Object Injection Vulnerability Discovery Based on Latent Semantic Indexing

Hossain Shahriar and Hisham Haddad
College of Computing and Software Engineering
Kennesaw State University, USA
{hshahria, hhaddad}@kennesaw.edu

ABSTRACT

Object Injection Vulnerability (OIV) is an emerging threat for web applications. It involves accepting external inputs during deserialization operation and use the inputs for sensitive operations such as file access, modification, and deletion. The challenge is the automation of the detection process. When the application size is large, it becomes hard to perform traditional approaches such as data flow analysis. Recent approaches fall short of narrowing down the list of source files to aid developers in discovering OIV and the flexibility to check for the presence of OIV through various known APIs. In this work, we address these limitations by exploring a concept borrowed from the information retrieval domain called Latent Semantic Indexing (LSI) to discover OIV. The approach analyzes application source code and builds an initial term document matrix which is then transformed systematically using singular value decomposition to reduce the search space. The approach identifies a small set of documents (source files) that are likely responsible for OIVs. We apply the LSI concept to three open source PHP applications that have been reported to contain OIVs. Our initial evaluation results suggest that the proposed LSI-based approach can identify OIVs and identify new vulnerabilities.

CCS Concepts

• Security and Privacy➞Software and Application Security;
• Information Retrieval ➞ Document Representation; Information Retrieval Query Processing;

Keywords

object injection attack; code reuse; latent semantic analysis; information retrieval; web security

1. INTRODUCTION

Web applications are widely used to perform many useful activities. Despite the availability of many tools to design and implement secure web applications, vulnerabilities are widely discovered [21].

Among the known security vulnerabilities, SQL Injection and Cross-Site Scripting have received broader level of attention from the research community [2,3]. However, recently a new form of security threat has emerged, known as *Object Injection Vulnerability* (*OIV*) [6,7]. OIV was first demonstrated to be practical for PHP web applications [7]. A number of vulnerability reports have emerged in Open Source Vulnerability Databases (OSVDB) about object injection, particularly in PHP web applications [26,27,28]. The vulnerability is commonly found in applications having programmer defined classes. An attacker may supply deserialized objects to an application which would be subsequently used within application's scope (*i.e.*, setting class member variables). An attacker can provide an arbitrary file name; perform path traversal, or even SQL Injection.

There remains some degree of similarity between OIV exploits and code reuse attacks [8]. In a code reuse attack, the goal of the attacker is to invoke a method with malicious payload, violating the expected control flow integrity of an application. In OIV exploit, an attacker provides serialized JSON (or XML) objects to alter the attributes of class members, and behavior changes during the invocation of methods that are defined or overridden in the object's class.

Very few research works have proposed countermeasures against OIV [6,7,8]. The object oriented nature of web applications makes the systematic discovery of OIV challenging. For example, one source file may deserialize an object; another two files may be responsible for setting object member attributes with deserialized objects, and invoking sensitive operations. It is common to find inheritance (one class extending a base class, which may define a function performing sensitive operations using attacker supplied deserialized objects). Developers need to identify a set of relevant source files to look for OIV presence and fix it accordingly. Thus, OIV discovery still remains a challenging task for large scale applications having hundreds and thousands of source files. Further, existing approaches do not have enough flexibility to let developer query for specific patterns of OIV of interest. For example, a developer may be interested to find only OIVs that involve accepting inputs with POST data and exploitation may involve object's constructor methods. We believe that these challenges could be addressed by considering OIV discovery as a process of querying an application's source code base for the presence of signatures (or method calls) and identifying a small set of source files that would be highly relevant to the query. A relevant and small set of source files could aid developers to inspect closely rather than exploring large amount of source code.

