



Московский государственный университет имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Кафедра автоматизации систем и вычислительных комплексов

Павлов Дмитрий Сергеевич

**Эксплуатация уязвимостей десериализации  
в веб-приложениях на языке PHP  
(Hypertext Preprocessor)**

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

**Научный руководитель:**

к.ф.-м.н., с.н.с

Гамаюнов Д.Ю.

Москва, 2019

## Аннотация

В работе исследуется возможность применения статического анализа для построения входных данных, эксплуатирующих недостаток, связанный с возможностью внедрения объектов на языке PHP (PHP Object Injection, POI).

В работе описан метод для автоматизированного поиска цепочек для эксплуатации недостатка POI и инструментальное средство, реализующее его. Применимость метода и средства подтверждена экспериментальными исследованиями на реальных проектах, что также отражено в данной работе.

# Содержание

Введение.....	4
Сериализация.....	4
„Опасная” десериализация .....	5
PHP Object Injection.....	6
Property Oriented Programming.....	8
Постановка задачи.....	11
Цель работы .....	11
Задачи .....	11
Обзор существующих инструментальных средств эксплуатации POI .....	12
RIPS.....	12
PHPGGC.....	14
Вывод .....	15
Формальные обозначения .....	16
Анализ задачи .....	18
Задачи, возникающие при поиске потенциальных цепочек .....	18
Задачи, возникающие при оценке применимости потенциальных цепочек .....	18
Реализация инструментального средства .....	20
Поиск потенциальных цепочек.....	20
Оценка применимости потенциальных цепочек .....	21
Экспериментальное исследование.....	23
Проверка работоспособности цепочки .....	23
Оценка качества поиска цепочек .....	25
Экспериментальное исследование на реальной библиотеке.....	28
Результаты .....	30
Приложение А.....	31
Список литературы .....	34

# Введение

## Сериализация

Современные веб-приложения реализуют клиент-серверную архитектуру, в которой предусмотрены клиентские устройства, отправляющие запросы на сервер, и сам сервер, обрабатывающий запросы и хранящий основной объем данных для этого. Запросы между клиентом и сервером передаются по протоколу HTTP [1].

Изначально в протоколе не было предусмотрено понятие сессии работы клиента с сервером и механизма поддержки состояний между HTTP-запросами от одного и того же клиента; позднее был разработан документ IETF RFC 6265 [2], описывающий механизм cookie.

Cookie позволяют реализовать два подхода по хранению состояния работы клиента в приложении:

1. Хранить все данные о состоянии работы клиента в приложении на сервере, а клиенту передавать уникальный идентификатор сессии, на основании которого можно извлечь эти данные.
2. Хранить все данные на стороне клиента в cookie, закодировав их в строку.

Первый подход характерен для монолитных нераспределенных ненагруженных приложений. Для современных нагруженных распределенных приложений, для которых характерна микросервисная архитектура [3], сохранение состояния на клиенте часто является наиболее оправданным решением. Соответственно, возникает задача о том, как упаковать в общем случае произвольную иерархическую структуру данных, отражающую состояние сессии пользователя приложения, в строку, пригодную, например, для передачи между клиентом и сервером в cookie или HTTP-заголовке.

**Сериализация** — процесс преобразования структуры данных или объекта (экземпляра класса) в формат, который может быть сохранен (на диске, в базе данных), передан (по сети, между процессами) и восстановлен обратно (**десериализован**).

Для решения данной задачи можно использовать различные стандартные форматы хранения и передачи данных такие, как XML [4], JSON [5]. При выборе их для использования в приложении разработчику придется реализовать собственные

процедуры сериализации необходимых структур в требуемую нотацию и десериализации обратно.

Наряду с универсальными форматами сериализации, которые требуют реализации разработчиками собственных функций упаковки и восстановления объектов, многие языки программирования (например PHP, Java, Object Pascal, OCaml) предлагают собственный встроенный формат сериализации и готовые функции по упаковке и восстановлению даже самых сложных структур данных.

В PHP с помощью стандартных функций *serialize* и *unserialize* можно сериализовывать и десериализовывать все встроенные в язык типы данных, кроме ресурсов. Сериализованные объекты представляют собой бинарную строку в формате, описанном в Приложении А (в сериализованной строке могут присутствовать нулевые байты, во всех примерах они будут обозначаться символом  $\emptyset$ ).

## „Опасная” десериализация

Сериализованные объекты требуется где-то хранить: либо на сервере, в файлах или базе данных, либо на стороне клиента — в локальном хранилище (механизм браузера *localStorage*) или файлах *cookie*. Хранение данных на стороне клиента и последующее их использование без должных проверок на корректность (например, на целостность) может привести к появлению в приложении уязвимостей.

При использовании универсальных форматов сериализации атакующий может изменить значение переменной, которая влияет на проверки прав доступа и привилегий, что может привести к уязвимостям авторизации. OWASP Top 10 2017 [6] выделяет уязвимости десериализации в отдельную категорию — А8. Рассмотрим пример ниже (см. Рисунок 1).

```
1  <?php
2  function webshell()
3  {
4      $data = base64_decode($_COOKIE['info']);
5      $user_info = json_decode($data);
6      var_export($user_info);
7      if ($user_info['admin'] === true) {
8          system("$_GET['cmd']");
9      }
10 }
```

Рисунок 1

В этом примере десериализованные данные, полученные от пользователя, влияют на поток управления выполняемого кода. Атакующий может подставить произвольное значение в параметр `$user_info['admin']`, в частности `true`, таким образом обойдя проверку и получив удаленное выполнение кода на сервере веб-приложения (Remote Code Execution).

В случае, если контролируемые клиентом данные не влияют на поток управления и не передаются в качестве аргумента в критические функции, их изменение не приведет к нарушению целостности приложения. **Критические функции** — функции для работы с файловой системой, СУБД или позволяющие выполнить произвольный код на сервере с веб-приложением.

## PHP Object Injection

При использовании внутреннего формата сериализации PHP для хранения данных из-за особенностей языка у атакующего появляется больше возможностей. Начиная с пятой версии PHP стал поддерживать объектно-ориентированную парадигму программирования. Класс определяется набором свойств и функций, которые называют полями и методами. В процессе сериализации сохраняются значения полей объекта, которые потом можно восстановить с помощью десериализации.

