Символьный SAST из опенсорсных компонентов

Автор: Андрей Погребной, CyberOK

Мы в CyberOK проводим пентесты каждый день и, естественно, ищем способы как автоматизировать этот процесс. Мы перепробовали самые разные инструменты – коммерческие, открытые, да всякие. В итоге пришли к тому, что фраза «хочешь сделать хорошо – сделай это сам» очень нам откликается и запилили свой символьный SAST без приватных компонентов. А как именно нам удалось заставить SAST и символьное исполнение работать вместе, мы подробно рассказываем в данной статье. Также следует упомянуть, что впервые это исследование было представлено на международной конференции по практической кибербезопасности *OFFZONE 2023*.

1. Мотивация

**В статье делается акцент на Java и web, однако подход применим и к иным областям.*

В данной статье предлагается концепция модификации существующих генераторов тестов с целью адаптировать их к поиску уязвимостей и, соответственно, получить полноценный SAST.

В рамках исследования были выделены ключевые характеристики анализатора:

1. Мощность (умение находить уязвимости в сложных ситуациях)
2. Способность работать в общем случае, а не ориентироваться на конкретные типы уязвимостей
3. Расширяемость (для простоты развития в дальнейшем)
4. Проверяемость, а именно возможность получить не просто отчёт со множеством потенциальных угроз, а конкретный *case*, на котором всё ломается.

Но как всего этого достичь? Об этом и пойдёт речь дальше.

2. Открытые генераторы unit-тестов

Одной из современных технологий статического анализа является символьное исполнение. Эта техника позволяет преодолевать сложные условия, с которыми нельзя справиться стандартными методами, например, фаззингом. Подробнее об этой технике можно почитать тут.

В качестве основы для своего анализатора мы решили взять генератор unit-тестов. Современные генераторы, как правило, используют один из готовых или собственный символьный движок. На рисунке ниже приведено несколько из них.

Engine	Settings	Result format	String support	Library support
Symbolic (Java) PathFinder	very specific .jpf config file	stacktrace	-+	-
JBSE	configuration from code	tests	+	-
UTBot	cli, fine tuning for difficult cases	tests	+	+
EvoSuite	cli, few simple settings	tests	+	+-

Важно отметить, что все они сильно отличаются друг от друга и обладают уникальными настройками для запуска, которые часто даже представлены в разном формате.

В ходе тестирования мы выделили две ключевых фичи: поддержка строк и использование сторонних библиотек. Следует подчеркнуть, что их имплементация в достаточном объёме является сложной задачей, поэтому некоторые движки поддерживают их, но этого не всегда хватает, в то время как другие даже не пытаются это сделать.

Неэффективная реализация этих фичей препятствует нормальной работе анализатора, так как строки и библиотечные вызовы являются неотъемлемой частью реального кода. Более того, сами опасные данные часто представляются в виде строк. Рассмотрим несколько примеров.

Строки

Взглянем на рисунок ниже:

```
public boolean example(String s, String s2, String s3, String s4) {
    if ((s + s2).equals(s3 + "." + s4)) {
        return true;
    }
    return false;
}
```

На первый взгляд кажется, что существует понятное условие и можно легко подобрать входные данные, при которых функция вернет значение *true*. Однако даже данное простое условие может вызвать сложности у символьных движков. В случае, если оно всё же отработает на каком-либо из них, можно немного усложнить пример так, чтобы он

перестал быть выполнимым. И это без участия сложных, тяжело анализируемых функций со строками по типу *match* и работы с регистрами, с которыми проблем ещё больше.

Библиотеки

Далее рассмотрим, что произойдет, если потребуется работать с библиотеками.

К сожалению, и здесь возникают сложности. В лучшем случае придётся тонко настраивать движок, чтобы он с этим справлялся, а в худшем – ничего не заработает. Так как обычный код использует библиотеки практически всегда, то подобные трудности становятся критичными и требуют дополнительных решений.

```
public boolean example(HttpServletRequest request, String s) {
    if (request.getHeader("test").equals(s)) {
        return true;
    }
    return false;
}
```

В примере используется библиотечный вызов *HttpServletRequest.getHeader*, символьный движок должен «понимать» как дальше используются данные, полученные из этого внешнего источника. Это может приводить к возникновению уязвимостей, таких как XSS или SQL Injection в зависимости от последующей обработки.

