dir/text()");
Document d = DocumentBuilderFactory.newInstance().newDocumentBuilder().
parse(new File("db.xml"));
String homedir = xlogin.evaluate(d);
```

Figure 15.0 Code vulnerable to the XPath Injection Exploitation

If a threat actor enters `'admin'` or `"="` for username and password as `'` or `'='`, the Xpath expression becomes

```
//users/user[login/text()='admin' or "" and password/text() = " or  
""]/home_dir/text()
```

The condition `or ""` is always true and allows the threat actor to logon to the system without authentication.

5.1 Detection of XPath Injection Exploitation

The algorithm used to detect the XPath injection attack makes use of the application-level hooks to construct the program dependency graph (PDG). The PDG captures the flow of data and control from methods which accept external input such as GET, POST, Cookies, etc. to the functions such as `xpath.compile()` which construct the XPath expression, and functions which executes XQuery such as `xdmp:eval()`, `xdmp:value()`, `fn:doc()`, and `fn:collection()`. Once the PDG is generated, it is used to identify the legitimate XPath expression or XQuery which accepts the data from the methods which takes user input such as GET, POST, Cookies, etc. Any injected malicious XPath expression or XQuery by a threat actor to the legitimate XPath expression or XQuery will show up as a change in the abstract syntax tree (AST) of the normalized legitimate XPath expression or XQuery. For every database access, AST of the executing XPath expression or XQuery is compared with the legitimate XPath expression or XQuery. If there is an additional node in the AST of the executing XPath expression or XQuery, then this change in the AST is matched with the value or with the part of the value passed to the methods which accept user inputs such as GET, POST, Cookies, etc. In the case of a match, an alert for XPath injection is raised.

6.0 Technical details of LDAP injection Exploitation.

LDAP injection is another injection technique which has been used by threat actors to extract information from LDAP directory. LDAP filters are defined in RFC 4515. RFC 4256 allows the use of the following standalone symbols as two special constants:

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

In the case of LDAP injection exploitation a threat actor can inject a malicious filter. If the web application does not sanitize the query, inserted filter will be executed by the threat actor, which can lead to providing additional information to the threat actor, or to bypassing access control.

```
find("(&(uid="+username+")(userpassword="+password+"))"))
```

Figure 16.0 Legitimate LDAP query filter in an application.

Figure 16.0 shows an example of an LDAP filter in an application which accepts the value of the variables *username* and *password* from the user-supplied input methods such as GET, POST. If the value of the variable *username* is *abhi* and the value of the variable *password* is *pass*, then the parse tree of the LDAP filter is as shown in Figure 17.0

```
#<Net::LDAP::Filter:0x000055b71ef93708
@left=
  #<Net::LDAP::Filter:0x000055b71ef78908 @left="uid", @op=:eq, @right="abhi">,
  @op=:and,
  @right=
    #<Net::LDAP::Filter:0x000055b71ef93870
      @left="userpassword",
      @op=:eq,
      @right="pass">>
```

Figure 17.0 Parse tree of the legitimate LDAP filter.

In an application, the LDAP filter will remain the same only the value of the user input variables to the LDAP filter will change. In the case of an LDAP injection exploitation, a threat actor can introduce malicious LDAP filter which will modify the legitimate LDAP filter and will be executed by an application. Explaining it with an example, for the LDAP filter shown in Figure 16.0, if a threat actor enters *abhi)(&))* as the value of variable *username* and *userpassword* is left empty, the modified LDAP filter query is shown in Figure 18.0

```
(&(uid=abhi)(&))(userpassword=))
```

Figure 18.0 LDAP filter due to the injection attack

In the modified filter, only the first filter is processed by the LDAP server, that is, only the `(&(uid=abhi)(&))` is processed. This query is always true, so the threat actor will be able to authenticate without having a valid password. This is also reflected in the modified parse tree shown in figure 19.0

