УДК 004.056.52 - 004.056.57

А.В. Благодаренко

СХЕМА ВЫЯВЛЕНИЯ ВРЕДОНОСНЫХ ДАННЫХ, ПОСТУПАЮЩИХ ИЗ СЕТИ НА ОСНОВЕ ДИНАМИЧЕСКОГО АНАЛИЗА ИСПОЛНЯЕМОГО КОДА

Предложена схема исследования воздействия данных из сети на программное обеспечение без исходных текстов. Рассмотрен метод слежения за распространением внешних по отношению к исследуемому ПО данными и выявления их использования с целью компрометации ПО. Предложен способ сбора образцов данных для схемы. Файлы с данными в формате, поддерживаемом приложением, получаются из открытого спутникового канала и представляют собой типовой интернет-трафик.

Уязвимости ПО; тестирование; верификация; fuzzing; спутниковый интернет; потоки информации.

A.V. Blagodarenko

SCHEME FOR HARMFUL NETWORK DATA DETECTION WITH EXECUTABLE CODE DYNAMIC ANALYSIS

The article offers the approach for researching data from the network impact to software without sources code research. The method of tracking the spread of external data and identify their use in order to compromise the software shown. An approach of collecting data for the scheme offered. Data files in an application-defined format can be obtained from the open satellite channel and represent, in some way, the typical Internet traffic.

Software vulnerabilities; testing; verification; fuzzing; satellite Internet; dataflow.

Уязвимости ПО – важная проблема информационной безопасности. Сложность разрабатываемых продуктов постоянно увеличивается, а вместе с этим увеличивается сложность их тестирования. Не найденные в процессе тестирования ошибки могут привести к неработоспособности части функционала, но самое неприятное, что они могут создать брешь в системе безопасности. Уязвимость в системе безопасности сетевого приложения может привести к удаленному захвату управления над системой. По этой причине в дополнение к стандартным процедурам тестирования ПО применяются специальные, ориентированные на проверку безопасности. Такие проверки, к примеру, являются неотъемлемой частью методологии SDL [1], которую применяет в разработке своих продуктов компания Microsoft.

При анализе могут использоваться различные подходы. Принято разделять методы на следующие типы: тестирование «черного ящика», статический и динамические анализ. Также методы разделяют на автоматические и ручные. С применением исходного кода и без. Каждый метод имеет свои положительные и отрицательные стороны в зависимости от того, на сколько доступно исследуемое ПО. Для проведения статического и динамического анализа необходим доступ к

исполняемому коду, а для некоторых методов и к исходному коду. Тестирование методом «черного ящика» может проводиться и удаленно, без доступа к коду.

По данным Web Application Security Consortium (WASC) [2], «Около 49 % Web-приложений содержат уязвимости высокой степеней риска (Urgent и Critical), обнаруженные при автоматическом сканировании систем (Т.1). Однако при детальной ручной и автоматизированной оценке методом белого ящика вероятность обнаружения таких уязвимостей высокой степени риска достигает 80 – 96 %. Вероятность же обнаружения уязвимостей степени риска выше среднего (критерий соответствия требованиям PCI DSS) составляет более 86 % при любом методе работ. В то же время при проведении более глубокого анализа 99 % Web-приложений не удовлетворяет требованиям стандарта по защите информации в индустрии платежных карт».

Согласно статистике из этого же источника [2] количество сайтов, на которых обнаружено более одной уязвимости, зависит от применяемого метода исследования, но различия эти незначительны (рис. 1). Столбец «Scans» описывает уязвимости, найденные с помощью автоматических средств со стандартными настройками. Статистика из столбца «BlackBox» достигнута с помощью детального анализа, но без доступа к кодам ПО. Столбец «WhiteBox» – уязвимости, найденные с помощью статического и динамического анализа исполняемых и исходных кодов ПО серверов. Улучшение результатов в зависимости от детальности изучения ПО связано с тем, что оценка рисков становится более адекватной, и учитывается не только тип уязвимости, но и реальные последствия ее эксплуатации с учетом архитектуры и реализации приложения. В статистике об автоматическом тестировании участвуют также сайты, которые не имеют активных элементов, а такие сайты заведомо будут иметь меньше уязвимостей [2]. Результаты автоматизированных сканирований описывают среднестатистические интернет-сайты, результаты более детальных тестов относятся к интерактивным Web-приложениям.