Системные вызовы

Отдельный класс проблем возникает при попытке обратиться к функциям операционной системы или использовать её ресурсы – это требует значительных усилий со стороны разработчиков символьного движка, и, как правило, этим уже никто не занимается. Однако на практике в реальном коде, глубоко в вызовах функций, такое не редко встречается.

```
public boolean exampleEnv(String s) {
    String value = System.getenv().get("key");
    if (value.equals(s)) {
        return true;
    }
    return false;
}
```

В данном примере ход исполнения зависит от значения переменной окружения *“key”* и, для правильного анализа, символьный движок должен эмулировать ответ от системы или как-то иначе корректно обрабатывать подобные ситуации.

3. Собираем все вместе

В ходе разработки мы пришли к тому, что анализ должен проводиться в несколько этапов. Рассмотрим их подробнее.

Преданализ: taint

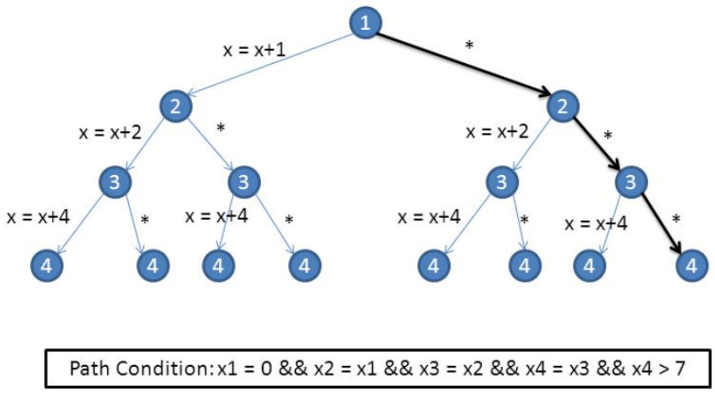
Перейдём к основным особенностям нашего решения. Первоначально следует рассмотреть проблемы, связанные не столько с символьными движками, сколько с самим символьным исполнением, а именно речь пойдет о *path explosion*.

Символьное исполнение пытается анализировать пути, но это значит, что после каждого *if* анализируемых состояний становится вдвое больше. Это приводит к экспоненциальному росту количества состояний и требует специфичных решений.

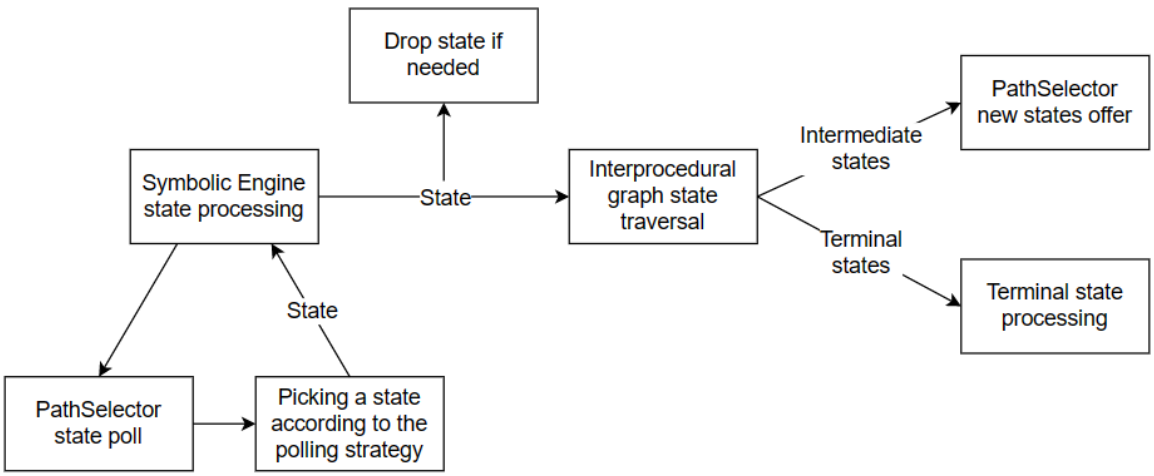
Здесь важно вспомнить о задаче, которую мы хотим решить – это поиск уязвимостей, а не анализ всего кода. В этом месте можно применить *taint analysis* – предварительный анализ кода позволит выявить потенциально опасные вызовы и пути до них. Затем эти пути можно сопоставлять с анализируемыми состояниями и таким образом направлять символьное исполнение.

