# Метод поиска уязвимости форматной строки

*[1]И.А. Вахрушев <ilia.vahrushev@ispras.ru>*
*[1]В.В. Каушан <korpse@ispras.ru>*
*[1, 2]В.А. Падарян <vartan@ispras.ru>*
*[1]А.Н. Федотов <fedotoff@ispras.ru>*
*[1] Институт системного программирования РАН,*
*109004, Россия, г. Москва, ул. А. Солженицына, дом 25*
*[2]Московский государственный университет имени М.В. Ломоносова,*
*119991 ГСП-1 Москва, Ленинские горы, 2-й учебный корпус, факультет ВМК*

**Аннотация.** В статье рассматривается метод поиска уязвимостей форматной строки в исполняемом бинарном коде. Предлагаемый метод использует символьное выполнение и применяется к бинарным файлам программ, не требуя какой-либо отладочной информации. Метод был реализован в виде программного инструмента. Возможности инструмента были продемонстрированы на известных уязвимостях приложений, работающих под управлением ОС Linux.

**Ключевые слова:** уязвимость форматной строки; бинарный код; эксплуатация уязвимостей; динамический анализ; символьное выполнение.

**DOI:** 10.15514/ISPRAS-2015-27(4)-2

**Для цитирования:** Вахрушев И.А., Каушан В.В., Падарян В.А., Федотов А.Н. Метод поиска уязвимости форматной строки. Труды ИСП РАН, том 27, вып. 4, 2015 г., стр. 23-38. DOI: 10.15514/ISPRAS-2015-27(4)-2.

## *1. Введение*

Обеспечение безопасности компьютерных программ становится все более злободневным вопросом. Наследственное ПО, разработанное без учета современных требований безопасности, продолжает эксплуатироваться, довольствуясь фрагментарными исправлениями. Да и современные технологии разработки безопасного ПО не решают проблемы безопасности в полной мере. В частности, несмотря на развитие графических интерфейсов, программы продолжают пользоваться функциями форматного вывода.

Форматный вывод используется для ведения системных журналов, в сообщениях об ошибках и т.д. Еще в работах 90-х годов [1] было показано, что неправильная работа с форматной строкой может привести к серьезным

уязвимостям в безопасности ПО, таким как выполнение произвольного кода, повышение привилегий, а также утечка чувствительных данных. Уязвимость форматной строки позволяет злоумышленнику целенаправленно перезаписывать определенные участки оперативной памяти. Одним из таких участков может быть ячейка памяти на стеке, которая содержит адрес возврата из функции. Перезапись адреса возврата приводит к перехвату потока управления и выполнению произвольного кода, размещённого в памяти. Размещение в стеке перед адресом возврата специального значения «канарейки», призванного выявлять ошибки переполнение буфера на стеке, не защищает от уязвимости форматной строки, так как перезаписывается не диапазон адресов памяти, а только адрес возврата.

Поиск ошибок и уязвимостей можно проводить как на уровне исходных текстов [2], так и в бинарном коде [3]. Поиск ошибок на уровне бинарного кода имеет преимущества не только, когда исходные тексты исследуемого ПО недоступны. Наиболее важная причина – исполняемый код позволяет гораздо точней оценить критичность ошибки, может ли она эксплуатироваться или нет [4], т.к. на уровне исходного кода еще не известно точное размещение переменных в памяти.

Существуют различные методы поиска ошибок и уязвимостей в бинарном коде. Активно изучаемым подходом является символьное выполнение [5], получившее «второе рождение» в последние годы [6].

Во время символьного выполнения конкретные значения переменных заменены символьными значениями, операции над переменными порождают формулы над символьными значениями и константами. На практике символьные значения задаются на основе входных данных программы, получаемых из различных источников: сеть, файлы, аргументы командной строки, переменные окружения и т.д. Каждое условное ветвление, на которое влияют входные данные, порождает дополнительное уравнение, характеризующее прохождение потока управления по определённой ветке. Совокупность таких уравнений, описывающих прохождение некоторого пути выполнения, называется предикатом пути. Этот набор уравнений подаётся на вход SMT-решателю, где символьные переменные выступают в качестве неизвестных величин. Результат решения – набор входных данных, при обработке которого поток управления программы пойдёт по заданному пути.

Для выявления уязвимости к предикату пути добавляются дополнительные уравнения, описывающие срабатывание уязвимости. Если полученная система уравнений совместна, то результатом её решения является эксплойт – набор входных данных, эксплуатирующий уязвимость. Из-за погрешностей в моделировании уязвимости необходимо дополнительно проверить работоспособность полученного эксплойта, запустив с ним программу.

Существует ряд инструментов, например, AEG [7], MAYHEM [8], разработанных в ведущих университетах США, которые решают задачу поиска уязвимости форматной строки. К сожалению, эти инструменты

недоступны. Существующие системы символьного выполнения, находящиеся в открытом доступе, например, S2E [9], предоставляют только инфраструктуру перебора состояний, инструменты поиска уязвимостей в этом случае приходится реализовывать самостоятельно.

