АНАЛИЗ УЯЗВИМОСТЕЙ ПРОГРАММНОГО КОДА МЕТОДОМ ШАБЛОНОВ

Е. Л. Евневич, С. В. Перминов, Е. В. Белаш, М. А. Внуков

Санкт-Петербургский институт информатики и автоматизации РАН

СПИИРАН, 14-я линия ВО, д. 39, Санкт-Петербург, 199178

<eva@iias.spb.su>

УДК 681.3

Евневич Е.Л., Перминов С.В., Белаш Е.В., Внуков М. А. **Анализ уязвимостей программного кода методом шаблонов** // Труды СПИИРАН. Вып. 7. — СПб.: Наука, 2008.

Аннотация. *Программа представляется в виде последовательности формализованных инструкций, из которых можно выделить характерные подпоследовательности команд - уязвимости. Нахождение таких подпоследовательностей и определение их функционального назначения производится с помощью семантического анализа кода на базе шаблонов.* — Библ. 4 Назв.

UDC 681.3

Evnevich E.L., Perminov S.V., Belash E.V., Vnukov M.A. **Vulnerability analysis of program code by pattern techniques** // SPIIRAS Proceedings. Issue 7. — SPb.: Nauka, 2008.

Abstract. *Program code is represented in the form of a sequence of formalized instructions, in which characteristic subsequences of instructions – vulnerabilities – may be marked out. Detection of such subsequences and determination of their functions is fulfilled by means of code semantic analysis on a pattern basis.* — Bibl. 4 items.

1. Введение

Анализ программного кода – часто встречающаяся задача в области компьютерных технологий. Его автоматизированная смысловая переработка позволит расширить возможности исследования программных языков и создать новые технологии разработки кода.

Анализируемая программа может быть представлена алгоритмической последовательностью формализованных инструкций, из которых выделяются некоторые подпоследовательности команд, характеризующие алгоритм. Эти подпоследовательности представляют из себя самостоятельные логические сущности, несут определенную атомарную функциональность и обычно повторяются с небольшими различиями в других алгоритмах. В самом очевидном случае эти подпоследовательности объединяются в отдельные процедуры, методы, классы, паттерны (шаблоны) проектирования. В общем случае обычно используется термин «закономерности».

Нахождение таких подпоследовательностей и определение их функционального назначения – задачи, решение которых позволит производить семантический анализ алгоритмов и улучшит понимание законов и принципов их построения. Именно выявление общих закономерностей позволяет перейти от простого перечисления инструкций к процедурной парадигме программирования.

Из множества источников, существующих в настоящее время, можно отметить [1–3], в которых излагаются методики анализа рисков оценки защищенности информационных систем, работы [4,5] посвящены взаимосвязи задач анализа защищенности и обнаружения вторжений с задачей управления

рисками. В работах [6, 7] изложена разработка руководства по метрикам безопасности информационных систем.

Общие закономерности в использовании переменных (например, массив часто сопровождается переменной, содержащей его длину) привели к понятию структуры данных. Постоянное использование данных и структур данных вместе с определенным набором функций привело к разработке полноценной объектно-ориентированой парадигмы. Таким же образом возникла идея паттернов проектирования, аспектно-ориентированого программирования и т.п. До сих пор все выводы о наличии тех или иных закономерностей в программировании имели случайный характер, основанный исключительно на опыте программистов. Очевидно, что автоматизированное средство анализа программных реализаций алгоритмов представляет ценность для дальнейшего развития всей сферы компьютерных технологий. Перспективным направлением представляется разработка интеллектуальных методов определения уязвимого кода. Их суть сводится к определению алгоритма и спецификации программы по ее коду и выявлению программ, осуществляющих несанкционированные действия [8]. Именно к этому классу относится метод, предлагаемый в данной работе. Новизна метода заключается в том, что фрагменты уязвимого кода обладают схожими функциональными элементами, наличие или отсутствие которых можно определить в программе при помощи семантического анализа. В данной статье предложена программа, осуществляющая семантический анализ кода на основе построения его дерева и выделения из последнего закономерностей, представляющих интерес для решения задачи описания деструктивных участков кода.

