DOI: 10.15514/ISPRAS-2023-35(1)-14

Исследование встречаемости небезопасно сериализованных программных объектов в клиентском коде веб-приложений

[1] *Д.Д. Миронов, ORCID: 0000-0002-0092-0806 <denis.mironov@solidwall.io>*
[1,2] *Д.А. Сигалов, ORCID: 0009-0005-2781-6493 <asterite@seclab.cs.msu.ru>*
[1,2] *М.П. Мальков, ORCID: 0009-0000-7019-7556 <wgh@seclab.cs.msu.ru>*

[1] *ООО «СолидСофт»,*
117312, Россия, Москва, ул. Вавилова, д. 47А
[2] *Московский государственный университет им. М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. В данной статье проведено исследование встречаемости случаев использования небезопасной десериализации при взаимодействии между клиентским кодом и серверной стороной веб–приложения. Особое внимание было уделено сериализованным объектам, отправляемым из JavaScript-кода. Были выявлены характерные особенности шаблонов использования сериализованных объектов внутри клиентского JavaScript-кода и составлены уникальные классы, главной целью которых является облегчение ручного и автоматического анализа веб-приложений. Было разработано и реализовано инструментальное средство, выявляющее сериализованный объект в коде веб-страницы. Данный инструмент способен найти закодированные сериализованные объекты, а также сериализованные объекты, закодированные с помощью нескольких последовательно примененных кодировок. Для найденных экземпляров сериализованных объектов, инструмент определяет контекст, в котором находится найденный объект на странице. Для объектов, находящихся внутри JavaScript-кода, инструмент выявляет ранее упомянутые классы, при помощи сопоставления вершин абстрактного синтаксического дерева кода. После получения результатов исследования был проведен анализ серверных точек ввода данных на предмет десериализации найденных программных объектов на стороне сервера. В результате данной проверки были найдены ранее неизвестные публично уязвимости, о которых было сообщено разработчикам данного программного обеспечения. Одна из них получила идентификатор CVE-2022-24108. По результатам проведенного исследования был предложен метод, позволяющий облегчить как ручной, так и автоматизированный поиск уязвимостей типа “Десериализация недоверенных данных”. Предложенный алгоритм был протестирован на страницах более чем 50000 веб-приложений из списка Alexa Top 1M, а также на страницах 20000 веб-приложений из программ Bug Bounty.

Ключевые слова: небезопасная десериализация; веб-приложения; анализ клиентского кода; автоматизация анализа безопасности; поиск уязвимостей.

Для цитирования: Миронов Д.Д., Сигалов Д.А., Мальков М.П. Исследование встречаемости небезопасно сериализованных программных объектов в клиентском коде веб-приложений. Труды ИСП РАН, том 35, вып. 1, 2023 г., стр. 223-236. DOI: 10.15514/ISPRAS-2023-35(1)-14

Research into Occurrence of Insecurely-Serialized Objects in Client-Side Code of Web-Applications

[1] *D.D. Mironov, ORCID: 0000-0002-**0092**-0806 <denis.mironov@solidwall.io>*
[1,2] *D.A. Sigalov, ORCID: 0009-0005-2781-6493 <asterite@seclab.cs.msu.ru>*
[1,2] *M.P. Malkov, ORCID: 0009-0000-7019-7556 <wgh@seclab.cs.msu.ru>*

[1] *SolidSoft,*
47A, st. Vavilova, Moscow, 117312, Russia
[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia

Abstract. This paper studies the occurrence of insecure deserialization in communication between client-side code and the server-side of a web application. Special attention was paid to serialized objects sent from JavaScript client-side code. Specific patterns of using serialized objects within the client-side JavaScript code were identified and unique classes were formulated, whose main goal is to facilitate manual and automatic analysis of web applications. A tool that detects a serialized object in the client-side code of a web page has been designed and implemented. This tool is capable of finding encoded serialized objects as well as serialized objects encoded using several sequentially applied encodings. For found samples of serialized objects, the tool determines the context in which the found object appears on the page. For objects inside JavaScript code, the tool identifies the previously mentioned classes by mapping the vertices of the abstract syntax tree (AST) of the code. Web application endpoints were checked for whether programming objects were deserialized on the server side, after obtaining the results of the study. As a result of this check, previously unknown vulnerabilities were found, which were reported to the developers of this software. One of them was identified as CVE-2022-24108. Based on the results of this research, a method was proposed to facilitate both manual and automated searches for vulnerabilities of the "Deserialization of untrusted data". The proposed algorithm was tested on more than 50,000 web application pages from the Alexa Top 1M list, as well as on 20,000 web application pages from Bug Bounty programs.

