ПРОБЛЕМА МОНИТОРИНГА ИНФОРМАЦИОННЫХ ПОТОКОВ, ВОЗНИКАЮЩИХ В ХОДЕ СБОРКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Тихомиров Н. А.[1], Ключарёв П. Г.[2]

DOI: 10.21681/2311-3456-2025-1-128-135

Цель исследования: *доказательство невозможности выявления информационных потоков, возникающих в ходе сборки программного обеспечения.*

Метод исследования: *математическое моделирование типового процесса сборки с последующим анализом полученных результатов в контексте фундаментальных математических задач.*

Результаты исследования: *в настоящей работе доказана фундаментальная невозможность точного детектирования информационного потока в рамках работы сборочной программы, а также рассмотрены предпосылки этой задачи и предложен перечень шагов, которые могут быть предприняты при организации эвристического решения. В составе предложенного эвристического решения рассмотрены популярные методы реализации отдельных его шагов, а само оно основано на необходимых условиях существования информационных потоков, что говорит о потенциале низкого уровня ложноотрицательных срабатываний.*

Научная новизна: *заключена в анализе применимости фундаментального подхода к решению поставленной задачи, а также в представлении эвристического подхода с низким уровнем ложноотрицательных срабатываний. Настоящая работа также в достаточно широкой мере рассматривает предпосылки поставленной задачи, что подчеркивает её важность.*

Ключевые слова: *теорема Райса, безопасность цепи поставок, избыточность на уровне файлов, недекларированные возможности, заимствованные компоненты, мониторинг сборки, сборочные системы, эвристические методы обеспечения безопасности.*

Введение

Первоначальным назначением процессов сборки программного обеспечения являлась (на примере проектов на языках C и C++) автоматизированная компиляция большого количества файлов с последующей линковкой их друг с другом. С другой стороны, в возможности всех сборочных инструментов (Make, Gradle и многих других) всегда входили в том или ином виде, – пусть даже с помощью вызова сторонних программ, – автоматическая генерация исходного кода, прямая модификация результирующих бинарных файлов и даже загрузка дополнительных материалов из сети во время сборки [1-3].

Более того, на основании выросших темпов разработки и повысившейся доступности пригодных к заимствованию компонентов ПО с открытым исходным кодом можно без сомнения говорить о предпосылках к снижению уровня понимания состава программного обеспечения его же разработчиком. Вполне рядовой становится ситуация, когда разработчик располагает десятками гигабайт кода его приложения, однако не может точно сказать, все ли имеющиеся у него файлы нужны для сборки итогового продукта.

С другой стороны, при формировании перечня исходников разработчик почти наверняка не учитывает файлы, которые берутся из его операционной системы. К примеру, для языка C++ это могут быть заголовочные файлы стандартной библиотеки. В то же время эти файлы в значительной степени влияют на структуру итогового исполняемого кода и его безопасность – к примеру, для некоторых проектов с открытым исходным кодом было выявлено, что более 90% обрабатываемых компилятором функций и структур проекта попадает в него именно из заголовочных файлов, включая системные[3].

Всё это не могло не привести к появлению атак на программное обеспечение, связанных с компрометацией системы сборки в ходе работы с заимствованным кодом. Подобные атаки могут быть отнесены к подмножеству семейств атак на цепи поставок [4, 5]. К примеру, в конце 2023 года и начале 2024 года появилось сразу несколько новостей о внедрении

1 Тихомиров Никита Александрович, студент кафедры ИУ-8 «Информационная безопасность» МГТУ им. Н. Э. Баумана, Москва, Россия. E-mail: nicktikhomirov02@gmail.com

2 Ключарёв Пётр Георгиевич, доктор технических наук, профессор кафедры ИУ-8 «Информационная безопасность» МГТУ им. Н. Э. Баумана, Москва, Россия. E-mail: pk.iu8@yandex.ru

3 Савицкий В.О. Инкрементальный анализ исходного кода на языках C/C++ // Труды Института системного программирования РАН, 2012. № 22. С. 119–130.

вредоносного кода в открытые пакеты для проектов на языке Python: arrapi, tmdbapis, nagerapi, pmmutils и PyTorch.

Аналогично, атаки могут проводиться и при помощи используемых средств разработки, включая сами инструменты сборки. На сегодняшний день для разработчика крайне важной становится задача идентификации и контроля используемых инструментов. Подобный подход активно продвигается как со стороны самих разработчиков, так и на уровне отечественных нормативных документов для разрабатываемых средств защиты информации[4].

Ещё одной типовой задачей в рамках обеспечения безопасности собираемого программного обеспечения является избавление от избыточности его исходного кода – по крайней мере, на уровне файлов исходных текстов. В некоторых случаях это является обязательным требованием, предъявляемым к ПО [7].