Работы, проводимые в ИСП РАН, используют другой подход: вместо перебора состояний детально исследуется ограниченное число трасс, как правило, это одна трасса, обеспечивающая приемлемое покрытие кода. Был предложен метод символьной интерпретации трассы, нацеленный на поиск и оценку критичности сработавших ошибок работы с памятью. В своем развитии метод получил возможность анализировать подсемейство трасс, абстрактно интерпретируя длины буферов памяти [10]. Основываясь на результатах предыдущих работ, был разработан метод поиска и эксплуатации важного типа уязвимостей, а именно – уязвимости форматной строки.

Статья организована следующим образом. Во втором разделе описан механизм эксплуатации уязвимости форматной строки. В третьем разделе описан предлагаемый метод и схема работы инструмента, реализующего его. Результаты практического применения приведены в четвёртом разделе. В последнем, пятом разделе обсуждаются полученные результаты и рассматриваются перспективные направления исследований.

## *2. Эксплуатация уязвимости форматной строки*

Функции форматного вывода, такие как `printf`, `fprintf`, `vfprintf`, `syslog`, используются для вывода значений разных типов, отформатированных согласно заданному шаблону (форматной строке). Шаблон представляет собой строку, содержащую управляющие последовательности (спецификаторы). Уязвимость форматной строки возникает в случаях, когда разработчик программы в качестве форматной строки передаёт буфер, зависящий от входных данных.

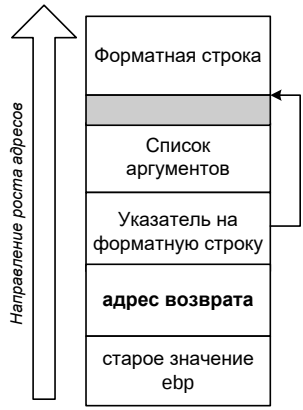


*Рис. 1. Состояние стека в момент вызова функции форматного вывода.*

На рис. 1 показан кадр стека функции форматного вывода `printf`. Перед вызовом функции в стеке размещается (в обратном порядке) список дополнительных аргументов и указатель на форматную строку. Во время вызова на стеке размещается адрес возврата в вызывающую функцию, после чего вызываемая функция может сохранять некоторые регистры и выделять память для локальных переменных.

Множество используемых дополнительных аргументов функции определяется спецификаторами форматной строки. При обработке форматной строки аргументы считываются по очереди, начиная с аргумента, следующего непосредственно за указателем на форматную строку. Если в форматной строке количество спецификаторов превосходит количество фактически переданных дополнительных аргументов, то вышележащие ячейки стека будут интерпретированы как потенциальные аргументы функции `printf`. Некоторые функции форматного вывода (например, `vfprintf`) в качестве списка аргументов принимают структуру `va_list`. В этом случае адреса потенциальных аргументов вычисляются с помощью содержимого этой структуры.

Помимо спецификаторов, позволяющих вывести значения в поток вывода, в функциях форматного вывода поддерживается спецификатор `%n`, который позволяет записывать в память по адресу, переданному в качестве аргумента, значение, равное выведенному в поток количеству символов. Поскольку адрес перезаписываемой ячейки памяти передаётся в качестве аргумента функции форматного вывода, этот спецификатор позволяет записывать данные в память по контролируемому адресу. Используя спецификатор несколько раз можно перезаписать произвольное количество ячеек памяти. Ячейки памяти, содержащие контролируемые адреса, должны находиться на стеке среди потенциальных аргументов. Как правило, эти аргументы размещаются внутри форматной строки. В этом случае, форматная строка должна располагаться на стеке.

С помощью спецификатора `%n` злоумышленник может перезаписать адрес возврата из функции указателем на произвольный код, что приведёт к выполнению этого кода и нарушению безопасности. Этот код также может быть размещён внутри форматной строки. Таким образом, можно сформировать такую форматную строку, обработка которой приведёт к выполнению кода, заданного злоумышленником.

При использовании спецификатора `%n` в память по заданному адресу записывается четырёхбайтное значение, равное количеству символов, выведенных в поток вывода. Это значение можно контролировать с помощью форматных спецификаторов, таких как: `%x`, `%d`. Спецификаторы форматной строки можно использовать с параметром ширины. Значение параметра ширины непосредственно влияет на значение, записываемое в память при обработке спецификатора `%n`. Таким образом, можно указать такое значение ширины, чтобы в память по заданному адресу записывалось требуемое

значение. В некоторых функциях форматного вывода может применяться ограничение на величину параметра ширины. В этом случае возможно использование нескольких спецификаторов с уменьшенным до допустимого значения параметром ширины. Помимо того, можно уменьшить величину записываемых значений, если перезаписывать заданную ячейку памяти по частям. Например, спецификатор `%hn` позволяет перезаписывать только два байта вместо четырёх. Уменьшение записываемых значений позволяет сократить объем выводимых текстовых данных, и как следствие – сократить размер форматной строки.

Адреса перезаписываемых ячеек памяти расположены выше по стеку относительно кадра стека функции форматного вывода и расположены среди аргументов этой функции. Аргументы функции считываются последовательно по мере обработки спецификаторов форматной строки, поэтому для доступа к нужному аргументу перед обработкой спецификатора `%n` необходимо предварительно обработать некоторое количество других спецификаторов вывода. Так как на стеке могут располагаться произвольные значения аргументов, следует использовать спецификаторы вывода с параметром ширины, позволяющим выводить фиксированное количество символов вне зависимости от значений аргументов. В качестве такого спецификатора можно использовать спецификатор `%8x`. В операционных системах Linux в спецификаторах форматной строки поддерживается возможность доступа к произвольному аргументу по его порядковому номеру (используется символ '`$`') [11]. Это значительно упрощает задачу доступа к нужному аргументу и ощутимо сокращает размеры форматной строки.