Keywords: deserialization of untrusted data; web-applications; client-side code analysis; security analysis automation; vulnerabilities.

For citation: Mironov D.D., Sigalov D.A., Malkov M.P. Research into Occurrence of Insecurely-Serialized Objects in Client-Side Code of Web-Applications. Trudy ISP RAN/Proc. ISP RAS, vol. 35, issue 1, 2023. pp. 223-236 (in Russian). DOI: 10.15514/ISPRAS-2023-35(1)-14

1. Введение

Веб-технологии стремительно развиваются в последние десятилетия, особенно ощутимый скачок произошел при переходе от так называемого “статического” веба к “динамическому” [1]. Раньше взаимодействие клиентской стороны веб-приложения с серверной было ограничено достаточно узким набором технологий, например: отправка GET-запроса с query-параметрами, содержащимися в URL (с помощью тегов <a>, <link>, и т.д.); отправка формы (с помощью тега <form>) с методом, указанным в атрибуте method, и данными из самой формы, которые могли отправляться в нескольких форматах, например: application/x-www-form-urlencoded, multipart/form-data (формат для отправки файлов).

В современных веб-приложениях, помимо вышеописанных способов и форматов взаимодействия клиента и сервера, появилось большое количество новых технологий. Для отправки запросов часто используются интерфейсы fetch и XMLHttpRequest, предоставляемые языком JavaScript. Сами же запросы зачастую могут содержать более сложные форматы данных, например: текстовые форматы, такие как JSON, XML; бинарные форматы, такие как protocol buffers, сериализованные объекты при помощи serialize в языке PHP, сериализованные объекты при помощи pickle в языке Python, сериализованные объекты при помощи Serializable в языке Java.

Такое разнообразие обусловлено несколькими факторами:

- технологии разработки и встроенные в них возможности обработки разных форматов данных;
- большое разнообразие данных, посредством которых пользователь должен коммуницировать с серверной частью веб-приложения;
- безопасность обработки пришедших данных;
- удобность их дальнейшего использования;

С появлением новых веб-технологий, помимо удобства, разработчики и пользователи столкнулись с новыми угрозами безопасности. В 2006 году появился термин “десериализация недоверенных данных”[1]. Этот недостаток также называют “небезопасной десериализацией”. Недостаток такого типа имеет высокий уровень опасности и может приводить к серьезным уязвимостям, таким как выполнение произвольного кода, отказ в обслуживании, чтение и запись локальных файлов [2]. Данный недостаток по сей день входит в классификацию OWASP Top Ten, в которой описаны наиболее часто встречаемые недостатки веб–приложений [3].

2. Методы поиска уязвимости типа “небезопасная десериализация”

Уязвимость типа “небезопасная десериализация” – это уязвимость серверной части приложения. Методы, позволяющие автоматически обнаруживать уязвимости такого рода, можно разделить на два класса.

- Анализ серверного кода, работающий в модели “белого ящика” (“white box”). Анализаторы такого типа имеют доступ к серверному коду; они, как правило, осуществляют преимущественно статический анализ.
- Анализ, работающий без доступа к серверному коду, в режиме “чёрного ящика” (“black box”). Средства, работающие таким образом, могут анализировать серверную часть, лишь взаимодействуя с ней – то есть, в случае веб-приложений, посылая на сервер запросы, и анализируя пришедшие ответы. Таким образом, это всегда динамический анализ.

Исследователи отмечают, что для современных средств поиска уязвимостей в веб-приложениях (как первого, так и второго типов) задача поиска уязвимостей типа "небезопасная десериализация" до сих пор остаётся сложной, существующие средства не предоставляют общего решения этой задачи [4], [5].

Анализаторы первого типа имеют полный доступ к информации об устройстве серверной части приложения; в связи с этим они имеют заведомо большее покрытие кода, чем анализаторы второго типа, и имеют больше шансов найти присутствующие недостатки. К анализаторам такого типа относятся SerialDetector [4], RIPS [6], а также такие промышленные инструменты, как PT Application Inspector [7], Solar appScreener [8], Fortify Static Code Analyzer [9], Checkmarx [10]. Недостатком данного типа анализаторов является необходимость предоставления кода серверной части приложения – в реальных условиях это не всегда возможно. На основе инструмента такого типа невозможно проводить эксперименты с поиском потенциально присутствующих уязвимостей на большой массе реальных сайтов, так как в большинстве случаев невозможно получить доступ к их серверному коду.