Допустим, что мы хотим проанализировать функцию *example* и знаем, что в одной из функций, вызываемой из неё, есть опасный вызов, а в другой нет. Применение данного решения позволит сфокусироваться только на анализе интересующей нас функции.

```
void interestingFun(String s) {
    // ...
}

void notInterestingFun(String s) {
    // ...
}

void example(String s, boolean flag) {
    if (flag) {
        interestingFun(s);
    } else {
        notInterestingFun(s);
    }
}
```



Говоря о механизме функционирования символьного исполнения изнутри, следует отметить, что оно работает с состояниями и за одну итерацию разбирает одно из существующих. Однако количество возможных состояний может быть не ограничено, что создает потребность в их систематизации. Поэтому в некоторых движках есть особая сущность, которая контролирует порядок анализа состояний – в используемом движке она называется *PathSelector*. Символьной машине передаётся состояние, оно разбирается и либо порождает новые состояния, которые попадают обратно в *PathSelector*, либо всё доходит до терминального состояния (*return/throw*) и оно обрабатывается определённым образом. Это общая идея *PathSelector*, в частности есть имплементация, интегрированная с *taint*-ом.



Vulnerability check

Важной частью *SAST*-анализатора является база знаний об опасных функциях и их диагностике. В коде существуют опасные функции, необходимо проверять, достижимы ли уязвимости, связанные с их использованием. Для каждой опасной функции заданы проверки *Check* (каждая состоит из проверочной функции и описания уязвимости. Проверочная функция возвращает *true*, если по данным, переданным в неё, можно утверждать, что изначальная функция уязвима). Пусть в ходе символьного исполнения вызывается опасная функция *f*. Тогда её вызов заменяется на вызов декорирующей её функции с проверками. Тело этой функции:

```
if (Check1.f(x)) {
    assert(Check1.description)
} else if (Check2.f(x)) {
    assert(Check2.description)
} else if (
    ...
} else {
    f(x)
}
```

Аргументы, которые должны передаваться в функцию *f*, до этого передаются в функции проверки. Если в результате была найдена уязвимость, то об этом сигнализируется исключением с её описанием. Вызов получившийся функции так же исследуется символьным исполнением. На рисунке ниже показано, какой вид могут иметь функции для проверки уязвимостей:

