электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

УДК 004.415.532.2

Обзор современных методов анализа программного обеспечения

Матакаев И.Р., *студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Защита информации»

Научный руководитель: Астрахов А.В., к. т. н., доцент
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
zi@bmstu.ru

Taint-анализ

Данный метод построен на предположении, что для изменения параметров исполнения программы атакующий должен обеспечить ввод определенных значений со своей стороны, в то время как при обычной работе программы данные значения поступают из доверенных источников. К примеру, такие значения как адреса для jump и форматные сроки обычно предоставляются самой программой, а не из внешних недоверенных источников. Однако атакующий может попытаться перезаписать эти значения своими данными.

Данные, приходящие из недоверенных источников называются тентированными. При динамическом тейнт-анализе происходит пометка входных данных из недоверенных источников, а затем наблюдение за исполнением программы с целью отследить их распространение. К примеру, использование jump-адресов или форматных строк в качестве тейнт-данных часто помогает определить эксплуатирование переполнения буфера и уязвимостей форматных строк.

Следует отметить, что данный подход определяет атаки только при их осуществлении, то есть когда помеченные данные используются в опасных целях [3]. Способ сильно отличается от основных путей использования taint-анализа, который заключается в обнаружении факта неавторизованной перезаписи атакующим участков памяти во время процесса записи. Не всегда возможно во время процесса записи определить, является ли перезапись разрешенной, особенно для методов, не использующих исходные коды или специально скомпилированные бинарные файлы в своей работе. В результате, рассматриваемый подход не полагается на обнаружение при

перезаписи и независим от метода перезаписи и, следовательно, может обнаружить более широкий спектр атак.

Следует отметить, что плагины могут применять различные taint-политики в зависимости от требований к применению. Например, для анализа некоторых приложений нет необходимости распространять метки через таблицы поиска. Для некоторых приложений нам нужно распространять метки через прямой операнд, если область кода занята и помечена. То есть, во время распространения меток инструмент анализа меток позволяет плагинам определить, как они хотят довести её до места назначения.

Для использование динамического taint анализа необходимо ответить на вопросы: какие источники должны быть помечены, как эти метки должны распространятся и какое использование помеченных данных должно служить индикатором атаки.

В качестве недоверенных источников стоит рассматривать сетевые интерфейсы, так как векторы многих атак направлены против них. Реализован механизм поступления taint данных от аппаратной части, как то клавиатура, жесткий диск. Также поддерживается пометка высокоуровневых абстрактных объектов данных (например файлы, выходные данные вызова функции или структура данных в определенных приложениях или ядре ОС).

Каждый байт памяти, включая регистры, стек, кучу и т.д. обладает четырехбайтной теневой памятью, хранящей указатель на помеченную структуру данных со значениями 1, если местоположение помечено и 0, если нет. Происходит проверка аргументов и результатов системных вызовов и определение, есть ли необходимость в пометке памяти, записанной системным вызовом. Если память помечена, то происходит запись номера системного вызова, создание копии текущего стека и копии данных, которые были записаны. Местонахождение теневой памяти переходит к указателю на эту структуру. Информация может быть использована в дальнейшем для анализа уязвимости после обнаружении атаки. Так же теневая память может хранить только бит, указывающий является ли память помеченной.

Распространения меток. После того, как источник данных помечен, происходит наблюдение за инструкциями, манипулирующими данными с целью обнаружения, является ли результат помеченным. Такие инструкции делятся на три категории: перемещение данных(load, store, move, push, pop, etc.), арифметические операции (add, sub, xor, etc.) и неменяющие операции (nop, jmp, etc.). Для перемещения данных пункт

назначения будет помечен если любой байт источника помечен; для арифметических операций результат будет помечен, только если любой из операндов помечен.

Особый случай – функции, результат которых не зависит от входных данных. Например *«xor eax,eax»* всегда обращает регистр eax в ноль независимо от его значений. Значит, для оптимизации анализа необходимо реализовать распознавание этих функций и не помечать результаты.

Для отслеживания распространения помеченных данных, добавляются выражения перед операциями перемещения и арифметическими действиями. После того, как результат инструкции помечен одним из операндов, происходит установка теневой памяти результата на указание на ту же отмеченную структуру, что и отмеченный операнд. При обнаружении атаки можно будет отследить последовательность структур, чтобы определить, как помеченные данные распространялись по памяти.