Анализаторы второго типа работают, как уже было упомянуто, в модели "чёрного ящика". В данной статье задача рассматривается именно в такой постановке – в условиях, когда доступ к исходному коду серверной части отсутствует. К анализаторам второго типа относятся такие средства, как PT BlackBox Scanner [11], Acunetix [12], Burp Suite Pro Scanner[13], HCL AppScan [14], Detectify [15], Qualys Web Application Scanning [16]. В связи с тем, что

[1] CWE-502: Deserialization of Untrusted Data (https://cwe.mitre.org/data/definitions/502.html)

анализаторы этого типа работают без доступа к коду сервера, они сталкиваются с рядом сложностей.

Для валидации данного недостатка нужно автоматически, по ответу сервера, распознавать факт выполнения десериализации данных на сервере. Помимо этого, для генерации вектора атаки необходимо иметь какие-то сведения о технологиях, используемых на серверной стороне, как минимум требуется знать, какой использовался язык программирования и какие существуют программные классы [17, 18]. Без доступа к серверному коду это нетривиально.

Следующим важным пунктом является поиск точек ввода данных на сервер, то есть URL конечной точки, а также параметры, тело запроса, и, возможно, заголовки. Современные средства могут найти все точки ввода данных, отправляемых из HTML-разметки, но не из JavaScript-кода [19. 20]. Для решения данной задачи обычно используют два подхода: динамический или статический анализ клиентского кода. Оба подхода имеют свои достоинства и недостатки, однако для современных веб-приложений ни один из этих подходов не решает задачу в общем виде [21, 22].

В данной статье предлагается иной подход, заключающийся в создании синтаксических шаблонов классов и разработке инструмента, выявляющего данные классы в клиентском коде веб-приложений.

3. Описание метода

В исходном клиентском коде веб-страницы выявляются сериализованные объекты, в том числе закодированные одной или более кодировок. Предполагается, что, если сериализованный объект находится в клиентском коде, значит он должен быть отправлен на сервер и десериализован. В общем случае это не всегда так, и в ходе исследования были обнаружены случаи, когда сериализованные объекты, присутствующие на странице, на самом деле не будут отправлены на сервер или не будут десериализованы после получения сервером. Тем не менее, в данной работе мы будем в дальнейшем исходить из того предположения, что наличие сериализованного объекта на клиентской стороне говорит о том, что скорее всего он будет отправлен на сервер, и в большей части выявленных классов (о них речь пойдёт ниже) это действительно так.

Далее, для сериализованных объектов, находящихся внутри JavaScript-кода, составляются синтаксические шаблоны, которые содержат какие-то характерные признаки для кода, в котором использован сериализованный объект. Затем, с помощью разработанного инструментального средства, полученные шаблоны выявляются в коде других веб-приложений.

Преимуществом предложенного подхода является то, что, при отнесении приложения к уже существующему классу, становится известна большая часть HTTP-запроса, который нужно выполнить, чтобы передать данные в функцию десериализации на сервере. Путь URL и ключи параметров обычно совпадают, так как используется одно и то же программное обеспечение. Преимуществом подхода также является то, что уже есть некоторые данные о сервере, и нет необходимости отправлять большое множество векторов атаки.

3.1 Алгоритм выявления сериализованных объектов внутри веб-страницы

Для большинства форматов сериализации можно написать достаточно точные сигнатуры. Например, сериализованные Java-объекты начинаются на байты “\xAC\xED\x00\x05”. То есть сигнатуре достаточно проверить, начинаются ли данные с “магической” константы. Сериализованные PHP-объекты (функция serialize) выглядят примерно таким образом: a:1:{i:20041001103319;s:4:\"test\";}. Для такого формата можно написать как “честный” детектор, работающий по тем же правилам, что и реализация unserialize из PHP, так и

несложную эвристическую сигнатуру, которая будет правильно работать в подавляющем числе случаев.

Эти объекты, когда они встречаются в веб-приложениях, обычно присутствуют не в “сыром” виде. Например, cookie допускают только печатные ASCII-символы, в то время как многие форматы сериализации могут содержать произвольные 8-битные байты. Также некоторые объекты могут быть очень большими, и поэтому их сжимают, чтобы они могли поместиться в те же cookie.

Таким образом, нередки подобные цепочки вложенности: Java Serializable → gzip → Base64. Вместо Base64 может быть Base32, URL encoding, 16-ричное кодирование (hex encoding), и т.д. Для сжатия может также использоваться zlib, DEFLATE, и т.д.