Уязвимость форматной строки рассматривается как обособленный, крайне важный тип уязвимостей. Практическое применение этой уязвимости – выборочная перезапись одного машинного слова, содержащего адрес. Как правило, это адрес возврата из функции, его изменение позволяет перехватить поток управления. Относительно недавно было показано [12], что уязвимость форматной строки удобна для реализации другого автоматизируемого типа атаки – перехвата потока данных. В результате успешной атаки злоумышленник получает доступ к чувствительной информации, как пользовательской, так и служебной (GOT, адреса стека, "канарейки", указатели и др.). Доступ к служебным данным позволяет получить значения рандомизированных адресов, что открывает дальнейшие возможности по конструированию работоспособных эксплойтов для данного экземпляра программы, содержащей уязвимость.

Важно отметить, что существующие механизмы защиты («канарейка», неисполняемый стек, ASLR, «безопасные» функции форматного вывода и др.) не способны противодействовать перехвату потока данных.

## 3. Схема работы

Представленные в данной статье исследования опираются на возможности среды анализа бинарного кода [13]. Основным предметом анализа являются трассы машинных инструкций, полученные в результате работы полносистемного эмулятора [14, 15]. Трассы содержат состояния регистров, информацию о прерываниях и взаимодействии с периферийными устройствами, что позволяет восстанавливать статико-динамическое представление [16] для всех работавших в системе программ и эффективно исследовать его свойства. Основным назначением среды анализа является автоматизация метода выделения алгоритмов из бинарного кода и повышение уровня представления этих выделенных алгоритмов.

Поиск уязвимостей форматной строки проводится по трассе выполнения анализируемой программы для каждого вызова функций форматного вывода. Поиск вызовов этих функций в трассе выполняется автоматически, рассматриваются только те вызовы, на форматную строку которых могут воздействовать входные данные. Для каждого анализируемого вызова функции из трассы отбираются инструкции, обрабатывающие соответствующие входные данные. Для отбора инструкций используется алгоритм динамического слайсинга трассы [17], дополненный анализом помеченных данных. В качестве входных данных выступают буферы, используемые для чтения файлов, получения данных из сети, а также содержащие параметры командной строки.

Для анализа машинных инструкций используется промежуточное представление Pivot [18], позволяющее унифицировано описывать операционную семантику инструкций различных процессорных архитектур. Промежуточное представление отвечает требованием SSA-формы, что позволяет значительно упростить анализ.

Поиск уязвимости форматной строки и генерация эксплойта разделена на пять последовательных этапов (рис.2).


*Рис. 2. Декомпозиция метода на пять этапов.*

Так как анализируются трассы выполнения программ, состоящих из последовательного набора машинных команд, то для начала работы алгоритма необходимо задать шаг трассы, начиная с которого в программе уже содержаться входные данные, полученные из внешнего источника. Также необходимо задать буфер памяти, в котором содержаться эти входные данные. Для поиска буфера с входными данными и нужного шага трассы, аналитик

может воспользоваться возможностями среды анализа бинарного кода, а именно – навигацией по вызовам функций. Как правило, программы используют для получения входных данных известные библиотечные функции. Заранее зная, какой источник входных данных (сеть, файлы, аргументы командной строки и т.п.) был использован, аналитик выбирает вызовы соответствующих библиотечных функций.

В результате работы алгоритма слайсинга формируется подтрасса, содержащая инструкции, которые участвуют в обработке входных данных. Точное выделение подтрассы позволяет рассматривать ограниченное число машинных команд, которые впоследствии будут оттранслированы в SMT-формулы, что в свою очередь позволяет ускорить решение результирующей системы уравнений. Начальному шагу работы алгоритма соответствует точка получения входных данных, а конечному – шаг трассы, на котором происходит вызов функции работы с форматной строкой.

Предикат пути строится по подтрассе и представляет собой набор уравнений на языке SMT [19]. Предикат пути используются для описания ограничений на входные данные, при прохождении потока управления от точки получения до вызова функции работы с форматной строкой. Каждая машинная инструкция подтрассы последовательно транслируется в промежуточное представление Pivot [18]. Затем по промежуточному представлению строятся SMT-формулы, описывающие работу машинной инструкции в целом. В результате трансляции всех инструкций подтрассы будет получен предикат пути. Более подробно применяющийся способ трансляции инструкций в SMT-формулы описан в работе [4].

Для поиска потенциальной уязвимости проверяется наличие помеченных данных в буфере, передаваемом в качестве форматной строки. Если этот буфер содержит помеченные данные, конструируется содержимое форматной строки, обработка которой приведет к выполнению заданного шелл-кода. На рис. 3 представлена структура генерируемой форматной строки. В начале создаваемой форматной строки размещается шелл-код, затем размещаются спецификаторы форматного вывода, а в самом конце строки расположен адрес перезаписываемой ячейки памяти.