Важной частью taint-анализа является сопоставление использования данных с политиками безопасности. Основная политика разработана для распознавания атак на форматные строки и атак, изменяющие адреса jump-функций, указатели на функции и отступы указателей на функции [3]. При обнаружении недопустимого использования помеченных данных, что скорее всего указывает на проведение атаки, необходимо начать анализ собранной информации. Вот некоторые возможные способы использования помеченных данных:

- **Адреса jump-функций.** По умолчанию происходит проверка, используются ли помеченные данные в качестве целей jump, как то адрес возврата, указатель на функцию или отступ указателя на функцию. Цель многих атак – перезаписать эти данные для перенаправления управляющего потока или на инструкции атакующего, или на стандартную функцию библиотеки (например exec), или на другой пункт программы (для обхода проверок безопасности). Есть небольшая вероятность того, что помеченные данные станут точкой назначения jump-функций. Тем не менее, эти проверки обнаруживают множество атак и генерируют очень мало ложно-позитивных результатов.
- **Форматные строки.** Осуществляется проверка на использование помеченных данных в качестве аргумента форматной строки (семейство функций printf). Эти проверки обнаруживают атаки форматных строк, во время которых атакующий передает программе вредоносную форматную строку для создания утечки памяти или записи нужных атакующему

значений по выбранному адресу. Эти проверки обнаруживают использование помеченных данных в качестве форматной строки, даже если она не содержит вредоносные спецификаторы формата для атаки. Это может использоваться для обнаружения прежде не известных уязвимостей форматных строк. Так же, сигнал может приходить только в случае, если строка помечена и содержит опасные определители формата (такие как %n). Эта опция полезна, если уязвимость уже известна и пользователь хочет обнаружить конкретную атаку.

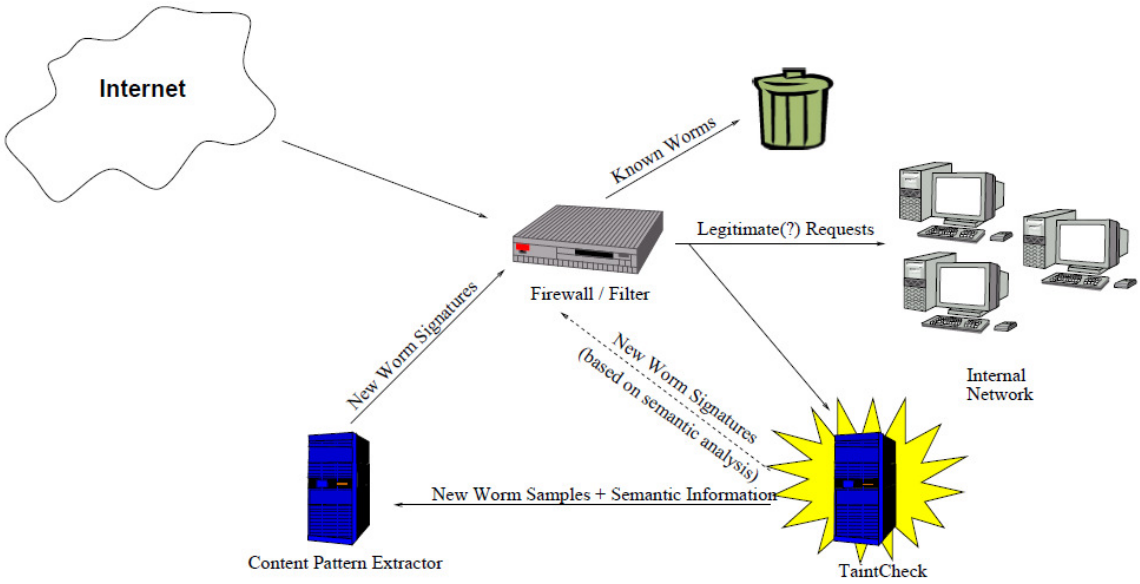
- Для введения данных проверок происходит перехват вызовов семейства функций printf (в том числе sylog) с обертками тейнт-анализа с целью убедится, что форматные строки не помечены, и затем вызывается первоначальная функция. У большинства программ данный способ не вызовет сбоев и сможет обнаружить любую атаку на форматные строки. Однако, если приложение использует собственный вариант данных функций, обертки вызываться не будут.
- **Аргументы системных вызовов.** Возможно проверит, является ли конкретный аргумент конкретной функции помеченным. Это может быть использовано для обнаружения атак, перезаписывающих данные, которые впоследствии используются в качестве аргументов вызова функций.
- **Проверка определенных приложений и библиотек.** Возможно так же обнаруживать атаки против определенных приложений и библиотек, осуществляя проверку конкретных областей памяти в определенных точках программы. Адреса могут быть точными и зависящими от текущего размера стека. В частности, можно проверять при вызове функции, является ли определенный аргумент этой функции помеченным. В качестве примера можно привести рассмотренные выше форматные строки, передаваемые printf-функциям.