```
#<Net::LDAP::Filter:0x0000564e8da76048 @left="uid", @op=:eq, @right="abhi">
```

Figure 19.0 Parse Tree of the modified query.

In the case of LDAP injection exploitation, if a threat actor introduces a malicious filter, then it will be reflected by a modification in the parse tree of the legitimate filter query.

6.1 Detection of LDAP Exploitation

The algorithm used to detect LDAP injection attacks makes use of application hooks to monitor the functions which executed LDAP filter queries such as `ldap_search()`, `ldap_search_st()`, `ldap_search_ext()` and `ldap_search_ext_s()`. Hooks to these functions helps to identify the legitimate LDAP filter queries will be executed by an application. Application hooking is also used to generate a program dependency graph which captures the flow of data and control from methods which accepts user inputs such as GET, POST etc. to the functions which executed LDAP filter queries. Once the program dependency graph has been generated, it is used to identify the legitimate LDAP filter queries in an application which can accept user input from methods such as GET, POST etc. If there is a flow of data from methods such as GET, POST etc. to the LDAP filter queries, then the parse tree of the normalized LDAP filter query is computed and stored. For every access to LDAP, the parse tree of the normalized executing LDAP filter query is compared with the parse tree of the normalized legitimate LDAP filter query. In the case of a mismatch between the parse tree, value from methods such as GET, POST, etc. are checked to validate if the mismatch is due to the value field. If the validation is found to be true alert for LDAP injection exploitation is raised.

7.0 Conclusion

The algorithm to detect the NoSQL, SQL, XPath, OS Command, LDAP injection attack makes use of application-level hooks and monitoring of systems calls. Injection attacks such as SQL, NoSQL, OS command, LDAP injection exploits adds additional code which leads to a change in the legitimate code of the application. The algorithm to detect the web injection based exploitation makes use of the abstract syntax tree (AST), program dependency graph (PDG) and the query parse tree to compute the changes in the original code due to the injection-based exploits. The computation of changes in the code is done for every access to database and invocation of program execution functions. If there is any deviation from the original code which is identified by changes in the AST, PDG, SQL parse tree or LDAP parse tree, it is checked to determine whether the deviation is due to the value or part of the value passed to the methods which accepts user's input such as GET, POST, Cookies, User-Agent, etc. In the case of a successful verification, alert for injection exploitation is raised.

The algorithm to detect the injection based exploitation has the following inherent advantage:

- It raises an alert during exploitation and is independent of the application and operating system on which the application is executing. Hence the algorithm is capable of detecting both known and zero-day vulnerabilities in the application.
- Once an alert is raised, the algorithm also identifies the vulnerable section of the code which has been exploited by the threat actor. This identification of the vulnerable part of the code will aid to patch the code preventing further exploitation.
- It complements the current detection techniques such as signature-based pattern matching of the SQL commands with the known SQL commands such as UNION, etc. at the network layer and source code analysis.
- The algorithm only leverages binary instrumentation of the application to detect injection-based exploitation. Hence the detection is independent of the deployment of an application and the manner it accepts external inputs. The application can be deployed as a backend microservices and can accept batched requests which get broken down by the middle layer and served to the rear end microservices. In this scenario the algorithm will raise an alert for injection based attacks.

These factors make the algorithms a recommended solution to detect the NoSQL, SQL, LDAP, OS Command, XPath, XQuery Injection exploitation.

References:

- [1] Top 10- 2017, Top 10 https://www.owasp.org/index.php/Top_10-2017_Top_10
- [2] Exploit Public - Facing Application <https://attack.mitre.org/techniques/T1190/>
- [3] SQL Map, <https://attack.mitre.org/software/S0225/>
- [4] NoSQL Injection,
https://ckarande.gitbooks.io/owasp-nodegoat-tutorial/content/tutorial/a1_-_sql_and_nosql_injection.html
- [5] XPath Injection, <https://cwe.mitre.org/data/definitions/643.html>
- [6] ECMAScript parsing infrastructure for multipurpose analysis, <http://esprima.org>
- [7] libpg_query, https://github.com/lfittl/libpg_query/blob/10-latest/README.md#resources
- [8] Compromised Web Servers and Web Shells - Threat Awareness and Guidance
<https://www.us-cert.gov/ncas/alerts/TA15-314A>
- [9] JSON -AST <https://github.com/rse/json-ast>
- [10] NoSQL injection in MongoDB <https://zanon.io/posts/nosql-injection-in-mongodb>
- [11] LDAP Injection & Blind LDAP injection in Web Application,
<https://www.blackhat.com/presentations/bh-europe-08/Alonso-Parada/Whitepaper/bh-eu-08-alonso-parada-WP.pdf>
- [12] LDAP Filter Parser,
<https://www.rubydoc.info/github/ruby-ldap/ruby-net-ldap/Net/LDAP/Filter/FilterParser>
- [13] Web Shell, <https://attack.mitre.org/wiki/Technique/T1100>
- [14] Compromise Web Servers and Web Shells - Threat Awareness and Guidance,
<https://www.us-cert.gov/ncas/alerts/TA15-314A>
- [15] PHP Shell Detector,
<https://github.com/emposha/PHP-Shell-Detector/blob/master/README.md>
- [16] What is a 0-day Exploit,
<https://www.fireeye.com/current-threats/what-is-a-zero-day-exploit.html>