In this paper, we propose a new approach to discover OIV in PHP web applications. We borrow the concept of *Latent Semantic Indexing* (LSI [9]) to identify OIVs. LSI is a widely used concept to find a set of relevant documents given that a user provides a set of words or terms of interests. In the context of program source code, this means finding a set of source files having responsible code elements for OIVs. The technique essentially reduces the

SAC 2016, April 04-08, 2016, Pisa, Italy
©2016 ACM. ISBN 978-1-4503-3739-7/16/04…$15.00.
DOI: http://dx.doi.org/10.1145/2851613.2851865

dimensionality of search space while processing the textual input data. Our contribution includes the application of LIS in the context of OIV discovery. We provide guidelines to choose query terms, and dealing with two common issues, ambiguity and synonymity, to apply LSI. We evaluated our approach with three open source large scale PHP applications reported to contain OIVs. Our evaluation results show that the LSI approach can detected OIVs. Further, our approach enables the detection of new OIVs based on the queries.

This paper is organized as follows: Section 2 presents an example of OIV and attack; Section 3 highlights related work; Section 4 presents the proposed approach; Section 5 discusses evaluation results; and Section 6 concludes the paper.

2. OBJECT INJECTION VULNERABILTY

Object Injection vulnerability (OIV) occurs when an application accepts unsanitized objects (e.g., JSON or XML), deserializes them, and then apply them to various sensitive operations such as accessing or modifying files, cookies, and sessions [1]. Deserialized objects may be used to set class members that might be used during sensitive operations.

The attack requires satisfying two generic conditions: first being able to accept and deserialize objects, invocation of methods from classes that may be using these objects within the scope of the objects instantiated already.

PHP supports serialization convert to any object to plain string representation with *serialize()* method [5]. The object can be converted back to the original representation by invoking method *unserialize()* [4]. The representation follows JavaScript Object Notation (JSON) syntax format [20]. An attacker can easily gain the knowledge and construct payloads accordingly to exploit OIV in web applications.

We now provide an example of deserialization of unfiltered input leading to object injection attack for PHP application. The attack input would create a file in an expected location trough path traversal technique.

```
<?php
1.  class Person {
2.     private $name;
3.     function __construct($name) {
4.        $this->name = $name;
5.     }
6.     function set_name($new_name) {
7.        $this->name = $new_name;
8.     }
9.     function get_name() {
10.       return $this->name;
11.    }
12.    function writeFile($fname, $o){
13.       $myfile = fopen($fname, "w") or die("Unable to open file!");
14.       fwrite($myfile, $o->get_name());
15.       fclose($myfile);
16.    }
17. }
?>
```

Figure 1. Example of PHP class (*person.php*).

Figure 1 shows PHP code for an example class (*person.php*) in Lines 1-17. Here, the name of a person is captured with private variable *$name* whose value can be set with *set_name()* and retrieved with *get_name()* method calls, respectively. The *writeFile()* method accepts file name and object as input (*$fname, $o*) and writes back the current user name to a disk file (Lines 12-16). Here, *writeFile()* method is vulnerable to object injection as it does not check if supplied file name has any malicious content.

The OIV vulnerability can be exploited by invoking the code shown in Figure 2 (*save.php*). Here, the code includes the previous class file (*Person* defined in *person.php*). It receives an input (Line 2) through $_GET["data"] and deserializes to obtain the file name (*$fname*) at Line 3. Line 4 creates a new object (*$o1*). It then invokes method *writeFile()* with the file name (obtained through deserialization) and the created object to save the name to the supplied file name.

```
<?php
1.   include ("person.php");
2.   $data = $_GET["data"];
3.   $fname = unserialize($data);
4.   $o1 = new Person ("John Doe");
5.   $o1->writeFile($fname, $o1);
?>
```

Figure 2. Example of OIV exploit (*save.php*).

It is possible to create arbitrary filename in serialized format. An attacker is able to create file in an unwanted location. Let us assume, the value of *$_GET["data"]* is provided with "s:8:\"file.txt\"". If deserialization is performed, it would generate the file name string as *file.txt*. Thus, the code would write the name "John Doe" in "*file.txt"* created in the same directory where the application code resides. An attacker may provide "s:11:\"..\file.txt\"" through *$_GET["data"]* input field. The deserialization operation would result in the file name as "*..\file.txt*". Thus, the name "John Doe" would be saved in file.txt which would be created one level up to the current directory (path traversal).