2. Постановка задачи

Предлагаемая система семантического анализа программ (далее – ССАП) представляет собой пример решения задачи нахождения закономерностей. Предложенные алгоритмы имеют широкий спектр применения, но в рассматриваемом случае они используются в достаточно узкой проблемной области – поиск и сбор шаблонов, характерных для деструктивных программных средств, в частности вирусов. Сбор информации и анализ осуществляется для вирусов, написанных на языке Java, чтобы облегчить технические сложности с нахождением исходного кода для произвольной анализируемой программы.

3. Описание системы

Данная система предназначена для поиска определенных шаблонных участков кода в исследуемом программном коде. Спектр задач, выполняемых системой, определяется используемыми шаблонами. Самым очевидным ее использованием является поиск деструктивных участков кода в программе, другими словами, анализ программы на предмет наличия в ней функциональных элементов, которые могли бы указать на выполняемые функции, которые идут вразрез с ожиданиями пользователя и наносят ему вред. Для этого достаточно собрать базу данных элементов, характерных для вирусных средств, но

малохарактерных для обычных программ, и произвести поиск таких функциональных шаблонов в исследуемой программе.

ССАП состоит из двух подсистем: подсистемы синтаксического разбора и подсистемы сравнения синтаксических деревьев.

Работа происходит в два этапа (рис. 1).

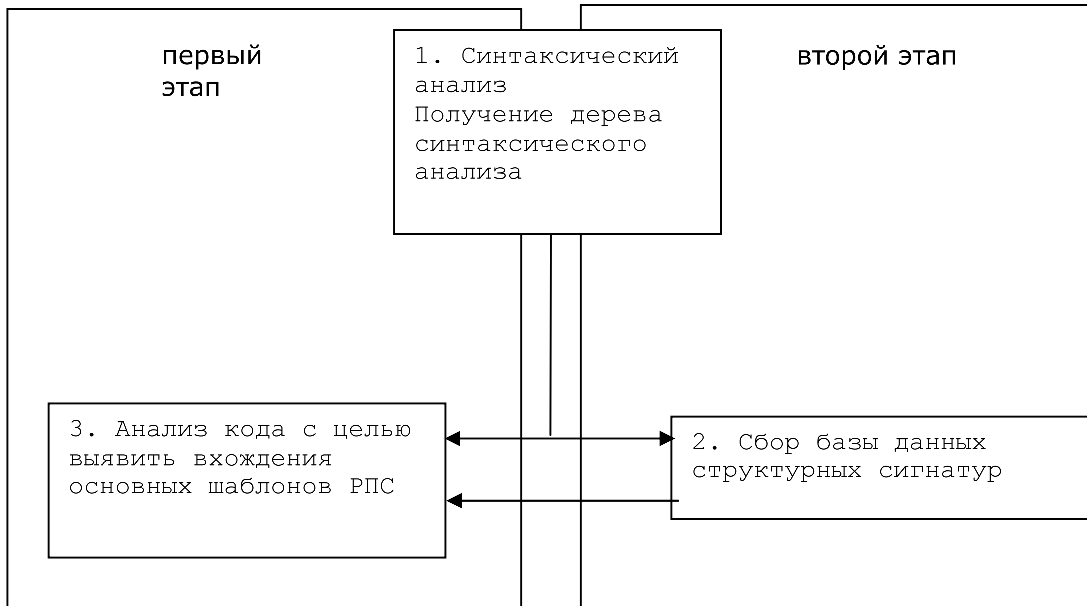

Рис. 1. Блок-схема системы семантического анализа программ.

Первый этап — формирование базы данных функциональных шаблонов, второй этап — анализ программ с помощью данных, полученных на первом этапе. Обе подсистемы используются на каждом этапе.