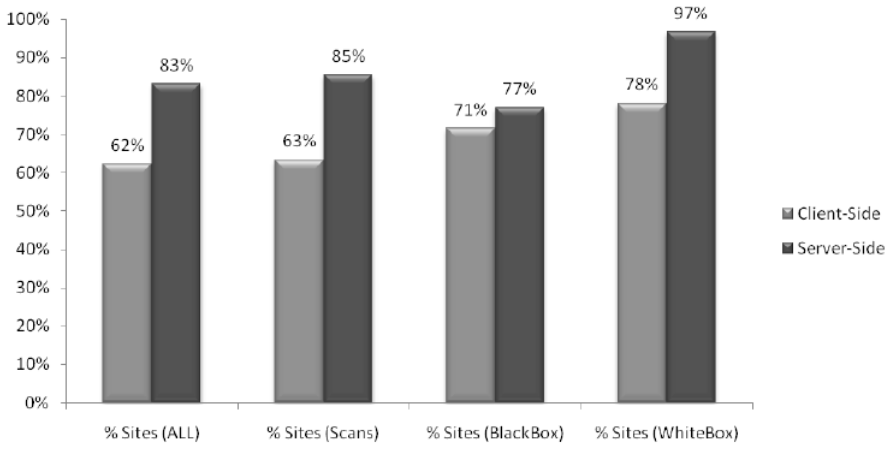

Рис. 1. Вероятность обнаружения уязвимости по типу воздействия

Если рассматривать зависимость среднего количества найденных уязвимостей на одном сайте от выбранного метода, то статический и динамический анализ сетевого ПО значительно эффективнее исследований методом «черного ящика». На рис. 2 колонка “WhiteBox” описывает эффективность работы методов, для которых необходим доступ к исходным или исполняемым кодам приложений. Следует учесть, что методики из ряда

«WhiteBox» могут содержать элементы методик «черного ящики». Данные для приложения могут генерироваться специальным образом и соответствующая реакция на них фиксироваться с помощью динамического анализа.

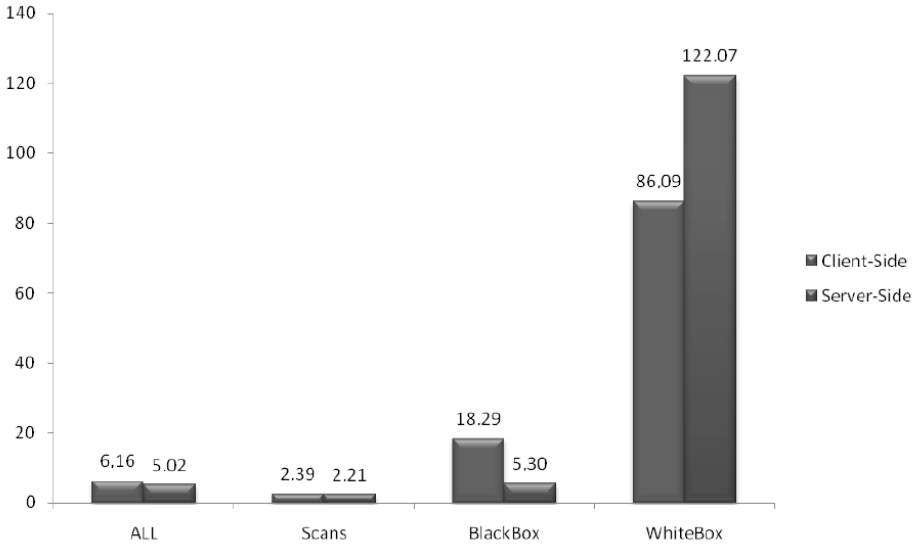

Рис. 2. Распределение уязвимостей на один сайт при использовании различных методов их поиска

Методы, к которым относится статистика из столбца «Scans», являются полностью автоматическими. С их помощью можно за небольшой промежуток времени провести анализ большого количество образцов. Однако ручной анализ или исследования, в которых применяется сочетание ручных и автоматических методов, более эффективны, если цель – наиболее глубокое изучение одного образца.

Следует, однако, учесть, что большое число программных продуктов распространяется без исходных кодов. В данной статье внимание уделено исследованию именно такого ПО. Отсутствие доступа к исходных кодам, с одной стороны усложняет статический анализ, а с другой – максимально приближает к рабочим условиям приложения. По этой причине целесообразно статический анализ дополнить исследованиями реакции на входные данные. Под входными данными понимается любой ввод извне: данные из сети или пользовательский ввод.

