НАУЧНОЕ ИЗДАНИЕ МГТУ ИМ. Н. Э. БАУМАНА
НАУКА и ОБРАЗОВАНИЕ
Эл № ФС77 - 48211. Государственная регистрация №0421200025. ISSN 1994-0408
электронный научно-технический журнал

Применение метода статического сигнатурного анализа для выявления дефектов безопасности веб-приложений

 # 09, сентябрь 2012
DOI: 10.7463/0912.0461281

Медведев Н. В., Марков А. С., Фадин А. А.

УДК 681.3.06

Россия, МГТУ им. Н.Э. Баумана
Россия, г. Москва, ЗАО «НПО «Эшелон»
mail@cnpo.ru
a.markov@npo-echelon.ru
a.fadin@npo-echelon.ru

Введение

Большинство компаний в современных рыночных условиях уделяют недостаточно внимания безопасности своих веб-приложений. Зачастую руководство может оправдывать такой подход низкой степенью влияния веб-приложений на бизнес-процессы компании.

Если речь идет о сайте-визитке, то потери от скомпрометированного веб-узла действительно можно расценить как минимальные. Однако не стоит забывать, что и в этом случае компания может нести значительные репутационные риски, связанные, например, с ситуацией, когда на сайте происходит малозаметная подмена контента.

Совсем иная ситуация возникает, когда бизнес-процессы компании полностью основаны на функционировании веб-приложения. К данной категории относятся интернет-магазины, корпоративные порталы, электронные торговые площадки, SaaS-приложения и т.д.

Возможные финансовые потери компании в случае вывода приложения из строя или хищения/подмены данных могут варьировать в очень широком диапазоне.

В соответствии с №152-ФЗ все операторы ПД обязаны привести свои информационные системы обработки персональных данных в соответствие

требованиям закона «О персональных данных». Под действие данного закона попадает большинство веб-ресурсов сегмента электронной коммерции.

Однако большинство существующих методов тестирования работы и оценки веб-систем основаны на подходе типа «черный ящик», когда аудитором создаются высокие нагрузки на систему или имитируются действия потенциального злоумышленника. Не умаляя ценности этого подхода, необходимо заметить, что практика показывает, что большинство уязвимостей связаны с ошибками разработчиков и администраторов веб-ресурсов, а комплексная оценка безопасности требует изучения внутренней архитектуры системы и в первую очередь её программного кода.

Методы статического анализа исходных текстов программ, в отличие от динамического тестирования, позволяют избежать проблемы размерности области тестовых данных и добиться большей степени автоматизации проверок на наличие дефектов безопасности, исходя из их конструктивных признаков. В настоящее время известен ряд техник статического синтаксического анализа, позволяющих эффективно определять некорректности кодирования (нефункциональные ошибки) [1]. Однако некоторые дефекты безопасности программного обеспечения (далее - ПО) могут иметь семантически или синтаксически корректную структуру, например, логические бомбы, ошибки конфигурирования, генераторы парольной информации и др. Для поиска подобных ошибок требуется привлечение эксперта, который исследует фрагменты кода повышенного риска, определяемые по некоторому шаблону [2]. В работе будет исследована возможность использования сигнатурного анализа для выявления дефектов безопасности.

В первом разделе будет определено понятие сигнатурного анализа и описана общая схема его работы. Второй раздел содержит обзор типовых дефектов безопасности и способов их выявления. Приведены примеры на языках программирования Java, Perl и PHP. В третьем разделе рассмотренные методы обобщаются, и вырабатывается методика проведения анализа. В четвертом разделе приведен пример реализации разработанной методики.

1 Место сигнатурного подхода в статическом анализе программ

Фактически под понятие сигнатурного анализа подпадает общий подход поиска тех или иных признаков объекта на основе сопоставления его содержимого с образцами из базы уже имеющихся признаков. Перспективным представляется использование данного подхода к выявлению ошибок в конфигурации приложений, а также потенциально опасных конструкций в коде приложений, в том числе программных закладок и дефектов кода.

Рассмотрим общую схему проведения сигнатурного анализа (рис. 1). В полученном на входе анализатора исходном тексте ПО, либо в оригинальном виде, либо после определенных преобразований (препарсинг, построение синтаксического дерева, канонизация, которые позволяют снизить влияние особенностей синтаксиса языка и его форматирования), производится поиск подозрительных конструкций на основе существующей базы сигнатур (шаблонов, паттернов) кода.