Возможность внедрения объекта PHP (PHP object injection, POI [7]) — тип недостатка в приложении, когда контролируемые пользователем данные без должных проверок попадают в функцию `unserialize`. Так как PHP позволяет десериализовывать объекты, атакующий может передать специально подготовленную строку в вызов функции `unserialize`, и получить объект нужного ему класса в контексте<sup>1</sup> приложения.

Каждый класс в PHP имеет набор зарезервированных методов, называемых **магическими** [8], которые выполняются автоматически при определенных событиях, происходящих с объектом в ходе выполнения программы. События могут быть разных типов: удаление<sup>2</sup> объекта, преобразование объекта к строке, вызов несуществующего

---

<sup>1</sup> PHP является интерпретируемым языком, что позволяет создавать во время выполнения объекты классов, которые находятся в области видимости (т.е. в том же файле, в подключенных файлах, либо любой класс проекта если подключен автозагрузчик). Все объекты и данные приложения, загруженные в память во время обработки конкретного запроса, будут называться контекстом выполнения запроса или просто контекстом.

<sup>2</sup> В PHP реализована динамическая сборка мусора через подсчет ссылок на объекты. Значения переменных на самом деле хранятся в контейнерах, а переменные являются ссылками на них. Сборщик мусора следит за количеством ссылок на контейнеры, и когда оно достигает нуля, удаляет контейнер вместе с содержащимся в нем объектом, освобождая память.

метода<sup>1</sup> и так далее. События можно разделить на те, которые обязательно произойдут в течение жизни объекта (например, рано или поздно объект будет освобожден из памяти) и которые зависят от логики выполнения программы (например, приведение переменной к строке может и не случиться, если не предусмотрено разработчиком). Магические методы, выполняющиеся при этих событиях, называются соответственно контекстно-независимыми или контекстно-зависимыми.

Заметим, что если в магическом методе какого-либо объекта, определенного в контексте выполнения запроса, происходит вызов критической функции, то нарушителю **достаточно** передать приложению именно такой сериализованный объект, сформированный особым образом: в итоге при его десериализации в приложении будет вызвана критическая функция с контролируемым злоумышленником данными (напомним, что нарушитель контролирует все поля объекта при его передаче в сериализованном виде).

В данной работе будет рассматриваться контекстно-независимый случай, который не сложно обобщить до контекстно-зависимого. Рассмотрим уязвимость на примере (см. Рисунок 2).

```
1  <?php
2  class Worker
3  {
4      private $hook;
5      //Some PHP code
6      function __wakeup()
7      {
8          if (isset($this->hook)) eval($this->hook);
9      }
10 }
11 // Some PHP code
12 $user_data = unserialize($_COOKIE['data']);
```

Рисунок 2

В строке 12 производится вызов функции *unserialize* с данными, контролируемым пользователем, без должных проверок. В строке 6 объявлен магический метод *\_\_wakeup*, который вызывается при десериализации объекта класса *Worker*. Так как атакующий при десериализации может контролировать значения полей десериализованного объекта, то

---

<sup>1</sup> Так как PHP является динамически типизированным языком, то переменная может указывать на произвольный объект, и возможность вызова метода через переменную определяется во время интерпретации — при выполнении конкретной инструкции.

при эксплуатации уязвимости из примера можно выполнить произвольный код на сервере с веб-приложением. Для этого надо в запросе передать cookie с названием *data* и следующим значением:

*O%3A6%3A%22Worker%22%3A1%3A%7Bs%3A12%3A%22%00Worker%00hook%22%3Bs%3A10%3A%22phpinfo%28%29%3B%22%3B%7D*, которая раскодируется веб-сервером в строку *O:6:"Worker":1:{s:12:"0Worker0hook";s:10:"phpinfo()";}*. В результате десериализации будет вызван магический метод *\_\_wakeup*, внутри которого выполнится *eval*, с контролируемым атакующим кодом (*phpinfo* в примере).

В зависимости от того, какие критические функции вызываются в магических методах объектов, доступных в контексте обработки запросов приложением, уязвимость рассматриваемого класса позволит провести различные атаки: внедрение операторов SQL, удаленное выполнение кода, запись в файлы, отказ в обслуживании и т.п.

## Property Oriented Programming

В 2009, 2010 годах Стефан Эссер на конференции Black Hat представил новую технику для эксплуатации уязвимостей типа POI — **Property Oriented Programming**, технику повторного использования кода [9][10]. Техники повторного использования кода появились при эксплуатации бинарных уязвимостей для обхода таких защитных механизмов, как рандомизация размещения адресного пространства (address space layout randomization — ASLR) и предотвращение выполнения данных (data execution prevention). Известными примерами являются техники return-to-libc, return-oriented programming (ROP), jump-oriented programming (JOP). В ROP и JOP атакующий повторно использует фрагменты кода, которые уже находятся в памяти (они получили название **гаджетов**), и соединяет их вместе, конструируя нужную последовательность кода по кусочкам. Полученная последовательность называется цепочкой гаджетов (**gadget chain**).

Эссер предложил использовать такую же идею для эксплуатации уязвимостей класса POI в веб-приложениях на PHP. Динамическая типизация в PHP позволяет присвоить любому полю объекта в качестве значения не только простое значение, но и объект любого типа. Таким образом, если:

- в каком-либо методе объекта А вызывается метод другого объекта В;
- и экземпляр объекта В хранится в поле объекта А;



- то при десериализации строки, которую контролирует пользователь, можно подменить тип объекта В, определив соответствующее поле объекта А по своему усмотрению.

В результате при десериализации вызовется одноименный метод другого объекта и изменится поток управления. Описанная логика иллюстрируется примером (см. Рисунок 3).

Рисунок 3

```
1  <?php
2  class Database
3  {
4      private $connection;
5      //Some PHP code
6      public function __destruct()
7      {
8          $this->connection->close();
9      }
10 }
11
12 class TempFile
13 {
14     private $filename;
15     //Some PHP code
16     public function close()
17     {
18         unlink($this->filename);
19     }
20 }
21
22 //Some PHP code
23 $user_data = unserialize($_COOKIE['data']);
```

В строке 23 производится вызов функции *unserialize* с данными, контролируемые пользователем, без должных проверок. Если атакующий передаст в вызов функции строку `O:8:"Database":1:{s:20:"0Database0connection";O:8:"TempFile":1:{s:18:"0TempFile0filename";s:9:".htaccess";}}`, то в контексте приложения создастся объект класса *Database*, которому в поле *connection* будет присвоен объект класса *TempFile* с полем *filename*, равным *.htaccess*. После

того, как запрос будет обработан, процесс-обработчик завершится, предварительно вызвав метод `__destruct` для всех существующих объектов. В результате вызова деструктора объекта *Database* вызовется метод *close* для объекта, который хранится в поле *connection* (*TempFile*), и в результате будет удален файл *.htaccess*.