Таким образом, схема верификации программного обеспечения на отсутствие уязвимостей безопасности может выполняться по схеме, изображенной на рис. 3 [3].

Стенд работает с бинарными файлами приложения. Файлы с отладочной информацией, если они есть, могут использоваться для получения дополнительной информации. Исследованное ПО запускается под управлением стенда. Сразу же после старта проводится статический анализ главного исполняемого модуля. Статический анализ подгружаемых библиотек проводится по мере необходимости. Все полученные в ходе статического анализа данные сохраняются в базе данных. Начиная с этого момента, они могут быть доступны для анализа через графический интерфейс пользователя. Далее в ходе динамического анализа

уточняются данные, полученные при статическом разборе. Так, определяются адреса передачи управления, которые нельзя определить статически, – это переходы по значению регистра или ячейки памяти. После того как получено достаточно статистических данных о работе приложения в штатных режимах, можно переходить к применению методик «черного ящика». На основе заданных правил генерируется контент. Правило может предписывать создавать случайные или по заданному алгоритму последовательности байтов входных данных. Для генерации контента может быть использована спецификация формата входных данных. Далее в статье будет предложен способ получения данных для схемы на основе фильтрации спутникового трафика.

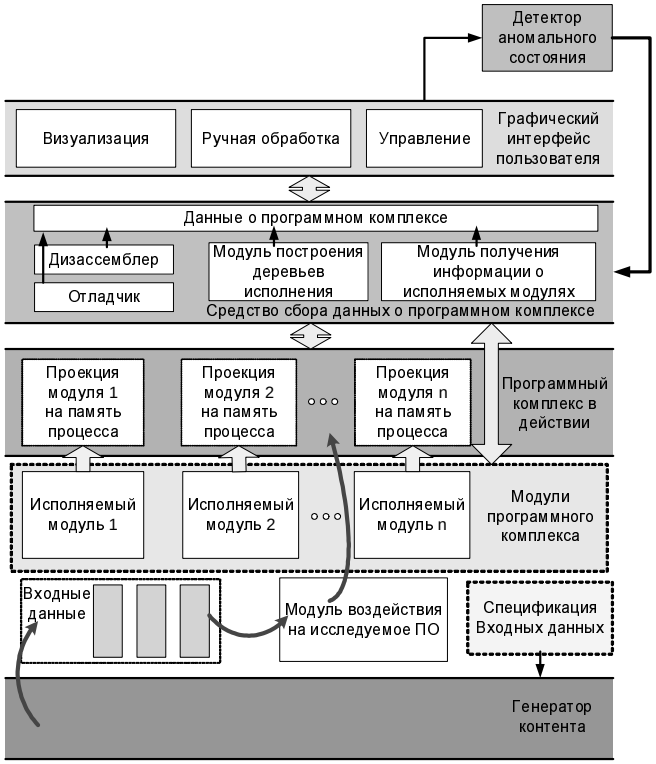

Рис. 3. Схема стенда для верификации программного обеспечения

Модуль воздействия на исследуемое ПО подставляет сгенерированные данные в нужный участок памяти программы или иным способом добивается обработки кодом этих данных. Реакция на входное воздействие фиксируется средствами сбора данных об исследуемом ПО. К подобным средствам относится модуль построения деревьев исполнения и модуль получения информации о модулях ПО.

Графический интерфейс пользователя используется для визуализации собранных данных. Данное представление может быть полезным при дополнительном ручном анализе. Информация представляется в удобном для анализа виде: граф исполнения (CFG) с непрерывными участками кода в вершинах (дизассемблированный листинг кода) и схемой передачи управления между ними.

Важной частью схемы является детектор аномальных состояний. Его функция – определять момент, когда входные данные приводят к непредвиденному поведению программы. Примером подобного поведения является передача управления на буфер, заполненный данными извне или использование ввода в качестве строки форматирования для функции языка C sprintf. Такая реакция может означать сбой или атаку на ПО. Важно, чтобы детектор не только зафиксировал сам факт атаки, но и мог предоставить данные о том, что привело систему в данное состояние.

Рассмотрим существующие методики, на которых может основываться подобный детектор.

В настоящее время существует множество подходов и инструментов для выявления атак на ПО. Они отличаются тем, какой класс атак способны выявить, а также какие для этого необходимы условия. Для противодействия атакам на переполнение буфера используются методы, требующие модификации исходных кодов [4, 5]. Метод [6] выявления атаки на форматную строку требует модификации библиотеки libc.