В процессе работы анализатора выполняются сопоставления участков кода всем находящимся в базе паттернам (сигнатурам) потенциально опасных конструкций.

Результатом работы сигнатурного анализатора как правило является набор найденных подозрительных участков кода с указанием типа вероятного дефекта или программной закладки).

К примеру, сигнатурный анализатор безопасности кода можно представить кортежем [3]:

$CSA = < SDB, HDB, MLA, MSA, MSY, MLO, MRP >$,

где: *SDB* – база данных сигнатур (паттернов) ПОК, *HDB* – база данных структурной информации о коде (иерархического представления программы), *MLA* – программный модуль лексического анализа, *MSA* – программный модуль синтаксического анализа, *MSY* – программный модуль сигнатурного анализа, *MLO* – программный модуль логического вывода, *MRP* - программный модуль построения отчетов.

Сигнатурный анализатор осуществляет сквозной поиск в исходных текстах программы паттернов ПОК, используя базу сигнатур, которая представляет собой набор следующих кортежей:

$SDB = <PE, DL>$,

где: *PE* – описание фрагмента ПОК (например, используя регулярные выражения), *DL* – оценка степени их опасности (например, числовая шкала 1-10).

Рисунок 1 - Общая схема выполнения сигнатурного анализа

2 Типовые дефекты безопасности, выявляемые методом сигнатурного анализа

В статье для демонстрации возможности выявления дефектов выбрана система классификации дефектов CWE (Common Weakness Enumeration) [4]. На сегодняшний день последняя версия стандарта 2.1 доступна по адресу cwe.mitre.org. Примеры дефектов, их признаки и способы выявления представлены в таблице 1.

Таблица 1 - Примеры способов выявления дефектов безопасности

Элемент CWE верхнего уровня	**Элемент CWE нижнего уровня**	**Признаки наличия**	**Способ обнаружения**
Проблемы поведения (Behavioral Problems) (438)	Изменение поведения в новой версии окружения (Behavioral Change in New Version or Environment) (438)	<совпадение вызова со списком проблемных для зависимости, используемой в проекте>	Определить версию среды функционирования Сравнить вызов со списком вредоносных функций
	Неверный контроль частоты взаимодействия (Improper Control of Interaction Frequency) (799)	<нет таймаута или запрета попыток> в интерактивной функции (взаимодействующей с пользователем или внешней стороной).	Определить интерактивность функционального объекта (по наличию API заданного типа) Определить внутри интерактивного объекта наличие API по обработке задержки

Элемент CWE верхнего уровня	Элемент CWE нижнего уровня	Признаки наличия	Способ обнаружения
Ошибки каналов и путей (Channel and path errors) (417)	Неверная защита альтернативного пути Improper Protection of Alternate Path - (424)	<отсутствие проверки при показе закрытой страницы>	Определить вызов функционала, отображающего содержимое страницы Удостовериться в наличии функционала контроля сессии доступа пользователя
	Скрытый канал памяти (Covert storage channel) (515)	<обращение к нестандартным информационным объектам (файл подкачки, реестр, прямой доступ к диску, операции с кучей)>	Поиск вызовов функционала из списка объектов потенциально скрытых каналов
	Скрытый канал времени (Covert timing channel) (385)	<обращение к нестандартным информационным объектам (бесконечные циклы, работа с очередью сообщений, таймером, высокопроизводительным таймером)>	Поиск вызовов функционала из списка объектов потенциальных скрытых каналов
Обработка данных (Data Handling) (19)	Неверная валидация ввода (Improper Input Validation)(20)	<поиск включения введенных пользователем данных при формировании строк>	Поиск вызовов получения пользовательских данных Поиск вызовов потенциально опасных функций с использованием пользовательских данных
Обработка ошибок (Error Handling) (388)		<наличие в блоке catch-операции вывода в журнал>	Поиск обработчиков исключений и определение внутри вызовов вывода в журнал
Ошибки обработчика (Handler Errors) (429)		<выброс исключения, у которого нет обработчика>	Создание списка всех выбросов исключений Создание списка всех обработчиков Сравнение двух списков
Злоупотребление API (API Abuse) (227)		<использование внешних компонент (библиотек) с некорректными версиями>	Составление списка всех внешних зависимостей с версиями Поиск потенциально опасных вызов из этих библиотек