В реальных приложениях находить подобные цепочки вручную затруднительно из-за большого количества классов и методов. Так, например, в популярной системе управления содержимым — WordPress [11] определено 383 класса. Актуальной задачей представляется автоматизация поиска возможных цепочек объектов, позволяющих проэксплуатировать уязвимость класса PHP Object Injection.

# Постановка задачи

## Цель работы

Разработать метод и инструментальное средство поиска цепочек для эксплуатации известной уязвимости десериализации в заданном коде на языке PHP и провести экспериментальное исследование эффективности разработанного метода и средства.

## Задачи

Для достижения поставленной цели необходимо решить следующие задачи:

1. Формализовать понятие цепочки для эксплуатации уязвимости десериализации для языка PHP.
2. Предложить метод поиска потенциальных цепочек.
3. Предложить метод оценки применимости потенциальных цепочек для эксплуатации уязвимости десериализации.
4. Реализовать средство, автоматизирующее предложенные методы для поиска и оценки применимости цепочек.
5. Провести экспериментальное исследование эффективности разработанного метода и средства.

# Обзор существующих инструментальных средств эксплуатации POI

С момента первого упоминания POI появилось не так много исследований связанных с поиском и эксплуатацией данной уязвимости. Методически она не отличается от других недостатков, связанных с некорректным использованием данных, полученных от пользователя, в результате чего открываются возможности проведения injection-атак. Исследования по поиску недостатков данного типа широко распространены [12][13][14], в то время как метод построения входных данных для эксплуатации недостатка (вектора для атаки) отличается в случае недостатка POI и исследования на эту тему немногочисленны, как и инструменты.

Критерии сравнения:

- Возможность сконструировать цепочку гаджетов с подставленными данными (например, командой, которая должна будет выполняться при эксплуатации или имя файла для удаления), так как это упрощает использование инструмента.
- Поддержка различных версий PHP. Парадигма ООП появилась в PHP не сразу, а только в пятой версии языка. Если инструмент будет ограниченно поддерживать ООП, это отрицательно скажется на полноте анализа. Также между разными мажорными версиями язык сильно меняется в плане поддержки новых выразительных средств, что так же влияет на полноту анализа.
- Требуется ли доступ к коду приложения. При исследовании веб-приложений на наличие уязвимостей и возможности их эксплуатации очень часто код приложения недоступен, поэтому возможность получения цепочки без кода является несомненным плюсом.

## RIPS

Одним из известных методов нахождения в коде приложения недостатков некорректного использования данных, полученных от пользователя, является статический анализ. Существует несколько статических анализаторов, которые поддерживают поиск таких недостатков в PHP [15][16][17]. Но так как статические анализаторы обычно используются именно для поиска недостатков и в дальнейшем

исправления их разработчиками, то задача генерации вектора для атаки обычно не ставится.

Существует две версии анализатора RIPS:

1. Открытая бесплатная версия RIPS 0.55 [18], которая не поддерживается с 2014 года.
2. Платная закрытая версия RIPS 2.9.0 [19], которую нельзя свободно получить для экспериментов.

Главные отличия разных версий анализатора представлены в Таблице 1.

**Таблица 1 — сравнение версий RIPS**

RIPS 0.55	RIPS 2.9.0
Ограниченная поддержка функциональности PHP, классы почти не поддерживаются	Поддерживаются версии PHP 3-7
Обнаружение 15 типов недостатков	Обнаружение более 100 типов недостатков
Поиск начальных звеньев цепочки гаджетов	Полное построение цепочки гаджетов

Открытая версия RIPS была выпущена во время „Месяца безопасности PHP” (MOPS) [20]. Это была вторая попытка реализовать сканер уязвимостей. Первая недоступная публично версия основывалась на регулярных выражениях, а уже следующая версия основывалась на использовании токенизатора. Решение о переходе улучшило качество анализа, но количество ложных срабатываний осталось большим. Основные характеристики инструмента:

- Осуществляется поиск уязвимого вызова функции *unserialize*.
- Осуществляется поиск классов, которые можно использовать для начала цепочки гаджетов.
- Очень ограниченная поддержка ООП.
- Цепочки гаджетов не генерируются.
- Требуется код приложения.
- Присутствует веб-интерфейс для управления;

Платная версия была выпущена разработчиками открытой версии RIPS. На основании опыта полученного при разработке предыдущей версии, авторами с нуля был разработан новый механизм анализа, который учитывает особенности PHP. Из-за закрытой разработки новой версии RIPS оценить ее на практике не удалось, но об

особенностях инструмента можно судить на основании статей авторов [21][22][23].

Основные характеристики инструмента:

- Осуществляется поиск уязвимого вызова функции *unserialize*.
- Осуществляется поиск цепочек гаджетов, которые можно использовать для эксплуатации найденных уязвимых вызовов функции *unserialize*.
- Нельзя сконструировать цепочку гаджетов с собственными данными.
- Поддержка PHP версий 3-7.
- Требуется код приложения.
- Присутствует веб-интерфейс для управления.

## RHPGGC

RHPGGC — это библиотека цепочек гаджетов для эксплуатации POI с утилитой для генерации векторов [24].

При поиске уязвимостей в веб-приложении методом черного ящика у аналитика нет доступа к коду приложения, что затрудняет создание цепочки гаджетов для эксплуатации уязвимости POI. Все чаще в разработке используются фреймворки и библиотеки, код которых находится в открытом доступе и может быть проанализирован. А вкупе с повсеместным использованием в PHP механизмов автозагрузки используемых классов, эксплуатация приложений с недоступным кодом становится возможной с использованием цепочек гаджетов найденных в распространенных библиотеках и фреймворках.

RHPGGC содержит известные цепочки гаджетов для популярных библиотек, которые могут использоваться для конструирования вектора эксплуатации для любого приложения, которое использует один из поддерживаемых фреймворков. Так как библиотека является открытой и свободно распространяемой, любой желающий может добавить в нее найденную им цепочку гаджетов.