The input may come from other available objects such as COOKIE, SESSION, and user defined classes. It can be demonstrated to construct arbitrary SQL queries with deserialized string. Moreover, magic methods may be invoked with unfiltered deserialized objects to perform sensitive operations. Magic methods are those that get automatically invoked during certain events. PHP supports a set of magic methods. For example, the construction of a new object event leads to the invocation of *__construct()*. The destruction of an object invokes *__destruct()*, the output generation with *echo* method invokes *__toString()*, serialization of objects invoke *__sleep()* method to perform cleanup of unused memory objects and spaces, and deserialization of objects leads to the invocation of *__wakeup()* method to reconstruct objects [19]. An attacker provided serialized objects can be part of these magic methods and exploit the vulnerability.

The scope of injected object could be in multiple files due to the supported object oriented programming concept in web applications. This brings further complexity for identifying OIV straightforward manner. For example, a PHP class may be extending a base class through inheritance; a method may be redefined or overloaded. Thus, OIV discovery process is challenging for applications having large amount of source code and files. Our approaches address these issues.

3. RELATED WORK

Here we provide a brief overview of literature work related to OIV detection. The latest effort addressing OIV detection from PHP source code is from Dashe *et al.* [6]. They construct abstract syntax tree from PHP source code and tainted perform data flow analysis. They summarize all defined methods by simulating basic blocks from the source code. They find if deserialized data sources can reach any sensitive operations such as file deletion. Their results provide developers an initial glimpse of the deserialized objects that may be redefined or classes that may be inherited from base class. However, their approach does not provide flexibility to perform various queries related to OIVs. In contrast, our approach is based on information retrieval concept of Latent Semantic Indexing (LSI). The approach can find relevant source files and scales up well if the number of source file is huge to point out the set of closely related files causing vulnerabilities.

OIV is a variation of code reuse attacks which got much broader attention lately from the research community through well-known buffer overflow exploits in legacy application implemented in C/C++. In a buffer overflow attack, an attacker provides malicious shell command to run arbitrary code. The payload gets copied in program buffers space and alters the expected control flow of applications [15]. Recently, there has been attempt to make program data segment and stack region non-executable to defend against buffer overflow exploits. Currently, most of the known operating systems supports non-executable pages or segments concept. For example, in NetBSD, stack and heap regions are non-executable by default. However, attackers have found sophisticated technique known as *return-to-libc* [17]. In this approach, instead of providing arbitrary injected code into a program's data buffer, an attacker provides new instruction address in the stack pointing a dynamic library routine.

Sadeghi *et al.* [8] study a number of latest techniques that bypass code injection attacks prevention measure such as address space layout randomization. They proposed an architecture to counter against code reuse attacks by preserving the control flow integrity at both hardware and software level. Readers are suggested to see surveys [21,22] addressing mitigation approaches of other traditional code injection attacks.

We now discuss some literature work related to the application of LSI below. LSI has been applied in several domains such as document classification [10,11], recovering connection between documents and program source code [12,13], identifying similarities among program source code [24], and automating the task of assigning reviewers to manuscripts [23]. However, we are not aware of any approach that explores the application of LSI for discovering application security vulnerabilities. Below, we briefly describe some earlier approaches that motivate our work.

Price *et al.* [10] reported that LSI is a robust approach to classify data in presence of noise. They studied known benchmark text data corpus (Reuters) by randomly inserting, updating, and deleting texts, and similarly conducted study for OCR documents having poor quality to see performance during retrieval of information.

Wang *et al.* [12] applied advanced LSI model to improve software engineering tasks. They define terms by capturing domain specific vocabulary from source code; construct a similarity dictionary with synonymity and abbreviations, clustered source code to improve precision of retrieval process.