4. Подсистема синтаксического разбора

Сам по себе программный код не является достаточно абстрактным выражением семантики алгоритма; алгоритм может иметь несколько внешне отличающихся реализаций на одном языке, не имеющих между собой никаких общих частей. Сравнение таких кусков кода, естественно, не имеет смысла. Для простейшего примера рассмотрим два листинга:

Листинг 1

```
public Beyond() {
try {
   PrivilegeManager.enablePrivilege("SuperUser");


   String sysname = System.getProperty(
```


Труды СПИИРАН. 2008. Вып. 7. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2008. Issue 7. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

```
            "os.name").toLowerCase();
        if (sysname.startsWith("windows")) {
            String cmd[] = new String[1];
            cmd[0] = "notepad.exe";
            Process p = Runtime.getRuntime().exec(
                    cmd);
        }
    } catch (Throwable t) {
    }
```

Очевидно, что алгоритмически он эквивалентен следующему коду:

Листинг 2

```
  public Beyond() {
    try {
        String sysname = System.getProperty(
            "os.name").toLowerCase();

        if (sysname.startsWith("windows")) {
                escalatePrivilegies();
                runNotepad();
                }
    } catch (Throwable t) {
    }
}

private void runNotepad() throws IOException {
    String cmd[] = new String[1];
    cmd[0] = "notepad.exe";
    Process p = Runtime.getRuntime().exec(
            cmd);
}
private void escalatePrivilegies() {
    PrivilegeManager.enablePrivilege("SuperUser");
}
```

Разница между этими двумя листингами минимальна; алгоритм, представленный в них, один и тот же, но, хотя здесь применен только простейший рефакторинг – выделение метода, реализации существенно отличаются друг от друга.

Синтаксический разбор программы необходим для того, чтобы перевести программный код на более абстрактный уровень семантики, т.е. выделить из кода его алгоритм в удобной для анализа форме. Необходимо выделить основные структуры: циклы, процедуры и т.п. [9].

Для синтаксического анализа любого языка необходимо иметь его грамматику, описывающую, по каким правилам из лексем (терминальные символы или терминалы) строятся его предложения. Применительно к рассматриваемой задаче используется грамматика языка Java (см. Листинг 1), однако дерево синтаксического разбора в памяти будет иметь более сложную структуру, которая лучше отвечает целям анализа.

Построение дерева проводится на примере вируса Trojan.Java.ClassLoader.d, при декомпиляции создается файл Parser.java. Для отдельных участков кода ниже дан дамп синтаксического дерева соответствующего участка с объяснением обозначений:

Листинг 3

```
import com.ms.security.SecurityClassLoader;
public class VerifierBug extends SecurityClassLoader
{
          <CompilationUnit>
             <!-- начало модуля -->
            <Import>
               com.ms.security.SecurityClassLoader
               <!-- строка импорта -->
            </Import>
            <Class>
              VerifierBug
               <!-- имя класса -->
             <SuperClass>
               SecurityClassLoader
               <!-- имя суперкласса -->
             </SuperClass>
          […]
            </Class>
```


Труды СПИИРАН. 2008. Вып. 7. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2008. Issue 7. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

```
public VerifierBug(int i)
{
}
        <ConstructorDeclaration>
         VerifierBug
        <!--дополнительный конструктор -->
           <FormalParameter>
             null
           </FormalParameter>
        <!--с входным параметром -->
            <Type>                - типа
              <INT>               - int
                <VariableDeclaratorId>
                   I
         <!-- и именем I -->
                </VariableDeclaratorId>
              </INT>
            </Type>
        </ConstructorDeclaration>
```

Данное представление кода не очень удобно для чтения, однако видно, что в полученном дереве выделены такие блоки, как циклы, условия, вызовы функции. В таком виде оно предоставляет определенные возможности для анализа кода и выделения общих подпоследовательностей. В реальности для большей эффективности и для того, чтобы избежать «запутывания» в простейших случаях, необходимо создавать более сложные структуры данных.