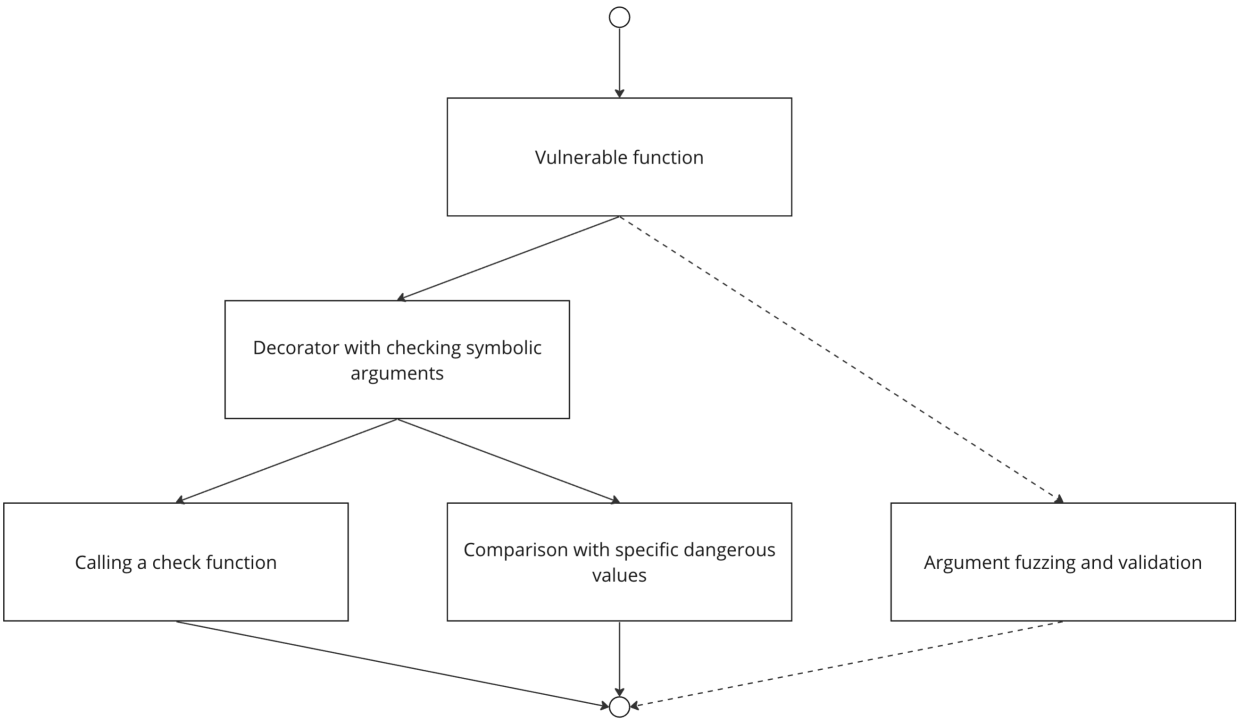
```
boolean pathCheckString(String s) {
    return s.equals("../etc/passwd");
}

boolean pathCheckStringWithAccess(File f, String access) {
    return access.equals("rw") && pathCheckString(f.getPath());
}
```

Кроме того, вместо функций можно альтернативно задать *JSON* с сигнатурой проверяемой функции и аргументами, считающимися уязвимыми:

```
{
  "signature": ["string", "int"],
  "arguments": [
    ["../etc/passwd", 666],
    ["admin", 777],
    ["", 42]
  ]
}
```

Таким образом, если к функции *f* подключена проверочная функция *pathCheckString* и аргумент, попадающий в *f* , может быть равен *../etc/passwd*, то возникнет соответствующее исключение, сообщающее об уязвимости.

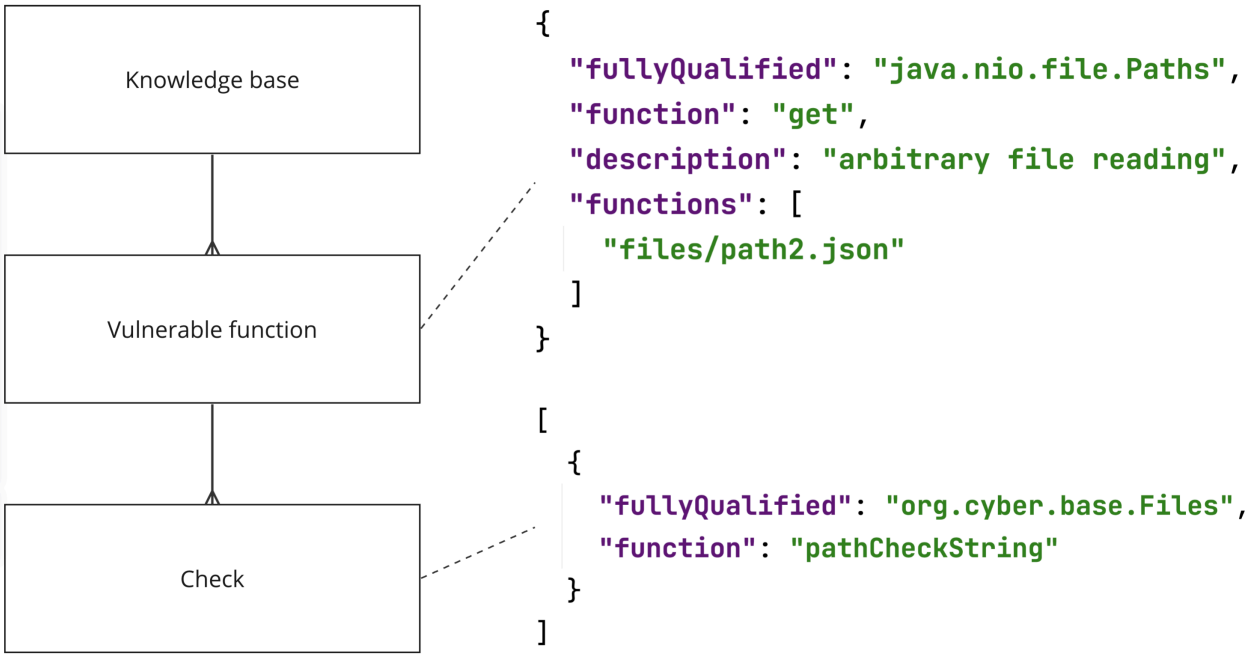
На рисунке ниже изображены все варианты проверок схематично. Первые два были описаны выше – они работают через декоратор с символьными аргументами. Помимо них существует адаптивный метод проверки – фаззинг аргументов. На данный момент он находится в разработке и реализован лишь частично. О нём подробнее будет рассказано позже.