Marcus *et al.* [13] applied LSI to recover the link between documents and source code in legacy software. They consider the written comments in the source code as part of the documentation and match them with the appropriate variable identifier names. Later McMillan *et al.* [18] explored recovering traceability links between requirement documentation and source code by performing textual and structural analysis (*i.e.*, methods and variables are modeled). They also compared their approach with other information retrieval approaches. In comparison to all these effort, we apply LSI concept to discover vulnerable sources given that there is a query to identify various types OIVs.

4. LSI-BASED OVI DISCOVERY

4.1 LSI APPROACH

Our approach is guided by Latent Semantic Indexing which is widely used to rank a set of documents based on how closely related to a set of query terms. The approach is applied based on the following notion of terms and documents that we adapt in the context of discovering OIVs.

- Application source files are considered as documents.
- Terms are defined based on known method calls or signatures that can exploit OIVs. We consider sensitive methods that accesses local computer resources (file systems) or web application resources (systems, cookies).
- Queries are formulated based on a set of terms of interests which captures combination of method calls that may indicate the presence of vulnerabilities.
- Similarity measures are based on cosine metrics from information retrieval literature. The closer a query vector and a document vector is, the higher the distance. This implies that the document is more likely relevant to OIV discovery process and likely to include the cause of vulnerabilities.

Though we provide examples for PHP code, the approach is generic enough to apply other implementation languages. Below are the six steps we follow to apply the LSI approach.

Step1: Build Term-Document matrix $A = [m*n]$, where m = number of terms, n = number of documents; $q = [m*1]$ as query vector. We denote terms as $t_1 \ldots t_m$, documents as $d_1 \ldots d_n$.

Step2: Perform Singular Value Decomposition (SVD) [14]. This means expressing A in terms of three matrices U, S, and V^T such that $A = U * S * V^T$. Here, * is matrix multiplication operator, U is the Eigen vector of the matrix A, S is the diagonal matrix having singular value, V is Inverse of U matrix, V^T is the transpose of V matrix.

Step3: Choose k out of n ($k < n$) to reduce the dimensionality of the A matrix.

Step 4: Define A_k by reducing the dimensions of A from $m*n$ to $m*k$. We define S_K from S by reducing dimension from $m*n$ to $m*k$ ($k < n$). Finally, define V_K from V by reducing the dimension from $n*n$ to $k*k$. Now, each document d_i is approximated by the corresponding row vector of v_i from V_K matrix.

Step 5: Obtain the approximate query vector in reduced space from q to q_k, by multiplying the three matrices $q* U_K * S_K^{-1}$. This means the original query vector is reduced from $m*1$ to $k*1$. Here, S_K^{-1} is the inverse matrix of S_K.

Step 6: Measure similarity using cosine distance formula as follows:

$$Sim(q_k, d_k) = q_k.d_k / (|q_k||d_k|)$$

Here, $|\ldots|$ represents the norm operation of a vector, $q_k.d_k$ represents the dot product for the two column vectors. We rank documents (d_i) in decreasing value of the distance.

4.2 EXAMPLE APPLICATION

We assume d_1 - d_4 are four source files. We define a set of terms related to OIV and find the presence of the terms in the documents to define A matrix (step1) as shown in Table 1.

Table 1: Term Document Matrix (A)

Term	d_1	d_2	d_3	d_4	q
Deserialize	1	0	0	1	1
Serialize	0	1	1	0	0
$_GET[]	0	1	0	0	1
$_POST[]	0	0	1	0	0
$_COOKIE[]	1	0	1	1	0
$_SESSION[]	0	1	0	1	0
__construct	0	0	1	1	1
__destruct	1	1	0	1	0
__toString	0	1	0	0	1

Let us assume a query (q) is intended to find OIV causing due to the acceptance of input through GET method ($GET[]) and invocation of two magical methods *__construct()* and *__toString()*. After applying SVD[1] of Step2, we obtain S, U, V^T matrices by performing Singular Value Decomposition of A matrix. These matrices are shown in Tables 2, 3, and 4. We show the V matrix (transpose of V^T) in Table 5.