Элемент CWE верхнего уровня	Элемент CWE нижнего уровня	Признаки наличия	Способ обнаружения
Индикатор плохого качества кода (Indicator of Poor Code Quality) (398)	Присваивание вместо (Assigning instead of Comparing) (481)	<наличие «=» вместо «==» в блоке if, где сравниваются лишь переменные>	Проверка заданного списка регулярных выражений для содержимого заданных типов блоков
Ошибки очистки и инициализации (Initialization and Cleanup Errors) (452)	Неверная инициализация (Improper Initialization) (665)	Отсутствие инициализации значений объектов (например, строк) перед их использованием	Поиск объявлений объектов, которые требуют инициализации перед своим использованием (например, статический массив строки) Поиск первого использования объекта в вызовах и операциях (например, правая часть в присваивании; функции, использующие их в качестве входных значений), при отсутствии их инициализации (левая часть в присваивании, функции использующие их в виде выходных значений)
Недостаточная инкапсуляция (Insufficient Encapsulation) (485)	Остаточный отладочный код (Leftover Debug Code) (489)	<наличие отладочного кода>	Поиск присутствия вызовов функций из списка отладочных Поиск подозрительных правил в названия отладочных функций (ключевые слова по debug)
Проблемы указателей (Pointer Issues) (465)	Разыменование нулевых указателей (NULL Pointer Dereference) (476)	<вызов операций по освобождению памяти для указателей, которые равны NULL>	Поиск вызовов функций, возвращающих указатель (на которую может повлиять пользователь), при отсутствии проверки ее значения следом за этим
Механизмы безопасности (Security Features) (254)	Использование жестко прописанных паролей (Use of hard-coded password) (259)	<наличие паролей и других данных авторизации непосредственно в коде приложения>	Поиск вызовов функций с параметрами авторизации, использующих аргументы со строковыми константами Поиск функций, требующих авторизации вместе с переменными, которым были присвоены строковые константы
Время и состояние (Time and State) (361)	Внешнее влияние на сферу определения (External Influence of Sphere Definition) (673)	<проверка наличия функций, использующих значения переменных окружения>	Поиск вызовов функций, читающих содержимое переменных окружения

Элемент CWE верхнего уровня	Элемент CWE нижнего уровня	Признаки наличия	Способ обнаружения
Ошибки интерфейса пользователя (User Interface Errors) (445)	Нереализованная или неподдержимая функция в GUI (Unimplemented of unsupported feature in UI) (447)	<наличие скрытых элементов GUI, а также некорректностей в обработчиках событий>	Поиск наличия disabled=true hidden=true и других подобных свойств в файле GUI> Поиск отсутствия кода в функции-обработчике (Qt, Builder)

В соответствии с рейтингом TIOBE наиболее популярными языками веб-программирования являются Perl, Java и PHP[6].

Подробнее рассмотрим некоторые распространенные дефекты с примерами кода на этих языках.

- Валидация ввода (Improper Input Validation)

Основные проблемы возникают, когда приложение полностью доверяет всем входным данным, подобный подход ведет к переполнениям буфера, межсайтовому скриптингу, SQL-инъекциям, отравлению кэша. Для большинства приложений — это приоритетная проблема.

Рассмотрим пример PHP-кода с дефектом, позволяющим выполнить манипулирование настройкой (setting manipulation):

```
<?php
   ...
   $table_name=$_GET['catalog'];
   $retrieved_array = pg_copy_to($db_connection, $table_name);
   ...
?>
```

В данном фрагменте кода происходит чтение параметра из HTTP-запроса и использование его значения в качестве активного каталога в соединении с БД.

Другой яркий пример – это дефект, дающий потенциальную возможность манипулирования процессами в системе с JVM.

Допустим, в Java-программе происходит загрузка динамической библиотеки, расположенной по стандартному пути (например, системная директория):

System.loadLibrary("library.dll");

Если посмотреть спецификацию метода System.loadLibrary() в Java 1.4.2 API, то мы увидим, что он принимает на вход имя библиотеки, а не путь, по которому она находится. Очередность просмотра папок с целью поиска этой библиотеки целиком определяется целевой системой, где запущена виртуальная машина Java (в данном случае это – ОС Windows). Для предотвращения подобного поведения рекомендуется настроить соответствующие политики и привилегии, также можно применять вызов System.load для загрузки библиотек по абсолютному пути.