База знаний

Для удобства проверок используется общая структура для хранения – база знаний. Она состоит из *Java*-классов с функциями проверок и *JSON*-файлов, которые на них ссылаются. Каждая уязвимая функция в базе представляется в виде *JSON*-объекта,

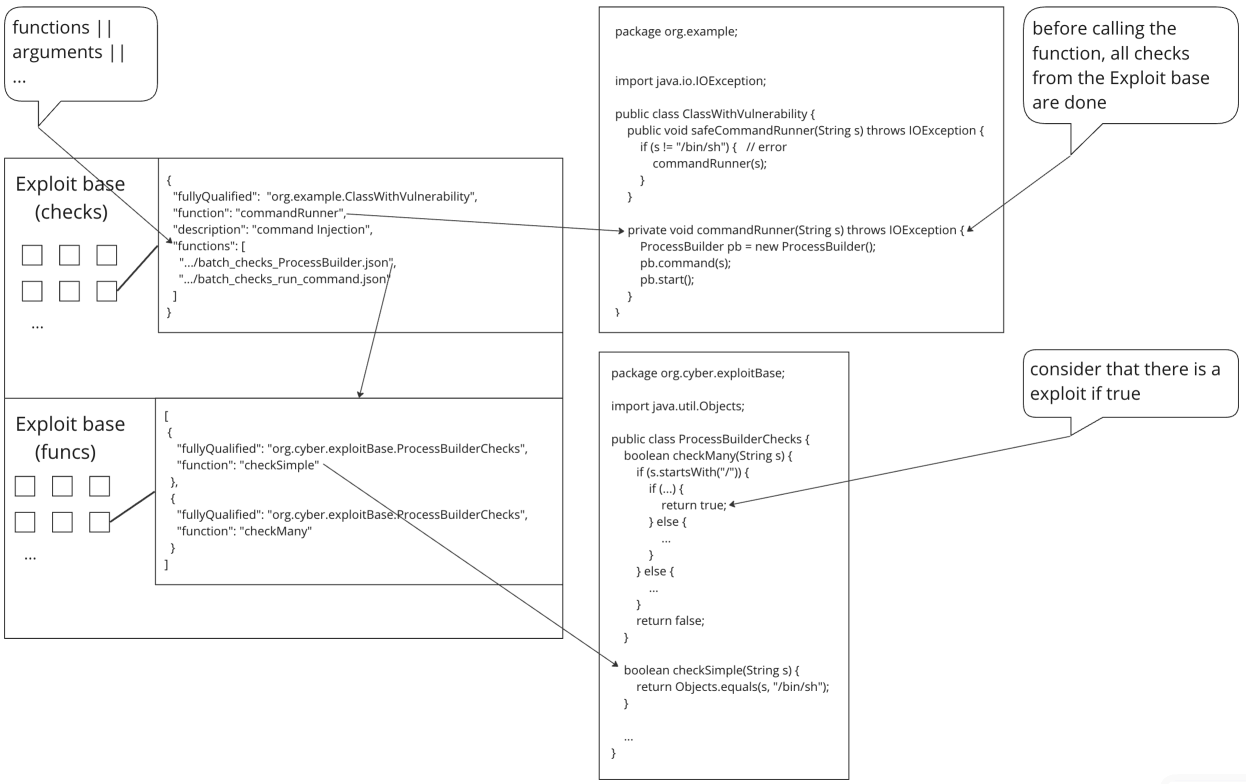
который содержит идентификатор функции, описание уязвимости и ссылки на *JSON*-файлы с описанием проверок. Каждый такой *JSON* в свою очередь имеет список идентификаторов проверочных функций. Базы знаний являются отдельными компонентами, их можно подключать к анализатору.