Table 2: Singular value matrix (S)

2.79	0	0	0
0	1.91	0	0
0	0	1.69	0
0	0	0	0.82
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Table 3: U matrix

0.32	-0.29	-0.51	-0.31	0.01	0.03	-0.02	-0.65	0.07
0.35	0.43	0.32	-0.23	-0.35	-0.34	-0.39	-0.21	-0.3
0.23	-0.04	0.44	-0.04	0.31	0.00	0.67	-0.25	-0.35
0.11	0.47	-0.11	-0.19	-0.35	0.70	0.27	0.05	0.07
0.11	0.47	-0.11	-0.19	0.79	0.06	-0.23	0.07	0.10
0.44	-0.16	0.13	0.63	0.08	0.43	-0.35	-0.08	-0.12
0.33	0.35	-0.42	0.48	-0.08	-0.43	0.35	0.08	0.12
0.56	-0.33	-0.07	-0.35	-0.01	-0.03	0.02	0.65	-0.07
0.23	-0.04	0.44	-0.04	-0.03	-0.06	0.04	-0.10	0.85

Table 4: V^T matrix

0.31	0.65	0.32	0.59
-0.33	-0.082	0.91	-0.23
-0.35	0.748	-0.19	-0.52
-0.81	-0.032	-0.15	0.55

Table 5: V matrix

0.31	-0.33	-0.35	-0.81
0.65	-0.08	0.74	-0.03
0.32	0.91	-0.19	-0.15
0.59	-0.23	-0.52	0.55

[1] Online tools are available for SVD, for example http://www.bluebit.gr/matrix-calculator/

Let us assume that we reduce the dimensionality of S from 9 X 4 to 2 X 2 (setp3). The S_K, U_K, and V_K we now obtain are shown in Tables 6, 7, and 8, respectively (following step 4).

Table 6: S_K matrix

2.795	0
0	1.911

Table 7: U_K matrix

0.328	-0.296
0.353	0.433
0.235	-0.043
0.118	0.476
0.118	0.476
0.449	-0.165
0.332	0.354
0.563	-0.339
0.235	-0.043

Table 8: V_K matrix

0.3191	-0.3323
0.6567	-0.0823
0.3294	0.9101
0.5986	-0.2335

Now, the new document vectors in the reduced space would be the V_K matrix as shown in Table 9 (step 4):

Table 9: Refined document matrix

d_1	0.3191	-0.3323
d_2	0.6567	-0.0823
d_3	0.3294	0.9101
d_4	0.5986	-0.2335

We now find q_K (step 5) by multiplying $q^T{*}U_K{*}S_K^{-1}$ which is shown in Table 10. Here, S_K^{-1} is the inverse matrix of S_K.

Table 10: q_k vector

q_k	0.40	-0.01

We now apply step 6 to compute the distance between q_k and each of the documents as shown in Table 11. We now have the ranking of documents as follows: $d_2 > d_4 > d_1 > d_3$. The results indicate that d_2 would be more relevant for OIV, followed by d_4 and d_1. This enables developers to identify OIV among these source files and fix it (e.g., sanitizing input).

Table 11: Similarity between query and document

Cosine Similarity	Distance
Sim (q_k, d_1)	0.718304
Sim (q_k, d_2)	0.996096
Sim (q_k, d_3)	0.306054
Sim (q_k, d_4)	0.944181

4.3 ADDRESSING SYNONIMITY AND ABBREVIATION

OIV takes advantage of sensitive operations performed at local server machines with unfiltered serialized object inputs. It is common to see API methods that access to a common resource type. Given that having too many unique APIs in the term sets unnecessary increases the search space. We address it by categorizing similar types of APIs. For example, all APIs accessing file system can be classified as one generic term. Table 12 shows a list of file related operations. We can have similar category for string (Table 13) and cookie operations (Table 14).