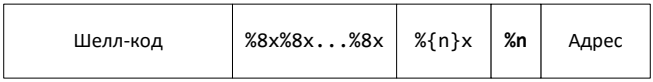


*Рис. 3. Структура форматной строки, используемой в эксплойте.*

Форматная строка представляет собой нуль-терминированную строку, следовательно, размещение нулевых байтов внутри строки недопустимо. Если адрес перезаписываемой ячейки памяти в старшем байте содержит ноль, то его можно совместить с терминатором строки при условии, что используется порядок байтов «от младшего к старшему». Наличие других нулевых байтов в адресе не допускается.

Последовательность спецификаторов `%8x` готовит доступ к адресу во время обработки спецификатора `%n`. Спецификатор `%{n}x` с параметром ширины `n` используется для достижения требуемого значения суммарной ширины. Спецификатор `%n` осуществляет запись суммарной ширины по адресу, записанному в конце строки. В процессе обработки форматной строки функция поддерживает указатель на текущий аргумент в списке потенциальных аргументов, в также указатель на текущий обрабатываемый символ форматной строки. При обработке спецификатора `%n` указатель на текущий аргумент должен указывать на ячейку, содержащую адрес перезаписываемой ячейки памяти. Для изменения этого указателя необходимо обработать некоторое количество спецификаторов `%8x`. Каждая обработка спецификатора `%8x` сдвигает указатель на текущий аргумент на 4 байта вверх по стеку, а указатель на текущий обрабатываемый символ форматной строки – на 3 байта (количество символов, занимаемых спецификатором форматной строки). Требуемое количество спецификаторов `%8x` подбирается, исходя из следующих соображений. Пусть перед обработкой последовательности спецификаторов `%8x` значение указателя на текущий аргумент равно *ArgStart*, а значение указателя на текущий обрабатываемый символ форматной строки равно *FmtStart*. Соответственно, в момент обработки спецификатора `%n` значение указателя на текущий аргумент равно *ArgEnd*, а на текущий обрабатываемый символ – *FmtEnd*. Структура списка аргументов и размещение указателей изображены на рис. 4.

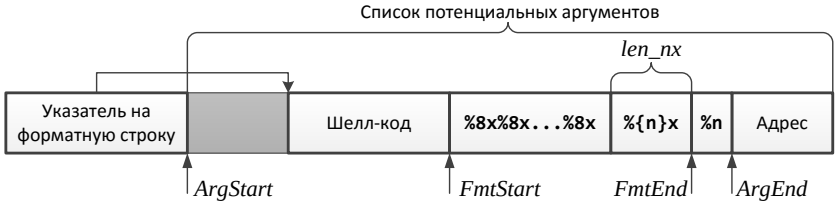


*Рис. 4. Структура списка аргументов.*

Пусть форматная строка содержит k спецификаторов %8x. В результате обработки форматных спецификаторов указатели должны соотноситься следующим образом:

$$ArgEnd = ArgStart + (k+1)*4,$$
$$FmtEnd = FmtStart + k*3 + len\_nx,$$
$$ArgEnd = FmtEnd + 2.$$

Первое уравнение описывает сдвиг указателя на текущий аргумент при обработке спецификаторов. Так как после последовательности спецификаторов `%8x` располагается спецификатор `%{n}x`, количество обработанных спецификаторов равно *(k+1)*. Второе уравнение отражает сдвиг

указателя на текущий обрабатываемый символ форматной строки с учётом обработки спецификатора `%{n}x`, длина записи которого обозначена как *len_nx*. Третье уравнение описывает требуемое состояние указателей на момент выполнения спецификатора `%n`. В этот момент указатели должны различаться на 2 байта – длину спецификатора `%n`. С учётом данных соотношений, требуемое количество спецификаторов `%8x` вычисляется по формуле:

$$k = FmtStart - ArgStart - 2 + len\_nx.$$

При размещении спецификатора `%{n}x` значение ширины n подбирается таким образом, чтобы в совокупности с шириной остальных спецификаторов, а также длиной шелл-кода получить требуемое суммарное значение ширины, которое в дальнейшем будет записано в память при обработке спецификатора `%n`. Завершается генерация форматной строки размещением спецификатора `%n`, вслед за которым размещается адрес перезаписываемой ячейки памяти. Если расстояние между ячейкой памяти, в которой будет лежать адрес, и ячейкой памяти первого элемента из списка потенциальных аргументов, не кратно 4, то в форматную строку после шелл-кода добавляется необходимое количество произвольных байтов.

При генерации эксплойта, перезаписывающего адрес возврата, в операционной системе Windows адрес перезаписываемой ячейки часто содержит нулевой байт в старшем разряде, поэтому его можно размещать только в конце форматной строки. В операционной системе Linux адреса ячеек стека не содержат нулевой байт в старшем разряде, поэтому адрес перезаписываемой ячейки, если он не содержит нулевых байтов, можно размещать в любом месте форматной строки. Также использование таких адресов позволяет размещать более одного адреса в форматной строке и перезаписывать память по частям. Перезапись ячеек памяти по частям позволяет использовать меньшие значения суммарной ширины, что, в свою очередь, позволяет использовать меньшие значения ширины в спецификаторах `%{n}x`. Так как в функциях библиотеки libc ОС Linux поддерживается возможность обращения к произвольному аргументу в спецификаторе форматной строки, целесообразно использовать эту возможность вместо генерации последовательности спецификаторов `%8x`.