Взглянем на то, каким образом можно добавить проверку на уязвимость. База знаний должна, как правило, содержать проверочные функции для библиотечных, но на рисунке ниже, для наглядности, приведен пример с кастомным классом. *ClassWithVulnerability* где-то объявлен в нашем коде и содержит функцию *commandRunner*, которая выполняет опасную операцию. Мы хотим быть уверенными, что в эту функцию не попадают данные, удовлетворяющие нашим условиям.

Для этого в базу знаний нужно добавить сами проверки (здесь они содержатся в классе *ProcessBuilderCheck*), добавить *JSON*-файл с ссылками на проверки, которые должны выполняться, и при помощи ещё одного *JSON* связать все объявленные проверки с функцией *commandRunner*.

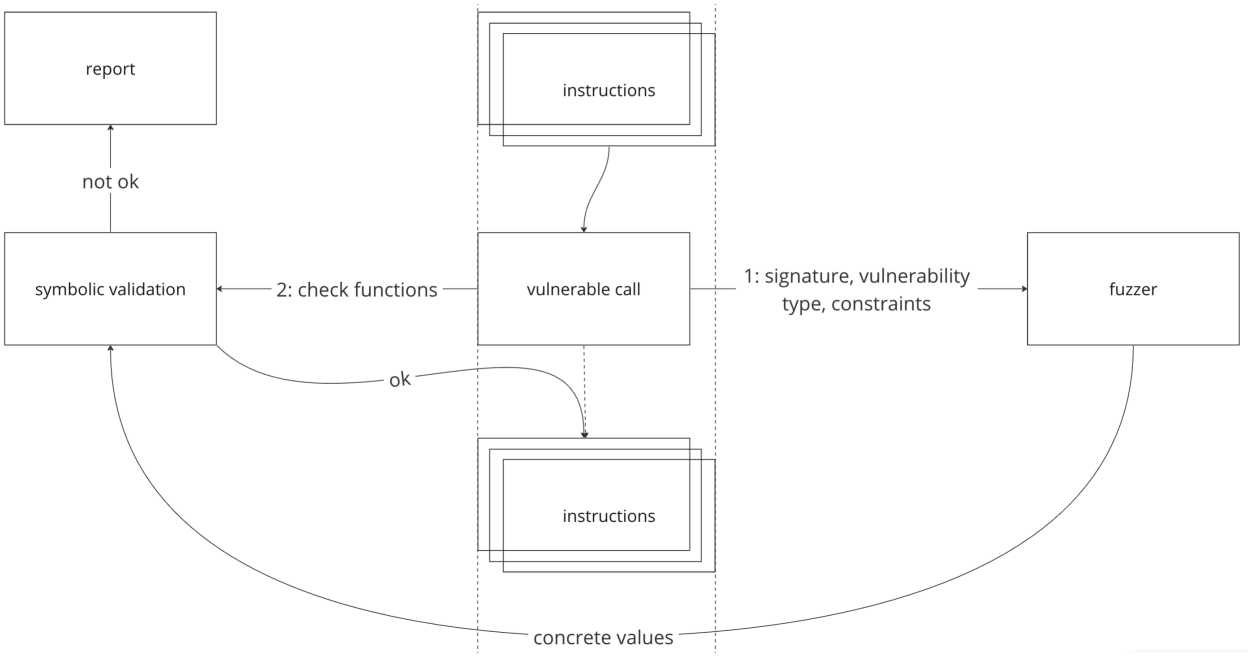
Теперь, при запуске анализатора, если исполнение доходит до вызова функции *commandRunner*, выполнятся объявленные проверки и будет сообщаться, если они прошли успешно (т.е. если была найдена уязвимость).