Table 12: APIs for file operations.

File	Description
fopen ()	Opens a file
fwrite()	Writes to a file
fclose()	Closes a file
chmod()	Changes file permission
unlink()	Deletes a file

Table 13: APIs for string category operations.

String	Description
strlen()	Return string length
strrev()	Reverses string
str_replace()	Replaces some characters with some other characters in a string.

Table 14: APIs for cookie related operations.

Cookie	Description
setcookie ()	Cookie gets created or deleted.
$_COOKIE [...]	Cookie object gets retrieved.

Abbreviation refers to terms in documents expressed or referred in similar context (shorthand notation). For PHP, we address abbreviation in the context of inheritance and file inclusion.

Table 15: Addressing abbreviation

Abbreviation context	Example code leading to OIV

Inheritance (base class extension)	`<?php`
	`  class A {`
	`    function print(){`
	`      deserilze();`
	`    }`
	`  }`
	`  class B extends A{`
	`    print ();`
	`  }`
	`}`
	`?>`

Local file inclusion	`<?php`
	`  include 'save.php';//defined in Fig2`
	`?>`

Inheritance: If a class extends another base class, and base class may define magic methods or deserializes inputs, we consider the extension as a form of abbreviation. Thus, we include "extend" in our term set. We show an example in the first row of Table 15. Here, B is an extension of base class A.

File inclusion: If a source file includes another source file, this provides the context to consider abbreviated scope of using deserialized inputs leading to the exploitation of OIV. Thus, we consider *include* as abbreviation in the context of OIV discovery. We show an example of local file inclusion in the second row of Table 15. Here, the source file "save.php" is defined earlier in Figure 2.

5. EVALUATION

We evaluate the proposed approach using three PHP applications that have been reported to contain OIV in OSVDB [25]. These include Contao CMS, SilverStripe, and CMS Made Simple. We show the version number, total number of source file, class, file inclusion, and inherited class in Table 16. All these applications are large in size considering the number of source file, classes, and inheritance.

Table 16: Characteristics of evaluated applications.

Name	Version	# of PHP files	# of classes	# of inclusion	# of inheritance
Contao CMS	3.2.19	584	725	0	508
SilverStripe	2.4.9	511	669	2	444
CMS Made Simple	2.0	187	177	0	97

Table 17 shows some examples of sensitive methods present in the three applications. These include file operations (fopen, unlink, fwrite, chmod), session, cookie, and input taking operations (GET and POST). Table 18 shows a summary serialization, deserialization, and magic methods invoked or defined in these applications.

Table 17: Sensitive methods in evaluated applications.

Name	fopen	unlink	fwrite	chmod	Session	Cookie	GET	POST
Contao CMS	38	33	19	47	130	9	47	44
SilverStripe	32	26	28	5	39	9	94	7
CMS Made Simple	11	14	7	5	3	2	1	25

Table 18: Magic methods in applications.

Name	serialize()	deserialze()	__construct()	__destruct()	__sleep()	__wakeup()	__toString()
Contao CMS	53	227	351	19	2	4	16
SilverStripe	34	1	402	6	0	0	8
CMS Made Simple	1	1	74	4	0	0	34

Table 19 shows a list of four example queries that we apply in our initial evaluation. The first query (Q1) shows that we are interested to discover OIVs in related documents that include the presence of deserialization method, inputs may be received by GET or POST method, and deserialized inputs may be used for file operations. Similarly, Q4 represents the query that attempts to find relevant documents pointing OIVs for both serialization, deserialization, inputs coming from GET or POST, operations performed on cookie or session object and a set of magic methods may be invoked.

Table 20 shows the result of applying the four queries for the three applications. We cross check the accuracy of the obtained OIV related documents from OSVDB # already known. We find that Q1-Q4 find vulnerabilities in Contao CMS where OIV is in magic methods. Q4 discovers a new vulnerability.

Table 19: Query applied in evaluation.