Более того, можно утверждать, что отсутствие мер по контролю избыточности исходных текстов программного обеспечения ведёт к затягиванию процесса разработки. К примеру, если применяемый разработчиком инструмент статического анализа не использует механизмы перехвата сборки, то исследование будет проведено для всей кодовой базы разработчика, – в этом случае всякая избыточность в исходных текстах приведёт к увеличению количества ложноположительных срабатываний.

Таким образом, можно говорить о необходимости контроля сборочной системы, что на формальном уровне может быть представлено в виде задачи валидации наличия или отсутствия соответствующих информационных потоков, возникающих в ходе процесса сборки программного обеспечения.

Основные понятия, используемые в поставленной задаче

Определение 1. В рамках данной работы будем определять процесс сборки программного обеспечения как процесс получения целевых файлов сборки при помощи некоторой программы, которую далее будем называть сборочной программой.

Для простоты модели будем считать список целевых файлов заданным (известным поэлементно) и не требующим валидации. Данный список конечен.

К примеру, в состав процесса сборки (для компилируемых языков) могут входить препроцессинг, компиляция и ассемблирование множества отдельных объектных файлов с их последующей линковкой в один общий исполняемый файл. Также в рамках настоящей работы допускается возможность генерации кода или ресурсных файлов, упаковки каких-либо файлов в архивы или же, наоборот, их распаковки. Типичными примерами сборочных программ могут служить Make, Maven, Gradle и их аналоги.

В рамках настоящей работы сборочная программа обладает следующими свойствами:

- исполняется пошагово, то есть ход исполнения программы может быть представлен в виде конечного или счётного списка состояний;
- состояние памяти процесса сборки может быть закодировано строкой конечной длины в некотором конечном алфавите (при этом не вводятся ограничения, что все состояния должны кодироваться строками одной и той же длины);
- число шагов процесса сборки конечно;
- исходный текст программы можно представить в виде строки конечной длины в некотором конечном алфавите (при этом не вводятся ограничения, что все сборочные программы должны кодироваться строками одной и той же длины).

Очевидно, что приведённый перечень ограничений не создаёт противоречия с практической областью.

Аналогично предположим, что любой файл (в том числе, любой целевой файл) может быть представлен конечной строкой в некоторой конечном алфавите, описывающей состояние его свойств (например, конкретное содержимое файла и конкретные права доступа к нему). Будем называть такую строку состоянием свойств файла. При этом не вводятся ограничения, что все состояния свойств файлов должны кодироваться строками одной и той же длины.

Информационным потоком, неформально говоря, можно называть перенос (с возможным преобразованием) информации между двумя файлами (объектами) при участии некоторого субъекта, являющегося инициатором этого преобразования. Эквивалентное, но более формальное определение можно встретить в национальном стандарте ГОСТ Р 59453.1-2021. Например, при распаковке архива можно говорить об информационном потоке из него в получаемые файлы при участии программы распаковки, а при компиляции можно говорить об информационном потоке из препроцессированного файла в файл с кодом на языке ассемблера при участии компилятора. Формальное определение информационного потока будет дано далее.

Используемая вычислительная модель

На практике в качестве вычислительной модели используются вычислительные устройства с достаточно сложной внутренней организацией. Упрощённой моделью этих вычислительных устройств можно считать равнодоступную адресную машину (РАМ).

Необходимо отметить, что данная модель не в полной мере соответствует её реальным аналогам, так как характеризуется бесконечной внутренней памятью, что неверно для прикладных устройств, конечность памяти которых говорит о, соответственно,

4 Сертификация программного обеспечения по требованиям доверия / Бегаев А. Н., Кашин С. В., Макаревич Н. А., Марченко А. А., Павлов Д. Д. // СПб.: Университет ИТМО, 2020. 40 с.

конечности множества состояний свойств файлов и множества состояний шагов исполнения сборочной программы.

Будем полагать это несоответствие незначительным, поскольку на практике затруднительно было бы использовать какие-либо из специфических для конечных множеств приёмов решения математических задач, включая, к примеру, табличные методы и методы полного перебора.

В представленной модели будем считать файлом некую последовательность ячеек памяти RAM (длина этой последовательности может меняться), а адресация по множеству ячеек осуществляется при помощи некоторой таблицы, подобной файловой системе в прикладных моделях. Во избежание необходимости математической формализации проблемы фрагментации памяти будем считать, что ячейки памяти одного файла не обязаны идти подряд и собраны в связный список.

Состояния свойств файлов могут быть представлены в виде конечных битовых строк, включающих сведения об имени файла, сведения о нём из таблицы, а также сведения, записанные в ячейки файла.