Основные характеристики инструмента:

- Не осуществляется поиск уязвимого вызова функции *unserialize*.
- Из-за отсутствия анализа кода приложения можно считать, что поддерживаются все версии PHP.

- Не требуется код приложения.
- Можно сформировать цепочку гаджетов с собственными данными.
- Наличие большого количества преобразований итогового вектора (кодирование, упаковывание в PHAR).
- Цепочки гаджетов добавляются в библиотеку вручную.

## Вывод

Рассмотренные инструменты имеют свои преимущества и недостатки, которые четко определяют, в каком случае какой инструмент лучше использовать.

Ключевым недостатком RIPS является его закрытость, что приводит к тому, что основная сфера применения данного инструмента — анализ приложений собственной разработки в коммерческих компаниях для повышения безопасности.

RHPGGC может свободно использоваться для тестирования на проникновение приложений, которые используют популярные фреймворки, методом черного ящика. Ключевым недостатком является, что RHPGGC — это библиотека **известных** цепочек, поэтому может использоваться для ограниченного круга приложений.

Существующие инструменты не помогают искать новые цепочки в приложениях с открытым исходным кодом. А постоянное пополнение новыми цепочками библиотеки RHPGGC показывает актуальности задачи поиска, поэтому разработка такого инструмента является актуальной задачей.

## Формальные обозначения

В дополнение к обозначениям внутривычислительного анализа, введенным в книге „Principles of Program Analysis” [25] на странице 3, введем еще два оператора:

- Вызов функции —  $func\_call(name, args\_list)$
- Вызов метода —  $method\_call(var, name, args\_list)$

Для того чтобы последовательность функций/методов была цепочкой необходимо (необходимое условие):

- первый элемент — магический метод из списка заданных;
- последний элемент — критическая функция из списка заданных;
- для любых двух последовательных элементов цепочки А, В выполняется одно из следующих условий:

**Таблица 2 — необходимые условия перехода между элементами цепочки**

Если элемент В...	То в теле элемента А...
функция с именем $nameB$ и $n$ обязательными аргументами	найдется выражение $func\_call(nameB, args\_list)$ , где $len(args\_list) \geq n^1$
метод класса $classB$ с именем $nameB$ и $n$ обязательными аргументами	найдется выражение $method\_call(some\_var, nameB, args\_list)$ , где $len(args\_list) \geq n$
	найдется выражение $method\_call("this", nameB, args\_list)$ и А так же является методом класса В

Последовательность элементов, удовлетворяющую **необходимому** условию цепочки, будем называть потенциальной цепочкой.

Будем считать, что класс участвует в потенциальной цепочке, если хоть один из элементов цепочки является методом данного класса.

Потенциальная цепочка является искомой цепочкой, если существуют такие значения полей классов, которые участвуют в потенциальной цепочке, что при выполнении первого элемента цепочки выполнится последний.

<sup>1</sup> Количество аргументов у вызова функции может быть больше, чем параметров у вызываемой функции в силу особенностей языка PHP, который передаст „лишние параметры” через специальные переменные



Тогда оценкой применимости потенциальной цепочки будем называть проверку того, можно ли контролируя поля классов сделать из нее искомую цепочку.

# Анализ задачи

## Задачи, возникающие при поиске потенциальных цепочек

Для поиска цепочек требуется получить информацию обо всех доступных методах, функциях и их вызовах в заданном коде. Данную задачу можно разбить на три подзадачи:

1. Поиск всех объявлений функций и методов.
2. Сбор информации об иерархии классов для извлечения информации об унаследованных методах.
3. Поиск вызовов функций и методов.

Абстрактное синтаксическое дерево (AST) содержит в себе информацию о структуре программы. Внутренние вершины дерева содержат информацию об операторах языка, а листовые вершины об операндах. Таким образом решение подзадач сводится к обходу AST с определенными обработками вершин:

1. Поиск вершин-операторов, объявляющих функции и методы.
2. Поиск вершин-операторов наследования классов и сохранения информации об операндах.
3. Поиск вершин-операторов, объявляющих функции и методы, с дальнейшим поиском в поддеревьях вершин-операторов вызова функций и методов.

## Задачи, возникающие при оценке применимости потенциальных цепочек

На вход этой задачи дана потенциальная цепочка, представляющая последовательность функций и методов, которые могут быть последовательно вызваны после вызова первого элемента.

Оценка применимости потенциальной цепочки сводится к проверке, возможно ли контролируя значения полей классов, которые участвуют в цепочке, добиться выполнения хотя бы одного из потоков вычислений, который приведет к последовательному вызову элементов вплоть до критической функции с контролируемыми аргументами.

Данную задачу можно разбить на подзадачи — начать анализировать потенциальную цепочку с конца, начиная с последовательности из двух последних элементов, потом из трех и так далее. При добавлении нового элемента в начало последовательности требуется выяснить, от каких переменных зависят параметры вызова метода или функции, соединяющего его со следующим элементом. Это известная задача анализа программ — построение обратного среза (backwards slicing) [26][27]. После того, как получено множество переменных, которые влияют на поток управления и значения аргументов вызова критической функции, требуется определить, при каких значениях будет достигнут нужный поток вычислений. То что РНР является языком с динамической типизацией и большим количеством динамических возможностей серьезно затрудняет статический анализ. Разделим полученное множество переменных на классы:

- константы;
- поля классов;
- параметры функций/методов;
- все оставшиеся переменные, которые будем называть динамическими.

Деление на представленные классы обусловлено тем, что для каждого класса можно провести специфическую обработку для получения выводов.

Для констант известна классическая задача анализа программ — распространение констант (constant propagation) [28]. После решения этой задачи можно отсеять множество потоков вычислений, которые точно не будут достигнуты.

Значения полей классов являются полностью контролируемыми переменными при десериализации, поэтому при их анализе можно допустить, что их значение будет равно таким, какое требуется для достижения нужного потока вычисления.

В класс динамических переменных попадают те переменные, про которые при статическом анализе сделать точных выводов не представляется возможности. Про данные переменные можно только собрать некоторые ограничения, по которым потом можно оценить недостижимость некоторого потока вычислений.

Так как параметры функций и методов зависят от переменных из предыдущих элементов потенциальной цепочки, то после полного анализа их можно отнести к одному из предыдущих классов, в зависимости от переменных от которых они зависят.

# Реализация инструментального средства

## Поиск потенциальных цепочек

При разработке инструментального средства рассматривались уже готовые реализации для решения выделенных подзадач.