Terms	Q1	Q2	Q3	Q4
deserialize()	1	1	1	1
serialize()	0	1	0	1
$_GET[]	1	1	1	1
$_POST[]	1	1	1	1
$_COOKIE[]	0	0	1	1
$_SESSION[]	0	0	1	1
__construct()	0	1	0	1
__destruct()	0	1	0	1
__toString()	0	1	0	1
file	1	0	1	0

For SilverStripe, Q1 and Q2 do not lead to the discovery of OIV. However, Q3 and Q4 reveal the vulnerability as they include cookie operation as reported in OSVDB. For CMS Made Simple application, Q1 and Q3 discover OIV due to file operation with unfiltered deserialized inputs.

Table 20: OIV findings using top two documents (k=2).

Name	OSVDB #	Q1	Q2	Q3	Q4
Contao CMS	102856	1	1	1	1
SilverStripe	85714	0	0	1	1
CMS Made Simple	115313	1	0	1	0

Table 21 shows the result of applying the four queries for the three applications while analyzing top four documents (k=4) to identify OIVs. The more files are considered for analysis, new vulnerabilities can be identified.

Table 21: OIV findings using top four documents (k=4).

Name	OSVDB #	Q1	Q2	Q3	Q4
Contao CMS	102856	2	2	3	4
SilverStripe	85714	1	1	1	2
CMS Made Simple	56878	2	1	2	1

We show two examples of OIVs we identify in Contao CMS application. We show example code snippet in Figures 3 and 4, respectively. In Figure 3, a calendar feed is deserialized in an inherited class, which has been used in another source file in magic method.

```
//inherit:class Calendar extends Frontend
$arrCalendars = deserialize($arrFeed['calendars']);
```

Figure 3: Example of OIV in Contao CMS through deserialization

In Figure 4, we show an example of OIV in Ftp module, where a class is extended. Here, fopen() method accepts a file name ($strFile) and permission ($strMode), but the method does not check or sanitizes these inputs before opening or deleting.

```
public function fopen ($strFile, $strMode){
  $resFile = fopen(TL_ROOT . '/system/tmp/'.
md5(uniqid(mt_rand(), true)), $strMode);

  if (!@unlink(TL_ROOT . '/' . $strOldName))
      $this->chmod($strDestination,
$GLOBALS['TL_CONFIG']['defaultFileChmod']);
  ..
}
```

Figure 4: Example OIV in Contao CMS (Ftp extends\Files)

6. CONCLUSION

Object Injection Vulnerability (OIV) can be exploited by providing objects within application scope to perform unwanted activities that may lead to security breaches. It is challenging to find OIV as they may span over multiple source files. There are very limited research works till now that identify OIV in implemented applications. However, they do not provide enough flexibility to discover OIV for various contexts involving object oriented programming feature, presence of magic methods, etc. Developers may need to fix the vulnerability in multiple files. This work proposed Latent Semantic Index (LSI) based approach to identify OIV in web application source code. We adapted the steps followed in LSI computation, except, we defined terms, documents, and addressed the issue of abbreviation and synonimity while applying the approach. The approach provides flexibility to define customized query to identify relevant source files to discover the vulnerabilities. We evaluated the approach using three open source PHP applications. The approach has found the known OIV and enabled us to discover new vulnerabilities. Our future work includes evaluating on more applications and query types. We plan to explore other similarity measure (e.g., Euclidean distance), address more abbreviation contexts such as polymorphism and method overloading. Finally, we plan to apply similar LSI approach to other common web application vulnerabilities.

7. REFERENCES

[1] E. Romano, PHP Object Injection Vulnerability, https://www.owasp.org/index.php/PHP_Object_Injection

[2] The Open Web Application Security Project (OWASP) Top 10,www.owasp.org/index.php/Top_10_2013-Top_10, 2013.

[3] C. Kern, "Securing the Tangled Web," *Communications of the ACM*, Vol. 57 No. 9, 2014, pp 38-47.