После генерации форматной строки производится проверка того, что размер сгенерированной форматной строки не превосходит размер буфера, контролируемого пользователем. Размер этого буфера определяется с помощью вычисления размера непрерывной области помеченных данных, передаваемых в функцию форматного вывода качестве форматной строки.

Если проверка выполнена успешно, выполняется построение уравнений, описывающих передачу сгенерированной форматной строки в функцию форматного вывода. Эти уравнения объединяются с предикатом пути и описывают состояние программы, в котором активируется уязвимость форматной строки. Полученная система уравнений подаётся на вход SMT-

решателю. Если система уравнений совместна, её решением является набор входных данных, обработка которого приводит к срабатыванию уязвимости. Выполнение программы на полученном наборе входных данных непосредственно подтверждает наличие уязвимости.

## 4. Реализация метода и результаты практического применения

Предложенный метод был реализован и интегрирован в уже существующий модуль-расширение среды анализа бинарного кода [4]. Для решения систем уравнений используется решатель Z3 [20], так как он обеспечивает высокую скорость работы по сравнению с другими решателями. Для приложений под управлением ОС Linux применяется возможность перезаписи ячейки памяти по частям с помощью спецификатора `%hn`.

Разработанный инструмент был опробован на нескольких примерах известных уязвимостей. Для запуска приложений использовалась операционная система Arch Linux. В табл. 1 приведены основные результаты работы реализованного алгоритма: размер слайса обработки входных данных, размер блока входных данных, общее время работы алгоритма. Следует отметить, что время работы SMT-решателя не превышало одной секунды на всех примерах. Кроме того, в таблице приведены названия и версии анализируемых приложений, используемая функция форматного вывода, а также тип источника входных данных (аргументы командной строки или сетевой сокет).

*Табл.1. Результаты работы алгоритма генерации эксплойта для уязвимости форматной строки.*

| Приложение | Функция | Размер слайса | Размер данных, байт | Общее время работы, с | Тип входных данных |
|---|---|---|---|---|---|
| socat-1.4 | syslog | 2847 | 254 | 25 | Арг. |
| orzhttpd-r140 | fprintf | 7006 | 254 | 49 | Сеть |
| sharutils-4.2.1 | vfprintf | 1970 | 255 | 52 | Арг. |
| gpsd-2.7 | syslog | 320 | 255 | 1 | Сеть |

Для успешной эксплуатации уязвимости были отключены механизмы защиты, которые препятствуют выполнению внедренного кода. Исследуемые приложения компилировались с флагами компилятора `-z execstack -D_FORTIFY_SOURCE=0`. Флаг `–z execstack` даёт возможность выполнять код, располагающийся на стеке. Директива `-D_FORTIFY_SOURCE=0` отключает использование безопасных функций. Также в операционной системе Arch Linux был отключен механизм рандомизации адресного пространства (ASLR). Стоит отметить, что отключение «канарейки» не потребовалась, так как

перезапись адреса возврата из функции происходит, не затрагивая «канарейку».

Предложенный метод обладает следующими ограничениями. Генерируемая форматная строка не должна быть большей длины, чем форматная строка, обработка которой зафиксирована в трассе. Кроме того, размещаемый внутри форматной строки шелл-код не должен содержать символов "%". Также, при записи больших значений в память могут использоваться большие значения модификатора ширины. При обработке таких спецификаторов на поток вывода будет выведено большое количество данных. В некоторых случаях это может привести к срабатыванию защитных проверок, ошибкам выделения памяти, ошибкам файловой системы и другим аварийным ситуациям, что, в свою очередь, приводит к преждевременному завершению обработки форматной строки.

Иногда на момент вызова функции форматного вывода предикат пути накладывает существенные ограничения на содержимое форматной строки. В приложениях под управлением операционной системы Windows адрес перезаписываемой ячейки памяти содержит нулевой байт. Это приводит к необходимости размещения адреса в конце форматной строки, что в совокупности с ограничениями на последние байты форматной строки приводит к несовместности системы ограничений. В связи с этим данный алгоритм не позволил провести эксплуатацию нескольких приложений под управлением ОС Windows.

## *5. Заключение*

В статье представлен метод поиска уязвимости форматной строки. Метод основан на символьном выполнении бинарных трасс полученных с полносистемного эмулятора. Метод был реализован в рамках среды анализа бинарного кода и позволяет генерировать эксплойты для уязвимости форматной строки. При помощи данного метода успешно удалось сгенерировать рабочие эксплойты для четырёх приложений, в которых содержались известные уязвимости форматной строки.