Изначально рассматривался проект BetterReflection [29], который позволяет, используя предоставленные им интерфейсы, получить всю требуемую для поиска цепочек информацию написав минимальное количество кода. Но при экспериментальном исследовании было выяснено, что время работы и требуемая для работы память растут нелинейно при росте размеров анализируемого проекта. Данный недостаток ограничивал множество анализируемых проектов небольшими фреймворками, поэтому было решено от него отказаться.

Для получения информации об иерархии классов было найдено единственное инструментальное средство — PhpDependencyAnalysis [30]. У данного средства есть следующие недостатки:

- Специализация средства на построении графического отображения графов зависимостей.
- Архитектура средства предполагающая его использование как целостный проект (standalone-приложение).

В итоге было решено реализовать весь анализ на базе AST самостоятельно. Единственным парсером кода на языке PHP, доступным на момент начала разработки, который поддерживает новые особенности языка, активно развивается и на выходе выдает AST был PHP Parser [31] от nikic. Данный парсер широко используется в различных открытых проектах, связанных с анализом кода (в том числе и в BetterReflection, PhpDependencyAnalysis), что подтверждает его применимость.

После построения AST для всего проекта проводится поиск:

1. Магических методов из списка заданных.
2. Вызовов методов/функций в теле.
3. Поиск методов/функций, которые могут быть вызваны с помощью найденных вызовов.

4. Если функция входит в заданный список критических функций, то потенциальная цепочка найдена, иначе к пункту 2.

## Оценка применимости потенциальных цепочек

Готовых средств для решения задачи обратного среза для языка PHP найдено не было, поэтому задача была разбита на построение графа потока управления (CFG) и последующее построение обратного среза.

Для построения графа потока управления использовалось инструментальное средство PHP-CFG [32], разработанное ircmaxell и поддерживаемое nikic. Средство позволяет построить CFG в форме **Static Single Assign form** (специальное представление в котором каждой переменной значение присваивается только один раз). Данное представление так же упрощает решение задачи распространения констант, что является дополнительным плюсом.

На базе полученного SSA для каждого метода/функции из цепочки задается критерий среза — точка (вызов, который служит переходом к следующему элементу цепочки) и интересующие переменные (подмножество аргументов функции). Срез строится постепенным включением в множество переменных, которые находятся в правой части оператора присваивания, если левый операнд уже принадлежит множеству. Для всех переменных значение которых берется из  $\phi$ , в множество добавляются переменные, влияющие на данное „разветвление” в CFG.

После получения множества интересующих переменных, каждой цепочке сопоставляется некоторое число — ее рейтинг, который соответствует рейтингу переменных-параметров критического вызова (в отдельных случаях рейтинг может быть обнулен). Рейтинг считается на основе классов переменных и операций, которые с ними происходят. Наивысший рейтинг имеют переменные из категории поля классов, так как значения данных переменных полностью контролируются нарушителем. Каждая операция над переменной понижает ее рейтинг, так как уменьшает контроль над ее значением. Переменные из динамического класса изначально имеют низкий рейтинг, так как плохо поддаются оценке с помощью статического анализа.

Влияние на рейтинг констант зависит от места их появления. Если константа используется в условии ветвления CFG и имеет „правильное значение”, то рейтинг цепочки не меняется, в ином случае он обнуляется. В случае влияния константы на

итоговое значение переменной-параметра критической функции (например, параметр равен сумме константы с полем класса), рейтинг переменной снижается.

# Экспериментальное исследование

Экспериментальное исследование разделено на несколько этапов, которые исследуют применимость инструментального средства на практике.

В первую очередь требовалось оценить способность инструментального средства находить цепочки в заданном коде (полнота работы средства). Для данной цели известно несколько подходов:

1. Реализовать синтетический проект с заранее известными цепочками.
2. Запускать инструментальное средство на различных проектах, находящихся в свободном доступе в Интернет (например, на GitHub).
3. Взять в качестве исходных данных фреймворки из библиотеки известных цепочек (PHPGGC) и проверить средство на них.

К плюсам первого подхода можно отнести возможность оценки полноты поиска цепочек, но код синтетического проекта может сильно отличаться от реальных проектов. Второй подход обратен первому, потому что нивелирует минусы и не имеет тех же плюсов. В то время как третий подход сочетает в себе плюсы первых двух и не подвержен их минусам.

После этого проводилось исследование применимости средства для автоматизации решаемой задачи: насколько инструментальное средство упрощает поиск цепочек человеку-оператору. Для данной цели самым действенным подходом является прямое сравнение времени, затраченного аналитиком на исследование проекта с использованием инструментального средства и без.

## Проверка работоспособности цепочки

Для эксплуатации недостатка POI в веб-приложение передается специально сформированная строка. Строка представляет собой сериализованный объект класса, которому принадлежит магический метод из начала цепочки. Значения полей этого объекта определяются аналитиком таким образом, чтобы последовательность вызовов цепочки выполнялась после восстановления состояния объекта. Для проверки цепочки необходимо сформировать строку эксплуатации.

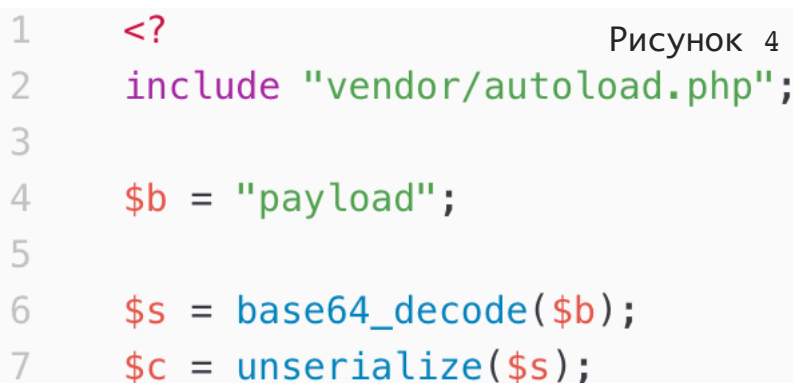
В зависимости от критической функции, которой заканчивается цепочка, эксплуатация недостатка приводит к разным последствиям. Наиболее значимые из них:

исполнение кода на сервере веб-приложения, запись в файл, удаление файла. Будем считать цепочку работающей, если при десериализации строки сформированной по данной цепочке произошло одно из перечисленных действий.