Пример применения тейнт-анализа – автоматическая генерация сигнатур угроз. При обнаружении нового червя или эксплоита важно быстро создать сигнатуру этой угрозы, для того, чтобы отфильтровывать ее запросы до тех пор, пока уязвимость не будет закрыта. Один из подходов в данной области – использование автоматического семантического анализа нагрузки. Будут рассмотрены достоинства данного метода и

показаны способы улучшения автоматических систем генерации сигнатур с помощью тейнт-анализа.

Предшествующие техники генерации сигнатур использовали извлечение шаблона содержимого для создания сигнатур. То есть они определяют нагрузку атаки как непрозрачную последовательность байт и пытаются найти паттерны, повторяющиеся в нагрузке для использования в качестве сигнатуры.

Новый подход в данной области – использование автоматического семантического анализа нагрузки атаки для идентификации частей нагрузки, которые с высокой вероятностью являются постоянными (то есть полезными для создания сигнатуры). Семантический анализ может позволить создание точных сигнатур при наличии меньшего числа нагрузок атак, чем необходимо для генерации сигнатур в системах извлечения шаблона содержимого. С уменьшением требований к количеству образцов вредоносного кода станет возможно создание сигнатуры на ранних стадиях эпидемии и уменьшению урона.



Тейнт-анализатор может проводить автоматический семантический анализ нагрузок атак, так как он проверяет, как используется каждый байт нагрузки каждой атаки уязвимой программой на уровне инструкций процессора. В качестве первого шага осуществляется идентификация значений, используемых для перезаписи указателя функции или адреса возврата. Рисунок показывает, как автоматическая система генерации сигнатур может использовать тейнт-анализатор для обнаружения новых атак и проводить семантический анализ нагрузок атак. В некоторых случаях семантическая информация

может напрямую формировать сигнатуру. В других случаях она может быть использована в качестве основы работы экстрактора шаблона содержимого, возможно позволяя ему создавать сигнатуры с меньшим количеством образцов, чем требовалось изначально.

Slicing

Slicing — выделение из программы ее определенных частей, называемых срезами программы. Каждый срез определяется по отношению к некоторому интересующему нас критерию среза (slicing criterion), который обычно задается парой (точка программы, множество переменных). Если критерий среза C задан, то те элементы программы, которые прямо или косвенно влияют на значения, вычисляемые в указанных переменных в заданной точке программы, составляют срез программы относительно критерия C [5].

Существует несколько отличных друг от друга вариантов понятия среза, а также методов их вычисления. Основной причиной такого разнообразия стал тот факт, что различные приложения требуют от срезов различных свойств. Срез — это подмножество операторов и выражений исходной программы, которые прямо или косвенно влияют на значения, вычисляемые в точке критерия среза, но не обязательно составляют исполняемую программу. Важное различие проводится между статическими и динамическими срезами. Вычисление статических срезов не требует принятия предположений о входных данных программы, тогда как вычисление динамических срезов предполагает определенные входные значения.

Срезы вычисляются путем создания последовательных множеств транзитивно подходящих операторов, в соответствии с имеющимися зависимостями по данным и по управлению. Для вычисления срезов используется только доступная статически информация; таким образом, этот тип срезов обозначается как статический. Существует также граф зависимости по данным, представляющий собой ориентированный граф с вершинами, соответствующими операторам и выражениям программы, и дугами, соответствующими зависимостям по управлению и по данным между ними. Критерий среза идентифицируется вершиной в данном графе, а срез соответствует всем вершинам графа, из которых достижима интересующая нас вершина. Различные подходы к срезам программ используют модифицированные и расширенные версии графа зависимости в качестве своих внутренних представлений. Все срезы вычисляются при помощи объединения операторов и предикатов (управляющих выражений) при обратном обходе (т.е. при движении по дугам графа против их направления) управляющего графа или

графа зависимостей по данным, начиная с той точки, где находится критерий среза. Поэтому полученные таким образом срезы называются обратными статическими срезами.