Для автоматического поиска сериализованных объектов с учетом таких “вложенностей” был разработан алгоритм, который будет сейчас описан. Входными данными для алгоритма являются HTTP-ресурсы (HTML-страницы, JavaScript-файлы) и их заголовки.

На первом шаге алгоритм выделяет строки-“кандидаты”, которые могут содержать в себе сериализованные объекты. В случае cookie это их значения непосредственно. JavaScript-файлы разбиваются на лексемы, и из них выделяются строковые константы. HTML-страницы синтаксически разбираются, из них извлекаются значения атрибутов (наиболее интересными являются атрибуты с именем “value” у элементов управления форм и атрибуты с именами, начинающимися с префикса “data-”). Инлайновые JavaScript-программы (текст которых непосредственно содержится в HTML-разметке страницы) обрабатываются вышеуказанным способом.

Для каждой такой строки-кандидата применяется рекурсивный алгоритм, псевдокод которого показа на листинге 1.

```
def find_chain(s, callback, max_depth):
    if max_depth <= 0:
          return
    for trans in TRANSFORMATIONS:
        if res := trans(s):
            find_chain(res, callback, max_depth-
1)
    for check in CHECKS:
        if check(s):
            callback(s)
```

Листинг 1: Алгоритм поиска цепочек
Listing 1: Chain search algorithm

`CHECKS` содержит определения функций, проверяющих строку на содержание конкретного типа сериализованного объекта: Java-объект, PHP-сериализованный объект, т.д. `TRANSFORMATIONS` содержит функции, производящие преобразования: вышеупомянутые кодирования вроде Base64, алгоритмы сжатия, синтаксический разбор JSON, и т.д.

То есть, например, если строка может быть раскодирована каким-то образом, то алгоритм вызывается рекурсивно для раскодированной строки. Если строка содержит что-то, что распознает сигнатура сериализованного объекта, то проверка завершается успехом, и вызывается функция `callback`.

H4sIAAAAAAAAAFvzloG1oLiIQTArs ... BZvHUM4CIisqAOqbFhijAQAA (base64)
'\x1f\x8b\x08\x00\x00\x00\x00\x00\x00\ … \xa3\x01\x00\x00' (gzip)
'\xac\xed\x00\x05psr\x00\x11java.util.HashMap\x05\x07\ … \x00\x00\x00w\x04\x00\x00\x00\x00xx' (Java Serializable)

3.2 Выявление контекста сериализованного объекта внутри веб–страницы

В проведенном исследовании важную роль играет анализ JavaScript-кода, подключенного на страницу. Поэтому неотъемлемым шагом исследования являлась реализация алгоритма, определяющего контекст нахождения сериализованного объекта внутри исходного кода веб-страницы. С помощью разработанного алгоритма были получены веб-страницы, на которых были найдены сериализованные объекты внутри JavaScript-кода для выполнения следующего шага работы.

Алгоритм принимает на вход сериализованный объект и контент веб-ресурса. Веб-ресурс представляет из себя HTML-разметку страницы или JavaScript-программу, которая подключается на веб-страницу.

При получении JavaScript-кода алгоритм выявляет наличие сериализованного объекта с помощью вхождения подстроки.

При получении HTML-разметки, она разбивается на лексемы, и происходит обход всех элементов. Проверяются значения всех атрибутов на равенство сериализованному объекту. При встрече тега script, подключенный на страницу JavaScript-код преобразуется в абстрактное синтаксическое дерево при помощи парсера Babel[2]. В полученном дереве происходит проверка всех строковых литералов на равенство сериализованному объекту. Такой способ необходим из-за того, что некоторые символы могут быть экранированы или закодированными специальными сущностями внутри HTML-разметки.

Предложенный алгоритм был применен на страницах более 50000 веб-приложений из списка Alexa Top 1 Million, а также на страницах 20000 веб-приложений из программ Bug Bounty. После выполнения данного шага была получена статистика того, в каких местах располагаются сериализованные объекты на страницах веб-приложения (табл. 1).

Табл 1. Статистика контекста найденных сериализованных объектов
Table 1. Statistics of found serialized objects contexts

Контекст сериализованного объекта	Количество
Сериализованные объекты в JavaScript-коде	12997
Сериализованные объекты внутри тега input	9804
Сериализованные объекты внутри тегов data*	2293
Сериализованные объекты внутри тега option	3
Сериализованные объекты внутри тегов script с	74
Сериализованные объекты внутри других тегов	476
Сериализованные объекты в других местах	46

Как видно из статистики табл. 1, 50% сериализованных объектов находится внутри JavaScript-кода, что подтверждает интерес к исследованию сериализованных объектов, найденных в JavaScript-коде.