Исследуемая задача

Введём следующие основные для данной работы множества:

- M – множество сборочных программ;
- V – множество состояний свойств файлов, каждый элемент $v \in V$ этого множества сообщает информацию о некотором состоянии одного какого-либо файла;
- Q – множество пошаговых состояний всех процессов сборки для программ из M, которые эти процессы принимают в ходе исполнения; для этого множества определена функция $S{:}M \longrightarrow 2^Q$, которая ставит сборочной программе в соответствие множество состояний соответствующего ей процесса;
- $P = \{p \subset V | p\text{-конечное}\}$ – множество конечных подмножеств V.

Заметим, что нельзя говорить о множестве V как о множестве файлов в файловой системе. Подразумевается, что в него входят все состояния свойств всех файлов, то есть если в некоторый файл с состоянием свойств $v \in V$ произвести запись, то он будет иметь новое состояние $v^* \in V$.

С учётом оговоренных в прошлом разделе свойств рассматриваемых объектов сделаем следующие заключения о мощностях введённых множеств:

- множество M счётное как объединение счётного числа конечных множеств, так как любая программа однозначно кодируется конечной строкой в конечном алфавите, поэтому для любой фиксированной длины можно выделить конечное количество программ, а сами длины, очевидно, образуют ряд натуральных чисел;
- множество V счётное по аналогичным рассуждениям;
- множество P счётное по свойству множества конечных подмножеств счётного множества (если бы в P входили бесконечные подмножества, то оно было бы континуально);
- множество Q счётное как объединение счётного числа конечных множеств, так как у шагов каждого процесса сборки конечное число состояний;
- $\forall m_i \in M$, $S(m_i)$ – конечное множество.

По сделанному в прошлом разделе замечанию для сборочной программы $m_i \in M$ без уменьшения общности будем считать заданным множество результатов сборки и соответствующее ему множество результирующих состояний свойств результирующих файлов $V_i^{(res)} \subset V$.

Определение 2. Определим информационный поток как трёхместное отношение $(src,rcv,init)$ (источник, приёмник, инициатор), которое свидетельствует о том, что при участии субъекта $init$ часть информации в объекте rcv была сформирована на основании информации src.

В контексте настоящей работы информационный поток инициируется сборочной программой и связывает два состояния свойств файлов (различных, или же одного и того же), поэтому можно уточнить определение как $(src,rcv,init) \in V \times V \times M$.

В рамках исполнения сборочной программы $m_i \in M$ реализуется множество $T_i = \{(src,rcv,m_i) | src,rcv \in V\}$, то есть множество потоков, инициированных одним и тем же субъектом m_i. С учётом этого можно исключить из всех элементов T_i общий компонент, задав множество двуместных отношений (рёбер) E_i, как показано в (1). Это множество примечательно тем, что вместе с V образует орграф $G_i = \langle V,E_i \rangle$, который далее будем называть графом сборки для сборочной программы $m_i \in M$.

$$E_i = \{(src,rcv) | (src,rcv,m_i) \in T_i\}. \quad (1)$$

Множество информационных потоков, возникающих в ходе работы сборочной программы, $m_i \in M$ определяется множеством состояний $S(m_i)$ по некоторому соотношению, которое обозначим для графа сборки как функцию φ: $M \times 2^Q \longrightarrow (V \times V) \cup \{\sigma\}$, где «$\sigma$» – некорректная связь (т. к. функция φ, очевидно, с практической точки зрения может не иметь смысла для некоторых сочетаний программ с множествами состояний). Тогда можем дать определение множеству рёбер сборочного графа через данную функцию, как показано в соотношении (2). Будем обозначать связь между программой и множеством рёбер её сборочного графа, как показано в (3), так как связь через нижний индекс не всегда удобна для

$$m_i \in M, \quad E_i = \{\varphi(m_i,s) | s \subseteq S(m_i)\} \setminus \{\sigma\}. \quad (2)$$

$$E_i = E(m_i). \quad (3)$$

Достижимость вершины $v_2 \in V$ из вершины $v_1 \in V$ на графе G_i будем обозначать $v_1 \mapsto_{G_i} v_2$. Тогда можно ввести определение множеству исходных файлов сборки как $V_i^{(src)} = \{v \in V | (\exists v_1 \in V_i^{(res)}, v \mapsto_{G_i} v_1) \wedge \wedge (\nexists v_2 \in V: v_2 \mapsto_{G_i} v\}$, то есть это множество таких состояний свойств файлов, из которых существует информационный поток в результирующее множество и которые при этом не являются промежуточными этапами преобразования информации.