В случае же использования loadLibrary злоумышленнику достаточно создать копию library.dll со своим кодом по пути, который система просмотрит раньше (например, в папке, где находится исполняемый модуль текущего процесса, либо в текущей папке при запуске приложения) и он сможет выполнить свой код в системе с привилегиями приложения.

Для выявления подобного типа дефектов в рамках сигнатурного анализа необходимо:

а) Определить строки, где происходит передача параметров от пользователя (в «чистом» PHP – это в основном GET и POST запросы);

б) Определить информационные объекты, которые получают значения от этих источников (как правило, это – присваивание);

в) Определить потенциально опасный внешний вызов с использованием выявленного ранее информационного объекта (или даже без него в случае loadLibrary).

- Злоупотребление API (API Abuse)

Интерфейс программирования приложений (Application Programming Interface, API) – по сути, это договоренность между вызывающим и вызываемым субъектами. К примеру, если программа не может вызвать функцию chdir() во время вызова chroot(), она нарушает свою сторону контракта в части активного корневого каталога. Ещё пример, когда от вызывающей стороны ждут достоверной информации о DNS-сервере, а получают значение полученное подменой. Или другой вариант: создание наследника класса SecureRandom и возврат через него неслучайного значения. В этом случае происходит нарушение правил архитектуры проектирования, поскольку новый наследник не соответствует возложенным на него требованиям.

Рассмотрим следующий пример, пусть есть код программы на языке Perl, который отображает информацию о пользователях, хранящуюся в файловой системе, открывая для этого соответствующий файл:

```
open (STATFILE, "/usr/stats/$username");
```

Однако помимо основного своего назначения функция open, согласно документации Perl, может дать возможность выполнить в конвейере произвольную команду системного шелл-скрипта:

If the filename begins with "|", the filename is interpreted as a command to which output is to be piped, and if the filename ends with a "|", the filename is interpreted as a command which pipes output to us.

Одним из наиболее известных вариантов предотвращения подобного развития событий является использование префикса “<”. Тогда вызов команды будет выглядеть следующим образом:

```
open (STATFILE, "</usr/stats/$username");
```

Для выявления подобного типа дефектов в рамках сигнатурного анализа необходимо:

а) Определить строки, где есть вызов потенциально опасного API (для этого нужны сигнатуры самих функций и объектов этого API, а также директив подключения его модулей);

б) Определить информационные объекты, либо параметры функций, которые неприемлемы (например, строк, в которых отсутствует “>” в качестве первого символа).

- Обработка ошибок

Здесь важны два аспекта: с одной стороны — отсутствие обработки ошибок или их неправильная обработка могут привести к неопределенному состоянию системы и создать уязвимость. С другой стороны, сообщая об ошибках и выдавая слишком подробную информацию об их аспектах, мы можем облегчить работу злоумышленника в исследовании системы, помочь ему в поиске «брешей» в механизмах безопасности. К примеру, в веб-приложениях имеет смысл отключить выдачу большинства ошибок и предупреждений веб-сервера и php-интерпретатора для проектов, которые уже находятся в production стадии и имеют базу постоянно работающих пользователей.

Рассмотрим фрагмент Java-программы, в котором присутствует перехват исключения NullPointerException:

```
try {
   mysteryMethod();
  }
  catch (NullPointerException npe) {
  }
```

Можно выделить три причины появления подобного кода в приложении:

а) В процессе работы программы происходит разыменование нулевого указателя, и разработчик вместо того, чтобы исправлять проблему в коде, просто сделал перехватчик этого исключения;

б) Программа намеренно выбрасывает исключение NullPointerException для выдачи сигнала об ошибочной ситуации;

в) Эта связка используется разработчиком лишь с целью отладки и тестирования в процессе разработки приложения.

Из перечисленных выше трех ситуаций лишь третью можно назвать приемлемой, да и то, лишь на стадии разработки.

Для выявления подобного типа дефектов в рамках сигнатурного анализа необходимо:

а) Определить все блоки catch в программе;

б) Проверить тип всех обрабатываемых исключений, сравнив его со списком нежелательных.