[2] Babel Parser spec. (https://babeljs.io/docs/en/babel-parser)

3.3 Формирование и выявление классов сериализованных объектов внутри JavaScript-кода

Для классификации сериализованных объектов проводился ручной просмотр найденных примеров и окружающего их кода. В этом коде выделялись повторяющиеся в разных веб-приложениях шаблоны, например:

- уникально названная переменная, содержащая сериализованный объект;
- определенная функция, вызываемая с переменной, содержащей сериализованный объект.

```
var ajaxurl = '/wp-admin/admin-ajax.php ';
var true_posts =
'a:63:{s:14:"posts_per_page";i:10..."order "; s :4:"DESC";}';
var current_page = 1;
var max_pages = '3 ';
...
$('#true_loadmore').click(function() {
    $(this).text('Loading ...') ;
    var data = {
        'action': 'loadmore',
        'query': true_posts,
        'page': current_page
    };
    $.ajax({
           url: ajaxurl,
        data: data,
        type: 'POST',
        success: function(data) {
            ...
        }
    });
});
```

Листинг 2: Пример кода одного из веб-приложений класса LoadMore
Listing 2: Code example from web-application with LoadMore class

В листинге 2 характерными являются следующие особенности:

- сериализованный объект содержится в переменной с идентификатором `true_posts`;
- переменная `true_posts` используется внутри объекта, отправляемого на сервер с помощью функции `$.ajax()`.

Для разметки всех найденных сериализованных объектов выполнялся итеративный процесс:

- выявление особенностей кода;
- создание синтаксического шаблона для выявления найденных особенностей;
- применение классификации для разметки элементов выборки, попадающих под созданные шаблоны;
- процесс повторяется для тех объектов, которые остались неразмеченными.

Алгоритм классификации заключается в следующем:

- происходит обход дерева;
- встречая вершины определенных типов, алгоритм производит сопоставление найденных вершин с разработанными шаблонами;
- при успешном сопоставлении классификатор подает на выход основному алгоритму название класса, к которому относится полученный экземпляр сериализованного объекта.

Предложенный алгоритм был протестирован на страницах, полученных в ходе работы алгоритма из подраздела 3.2. В табл. 2 приведена статистика наиболее интересных классов. Интересными были сочтены, прежде всего, классы, в которых были обнаружены недостатки. Кроме того, как таковые были выбраны наиболее представительные классы. А конкретнее, те, в которые входит большое (> 5) количество приложений. Наконец, интересными были также сочтены классы, имеющие нетривиальную цепочку кодировок (более чем одна кодировка перед сериализацией) – поскольку обнаружение сериализованных объектов с такой цепочкой требует более сложного алгоритма поиска.

Всего было выделено 60 классов, из которых 23 являются нетривиальными (более одного приложения) и 37 тривиальных. Более полную статистику можно найти в репозитории на GitHub[3].

Табл 2. Статистика найденных классов
Table 2. Statistics of found classes

Класс	Приложений	Страниц	Цепочка кодировок
LoadMoreWordPress	65	2273	urldecode, phpser
JCCatalogBitrixOnlineShopSoft	33	128	base64, phpser
Settings	17	33	phpser
QuizSoft	15	919	phpser
SimpleAjaxManagerWordPress	7	935	base64, phpser
MsgLogVAR	4	1528	base64, base64, base64, phpser
InitState	1	101	base64, zlib, phpser
GdnMeta	1	67	urldecode, urldecode, json, phpser

3.4 Описание некоторых классов

В табл. 3, как пример, приведено описание некоторых классов, которые были выбраны как интересные выше, в подразделе 3.3. Представленные сигнатуры описаны на псевдоязыке на основе JavaScript, при этом используются следующие специальные обозначения:

- * – на месте этого символа может быть любое количество (может быть нулевым) любых символов, допустимых в идентификаторе;
- | – один из предложенных вариантов;
- <*> - любое обращение к свойствам (например, .property, [*]);
- SERIALIZED_OBJ – сериализованный объект в той кодировке, в которой он был найден на странице;
- OBJ_WITH_SERIALIZE_OBJ_INSIDE – JavaScript-объект содержащий внутри себя SERIALIZED_OBJ.