Динамический срез программы исследует, как именно информация передается по программе для получения конкретного заданного значения: пользователь интерактивным образом обходит граф, представляющий зависимости по управлению и по данным программы. Например, если значение, вычисляемое оператором s, зависит от значений, вычисляемых оператором t, то пользователь может проследить путь от вершины, соответствующей s, к вершине, соответствующей t.

В случае динамического среза программ к рассмотрению принимаются только те зависимости между элементами программы, которые возникают при данном конкретном ее исполнении. Критерий динамического среза определяет входные данные и отсекает неподходящие вхождения операторов в истории исполнения; обычно этот критерий задается тройкой (входные данные, вхождение оператора, переменная). Другими словами, основное различие между статическим и динамическим подходами к построению срезов программ состоит в том, что динамический срез предполагает фиксированные входные данные, тогда как статический срез не делает предположений о входных данных программы.

Зависимости по данным и зависимости по управлению. Между операторами программы существуют два типа отношений, реализуемых в выполнениях программы: связь по передаче управления – управляющая связь, когда один оператор выполняется непосредственно вслед за другим; связь по передаче информации – информационная связь, когда один оператор при своем исполнении использует значение переменной, выработанное другим оператором. Поток управления (система управляющих связей) в программе обычно задается в виде так называемого управляющего графа. Это ориентированный граф, вершины которого соответствуют операторам, а дуги, соединяющие вершины, отражают возможность передачи управления от одного оператора к другому. Обычно управляющий граф содержит две выделенные вершины Start и Stop, называемые начальной (или входной) и конечной, а все остальные вершины делятся на преобразователи, из которых исходит по одной дуге, и распознаватели, из каждой из которых исходит не менее двух дуг. Таким образом, каждому выполнению программы соответствует некоторый путь по графу от его начальной вершины. В отличие от управляющего графа дуги графа программных зависимостей (ГПЗ) представляют отношения зависимостей по данным и по управлению между вершинами. Дуги зависимостей в ГПЗ определяют частичный порядок на множестве операторов

программы; операторы должны исполняться согласно этому порядку, чтобы семантика программы сохранялась.

Статический срез. Срез – исполняемую программу, полученную из исходной программы путем удаления нуля или большего числа операторов. Критерий среза C состоит из пары (n, V), где n — вершина в управляющем графе программы CFG, а V — подмножество переменных программы. Подмножество S операторов программы P называется срезом относительно критерия (n, V), если справедливы следующие два свойства:

1) S — допустимая исполняемая программа;
2) всегда, когда P останавливает исполнение на заданных входных данных, S также останавливает исполнение на этих входных данных, вычисляя те же значения для переменных из V, когда исполняется оператор, соответствующий вершине n.

Для любого критерия существует по крайней мере один срез: сама исходная программа. Срез является минимальным, если ни один другой срез для того же критерия не содержит меньшего числа операторов.

Итеративный алгоритм для вычисления приближенных минимальных срезов использует два «слоя» итерации, которые можно охарактеризовать следующим образом:

- трассировка транзитивных зависимостей по данным. Это итеративный процесс при наличии циклов;
- трассировка зависимостей по управлению, которое приводит к включению определенных распознавателей в срезы. Для каждого такого выражения шаг 1 повторяется для включения в срез операторов, от которых он информационно зависит.

Алгоритм определяет последовательные множества подходящих переменных, из которых выводятся множества подходящих операторов; результирующий срез получается из них как неподвижная точка в ходе последовательного схождения алгоритма.

Динамический срез. Формализация истории исполнения программы – траекторию, состоящую из последовательности «вхождений» операторов и выражений (управляющих предикатов). Критерий динамического среза представляет собой триплет (x, I^q, V), где x обозначает входное значение программы, вхождение оператора I^q является q-м элементом траектории, а V — подмножеством переменных программы. Динамический срез относительно критерия (x, I^q, V) определяют как исполняемую программу S, полученную

из программы P, путем удаления из нее нуля или большего числа операторов. Три ограничения налагаются на S. Во-первых, при исполнении с входным значением x траектория S идентична траектории P, если удалить из нее все операторы, не появляющиеся в S. Во-вторых, и программой, и ее срезом вычисляются идентичные значения для всех переменных из V в точке вхождения оператора, обозначенной в критерии. В-третьих, требуется, чтобы оператор I, соответствующий обозначенному в критерии экземпляру оператора I^q, входил в S. Динамический срез обладает следующим свойством: если цикл встречается в срезе, он проходится столько же раз, сколько и в исходной программе.