Для проверки работоспособности используется стенд на базе официального Docker-контейнера [33] Composer [34] в который добавляется код исследуемого веб-приложения (взятый из официального репозитория разработчика) или фреймворка. После производилась установка всех зависимостей.

Так как поиск цепочек осуществляется без учета вызова функции *unserialize*, и в общем случае в приложении или фреймворке может не присутствовать недостатка POI, то для проверки работоспособности будем использовать специальный скрипт, который в контексте веб-приложения или фреймворка производит десериализацию переданной ему строки.

Пример скрипта, используемого для тестирования (см. Рисунок 4).



```
1  <?
2  include "vendor/autoload.php";
3
4  $b = "payload";
5
6  $s = base64_decode($b);
7  $c = unserialize($s);
```

Во второй строке происходит подключение контекста веб-приложения/фреймворка<sup>1</sup>. В четвертой строке задается специально сформированная строка, закодированная в base64 (так как в строке могут быть непечатные символы). В 6 строке происходит ее декодирование и в 7 строке происходит десериализация, которая создает объект. Во время жизни данного объекта срабатывает один из магических методов, что приводит последовательному вызову элементов цепочки.

Если последовательность вызовов действительно происходит, то выполняется действие, которое заложено в строке, сформированной по цепочке. В рамках тестирования для удаленного исполнения используется код, который создает пустой файл */tmp/work*, а для цепочек с записью в файл пишется последовательность „AAAAA” в

---

<sup>1</sup> Популярными веб-приложениями и фреймворками на языке PHP сейчас в основном используют систему управления зависимостями Composer. Данная система при установке зависимостей генерирует вспомогательные файлы, в частности „vendor/autoload.php”. При подключении данного файла в скрипт Composer автоматически подгружает все доступные классы в область видимости скрипта.



файл `/tmp/work`. Если после запуска скрипта появляются указанные файлы, то цепочка считается работающей.

## Оценка качества поиска цепочек

Для проведения эксперимента была взята актуальная версия библиотеки RHPGGC. Для каждой цепочки в библиотеке указана информация, представленная в Листинге 1.

NAME	VERSION	TYPE	VECTOR	I
CodeIgniter4/RCE1	4.0.0-beta.1 <= ?	rce	__destruct	

Листинг 1

- название цепочки, которое содержит в себе информацию о фреймворке в котором она была найдена;
- ограничения на версию фреймворка для которой данная цепочка должна работать;
- тип цепочки (действие, которое будет выполнено при эксплуатации);
- начальный магический метод;
- дополнительная информация, если присутствует.

Эксперимент состоял из следующих частей:

- проверка работоспособности цепочек для заявленных версий фреймворков;
- поиск работающих цепочек в коде фреймворков проверенной версии.

Так как библиотека RHPGGC в информации о цепочке не содержит информации о последовательности вызовов, и соответственно о критической функции (которая нужна для запуска инструментального средства), во время проверки работоспособности цепочек так же собиралась данная информация. Было включено логгирования вызовов функций в расширении PHP — Xdebug [35], данный лог анализировался и из него выделялась последовательность вызовов от магического метода до критической функции.

Неработающие цепочки вместе с соответствующими им фреймворками были исключены из эксперимента. Для остальных был проведен запуск инструментального средства с входными данными — указанная версия фреймворка, магический метод и, полученная на этапе проверки работоспособности, критическая функция. Для оценки эффективности работы реализованного метода был выбран следующий критерий:

процент найденных работающих цепочек из известных. Были получены следующие результаты:

**Таблица 3 — Результаты экспериментального исследования**

Цепочка с именем фреймворка	Версия Фреймворка	Магический метод	Критическая функция	Результат поиска
CodeIgniter4/RCE1	4.0.0-beta.1	__destruct	Вызов значения переменной	+
Doctrine/FW1	2.1.0	__toString	file_put_contents	+
Drupal7/FD1	7,63	__destruct	unlink	+
Drupal7/RCE1	7,63	__destruct	call_user_func_array	
Guzzle/FW1	6.3.2	__destruct	file_put_contents	+
Guzzle/INFO1	6.3.2	__destruct	call_user_func	
Guzzle/RCE1	6.3.2	__destruct	call_user_func	
Laravel/RCE1	5.4.23	__destruct	call_user_func_array	+
Laravel/RCE2	5.4.23	__destruct	Вызов значения переменной	+
Laravel/RCE3	5.4.23	__destruct	Вызов значения переменной	+
Laravel/RCE4	5.4.23	__destruct	call_user_func_array	+
Magento/FW1	1.9.4.0	__destruct	file_put_contents	
Magento/SQLI1	1.9.4.0	__destruct		
Monolog/RCE1	1,23	__destruct	call_user_func	+
Monolog/RCE2	1,17	__destruct	call_user_func	+
Phalcon/RCE1	1.2.2	__wakeup		
Slim/RCE1	3.8.1	__toString	call_user_func_array	+
SwiftMailer/FW1	5.4.8	__toString	fwrite	+
SwiftMailer/FW2	6.0.0	__toString	fwrite	+
SwiftMailer/FW3	5.0.1	__toString	fwrite	+
SwiftMailer/FW4	4.0.0	__destruct	fwrite	+
Symfony/FW1	2.5.2	DebugImport		
Symfony/FW2	3,4	__destruct	file_put_contents	+

Symfony/RCE1	3,3	__destruct	Вызов значения переменной	+
Symfony/RCE2	2.3.42	__destruct		
Symfony/RCE3	2.8.32	__destruct		
Yii/RCE1	1.1.20	__wakeup	call_user_func	+
ZendFramework/FD1	1.12.20	__destruct	unlink	+
ZendFramework/RCE1	1.12.20	__destruct	preg_replace	+
ZendFramework/RCE2	1.12.20	__toString	call_user_func_array	
ZendFramework/RCE3	2.0.1	__destruct	call_user_func	

Из известных работающих цепочек в популярных фреймворках с помощью инструментального средства было найдено 80% цепочек. Не найденные цепочки в основном относятся к многоэтапной эксплуатации, либо требуют от атакующего каких-то дополнительных действий после десериализации.