[3] Репозиторий со статистикой: https://github.com/miron6/Insecurely-serialized-objects-research

Табл 3: Примеры описания некоторых классов
Table 3: Examples of descriptions of some classes

LoadMoreWordPress	
Сигнатура	**Пример кода**
(true_posts* | ajax_query* | loadmore*) = SERIALIZED_OBJ | OBJ_WITH_SERIALIZE_OBJ_INSIDE	var true_posts = 'a:63:{s:13:"category_name";s:7:"betsoft";...;s:5:"order";s:4:"DESC";}';
JCCatalogBitrixOnlineShopSoft	
Сигнатура	**Пример кода**
obbx_* = new JCCatalog*(OBJ_WITH_SERIALIZE_OBJ_INSIDE)	var obbx_117848907_56410 = new JCCatalogElement({...,'SKU_PROPS':'YTozOntpOjA7czo5OiJWWVNPVEFfTU0iO2k6MTtzOjc6IlNUT1JPTkEiO2k6MjtzOjU6IlRTVkVUIjt9', ...});
Settings	
Сигнатура	**Пример кода**
setting = SERIALIZED_OBJ	setting = 'a:75:{s:6:"action";s:9:"save_edit";s:4:"name"...s:8:"moduleid";s:3:"154";}'
QuizSoft	
Сигнатура	**Пример кода**
window.qmn_quiz_data<*> = OBJ_WITH_SERIALIZE_OBJ_INSIDE	window.qmn_quiz_data["1"] = {..., "answer_array":"a:5:{i:0;a:3:{i:0;s:25:"...";i:1;d:0;i:2;i:0;}}", ...};
SimpleAjaxManagerWordPress	
Сигнатура	**Пример кода**
samAjax = OBJ_WITH_SERIALIZE_OBJ_INSIDE	var samAjax = {..."clauses":"YTo0OntzOjI6IldDIjtzOjExMzg6IihJRihzYS5hZF91c2VycyA9IDAsIFRSVUUsIChzYS5hZF91…IDApIjt9","doStats":"1",...};
MsgLogVAR	
Сигнатура	**Пример кода**
MSLOG_var = SERIALIZED_OBJ	var MSLOG_var = "V1ZSdmVFN...UFE9PQ==";
InitState	
Сигнатура	**Пример кода**

___INITIAL_STATE__ = OBJ_WITH_SERIALIZE_OBJ_INSIDE	___INITIAL_STATE__={"intl":{"defaultLocale":"en", ...}, ..., "description":"eJyN...k5mw==",...};
GdnMeta	
Сигнатура	**Пример кода**
gdn.meta = OBJ_WITH_SERIALIZE_OBJ_INSIDE	gdn.meta={"AnalyticsTask": "tick",..., "[{\"HashType\":\"md5\",\"TestMode\":false,\"Trusted\":\"1\",..."}

4. Поиск уязвимостей

4.1 Методология, использованная при поиске уязвимостей

Для валидации того, происходит ли на сервере десериализация клиентских данных, использовались следующие методы.

- Отправка оригинального объекта, найденного на странице, а также некорректного, с точки зрения формата сериализации, объекта.
- Отправка измененного, но синтаксически корректного объекта. Например, добавление нового поля в словарь или изменение строкового литерала незначительным образом. Данная проверка нужна, чтобы отбросить случаи, когда на серверной части приложения сериализованный объект проходит проверку на строгое равенство с константной строкой.

При использовании всех методов выше сравнивались и анализировались HTTP-ответы.

Для веб-приложений, входящих в программу Bug Bounty, производилась попытка эксплуатации уязвимостей на самих приложениях. Для остальных приложений производился поиск компонентов с открытым исходным кодом, на основе которых они реализованы. В случае нахождения таковых проверялась возможность эксплуатации уязвимости путем ручного анализа кода и атака на стендовые приложения, сделанные на их основе.

В случае, когда подобные проверки не давали однозначных результатов, отправлялись сериализованные объекты стандартных классов используемого языка программирования. Например, в случае PHP таковым являлся класс DateTime. Отправлялся корректный и некорректный сериализованные объект. Десериализация некорректного PHP-объекта DateTime приводит к фатальной ошибке сервера, что являлось индикатором происходящей на сервере десериализации.

4.2 Найденные уязвимости

Класс Settings: было выявлено, что в данный класс входят приложения, сделанные на основе PHP-фреймворка OpenCart с использованием одного и того же плагина. В данном плагине присутствует недостаток небезопасной сериализации.

- Было реализовано стендовое приложение с использованием тех же технологий.
- Была найдена цепочка гаджетов с использованием классов базового фреймворка, позволяющая выполнить произвольный код на сервере.
- Данная уязвимость не была публично известной, поэтому разработчики продукта были оповещены о наличии данной уязвимости, а также был получен идентификатор

уязвимости CVE-2022-24108[4].