- Инкапсуляция (Insufficient Encapsulation)

В любых компьютерных системах существует своя иерархия объектов, их вложенность и границы между ними. Любое нарушение правил и пересечение этих границ потенциально создает серьезную уязвимость.

Например, если в production PHP-приложении остался следующий отладочный код:

```
<?php
  ...
  echo "Server error! Printing the backtrace";
  debug_print_backtrace();
  ...
```

?>,

то в процессе его работы будет происходить нарушения границ и утечка внутренней информации о системе, которая станет доступна рядовому пользователю бизнес-приложения.

Другим примером нарушения инкапсуляции является использования внутренних классов (inner class) в мобильном коде Java-апплетов, которые выполняются на удаленных машинах:

```
public final class urlTool extends Applet {
private final class urlHelper {
    ...
}
    ...
}
```

Проблема заключается в том, что на уровне байт-кода JVM не существует никаких inner-классов, поэтому компилятор Java должен преобразовать объявление inner-класса в обычный класс уровнем доступа на уровне пакета к изначальному внешнему классу. И как только inner-класс становится обычным, компилятор преобразует все private поля классов, к которым он обращается, в protected поля. А поскольку большинство браузеров запускают все используемые апплеты в одной и той же JVM, то потенциально возможно манипулирование объектами и состоянием полей ее окружения со стороны Java-апплета злоумышленника, работающего одновременно с апплетами других приложений.

Для выявления подобного типа дефектов в рамках сигнатурного анализа необходимо:

а) Определить перечень имен имплементируемых классов для всех классов приложения, а также классов-родителей.

б) Проверить отсутствие вложенных классов для всех классов наследующих Applet.

в) Выполнить поиск функционала из списка отладочного.

3 Разработка методики проведения сигнатурного анализа

Попробуем обобщить перечисленные выше подходы к выявлению дефектов безопасности.

Для выявления потенциально опасных конструкций необходимо решать задачи следующего рода:

1. Определение имени информационного объекта (как правило, переменной), которому происходит передача значения (как правило, присваивания) заданного потенциально опасного вызова (вызов функции, обращение к ассоциативному массиву значений).

2. Определение внешнего вызова (прежде всего функции) из списка потенциально опасных для данного языка, платформы или при условии подключения/не подключения заданного модуля/библиотеки и другого подобного контекста.

3. Определение внешнего вызова (прежде всего функции) из списка потенциально опасных, при условии использования в нем информационного объекта с известным именем.

4. Определение внешнего вызова (прежде всего функции) из списка потенциально опасных, при условии использования в нем аргумента (строкового параметра или информационного объекта) содержащего заданное строковое регулярное выражение.

5. Определение внешнего вызова из списка потенциально опасных, который использует тот же информационных объект, что уже был использован потенциально опасным вызовом.

6. Поиск всех определений блочных конструкций языка (например, перехват исключений или определение класса) использующих объект заданного типа (класс-родитель, аргумент перехвата исключения).

7. Определение для каждой блочной конструкции (например, класса) наличия заданного вложенного объекта (метода, другого класса) при условии выполнения условия (5).

Одним из решающих преимуществ сигнатурного анализа является его быстрота работы и легкость конструирования сигнатур (по сути, их можно оперативно дорабатывать и настраивать для решения любой узкой задачи распознавания наборов). В дальнейшем, на основе распознанных конструкций, можно построить любую, самую сложную логику работы анализатора.

Чтобы решить задачи, описанные выше и построить базис для подобной логики, для каждого из поддерживаемых анализатором языков программирования требуется решить следующие задачи:

1. Выполнить разбиение исходных текстов на строки операций с удалением комментариев, избыточных символов пробелов, переносов строк и другой незначимой для данного вида анализа информации;

2. Выделить в строке операции вызова функционального объекта и извлечь литерала его названия (для повышения точности операции имеет смысл добавить небольшой список ключевых слов-исключений, представляющий зарезервированные слова этого языка, например: for, while, if).

3. Выделить в строке операции передачу значения (в первую очередь, присваивание).

4. Выделить литералы имен объектов, предположительно получающих значения, и литералов имен объектов, служащих источником данных значений.

Данный вид анализа в литературе по созданию компиляторов получил название «анализ потоков данных» (data-flow analysis). Рассмотрим схему работу анализатора, использующего сигнатурный анализ и анализ потоков данных для выявления дефектов программного кода.