Рассмотрим, почему не были найдены некоторые цепочки:

- Drupal7/RCE1 — данная цепочка на самом деле приводит не к RCE, а к записи сериализованного объекта кэша в базу данных. И только потом, во время работы приложения, при использовании данной записи из базы данных происходит последовательность вызовов (**не начинающаяся с магического метода**), приводящая к RCE.
- Guzzle/INFO1, Guzzle/RCE1 — цепочки на самом деле используют класс из зависимости фреймворка — guzzle/psr7. Начиная с версии 1.5.2 цепочки больше не применимы, так как с фреймворком поставляется последняя версия зависимости.
- Magento/SQLI1 — цепочка не работает, точная причина не установлена.
- Phalcon/RCE1 — является расширением для PHP, написанным на C. Так как инструментальное средство принимает на вход PHP-код, поиск цепочки осуществить невозможно.

- `Symfony/FW1` — начало цепочки не является магическим методом, что не удовлетворяет ограничениям данной работы.
- `Symfony/RCE2`, `Symfony/RCE2` — цепочки не работают для заявленных версий проекта, так как в них присутствует некорректный переход между методами-элементами цепочки. В одном из методов вызывается следующий по цепочке без передачи аргументов, хотя следующий метод имеет один обязательный аргумент.
- `ZendFramework/RCE2` — в середине цепочки есть вызов метода название которого берется из значения переменной. Такие переходы на данный момент не поддерживаются разработанным методом и средством.
- `ZendFramework/RCE3` — многоэтапная цепочка, которая содержит в себе вызовы нескольких магических методов и используется механизм обратных вызовов.

## Экспериментальное исследование на реальной библиотеке

В качестве исследуемой библиотеки использовалась `SwiftMailer` [36] — популярная библиотека для отправки электронных писем, которая имеет восемь тысяч звезд на GitHub<sup>1</sup> и используется в самых популярных фреймворках для разработки веб-приложений (`Laravel`, `Symphony`). Дополнительным плюсом было большое количество работающих цепочек в `RHPGGS` для данной библиотеки, что говорит о большом количестве изучений ее на предмет поиска цепочек. Для исследования использовались последние доступные версии — 6.2.1 и 5.4.12.

Перед проведением исследования были выделены интересующие критические действия: исполнение кода на сервере веб-приложения, запись в файл, удаление файла. Был составлен список критических функций, вызывая которые с контролируемыми параметрами можно добиться данных действий.

Инструментальное средство было запущено со списком выделенных критических функций и магическими методами: `__destruct`, `__wakeup`, `__toString`. В результате обработки результата работы инструментального средства человеком-оператором было обнаружено три ранее неизвестных цепочки и еще двенадцать вариаций данных цепочек, получаемых с помощью замен классов на их родственные (для обеих версий были найдены одинаковые цепочки). На поиск было затрачено две минуты работы инструментального

---

<sup>1</sup> Звезды на GitHub реализуют механизм добавления репозитория в список интересных для разработчика. По количеству звезд у репозитория можно оценивать его популярность. Всего три тысячи репозитория из более ста миллионов имеет больше пяти тысяч звезд.

средства и тридцать минут на анализ вывода человеком-оператором и еще около часа на составление по цепочкам строк для эксплуатации и итоговую проверку работоспособности цепочек.

Такой же результат был достигнут аналитиком без использования инструментального средства за два с половиной часа анализа кода библиотеки. И еще час был бы затрачен на составление строк и проверку работоспособности, так как инструмент никак не оптимизирует этот процесс.

Проведенный эксперимент показывает, что даже на таком небольшом проекте инструментальное средство облегчает и ускоряет работу аналитика по поиску цепочек.

Одна из найденных цепочек была согласована с администраторами проекта PHPGGC и добавлена в библиотеку. Остальные цепочки находятся в обработке.

# Результаты

В рамках выполнения данной работы были достигнуты следующие результаты:

- Проведен обзор существующих инструментальных средств эксплуатации POI.
- Предложен метод поиска потенциальных цепочек.
- Предложен метод оценки применимости потенциальных цепочек.
- Разработано инструментальное средство автоматизирующее предложенные методы поиска и оценки цепочек.
- Проведено экспериментальное исследование, обнаружившее, что в библиотеке PHPGGC присутствуют неработающие цепочки.
- Подтверждена эффективность разработанного инструментального средства в задаче поиска цепочек.

Поставленная в работе задача была полностью выполнена, в качестве дальнейшего развития планируются следующие шаги:

- Улучшение удобства использования инструментального средства путем добавления графического пользовательского интерфейса.
- Автоматизация генерации строки для эксплуатации по цепочке.
- Повышение скорости работы.
- Улучшение оценки применимости цепочек.

Полученные результаты были представлены на международном форуме по практической безопасности Positive Hack Days 2019 [37].

# Приложение А

Для начала рассмотрим, как сериализуются простые типы, общая схема выглядит, как „<тип переменной>:<значение>”:

```
NULL:      N;
true:      b:1;
false:     b:0;
42:        i:42;

42.3789:    d:42.3789000000000002;
           ^-- Точность контролируется ini настройкой
              serialize_precision (по-умолчанию 17)

resource:   i:0;
           ^-- Ресурсы не могут быть сериализованы, поэтому
              вместо них записывается значение int(0)
```

Строки могут быть сериализованы двумя различными способами:

```
"foobar":   s:6:"foobar";
           ^-- длина строки „foobar”

"foobar":   S:6:"\102\111\10\98\97\114";
           ^-- длина строки „foobar”
```

Если использовать идентификатор *s*, то строка записывается в сериализованном виде как есть. В случае использования идентификатора *S*, любой символ может быть заменен на конструкцию „\<численное значение байта>”. Данная возможность может быть использована для обхода фильтров, которые удаляют из строк непечатные символы.

Рассмотрим, как сериализуются более сложные типы, начнем с массивов:

```
[10, 11, 12]:    a:3:{i:0;i:10;i:1;i:11;i:2;i:12;}
                 ^-- длина массива([10, 11, 12])

                    v-- ключ   v-- значение
["foo" => 4, "bo" => 2]:  a:2:{s:3:"foo";i:4;s:3:"bo";i:2;}
                        ^-- ключ   ^
                           |
                           значение
```

Структура сериализации массива выглядит так же, как и для обычных типов. Сначала идет идентификатор что это массив, с дополнительной информацией о

количестве элементов в массиве, а потом в фигурных скобках перечисляются элементы массива (парами *ключ;значение*).