Класс SimpleAjaxManagerWordpress: было выявлено, что приложения данного класса используют CMS WordPress (является одной из самых распространенных CMS в интернете [23, 24]) и плагин Simple Ajax Manager.

- Было реализовано стендовое приложение с использованием тех же технологий.
- Были найдены две уязвимости: небезопасная десериализация и SQL-инъекция через поля сериализованного объекта.
- Найденная SQL-инъекция не была публично известной уязвимостью, поэтому был отправлен запрос на получение идентификатора CVE в организацию WPScan. Уязвимость не получила нового идентификатора уязвимости, но информация о ней была добавлена в существующую запись о недостатке “Десериализации недоверенных данных”[5].

Класс LoadMoreWordpress: приложения данного класса используют CMS WordPress, но технология, где используется небезопасная десериализация, не является плагином. Код, где выполняется уязвимый запрос, был приведен на одном из форумов для разработчиков.

- Страницы данного класса были найдены на ряде веб-приложений, входящих в программу Bug Bounty.
- При помощи эксплуатации небезопасной десериализации получилось добиться выполнения произвольного кода на серверной стороне для всех этих приложений.
- Был отправлен отчет об уязвимости на один из главных агрегаторов Bug Bounty программ HackerOne [25].

5. Заключение

В данной статье предложен метод поиска недостатка “Десериализация недоверенных данных” при помощи поиска сериализованных объектов внутри исходного кода веб-страницы, создания синтаксических шаблонов, основанных на особенностях кода, содержащего в себе сериализованный объект, и последующей классификации страниц веб-приложений.

Данный метод был протестирован на страницах более чем 70000 веб-приложений. Получены синтаксические шаблоны для 60 классов, которые могут быть использованы при дальнейшем автоматизированном анализе веб-приложений. На приложениях трех найденных классов получилось установить наличие искомого недостатка. В ходе исследования был получен идентификатор уязвимости – CVE, дополнена информация о другой общеизвестной уязвимости и отправлен отчет в программу Bug Bounty для целого ряда приложений, имеющих схожий недостаток.

Список литературы / References

[1] Nath K., Dhar S., Basishtha S. Web 1.0 to Web 3.0 - Evolution of the Web and its various challenges. In Proc. of the International Conference on Reliability Optimization and Information Technology (ICROIT), 2014, pp. 86-89.

[2] Koutroumpouchos N., Lavdanis G. et al. ObjectMap: detecting insecure object deserialization. In Proc. of the 23rd Pan-Hellenic Conference on Informatics (PCI'19), 2019, pp/ 67-72.

[3] Bach-Nutman M. Understanding The Top 10 OWASP Vulnerabilities. arXiv preprint arXiv:2012.09960, 2020, 4 p.

[4] CVE-2022-24108 (https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-24108)

[5] Wordpress: Simple Ads Manager vulnerability page (https://wpscan.com/vulnerability/38787b49-c19c-49db-925a-69e2c9cf7a43)

[4] Shcherbakov M., Balliu M. SerialDetector: SerialDetector: Principled and Practical Exploration of Object Injection Vulnerabilities for the Web. In Proc. of the Network and Distributed Systems Security (NDSS) Symposium, 2021, 18 p.

[5] Sabatini A. Evaluating the Testability of Insecure Deserialization Vulnerabilities via Static Analysis. Tesi di Laurea Magistrale in Computer Science and Engineering. Politecnico Milano, 2022, 72 p.

[6] Dahse J., Holz T. Simulation of Built-in PHP Features for Precise Static Code Analysis. In Proc. of the Network and Distributed System Security (NDSS) Symposium, 2014, 15 p.

[7] PT Application Inspector. Available at: https://www.ptsecurity.com/ru-ru/products/ai/, accessed 02.04.2023.

[8] Solar appScreener. Available at: https://rt-solar.ru/products/solar_appscreener/, accessed 02.04.2023.

[9] Fortify Static Code Analyzer. Available at: https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer, accessed 02.04.2023.

[10] Checkmarx. Available at: https://checkmarx.com/, accessed 02.04.2023.

[11] PT BlackBox Scanner. Available at: https://bbs.ptsecurity.com/, accessed 02.04.2023.

[12] Acunetix. Available at: https://www.acunetix.com/, accessed 02.04.2023.

[13] Burp Suite's web vulnerability scanner. Available at: https://portswigger.net/burp/vulnerability-scanner, accessed 02.04.2023.