Предложим следующее представление кода. По сути, каждая значимая строка конструкций исходного кода (за исключением специфических макросов или других директив компилятора) может быть представлена в виде следующего кортежа:

$CD=<C, I, O>$,

где *C* – это вызываемый элемент, *I* – это перечень элементов, чьё значение считывается, *O* – это перечень элементов, чье значение изменяется на основе считанных элементов.

Каждое поле типа «элемент» может быть одного из трех типов:

– объект (в данном элементе объявляется или создается объект, использующий память);

– ссылка (в данном выражении находится лишь имя или косвенная ссылка на имя существующего объекта);

– литерал (в данном выражении находится непосредственное значение элемента – строка, число и т.п.).

4 Пример реализации сигнатурного анализа

Вернемся к рассмотренному ранее примеру PHP-кода с манипулированием настройками, но попробуем усложнить в нем задачу для анализатора, добавив дополнительную переменную, хранящую промежуточное значение переменной:

```
<?php
  ...
  $section_name=$_GET['catalog'];
  $table_name='catalog'.$section_name;
  $retrieved_array = pg_copy_to($db_connection, $table_name);
  ...
?>
```

Тогда с помощью следующих сигнатур, построенных на основе регулярных выражений, возможны:

1) поиск элементов типа «ссылка» на изменяемые информационные объекты (I):

```
\W*(\$\w+)\W*[=]
```

2) поиск элементов типа «ссылка», изменяющих информационные объекты (O):

```
[=]\W*(\$\w+)\W*
```

3) поиск элементов типа «литерал», изменяющих информационные объекты (O):

```
[=]\W*(\'\w+)'\W*
```

4) поиск вызываемых элементов типа «ссылка» (C):

```
\W*\w+\(\W*
```

5) Сигнатура потенциально опасной операции (вызов операции способной уничтожить большой объем данных СУБД):

```
pg_copy_to
```

6) Сигнатура потенциально опасной операции (использование данных полученных от пользователя):

```
\$_GET
```

Таким образом, с помощью сигнатур было получено следующее представление кода в виде кортежей C,I,O (знаком «0» обозначены пустые элементы):

```
(C,I,O)

1:(0,“section_name”,”_GET”)

2:(0,“table_name”,”section_name”)

3:(“pg_copy_to”,“retrived_array”,”table_name”)
```

Далее на основе самой простой модели потока управления (считаем, что будут выполнены безусловно ВСЕ операторы в рамках функционального объекта кода), мы

начинаем производить замену имен объектов, получающих значения, кортежем объектов, передающих значения:

Шаг 1:

```
1:(0,(”_GET”),”_GET”)
```

Шаг 2:

```
2:(0,(”_GET”),”section_name”)
```

Шаг 3:

```
3:(“pg_copy_to”,“retrived_array”, (”_GET”))
```

После третьего шага у нас срабатывают одновременно две сигнатуры потенциально опасных операций, и мы можем сигнализировать о наличии потенциально опасной конструкции, связанной с валидацией данных.

Заключение

Исследование существующих баз дефектов кода ПО и изучение реальных примеров кода показали, что достаточно широкий спектр задач, связанных с выявлением потенциально опасных конструкций, можно решить с помощью сигнатурного анализа с внутрипроцедурным анализом потока данных (intro-procedural data-flow) [5].

Среди достоинств сигнатурного метода статического анализа можно назвать относительную простоту реализации, очень высокую скорость работы, а также легкость портирования сигнатур на различные платформы и языки программирования.

Недостатком сигнатурного подхода является необходимость учитывать различные «вариации» конструкций и блоков кода, которые допускает синтаксис языка программирования и логика выполнения программы.

В связи с этим наиболее эффективной роль сигнатурного анализа видится в исследовании зависимостей модулей программ и внешних компонент, поиске вызовов функциональных объектов, а также проверке содержимого информационных объектов

программы. Для класса ошибок, связанных с синтаксисом программы и достижимостью кода, его следует применять совместно с другими методами.

Работа проведена на средства государственного контракта № 07.514.11.4114 Минобрнауки.

Список литературы

1. Глухих М.И., Ицыксон В.М. Программная инженерия. Обеспечение качества программных средств методами статического анализа. СПб.: Изд-во Политехн. ун-та, 2011. 150 с.