Для описанной выше схемы требуется выявлять атаки путем статического и динамического анализа исполняемого кода. Метод, описанный в [7], удовлетворяет этим требованиям. Остановимся на нем подробнее.

Метод эффективен для выявления атак перезаписи – атак, которые выполняются через изменение ячейки памяти или регистра там, где такая подмена не предусмотрена. Примерами такой атаки является атака переполнения буфера и атака форматной строки. Однако проблема не ограничивается только лишь приведенными примерами. Подобная атака подразумевает воздействие на внутренние данные программы через получаемые извне пользовательские данные. Корректно реализованная программа должна допускать только лишь контролируемое воздействие, то есть воздействие только лишь в рамках заранее определенных правил. В листинге 1 представлен пример кода, где входные данные влияют на содержимое внутренней переменной *cmdID*.

Листинг 1. – Пример контролируемого влияния

```
#define UNKNOWN_CMD_ID 0
#define PRINT_CMD_ID 1
…
int cmdID = UNKNOWN_CMD_ID;
char Cmd[4];
char Param[5];
char * Commands[] = { "Unknown", "Print" };

printf( "Format: CMD[3] PARAM[4]>" );
while( 2 != scanf_s("%03s %04s", Cmd, 4, Param, 5) )
{
      printf( "Invalid input." );
      printf( "Format: CMD[3] PARAM[4]\n" );
}

cmdID = UNKNOWN_CMD_ID;

// Command parser
if( !strcmp( Cmd, "PRN" ) )
```

```
        cmdID = PRINT_CMD_ID;

    // Command execution
    printf( "Command: %s\n", Commands[ cmdID ] );
    switch ( cmdID )
    {
        case PRINT_CMD_ID:
            printf( "Result: %s\n", Param );
            break;
        case UNKNOWN_CMD_ID:
            printf( "Result: undefined\n" );
            break;
        default:
            assert( false );
            break;
    }
```

Программа гарантирует, что значение переменной *cmdID* примет значение 0 или 1. Этот момент является важным по причине того, что значение используется в качестве индекса массива *Commands*.

Рассмотрим теперь пример кода, где входные данные напрямую меняют содержимое внутренней переменной (листинг 2).

Листинг 2. – Пример неконтролируемого влияния

```
#define UNKNOWN_CMD_ID 0
#define PRINT_CMD_ID 1

...

int cmdID = UNKNOWN_CMD_ID;
char Param[5];
char * Commands[] = { "Unknown", "Print" };

printf( "Format: CMD_ID PARAM[4]>" );
while( 2 != scanf_s("%01i %04s", &cmdID, Param, 5) )
{
    printf( "Invalid input\n" );
    printf( "Format: CMD_ID PARAM[4]\n" );
}

// Command execution
printf( "Command: %s\n", Commands[ cmdID ] );
switch ( cmdID )
{
case PRINT_CMD_ID:
    printf( "Result: %s\n", Param );
    break;
case UNKNOWN_CMD_ID:
    printf( "Result: undefined\n" );
    break;
default:
```

```
            assert( false );
            break;
      }
```

Значение параметра *cmdID* в данном примере напрямую получено из пользовательского ввода. И если указать индекс, равный 2, то программа попытается вывести строку, которая находится по адресу, соответствующему шестнадцатеричному представлению последних 4 байт буфера Param. Такой код является примером того, как пользовательский ввод неконтролируемо влияет на внутреннюю переменную программы (индекс массива).

Ситуация, иллюстрированная выше, не является частью реально работающего ПО, а дана для понимания проблемы. В качества реального примера можно привести уязвимость протокола SMB2 операционной системы Windows [8]. Передаваемое в запросе поле *PidHigh* использовалось в качестве индекса при выборе функции обработчика. При этом не проверялось условие выхода за пределы размеров таблицы функций обработчиков.

В качестве примера ситуации, когда входные данные не влияют на внутреннюю переменную в обычном режиме работы, но могут влиять в аварийном, рассмотрим листинг 3.