[14] HCL AppScan. Available at: https://www.hcltechsw.com/appscan, accessed 02.04.2023.

[15] Detectify Web Application Scanning. Available at: https://detectify.com/product/application-scanning, accessed 02.04.2023.

[16] Qualys Web Application Scanning. Available at: https://www.qualys.com/apps/web-app-scanning/, accessed 02.04.2023.

[17] Esser S. Shocking News in PHP Exploitation. Slides of the Presentation at the Power of Community Conference (POC), 2009. Available at: https://www.suspekt.org/wp-content/uploads/2019/12/POC2009-ShockingNewsInPHPExploitation.pdf, accessed at 04.03.2023.

[18] Esser S. Utilizing code reuse or return oriented programming in PHP applications. Slides of the Presentation at the BlackHat USA Conference, 2010. Available at: Available at: https://media.blackhat.com/bh-us-10/presentations/Esser/BlackHat-USA-2010-Esser-Utilizing-Code-Reuse-Or-Return-Oriented-Programming-In-PHP-Application-Exploits-slides.pdf, accessed at 04.03.2023.

[19] Сигалов Д.А., Хашаев А.А., Гамаюнов Д.Ю. Обнаружение серверных точек взаимодействия в веб-приложениях на основе анализа клиентского JavaScript-кода. Прикладная дискретная математика вып. 53, 2021 г., стр. 32-54. / Sigalov D.A., Khashaev A.A., Gamayunov D.Yu. Detecting server-side endpoints in web applications based on static analysis of client-side JavaScript code. Prikladnaya Diskretnaya Matematika, issue 53, 2021, pp. 32-54 (in Russian).

[20] Раздобаров А.В., Петухов А.А., Гамаюнов Д.Ю. Проблемы обнаружения уязвимостей в современных веб-приложениях. Проблемы информационной безопасности. Компьютерные системы, вып. 4, 2015 г., стр. 64-69. / Razdobarov A.V., Petukhov A.A., Gamayunov D.Yu. Problems overview for modern web applications vulnerabilities discovery. Information Security Problems. Computer Systems, issue 4, 2015, pp. 64-69 (in Russian).

[21] Pradel M., Schuh P., Sen K. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In Proc. of the IEEE/ACM 37th IEEE International Conference on Software Engineering, 2015, pp. 314-324.

[22] Park C., Ryu S. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In Proc. of the 29th European Conference on Object-Oriented Programming (ECOOP), 2015, pp. 735-756.

[23] Lin J., Sayagh M., Hassan A.E. The Co-evolution of the WordPress Platform and its Plugins. ACM Transactions on Software Engineering and Methodology, vol. 32, issue 1, 2023, article no. 19, 24 p.

[24] Patel S.K., Rathod V.R., Prajapati J.B. Performance Analysis of Content Management Systems - Joomla, Drupal and WordPress. International Journal of Computer Applications, vol. 21, issue 4, 2011, pp 39-43.

[25] Walshe T., Simpson A. An Empirical Study of Bug Bounty Programs. In Proc. of the IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF), 2020, pp. 35-44.

Информация об авторах / Information about authors

Денис Дмитриевич МИРОНОВ – исследователь проблем безопасности. Сфера научных интересов: статический анализ программ для задач безопасности приложений, безопасность веб-приложений.

Denis Dmitrievich MIRONOV – Security Researcher. Research interests: static program analysis for application security, web application security.

Даниил Алексеевич СИГАЛОВ – младший научный сотрудник лаборатории математических проблем компьютерной безопасности факультета ВМК МГУ, исследователь проблем безопасности в ООО «Солидсофт». Сфера научных интересов: безопасность веб-приложений, статический и динамический анализ программ для задач безопасности приложений.

Daniil Alekseevich SIGALOV – Junior Researcher of Laboratory of Mathematical Problems of Computer Cecurity at the CMC Faculty of Lomonosov Moscow State University, Security Researcher at SolidSoft LLC. Research interests: web application security, static and dynamic program analysis for application security.

Максим Петрович МАЛЬКОВ – ведущий программист лаборатории математических проблем компьютерной безопасности факультета ВМК МГУ, исследователь безопасности в ООО «Солидсофт». Сфера научных интересов: безопасность веб-приложений.

Maxim Petrovich MALKOV – Lead Programmer of Laboratory of Mathematical Problems of Computer Security at the CMC Faculty of Lomonosov Moscow State University, Security Researcher at SolidSoft LLC. Research interests: web application security.