2. Марков А.С., Миронов С.В., Цирлов В.Л. Выявление уязвимостей программного обеспечения в процессе сертификации // Известия Южного федерального университета. Технические науки. 2006. Т. 62, № 7. С. 82-87.

3. Марков А.С., Фадин А.А., Цирлов В.Л. Концептуальные основы построения анализатора безопасности программного кода // Программные продукты и системы. 2012. № 2.

4. Марков А.С., Фадин А.А., Цирлов В.Л. Систематика дефектов и уязвимостей программного обеспечения // Безопасные информационные технологии : сборник трудов 2-й Научно-Технической конференции. М.: МГТУ им. Н.Э.Баумана. 2011. С. 83-87.

5. Вылегжанин В.В., Маркин А.Л., Марков А.С., Уточка Р.А., Фадин А.А., Фамбулов А.К., Цирлов В.Л. Система для определения программных закладок : пат. 114799 Российская Федерация.- № 2011153967/08(081180); заявл. 29.12.11; опубл. 10.04.12, Бюл. № 10. 2 с.

6. TIOBE Programming Community Index for July 2012: Objective-C overtakes C++. Режим доступа: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html> (дата обращения 04.08.2012).

SCIENTIFIC PERIODICAL OF THE BAUMAN MSTU

SCIENCE and EDUCATION

EL № FS77 - 48211. №0421200025. ISSN 1994-0408

electronic scientific and technical journal

Application of static signature analysis to detect defects in web applications security

 # 09, September 2012

DOI: 10.7463/0912.0461281

Medvedev N.V., Markov A.S., Fadin A.A.

Russia, Bauman Moscow State Technical University

Russia, Moscow, CJSC «NPO «Echelon»

mail@cnpo.ru

a.markov@npo-echelon.ru

a.fadin@npo-echelon.ru

The article presents a signature analysis and its workability to detect security defects in a program code. The authors review typical security defects and techniques for detecting them. They developed a flow chart of the analyzer using signature analysis and data flow analysis for detection of program code defects. The proposed method can be applied in security analysis of web applications.

Publications with keywords:certification, flaw, vulnerabilities, software securityre, signature analysis, static code analysis

Publications with words:certification, flaw, vulnerabilities, software securityre, signature analysis, static code analysis

References

1. Glukhikh M.I., Itsykson V.M. *Programmnaia inzheneriia. Obespechenie kachestva programmnykh sredstv metodami staticheskogo analiza* [Software engineering. Ensuring the quality of the software by methods of static analysis].St. Petersburg, SPbSTU Publ., 2011. 150 p.

2. Markov A.S., Mironov S.V., Tsirlov V.L. Vyiavlenie uiazvimostei programmnogo obespecheniia v protsesse sertifikatsii [Identification of software vulnerabilities in the process of certification]. *Izvestiia Iuzhnogo federal'nogo universiteta. Tekhnicheskie nauki* [Bulletins of the southern Federal University. Technical Sciences], 2006, vol. 62, no. 7, pp. 82-87.

3. Markov A.S., Fadin A.A., Tsirlov V.L. Kontseptual'nye osnovy postroeniia analizatora bezopasnosti programmnogo koda [Conceptual foundations of construction of analyzer of security of programming code]. *Programmnye produkty i sistemy* [Software products and systems], 2012, no. 2.

4. Markov A.S., Fadin A.A., Tsirlov V.L. Sistematika defektov i uiazvimostei programmnogo obespecheniia [Systematics of defects and vulnerabilities of software]. *Bezopasnye informatsionnye tekhnologii :*

cbornik trudov 2-i Nauchno-Tekhnicheskoi konferentsii [Secure information technologies : proceedings of the 2nd Scientific and Technical conference]. Moscow, Bauman MSTU Publ., 2011, pp. 83-87.

5. Vylegzhanin V.V., Markin A.L., Markov A.S., Utochka R.A., Fadin A.A., Fambulov A.K., Tsirlov V.L. *Sistema dlia opredeleniia programmnykh zakladok* [System for determining program bookmarks]. Patent RF, no. 114799, 2012.

6. *TIOBE Programming Community Index for July 2012: Objective-C overtakes C++*. Available at: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.htm , accessed 04.08.2012.