5. Подсистема сравнения синтаксических деревьев

Полученное с помощью первой подсистемы синтаксическое дерево предоставляет гораздо больше возможностей для анализа, чем простой код. С помощью второй подсистемы производится сравнение двух синтаксических деревьев с целью поиска в них общих закономерностей.

Сравнение двух файлов с исходным кодом – известная задача динамического программирования, реализованная в таких популярных программных средствах, как diff [10]. Необходимо найти максимальную общую для файлов (в данном случае для синтаксических деревьев) подпоследовательность строк.

В процессе работы было обнаружено, что некоторые вирусы одного вида,

обозначенного в базе данных Касперского как Trojan, при выводе общих подпоследовательностей выдают только хаотические короткие обрывки, из чего следует, что файлы не имеют общих смысловых участков. Однако при сравнении таких файлов, как например VerifierBug.java (значится как Exploit.Java.ByteVerify), Parser.java (значится как Trojan.Java.ClassLoader.d), InsecureClassLoader.java (значится как Trojan-Downloader.Java.OpenConnection.o) и некоторых других получаются общие фрагменты кода следующего вида:

1.

Name SecurityClassLoader –имя суперкласса.

2.

FormalParameter

Type null

INT null

VariableDeclaratorId i

Method loadClass

Type null

Name Class

FormalParameter

Type null

Name String

FormalParameter l

Type

BOOL

Block

Это объявление метода loadClass, возвращающего переменную типа Class с двумя переменными типа String и BOOL, с уровнем доступа protected.

3.

IfStatement

UnaryExpression

Name s.equals

Arguments

Literal "com.ms.vm.loader.URLClassLoader"

UnaryExpression

Arguments

Literal "com.ms.vm.loader.URLClassLoader"

Name UCL_definition

Труды СПИИРАН. 2008. Вып. 7. ISSN 2078-9181 (печ.), ISSN 2078-9599 (онлайн)
SPIIRAS Proceedings. 2008. Issue 7. ISSN 2078-9181 (print), ISSN 2078-9599 (online)
www.proceedings.spiiras.nw.ru

Literal 0

Name UCL_definition.length

Этот фрагмент соответствует вызову в блоке if при условии выполнения условия равенства строки s строке com.ms.vm.loader.URLClassLoader, разных для каждого файла функций с параметрами:

1. "com.ms.vm.loader.URLClassLoader"

2. UCL_definition

3. 0

4. UCL_definition.length

И хотя общность файлов на этом не заканчивается, уже этих трех фрагментов достаточно, чтобы увидеть определенную тенденцию, указывающую на наличие известной уязвимости в java машине, описанной еще группой польских исследователей безопасности Last Stage of Delirium [11]. Для правильного ее использования необходимо создать класс, являющийся преемником ClassLoader (либо SecurityClassLoader), переопределить функцию loadClass, в которой необходимо вызвать собственно определенную функцию, подменяющую системный класс (в данном случае, очевидно, используется URLClassLoader) своим собственным созданным классом.

6. Характеристика результатов

Особенность результата заключается в том, что в принципе нет необходимости вдаваться в детали использования уязвимости. Полученных данных уже достаточно, чтобы определить все деструктивные программные средства, использующие подобную технологию, потому что все они имеют общую логику работы, легко определяемую в результате анализа.

Работа системы проходит в два этапа: первый этап – сбор базы данных закономерностей, второй этап – анализ программы.

Первый этап предварительный, его цель – выявление закономерностей программ определенного типа. Выделение общих шаблонов как для вирусных, так и для любых других программ дает дополнительные знания о работе алгоритма и, вероятно, может пригодиться для других средств, таких как средства разработки, рефакторинг кода, улучшенное представление кода, вывод полезной статистической информации, а также средства семантического поиска. Предлагаемый метод повышает эффективность статистического анализа ПС, как сами результаты статистического анализа (для рефакторинга), то есть выделенные типовые участки кода, так и системы поиска шаблонов в наборе программ (в качестве антивирусного средства).