4. Что можно сделать еще?

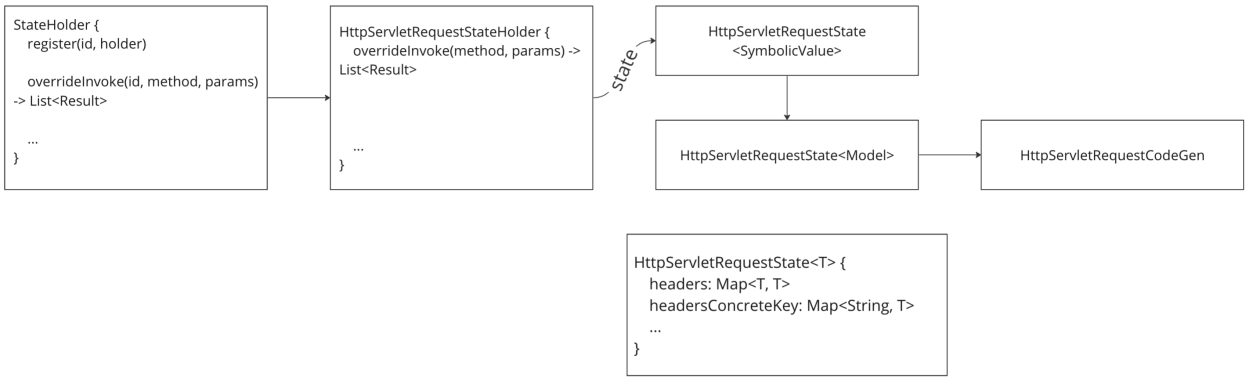
Fuzzing

Символьное исполнение не всегда справляется со сложными проверками, а использование *JSON* с конкретными аргументами может быть недостаточно гибким. Здесь на помощь приходит фаззинг – он используется вместо проверок с декоратором и позволяет создавать релевантные опасные входы и проверять их. Чтобы справляться с этой задачей, фаззеру передаются сигнатура самой функции, тип уязвимости для которого нужен опасный вход и ограничения на аргументы, собранные символьным исполнением. Он возвращает конкретные опасные входы, также проходящие символьную проверку, и, как и раньше, создаётся *report*, если она была успешна.



Wrappers

Стоит вспомнить и о библиотечных функциях. В ряде случаев обойти проблему с ними помогают *mock*-объекты (фиктивная реализация интерфейса, предназначенная для тестирования. Например, можно замокать класс и задать ему возвращаемое значение при вызове нужных функций). Тем не менее, они не являются универсальным решением, потому что иногда анализ библиотечных функций может быть необходим. С этой целью были созданы некоторые «обёртки», подменяющие реализацию метода, но при этом сохраняющие внутреннее состояние класса, корректно с ним работая. Их можно использовать в определённых случаях. Например, для анализа *web* можно создать обертку *HttpServletRequstStateHolder*, которая переопределяет нужные методы *HttpServletRequst*. В обёртках хранятся символьные состояния полей класса, что обеспечивает более точный анализ. Например, в *state* для *request* хранятся *headers* в символьном виде – они используются при обращении к ним через функции. В общем случае писать обёртки слишком затратно, но в конкретных и часто встречающихся ситуациях они способны оказать существенную помощь. Например, это полезно для точек входа приложения, откуда приходят пользовательские данные.



5. Примеры

Пример 1

Перейдём к примерам, чтобы продемонстрировать, как описанный подход работает. Рассмотрим функцию *doGet*, в которой используется путь до файла – *filename*. При выполнении некоторых условий на *cookie request*, этот файл читается и отправляется как результат обратно.

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    String filename = req.getParameter( s: "filename");
    Cookie[] cookies = req.getCookies();
    if (cookies[0].getName().equals("is_admin") && cookies[0].getValue().equals("true")) {
        List<String> lines = Files.readAllLines(Paths.get(filename));
        PrintWriter writer = resp.getWriter();
        writer.println(lines);
    }
}
```

Пример 1: результат

Если запустить анализатор, можно получить тест с аннотацией, содержащей описание уязвимости, взятое из прошедшей проверки в базе знаний - *vulnerabilityInfo*. В коде создаётся *mock* на *request*, в него добавляются необходимые *cookie* для прохождения условия, а именно устанавливается *cookie* с нужным именем и значением, далее по параметру *filename* записывается опасный вход. Это позволяет создать тест, демонстрирующий уязвимость.