Наиболее близкими являются работы коллектива из университета Карнеги-Меллон. В работе AEG [7] с помощью анализа исходного кода программы производится поиск двух видов уязвимостей: уязвимость переполнения буфера и уязвимость форматной строки. Затем уже по бинарному коду программы генерируется эксплойт для найденной уязвимости. Работа MAYHEM [8] является продолжением работы AEG и описывает метод генерации эксплойтов на основе анализа только бинарного кода. Предлагаемый инструмент имеет возможность перебирать различные траектории выполнения во время динамического символьного выполнения. К сожалению, разработанные этим коллективом инструменты недоступны. Стоит отметить, что в этих работах для успешной эксплуатации уязвимостей также требовалось отключение различных механизмов защиты.

Инструмент CRAX [21] использует инфраструктуру символьного выполнения S2E [8] и позволяет обнаруживать и эксплуатировать уязвимости форматной строки. В работе [22] также предложен метод поиска уязвимости форматной строки, но эксплуатация найденной уязвимости не производится.

Среди дальнейших направлений исследований можно выделить эксплуатацию уязвимостей с использованием перехвата потока данных [12], а не потока управления. В настоящее время повсеместно применяются механизмы защиты, препятствующие перехвату потока управления. Среди таких механизмов можно отметить DEP, ASLR, а также использование безопасных функций. В современных операционных системах, как правило, DEP и ASLR включены по умолчанию, а современные компиляторы используют безопасные функции при сборке программ. Таким образом, большая часть современных программ достаточно защищена от перехвата потока управления. Однако в некоторых случаях защита намеренно отключается разработчиком, а главное, указанные механизмы не защищают программу от перехвата потока данных. Посредством перехвата потока данных можно реализовать такие атаки, как утечка чувствительных данных, повышение привилегий и т.д. На данный момент защиты, которые препятствуют изменению потока данных, вносят значительное замедление в работу программ, поэтому их применение не всегда целесообразно.

Ещё одним перспективным направлением является адаптация существующих алгоритмов к архитектурам ARM и MIPS. Это позволит расширить круг устройств, для которых можно проводить анализ. Помимо настольных компьютеров, анализ сможет применяться к мобильным устройствам, планшетам и сетевым маршрутизаторам. Адаптация разработанных ранее алгоритмов потребует определенных усилий из-за особенностей упомянутых выше архитектур. Например, в архитектуре ARM некоторые функции могут сохранять адрес возврата на регистре, а не на стеке, что существенно осложняет перехват потока управления.

## Список литературы

[1]. B.P. Miller; L. Fredriksen; B So. An Empirical Study of the Reliability of UNIX Utilities. // Commun. ACM. – 1990. – No 33.

[2]. В.П. Иванников, А.А. Белеванцев, А.Е. Бородин, В.Н. Игнатьев, Д.М. Журихин, А.И. Аветисян, М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ. // Труды Института системного программирования РАН Том 26. Выпуск 1. 2014 г. Стр. 231-250. DOI: DOI: 10.15514/ISPRAS-2014-26(1)-7

[3]. И.К. Исаев, Д.В. Сидоров, А.Ю. Герасимов, М.К. Ермаков. Avalanche: Применение динамического анализа для автоматического обнаружения ошибок в программах, использующих сетевые сокеты. // Труды Института системного программирования РАН, том 21, 2011, стр. 55-70.

[4]. Падарян В.А., Каушан В.В., Федотов А.Н. Автоматизированный метод построения эксплойтов для уязвимости переполнения буфера на стеке. // Труды Института

системного программирования РАН, том 26, выпуск 3, 2014, стр. 127-144. DOI: 10.15514/ISPRAS-2014-26(3)-7.

[5]. King J.C. Symbolic execution and program testing. // Commun. ACM. – 1976. – No 19.

[6]. C. Cadar, K. Sen. Symbolic Execution for Software Testing: Three Decades Later. // Commun. ACM. – 2013. – No 56.

[7]. T. Avgerinos, S. K. Cha, Alexandre Rebert, Edard J. Schwartz, Maverick Woo, and D. Brumley. AEG: Automatic exploit generation // Commun. ACM. – 2014.– №2.

[8]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley. Unleashing MAYHEM on Binary Code // IEEE Symposium on Security and Privacy. – 2012.

[9]. G. C. Vitaly Chipounov, Volodymyr Kuznetsov. S2E: A platform for in-vivo multi-path analysis of software systems. // In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2011, pp. 265–278.

[10]. Каушан В.В., Мамонтов А.Ю., Падарян В.А., Федотов А.Н. Метод выявления некоторых типов ошибок работы с памятью в бинарном коде программ. // Труды Института системного программирования РАН, том 27, выпуск 2, 2015, стр. 105-126. DOI: 10.15514/ISPRAS-2015-27(2)-7

[11]. Ericson J. Hacking: The Art of Exploitation. 2nd Edition. // No Starch Press, 2008, pp.167-183.

[12]. Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, Zhenkai Liang. Automatic Generation of Data-Oriented Exploits. // 24th USENIX Security Symposium 2015

[13]. А.Ю. Тихонов, А.И. Аветисян. Комбинированный (статический и динамический) анализ бинарного кода. // Труды Института системного программирования РАН, том 22, 2012, стр. 131-152. DOI: 10.15514/ISPRAS-2012-22-9

[14]. П.М. Довгалюк, Н.И. Фурсова, Д.С. Дмитриев. Перспективы применения детерминированного воспроизведения работы виртуальной машины при решении задач компьютерной безопасности. // Материалы конференции РусКрипто’2013. Москва, 27 — 30 марта 2013.