Чтобы вычислить динамические срезы, вводятся три динамических представления потока, формализующие зависимости между вхождениями операторов в траектории. Отношение DU (Definition-Use) связывает использование переменной с последним ее определением. Для конкретной траектории это отношение определяется единственным образом. Отношение TC (Test-Control) связывает самое последнее вхождение управляющего предиката с вхождениями операторов траектории, зависимыми по управлению от него. Это отношение определяется синтаксически только для структурированных конструкций программы. Вхождения одного и того же оператора составляют отношение IR (Identity).

Динамические срезы вычисляются итеративным образом, путем определения последовательных множеств S^i, состоящих из прямо и косвенно подходящих операторов. Для критерия среза (x, I^q, V) начальное приближение S^0 содержит последние определения переменных из V на траектории перед вхождением экземпляра оператора I^q, а также тестовые операции на траектории, от которых I^q зависим по управлению. Приближение Si+1 определяется следующим образом:

$S^i+1 = S^i \cup A^i+1$,

где A^i+1 определяется по следующему правилу:

$A^i+1 = \{X^p \mid X^p \notin S^i, (X^p, Y^t) \in (DU \cup TC \cup IR)$ для некоторого $Y^t \in S^i, p < q\}$,

где q — «метка» вхождения оператора, обозначенного в критерии среза. Динамический срез является неподвижной точкой S^C этого итеративного процесса (поскольку q конечно, она всегда существует): любой оператор X, экземпляр X^p которого входит в S^C, будет принадлежать срезу. Затем оператор I, соответствующий I^q, добавляется в срез.

Отношения DU и TC обходят траекторию только в обратном направлении. Целью введения IR-отношения является как обход траектории в обоих направлениях, так и

включение в срез всех операторов и управляющих предикатов, необходимых для обеспечения завершения всех циклов среза.

Графы зависимостей. Миллер и Чои впервые ввели динамическую вариацию ГПЗ, назвав ее динамическим графом программных зависимостей. Разработанный ими параллельный отладчик программ использует упомянутые графы для проведения обратного потокового анализа, наращивая их по мере необходимости. Перед исполнением строится вариация статического ГПЗ, а объектный код программы дополняется кодом, генерирующим лог-файл. Вдобавок к этому генерируется пакет эмулятора. Программы разбиваются на так называемые блоки эмуляции (обычно это процедуры). Во время отладки отладчик, используя лог-файл, статический ГПЗ и пакет эмуляции, перевычисляет блок эмуляции и получает информацию, необходимую для построения части динамического ГПЗ, соответствующей этому блоку. В случае, когда пользователь желает применить обратный потоковый анализ к еще не построенным частям графа, перевычисляется больше блоков эмуляции.

Отладка и анализ программ. Срезы программы полезны для отладки, поскольку срез программы потенциально позволяет игнорировать большое число операторов программы во время локализации ошибки. Если программа вычисляет ошибочное значение в переменной x, только операторы, входящие в срез относительно x, с наибольшей вероятностью внесли вклад в появление этой ошибки. В данном случае наиболее вероятно, что ошибка появляется в одном из операторов среза. Однако не всегда ошибка находится именно в срезе, так как она может содержаться в операторе, который был неумышленно исключен из среза. Тем не менее, эта техника полезна, так как она позволяет определить возможное место ошибки с большей точностью, чем без ее применения.

Заключение

В работе были рассмотрены основные методики, применяющиеся в настоящее время при анализе программного обеспечения класса «черный ящик», описан метод детектирования атак на основе taint-анализа. В последних работах в области анализа программного обеспечения активно рассматривается метод символьного анализа, который станет предметом дальнейших исследований.

Список литературы

1. Miller B.P., Choi J.-D. A mechanism for efficient debugging of parallel programs // ACM SIGPLAN Notices. 1988. Vol. 23, no. 7. P. 141-150.
2. Agrawal H., Horgan J.R. Dynamic Program Slicing // ACM SIGPLAN Notices. 1990. Vol. 25, no. 6. P. 246-256.
3. Newsome J., Song D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. Available at: http://bitblaze.cs.berkeley.edu/papers/taintcheck-full.pdf, accessed10.02.2014.
4. Тихонов А.Ю., Аветисян А.И. Развитие taint-анализа для решения задачи поиска программных закладок // Труды института системного программирования РАН. 2011. Т. 20. С. 9-24.
5. Касьянов В.Н., Мирзуитова И.Л. Slicing: срезы программ и их использование. Новосибирск: Институт систем информатики, 2002. 116 с.