Листинг 3. – Пример неявного неконтролируемого влияния

```
      #define UNKNOWN_CMD_ID 0
      #define PRINT_CMD_ID 1

      int cmdID = UNKNOWN_CMD_ID;
      char Cmd[4];
      char Param[5];
      char * Commands[] = { "Unknown", "Print" };

      cmdID = UNKNOWN_CMD_ID;

      printf( "Format: CMD[3] PARAM[4]>" );
      while( 2 != scanf( "%s %s", Cmd, Param ) )
      {
            printf( "Invalid input. Format: CMD[3]
PARAM[4]\n" );
            printf( "Format: CMD[3] PARAM[4]\n" );
      }

      // Command parser
      if( !strcmp( Cmd, "PRN" ) )
            cmdID = PRINT_CMD_ID;

      // Command execution
      printf( "Command: %s\n", Commands[ cmdID ] );
      switch ( cmdID )
      {
            case PRINT_CMD_ID:
                  printf( "Result: %s\n", Param );
                  break;
```

```
        case UNKNOWN_CMD_ID:
              printf( "Result: undefined\n" );
              break;
        default:
              assert( false );
              break;
  }
```

Отличие приведенной программы от первого примера в использовании функции scanf вместо более защищенного аналога scanf_s. Данная функция не получает в качестве входного параметра размер буферов, которые используются для сохранения считанных из входного потока данных. Этот факт может быть злонамеренно использован для проведения атаки переполнения буфера. Заполняя данными буфер *Param* (впрочем как и *Cmd*), существует возможность изменить значение переменной *cmdID*. Это приведет к тому, что в качестве индекса массива будет использовано заданное при переполнении значение. Впрочем, при условии отсутствия дополнительной защиты со стороны компилятора, подобная атака может быть использована для замены адреса возврата из функции.

Для отслеживания ситуаций, подобных описанным выше, в статье [7] предлагается метод распространения пятна (от англ. «taint»). Априори предполагается, что данные, подверженные воздействию пользовательского ввода, не могут участвовать в качестве операндов некоторых операций. В листинге 1 был приведен код, где переменная *cmdID* используется в качестве индекса массива. Это пример операции, где не может быть использована переменная, на которую явно или неявно не контролировано повлиял пользовательский ввод. Другим примером является вызов функции, принимающий в качестве входного параметра строку форматирования (printf, scanf и т.д.), а также передача управления по адресам из памяти или регистров процессора.

Для построения детектора потребуется определять точки ввода пользовательских данных. Подобными точками являются вызовы API для получения данных из стандартного потока или сети.

Далее, для того чтобы отследить распространение влияния входных данных, требуется выделить подмножество команд процессора, которые изменяют содержимое ячеек памяти или регистров. При этом необходимо помечать ячейки, на содержимое которых повлияли входные данные или уже помеченные данные.

Другой тип команд, выполнение которых также необходимо отслеживать – команды, которые требуют в качестве операндов данные, не помеченные в ходе слежения данные. Если подобная команда выполняется над помеченными данными, то выявлена либо атака, либо уязвимое место программы.

Для проведения стендовых испытаний по предложенной выше схеме необходим метод генерации входных данных. В [9] представлен достаточно полный обзор методов генерации данных, объединенных общим названием “fuzzing”. Входной массив подготавливается по определенному алгоритму, учитывающему либо нет формат данных, принимаемых приложением.

Другим подходом генерации данных является использование образцов из трафика сети Internet. Это возможно при условии реализации одной из следующих схем:

1. Расстановка на множество рабочих станций специального ПО – датчиков, сбора образцов;

2. Сбор образцов из сети Интернет с помощью специальных программ (подобных тем, что используют поисковые машины);
3. Перехват трафика пользователей, его анализ и выделение интересующих образцов.

Пойти первым путем могут позволить себе разве что крупные компании, у которых существует большая сеть клиентов.

Второй способ выглядит более реализуемым, однако требует для своей реализации затрат большого количества трафика.

Третий способ на первый взгляд нереализуем, ведь при современной организации сетей перехват данных, адресуемых другим пользователям, в общем случае невозможен. Повторители повсеместно заменены маршрутизаторами, но даже если это было бы не так, то анализ трафика локальной сети не так интересен, как файлы, адресуемые среднестатистическому пользователю Интернета.

Но даже в современных условиях такую возможность нам предоставляют провайдеры спутникового Интернета [10]. Для организации асинхронного интернет-канала (исходящий трафик передается по наземному каналу, входящий – через спутник) используется спецификация DVB для построения систем широковещательной передачи данных. Неограниченное количество пользователей принимают на свои тарелки сигнал со спутника, который несет в себе mpg-2 поток.

Интернет-трафик, адресуемый клиентам, также передается в одном потоке. Поток делится на части (pids). Каждый клиент анализирует поток с нужным ему pid (оговаривается провайдером) и забирает данные, адресуемые карте с соответствующим MAC-адресом.