```
@Test
@DisplayName("doGet: lines = Files.readAllLines(Paths.get(filename)) : True → ThrowMockitoException")
@org.cyber.utils.VulnerabilityInfo("arbitrary file reading")
public void testDoGet_Cookies0GetNameEqualsAndCookies0GetValueEquals() throws IOException  {
    Example example = new Example();
    HttpServletRequest httpServletRequestMock = mock(HttpServletRequest.class);
    (org.mockito.Mockito.when(httpServletRequestMock.getParameter( name: "filename"))).thenReturn( value: "../etc/passwd");
    javax.servlet.http.Cookie[] cookieArray = new javax.servlet.http.Cookie[1];
    Cookie cookieMock = mock(Cookie.class);
    (org.mockito.Mockito.when(cookieMock.getName())).thenReturn( value: "is_admin");
    (org.mockito.Mockito.when(cookieMock.getValue())).thenReturn( value: "true");
    cookieArray[0] = cookieMock;
    (when(httpServletRequestMock.getCookies())).thenReturn(cookieArray);

    example.doGet(httpServletRequestMock,  resp: null);
}
```

Пример 2: реальный код

Приведем еще один пример, на этот раз на реальном коде. Рассмотрим функцию *get*, она вызывается выше из другой функции, с которой начинается анализ. Дальше, в *initHttp* открывается соединение, представляющее собой уязвимое место.

```
public static String get(String url, Map<String, String> params, Map<String, String> headers) {
    StringBuffer bufferRes = null;
    try {
        HttpURLConnection http = null;
        if (isHttps(url)) {
            http = initHttps(initParams(url, params), _GET, headers);
        } else {
            http = initHttp(initParams(url, params), _GET, headers);
        }
        InputStream in = http.getInputStream();
        BufferedReader read = new BufferedReader(new InputStreamReader(in, DEFAULT_CHARSET));
        String valueString = null;
        bufferRes = new StringBuffer();
        while ((valueString = read.readLine()) != null) {
            bufferRes.append(valueString);
        }
```

Пример 2: результат

Для данного кода был сгенерирован тест с *ServerSideRequestForgery* и подходящим опасным входом – *file:///etc/passwd*. Стоит отметить, что в рассмотренном примере также проверяется дополнительное условие для достижения самой *initHttp*. Это позволяет отсечь часть опасных *payload*-ов (в данном случае те, что начинаются с *https)*.

```
@Test
@DisplayName("get: return get(url, params, null) → ThrowStackOverflowError")
@org.cyber.utils.VulnerabilityInfo("ServerSideRequestForgery")
public void testGet_HttpKitGet_1() {
    org.mockito.MockedConstruction mockedConstruction = null;
    try {
        mockedConstruction = mockConstruction(java.net.URL.class,
                (java.net.URL urlMock, org.mockito.MockedConstruction.Context context) → {
        });
        String string = "file:///etc/passwd";

        assertThrows(StackOverflowError.class, () → HttpKit.get(string, null));
    } finally {
        mockedConstruction.close();
    }
}
```

6. Выводы

В заключение подведем итоги описанного подхода. Среди преимуществ следует выделить:

1) Способность справляться со сложными случаями за счёт техники символьного исполнения;
2) Возможность работы с большими исходниками при использовании описанного преданализа;
3) Гибкая конфигурация и возможность расширения за счёт отдельного компонента базы знаний и в целом разработанной архитектуры;
4) Понятный результат на выходе.

Также следует отметить некоторые минусы подхода:

1) Необходимость тонкой настройки для движка;
2) Ограничения для строк и библиотек среди существующих движков для *Java*.

7. Перспективы развития

В дальнейшем, для улучшения работы и усовершенствования подхода предполагается обеспечить более надежную поддержку строк и библиотек, реализовать в полной мере описанный ранее фаззинг, продолжить расширение базы знаний и имплементировать ряд обёрток.