[4] PHP unserialize method, Accessed from http://php.net/manual/en/function.unserialize.php

[5] PHP serialize method, Accessed from http://php.net/manual/en/function.serialize.php

[6] J. Dashe, N. Krein, and T. Holz, "Code Reuse Attack in PHP: Automated POP Chain Generation," ACM CCS 2014, Scottsdale, AZ, November 2014.

[7] Esser, S. Utilizing Code Reuse Or Return Oriented Programming in PHP Applications. In BlackHat USA (2010).

[8] A. Sadeghi, L Davi, and P. Larsen, "Securing Legacy Software against Real-World Code-Reuse Exploits: Utopia, Alchemy, or Possible Future?" *Proc. of ACM Symposium on Information, Computer and Communications Security (ASIA CCS)*, April 2015, pp. 55-61.

[9] C. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*, 2008, Cambridge University Press.

[10] R. Price and A. Zukas, "Application of Latent Semantic Indexing to Processing of Noisy Text," *Proc. of IEEE International Conference*

on Intelligence and Security Informatics, Atlanta, GA, USA, May 19-20, 2005, pp. 602-603.

[11] A. Zukas and R. Price, “Document Categorization Using Latent Semantic Indexing,” *Proceedings of Symposium on Document Image Understanding Technology*, Greenbelt, MD, April 2003, pp. 87–91.

[12] X. Wang, G. Lai, and C. Liu, “Recovering Relationships between Documentation and Source Code based on the Characteristics of Software Engineering,” *Electronic Notes in Theoretical Computer Science*, 243 (2009), pp. 121–137.

[13] A. Marcus and J. Maletic, “Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing,” *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, Portland, OR, USA, 2003, pp. 125-135.

[14] G. Salton, *Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer*, Addison-Wesley, 1989.

[15] A. One, “Smashing the Stack for Fun and Profit,” Phrack Magazine, Vol. 49, Article 14, 1996, Accessed from http://insecure.org/stf/smashstack.html

[16] NetBSD, Non-executable stack and heap, Accessed from http://www.netbsd.org/docs/kernel/non-exec.html

[17] A. Ihsahn, Advanced return-to-lib(c) exploits, 2001, Accessed from http://phrack.org/issues/58/4.html

[18] C. McMillan, D. Poshyvanyk, and M. Revelle, “Combining Textual and Structural Analysis of Software Artifacts for Traceability Link Recovery,” *Proc. of ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, Vancouver, Canada, pp. 41-48.

[19] PHP Magic Method, Accessed from http://php.net/manual/en/language.oop5.magic.php

[20] JSON Format, Accessed from http://json.org/

[21] H. Shahriar and M. Zulkernine, Mitigation of Program Security Vulnerabilities: Approaches and Challenges, *ACM Computing Surveys (CSUR)*, 44(3), Article 11, May 2012, pp. 1-46.

[22] H Shahriar, M Zulkernine, Classification of Static Analysisbased Buffer Overflow Detectors, *Proc. of Secure Software Integration and Reliability Improvement Companion (SSIRI-C)*, Singapore, June 2010, pp. 94-101.

[23] S. Dumais, and J. Nielsen, “Automating the Assignment of Submitted Manuscripts to Reviewers,” *Proc. of the 15th Annual International Conference on Research and Development in Information Retrieval, pp.* 233–244.

[24] A. Marcus and J. Maletic, “Using Latent Semantic Analysis to Identify Similarities in Source Code to Support Program Understanding,” *Proc. of 12th IEEE International Conference on Tools with Artificial Intelligence*, November 2000, pp. 46–53.

[25] Open Source Vulnerability Database (OSVDB), http://osvdb.org

[26] Contao CMS Object Injection Vulnerability, http://osvdb.org/show/osvdb/102856

[27] SilverStripe Object Injection Vulnerability, http://osvdb.org/show/osvdb/85714

[28] CMS Made Simple Object Injection Vulnerability, http://osvdb.org/show/osvdb/115504