[15]. Довгалюк П.М., Макаров В.А., Падарян В.А., Романеев М.С., Фурсова Н.И. Применение программных эмуляторов в задачах анализа бинарного кода. // Труды Института системного программирования РАН, том 26, выпуск 1, 2014, стр. 277-296. DOI: 10.15514/ISPRAS-2014-26(1)-9.

[16]. В.А. Падарян, А.И. Гетьман, М.А. Соловьев, М.Г. Бакулин, А.И. Борзилов, В.В. Каушан, И.Н. Ледовских, Ю.В. Маркин, С.С. Панасенко. Методы и программные средства, поддерживающие комбинированный анализ бинарного кода. // Труды Института системного программирования РАН, том 26, выпуск 1, 2014, стр. 251-276. DOI: 10.15514/ISPRAS-2014-26(1)-8.

[17]. Андрей Тихонов, Вартан Падарян. Применение программного слайсинга для анализа бинарного кода, представленного трассами выполнения. // Материалы XVIII Общероссийской научно-технической конференции «Методы и технические средства обеспечения безопасности информации». 2009. стр. 131

[18]. Падарян В. А., Соловьев М. А., Кононов А. И. Моделирование операционной семантики машинных инструкций. // Программирование, № 3, 2011 г. Стр. 50-64.

[19]. Silvio Ranise and Cesare Tinelli. The SMT-LIB Format: An Initial Proposal. Proceedings of PDPAR'03, July 2003

[20]. Nikolaj Bjørner, Leonardo de Moura. Z3: Applications, Enablers, Challenges and Directions // Sixth International Workshop on Constraints in Formal Verification Grenoble, 2009.

[21]. Huang S. K. et al. Software Crash Analysis for Automatic Exploit Generation on Binary Programs // Reliability, IEEE Transactions on. – 2014. – Т. 63. – №. 1. – С. 270-289.
[22]. Wu B. et al. Directed symbolic execution for binary vulnerability mining // Electronics, Computer and Applications, 2014 IEEE Workshop on. – IEEE, 2014. – С. 614-617.

# Search Method for Format String Vulnerabilities

*[1]I.A. Vakhrushev <ilia.vahrushev@ispras.ru>*
*[1]V.V. Kaushan <korpse@ispras.ru>*
*[1, 2]V.A. Padaryan <vartan@ispras.ru>*
*[1]A.N. Fedotov <fedotoff@ispras.ru>*
*[1] Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., 109004, Moscow, Russia*
*[2]Lomonosov Moscow State University, 2nd Education Building, Faculty CMC, GSP-1, Leninskie Gory, Moscow, 119991, Russian Federation*

**Abstract.** In this paper search method for format string vulnerabilities is presented. Format string vulnerabilities can cause serious security problems providing ability to write an arbitrary value to an arbitrary location. Using this opportunity an attacker may hijack the control flow of a program and execute a malicious code. Besides, it is possible to bypass some protection mechanisms such as the stack canary due to exact overwriting of function return address. The method is based on dynamic analysis and symbolic execution. It is applied to program binaries without requiring debug information. We use dynamic analysis to find possible unsafe usage of format string function. If user controls data in a format string parameter, we consider that it is an unsafe usage of format string. Then a path predicate is build. The starting point of path predicate is a place where input data was received, the ending point is an unsafe format string function call. After building the path predicate we need to generate a special format string that provides a control flow hijack and a payload execution. By asserting such constraints on the format string parameter we are able to determine whether unsafe function usage is vulnerable or not. If the generated constraints are solvable, the method produces an exploit. We present a tool implementing this method. We used this tool to detect known vulnerabilities in Linux programs.

**Keywords:** format string vulnerability; binary code; vulnerability exploitation; dynamic analysis; symbolic execution.

**DOI:** 10.15514/ISPRAS-2015-27(4)-2
**For citation**: Vakhrushev I.A., Kaushan V.V., Padaryan V.A., Fedotov A.N. Search Method for Format String Vulnerabilities. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 4, 2015, pp.23-38 (in Russian). DOI: 10.15514/ISPRAS-2015-27(4)-2.

## References


[1]. B.P. Miller; L. Fredriksen; B So. An Empirical Study of the Reliability of UNIX Utilities. // Commun. ACM. – 1990. – No 33.

[2]. V.P. Ivannikov, A.A. Belevantsev, A.E. Borodin, V.N. Ignatiev, D.M. Zhurikhin, A.I. Avetisyan, M.I. Leonov. Staticheskij analizator Svace dlja poiska defektov v ishodnom kode programm. [Static analyzer Svace for finding of defects in program source code] // Trudy ISP RAN [The Proceedings of ISP RAS], vol. 26, issue 1, 2014, pp. 231-250 (in Russian). DOI: DOI: 10.15514/ISPRAS-2014-26(1)-7

[3]. Isaev, I. K., Sidorov, D. V., Gerasimov, A. YU., Ermakov, M. K. (2011). Primenenie dinamicheskogo analiza dlya avtomaticheskogo obnaruzheniya oshibok v programmakh ispol'zuyushhikh setevye sokety [Using dynamic analysis for automatic bug detection in software that use network sockets]. Trudy ISP RAN [The Proceedings of ISP RAS], 2011, vol. 21, pp. 55-70 (In Russian).