У классов в PHP тоже есть два способа сериализации: стандартный и модифицируемый. Стандартный использует идентификатор *O* и похож на сериализацию массивов.

```
1  <?php
2  class Example
3  {
4      public $public = 1;
5      protected $protected = 2;
6      private $private = 3;
7  }
```

```
v-- длина("Example")  имя поля --v      значение --v
O:7:"Example":3:{s:6:"public";i:1;s:12:"0*0protected";i:2;
      ^-- имя поля ^-- значение
      количество полей класса
s:16:"0Example0private";i:3;}
^-- имя поля      ^-- значение
```

Для обозначения различных областей видимости полей при сериализации к именам полей добавляются специальные конструкции. В случае *public* области видимости имя остается неизменным, при *protected* области видимости перед именем поля вставляется *0\*0*, а в случае *private* *0<имя класса>0*.

Модифицируемый способ предполагает самостоятельное написание методов *serialize* и *unserialize* для класса, тогда используется идентификатор *C*.

```
1  <?php
2  class Example implements Serializable
3  {
4      public function serialize()
5      {
6          return "foobar";
7      }
8      public function unserialize($str)
9      {
10         // ...
11     }
12 }
```



```
C:5:"Test2":6:{foobar}  
      ^— длина(„foobar“)
```

В таком случае между фигурных скобок будет подставлено строковое значение, которое было возвращено из метода *serialize*.

Другая функциональность, поддерживаемая встроенной в PHP сериализацией — ссылки и указатели. Рассмотрим, какой у них формат сериализации.

```
1  <?php  
2  $a = ["foo"];  
3  $a[1] =& $a[0];  
4  var_dump(serialize($a));  
5  
6  $o = new stdClass;  
7  $o->foo = $o;  
8  var_dump(serialize($o));
```

```
v- первый объект  
a:2:{i:0;s:3:"foo";i:1;R:2;}  
      ^— второй объект  
O:8:"stdClass":1:{s:3:"foo";r:1;}  
      ^— первый объект
```

Указатели сериализуются в *R*, а ссылки в *r*, а формат у них одинаковый. За буквой пишется число, которое указывает на связанный объект (объекты нумеруются с самого начала по-порядку).

## Список литературы

1. Fielding R. et al. Rfc 2616, hypertext transfer protocol–http/1.1, 1999 //URL <http://www.rfc.net/rfc2616.html>. – 2009.
2. Barth A. RFC 6265. HTTP State Management Mechanism. IETF, 2011. – 2011.
3. Nadareishvili I. et al. Microservice architecture: aligning principles, practices, and culture. – " O'Reilly Media, Inc.", 2016.
4. Bray T. et al. Extensible markup language (XML) //World Wide Web Journal. – 1997. – Т. 2. – №. 4. – С. 27-66.
5. Bray T. The javascript object notation (json) data interchange format. – 2017. – №. RFC 8259.
6. OWASP T. Top 10-2017 The Ten Most Critical Web Application Security Risks // U R L : [https://www.owasp.org/images/7/72/OWASP\\_Top\\_10-2017\\_%28en%29.pdf.pdf](https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf)
7. Romano E. PHP Object Injection //URL:[https://www.owasp.org/index.php/PHP\\_Object\\_Injection](https://www.owasp.org/index.php/PHP_Object_Injection)
8. The PHP Group. PHP: Magic Methods. //URL:<http://php.net/manual/language.oop5.magic.php>
9. Esser, S. Shocking News in PHP Exploitation. In Power of Community (POC) (2009).
10. Esser, S. Utilizing Code Reuse Or Return Oriented Programming in PHP Applications. In BlackHat USA (2010).
11. Wordpress.org //URL:<https://wordpress.org>
12. Su T. et al. A survey on data-flow testing //ACM Computing Surveys (CSUR). – 2017. – Т. 50. – №. 1. – С. 5.
13. Wassermann G., Su Z. Sound and precise analysis of web applications for injection vulnerabilities //ACM Sigplan Notices. – ACM, 2007. – Т. 42. – №. 6. – С. 32-41.
14. Su Z., Wassermann G. The essence of command injection attacks in web applications //Acm Sigplan Notices. – ACM, 2006. – Т. 41. – №. 1. – С. 372-382.
15. Progpilot //URL:<https://github.com/designsecurity/progpilot>
16. Phpcs-security-audit //URL:<https://github.com/FloeDesignTechnologies/phpcs-security-audit>

17. PT Application Inspector //URL:<https://www.ptsecurity.com/ww-en/products/ai/>
18. RIPS — A static source code analyser for vulnerabilities in PHP scripts 0.55 //URL:<http://rips-scanner.sourceforge.net>
19. RIPSTech scanner //URL:<https://www.ripstech.com>
20. the Month of PHP Security //URL:<http://www.php-security.org>
21. Dahse, J., and Holz, T. Simulation of Built-in PHP Features for Precise Static Code Analysis. In Symposium on Network and Distributed System Security (NDSS) (2014).
22. Dahse, J., and Holz, T. Static Detection of Second-Order Vulnerabilities in Web Applications. In USENIX Security Symposium (2014).
23. Dahse, J., Krein, N., and Holz, T. Code Reuse Attacks in PHP: Automated POP Chain Generation.
24. PHPGGC: PHP Generic Gadget Chains //URL:<https://github.com/ambionics/phpggc>
25. Nielson F., Nielson H. R., Hankin C. Principles of program analysis. – Springer, 2015.
26. Weiser M. Program slicing //Proceedings of the 5th international conference on Software engineering. – IEEE Press, 1981. – C. 439-449.
27. Tip F. A survey of program slicing techniques. – Centrum voor Wiskunde en Informatica, 1994.
28. Wegman M. N., Zadeck F. K. Constant propagation with conditional branches //ACM Transactions on Programming Languages and Systems (TOPLAS). – 1991. – T. 13. – N°. 2. – C. 181-210.
29. Better Reflection //URL:<https://github.com/Roave/BetterReflection>
30. PhpDependencyAnalysis //URL:<https://github.com/mamuz/PhpDependencyAnalysis>
31. PHP Parser //URL:<https://github.com/nikic/PHP-Parser>
32. PHP-CFG //URL:<https://github.com/ircmaxell/php-cfg>
33. Docker Platform //URL:<https://www.docker.com>
34. Composer: Dependency Manager for PHP //URL:<https://getcomposer.org>
35. XDebug extension for PHP //URL:<https://xdebug.org>

36. SwiftMailer — PHP Mailer //URL:<https://swiftmailer.symfony.com>

37. Positive Hack Days //URL:<https://www.phdays.com/>