Для того чтобы выявить закономерности алгоритмов различных ПС, необходимо собрать большую выборку таких алгоритмов. Для этого собирается

массив экземпляров исследуемого типа программ, для каждой программы строится синтаксическое дерево, а затем с помощью серии сравнений выводятся общие закономерности. В данной экспериментальной системе использовалась база данных вирусов лаборатории Касперского. В общей сложности было собрано около 150 вирусных средств различных типов, проведены сравнения и выявлены всего порядка 20 функциональных элементов, определенных как «подозрительные», то есть элементы, которые в обычной программе встречаться не должны (как, например, запись в реестр для Java-апплета).

Второй этап рабочий, его цель – непосредственное выявление наличия определенных шаблонов в тексте программы. Очевидно, что если на первом этапе исследуются экземпляры известных программ с известным исходным кодом, то на втором этапе работа проходит с реальной программой, исходный код которой в общем случае неизвестен. Задача декомпиляции решается с помощью консольной программы Jad, производящей исходный программный код из заданного байт-кода Java класса. Следует отметить, что генерация абсолютно правильного или даже компилируемого кода совсем необязательна. Декомпиляция, допускающая до 20 процентов неразобранного кода (это часто происходит при декомпиляции Java вирусов, созданных в обход стандартных средств компиляции), все равно предоставляет достаточно информации для анализа, что позволяет с большой степенью уверенности делать утверждения насчет «дружелюбности» исследуемой программы.

7. Заключение

Семантический анализ показал свою высокую эффективность в задаче частичного определения целей кода (определение его «дружелюбности» или злонамеренности). Описанные в статье средства по трансформации и сравнению кода позволили эффективно определить злонамеренность кода. Таким образом, можно сформулировать следующие выводы:

1) предложенное средство анализа обеспечивает решение задачи определения целей алгоритма или его отдельных частей;

2) подобные средства эффективны в таких узкоспециализированных задачах, как например определение вирусов. Очевидно также, что задачей обнаружения злонамеренного кода не исчерпываются все задачи, которые могут быть решены с помощью двух представленных подсистем.

Литература

1. *Alberts C., Dorofee A*. Managing Information Security Risks: The OCTAVE Approach. Addison Wesley Professional, 2002.
2. *Chapman C., Ward S*. Proect Risk Management: Process, Techniques and Insights. Chichester, John Wiley, 2003.
3. *Peltier T. R. , Peltier J., Blackley J. A.* Managing a Network Vulnerabilty Assessmrnt. Auerbach Publications, 2003.
4. *Петренко С. А., Симонов С. В.* Управление информационными рисками. Экономически оправданная безопасность. М.: Компания АйТи; ДМК Пресс, 2004.
5. *Астахов А.* Анализ защищенности корпоративных автоматизированных систем //Jet Info. №7, 2002.

6. *Афанасьев С.В., Воробьев В.И.* Метрики для объектно-ориентированного проектирования сложных систем // Вестник гражданских инженеров. 2005. №4. С. 108 - 114.
7. *Котенко И.В., Степашкин М.В.* Метрики безопасности для уровня защищености компьютерных сетей на основе построения графов атак. Защита информации. 2006. №3. С. 36 - 45.
8. *Mila D. P., Mihai C., Somesh J., Saumya D.* A semantics-based approach to malware detection // ACM SIGPLAN Notices. 2007. Vol. 42, Issue 1. P. 377 – 388.
9. *Ахо А В, Сети Р., Ульман Д.* Компиляторы: принципы, технологии и инструменты // М.: Издат. Дом "Вильямс". 2001. 745 с.
10. Component Software Diff for Windows [Электронный ресурс] // <http://www.componentsoftware.com/products/CSDiff/index.htm > (по состоянию на 29.12.2008).
11. HITB 2003 Security Conference - Last Stage of Delirium (LSD) - Security Myths [Электронный ресурс] // <http://www.archive.org/details/hitb2003-Last-Stage-of-delirium> (по состоянию на 29.12.2008).