Так же как и у сетевых ethernet-карт, у сетевых DVB-карт есть возможность работать в режиме полного захвата пакетов (promiscuous mode). В этом случае карта будет перехватывать трафик (в зависимости от символьной скорости передачи со скоростью от ~25 Мб/с до 50 Мб/с), адресуемый всем пользователям спутникового Интернета, использующих тот же pid. Файлы, которые скачивают пользователи спутникового Интернета (в зависимости от выбора провайдера, можно ограничить аудиторию, например, территориально) могут быть использованы как образцы для стенда верификации ПО. При этом пользователи, чьи файлы перехватываются не против участия в подобной программе (ведь им известно, что файлы, передаваемые через спутник, не шифруются) и нет расхода трафика Интернета совсем.

Предложенная в статье схема является модульной. Состав модулей может меняться в зависимости от решаемой задачи. В статье предложены методы для одной из возможных конфигураций.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Корпорация «Майкрософт» и обеспечение безопасности: результаты изменения подхода к разработке продуктов. [Интернет] – режим доступа http://www.microsoft.com/rus/midsizebusiness/security/sdl.mspx, свободный.

2. WASC Web Application Security Statistics 2008. [Интернет] – режим доступа http://www.scribd.com/doc/21324267/WASC-Web-Application-Security-Statistics-2008-Russian, свободный.

3. *Благодаренко, А.В.* Инструментальное средство для проведения сертификационных испытаний программного обеспечения без исходных кодов // Известия ЮФУ. Технические науки. Тематический выпуск. “Информационная безопасность”. – 2007. – №1 (76). – С. 212 – 215.

4. *C. Cowan, S. Beattie, J. Johansen, and P. Wagle.* Point-Guard: Protecting pointers from buffer overflow vulnerabilities. In 12th USENIX Security Symposium, 2003.

5. *C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton.* Stack-Guard: automatic adaptive detection and prevention of buffer-overflow attacks. In Proceedings of the 7th USENIX Security Symposium, January 1998.

6. *C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman.* FormatGuard: automatic protection from printf format string vulnerabilities. In Proceedings of the 10th USENIX Security Symposium, August 2001.

7. *James Newsome, Dawn Song.* Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In Proceedings of the Network and Distributed System Security Symposium, 2005

8. Vulnerabilities in SMBv2 Could Allow Remote Code Execution (975517) http://www.microsoft.com/technet/security/Bulletin/ms09-050.mspx

9. *Michael Sutton, Adam Greene, Pedram Amini.* Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley Professional; 1 edition (July 9, 2007)

10. *Благодаренко, А.В.* Широковещательное распространение обновлений безопасности для ОС Linux через спутниковый канал // Сборник ЮФУ. Тематический выпуск. Информационная безопасность. Перспектива-2009. – 2009. – С. 210–214.

Благодаренко Артем Васильевич
Технологический институт Федерального государственного образовательного учреждения высшего профессионального образования «Южный федеральный университет» в г. Таганроге.
E-mail: artem.blagodarenko@gmail.com.
347928, г. Таганрог, ул. Чехова, 2, корпус "И".
Тел.: 8 (8634) 312-018.
Кафедра безопасности информационных технологий; аспирант.

Blagodarenko Artem Vasilyevich
Taganrog Institute of Technology – Federal State-Owned Educational Establishment of Higher Vocational Education “Southern Federal University”.
E-mail: artem.blagodarenko@gmail.com.
Block “I”, 2, Chehov str., Taganrog, 347928, Russia.
Phone: 8 (8634) 312-018.
The Department of Security of Information Technologies; post-graduate student.

УДК 004.942 – 056.57

С.Д. Жилкин

РАЗДЕЛЕНИЕ РАБОТЫ ПО НА ФАЗЫ С ЦЕЛЬЮ ПОСТРОЕНИЯ МОДЕЛИ РАБОТЫ ПО

Статья предлагает подходы к решению задачи моделирования поведения программного обеспечения (ПО) с целью дальнейшего выявления отклонений работы. Методы моделирования основаны на системах мониторинга и математическом аппарате нейронных сетей. Основная часть статьи посвящена механизму моделирования работы ПО на протяжении всего рабочего цикла путём выявления определённых фаз работы ПО. Рассмотрена эффективность различных способов.

Модель работы ПО; аномалии работы ПО; фазы работы ПО; нейронные сети.