[4]. V.A. Padaryan, V.V. Kaushan, A.N. Fedotov. Avtomatizirovannyj metod postroenija jeksplojtov dlja ujazvimosti perepolnenija bufera na steke. [Automated exploit generation method for stack buffer overflow vulnerabilities] // Trudy ISP RAN [The Proceedings of ISP RAS], vol. 26, issue 3, 2014, pp. 127-144 (in Russian). DOI: 10.15514/ISPRAS-2014-26(3)-7).

[5]. King J.C. Symbolic execution and program testing. // Commun. ACM. – 1976. – No 19.

[6]. C. Cadar, K. Sen. Symbolic Execution for Software Testing: Three Decades Later. // Commun. ACM. – 2013. – No 56.

[7]. T. Avgerinos, S. K. Cha, Alexandre Rebert, Edard J. Schwartz, Maverick Woo, and D. Brumley. AEG: Automatic exploit generation // Commun. ACM. – 2014. – №2.

[8]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley. Unleashing MAYHEM on Binary Code // IEEE Symposium on Security and Privacy. – 2012.

[9]. G. C. Vitaly Chipounov, Volodymyr Kuznetsov. S2E: A platform for in-vivo multi-path analysis of software systems. // In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2011, pp. 265–278.

[10]. Kaushan V.V., Mamontov A.Yu., Padaryan V.A., Fedotov A.N. // Metod vyyavleniya nekotorykh tipov oshibok raboty s pamyat'yu v binarnom kode programm. [Memory violation detection method in binary code]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 27, issue 2, 2015, pp. 105-126 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-7

[11]. Ericson J. Hacking: The Art of Exploitation. 2nd Edition. // No Starch Press, 2008, pp.167-183.

[12]. Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, Zhenkai Liang. Automatic Generation of Data-Oriented Exploits. // 24th USENIX Security Symposium 2015

[13]. Tikhonov A.YU., Avetisyan A.I. Kombinirovannyj (staticheskij i dinamicheskij) analiz binarnogo koda. [Combined (static and dynamic) analysis of binary code]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 22, 2012, pp. 131-152 (in Russian). DOI: 10.15514/ISPRAS-2012-22-9

[14]. Dovgalyuk P.M., Fursova N.I., Dmitriev D.S. Perspektivyi primeneniya determinirovannogo vosproizvedeniya rabotyi virtualnoy mashinyi pri reshenii zadach kompyuternoy bezopasnosti. [Prospects of using virtual machine deterministic replay insolving computer security problems]. Materialyi konferentsii RusKripto’2013 [The Proceedings RusCrypto'2013], 2014 (In Russian).

[15]. Dovgalyuk P.M., Makarov V.A., Romaneev M.S., Fursova N.I. Primenenie programmyih emulyatorov v zadachah analiza binarnogo koda.[Applying program emulators for binary code analysis] // Trudy ISP RAN [The Proceedings of ISP RAS], vol. 26, issue 1, 2014, pp. 277-296 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-9.

[16]. V.A. Padaryan, A.I. Getman, M.A. Solovyev, M.G. Bakulin, A.I. Borzilov, V.V. Kaushan, I.N. Ledovskich, U.V. Markin, S.S. Panasenko. Metody i programmnye sredstva, podderzhivayushhie kombinirovannyj analiz binarnogo koda [Methods and software tools for combined binary code analysis]. Trudy ISP RAN [The Proceedings of ISP RAS], 2014, vol. 26, no. 1, pp. 251-276 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-8.

[17]. Tikhonov A.YU., Padaryan V.A., Primenenie programmnogo slaysinga dlya analiza binarnogo koda, predstavlennogo trassami vyipolneniya.[Using program slicing for bynary code represented by execution traces] Materialyi XVIII Obscherossiyskoy nauchno-tehnicheskoy konferentsii «Metodyi i tehnicheskie sredstva obespecheniya bezopasnosti informatsii». [The Proceedings of XVIII Russian science technical conference "Methods and technical information security tools"] 2009. pp 131 (In Russian).

[18]. Padaryan V.A., Solov'ev M.A., Kononov A.I. Modelirovanie operatsionnoy semantiki mashinnyih instruktsiy. [Simulation of operational semantics of machine instructions]. Programming and Computer Software, May 2011, Volume 37, Issue 3, pp 161 – 170, DOI 10.1134/S0361768811030030.

[19]. Silvio Ranise and Cesare Tinelli. The SMT-LIB Format: An Initial Proposal. Proceedings of PDPAR'03, July 2003

[20]. Nikolaj Bjørner, Leonardo de Moura. Z3: Applications, Enablers, Challenges and Directions // Sixth International Workshop on Constraints in Formal Verification Grenoble, 2009.

[21]. Huang S. K. et al. Software Crash Analysis for Automatic Exploit Generation on Binary Programs // Reliability, IEEE Transactions on. – 2014. – Т. 63. – №. 1. – С. 270-289.

[22]. Wu B. et al. Directed symbolic execution for binary vulnerability mining // Electronics, Computer and Applications, 2014 IEEE Workshop on. – IEEE, 2014. – pp. 614-617.