Необходимо заметить, что если бы вместо множества состояний свойств файлов в модели использовалось бы множество файлов, то необходимо было бы ввести временной параметр, поскольку для файлов потоки данных могут возникать в неестественном порядке наподобие приведённого в системе (4). Очевидно, что в приведённом примере неуместно было бы говорить о связи объектов a и c.

$$\begin{cases} (b,c,m_i) \text{ при } time = t \\ (a,b,m_i) \text{ при time} = t+1 \end{cases} \quad (4)$$

Также очевидно, что существует конструирующая функция ψ: $P \longrightarrow V$, для любого конечного множества состояний свойств файлов она строит такую программу, что выполняются утверждения:

1. $\forall p \in P$, если $\psi(p) = m_i$, то в графе $G_i = \langle V, E(m_i)\rangle$ выполняется $\forall v_{res} \in p$, $\exists v \in V: (v, v_{res}) \in E(m_i)$ – сконструированная программа гарантировано кодирует в числе прочего те рёбра, которые заходят в эти вершины;
2. $\forall p \in P$, для $\psi(p) = m_i$ выполняется $p \subseteq V_i^{(res)}$.

В прикладной области эта совокупность условий означает, что для любого конечного множества состояний свойств файлов можно написать сборочную программу, которая каким-либо образом их генерирует (однако, конечно же, не гарантируется, что у результирующей программы не будет побочных эффектов).

Функция ψ считается вычислимой и одинаковой для любой частной задачи.

Для моделирования результата сборочной программы $m_i \in M$ введём семейство характеристических (т.е. бинарных) векторов β_i множества V. Вектор β_i является представлением файловой системы компьютера после работы сборочной программы m_i, ставя в соответствие имеющимся состояниям свойств файлов единицу, а отсутствующим – ноль. Вектор β_i обладает следующими (обоснованными с прикладной точки зрения) свойствами:

1. если после работы сборочной программы $m_i \in M$ имеется результат сборки $v_k \in V_i^{(res)}$, то $\beta_{ik} = 1$;
2. $\|\beta_{ik}\|$ – конечное число (т.е. по результатам работы сборочной программы $m_i \in M$ в файловой системе не может быть бесконечное число файлов).

Из приведённых свойств следует, что вектор β_i, хотя и является бесконечномерным как характеристический вектор счётного множества, может быть представлен конечной строкой, кодирующей конечный перечень индексов (натуральных чисел), обозначающих координаты, в которых характеристический вектор β_i принимает ненулевое значение. Таким образом, можно сказать, что программа $m_i \in M$ вычисляет вектор β_i.

В рамках настоящей работы для сборочных программ нет необходимости в формализации входных данных (можно считать входные данные зашитыми в код программы), однако для единообразия с принятой для Машин Тьюринга и вычисляемых ими функций нотацией введём λ – фиктивный вход функций, вычисляемых сборочными программами. Проще говоря, программа $m_i \in M$ будет вычислять функцию $m_i(\lambda) = \beta_i$ (вычисляемую программой функцию будем обозначать так же, как и саму программу). Здесь следует напомнить, что по изначальному предположению сборочные программы останавливаются за конечное количество шагов, то есть функция $m_i(\lambda)$ вычислима.

Таким образом, исследование некоторой программы сборки имеет следующие входные сведения:

- сборочную программу $m_i \in M$ с процессом, который завершается за конечное число шагов и имеет множество состояний $S(m_i)$;
- считающееся известным и (для простоты) неоспоримым конечное множество результирующих состояний свойств файлов $V_i^{(res)} \subset V$, являющихся результатами сборки для программы $m_i \in M$;
- общий для всех задач ранее описанный объект ψ.

Из приведённых ранее рассуждений следует, что задача выявления информационных потоков сводится к задаче восстановления рёбер графа G_i. Из тех же рассуждений справедлива и обратная сводимость. Далее будем рассматривать именно постановку задачи с использованием графа G_i. Выпишем её полную постановку.

Задача 1. Имеется сборочная программа $m_i \in M$, для неё известны её результирующие состояния свойств файлов $V_i^{(res)} \subset V$ и все состояния её исполнения $S(m_i)$ – для описанной программы необходимо восстановить граф.

В рамках настоящей работы далее будет продемонстрировано, что задача 1 не имеет математически корректного алгоритма решения.

Утверждение 1. Задача 1 алгоритмически неразрешима.

Анализ поставленной задачи

Построим семейство множеств $F_{vw} = \{m_j(\lambda) | (m_j \in M) \wedge \wedge ((v,w) \in E(m_j))\}$ с параметрами $v,w \in V$. Проще говоря, для пары состояний свойств файлов $v,w \in V$

множество F_{vw} содержит все вычисляемые сборочными программами функции, сборочные графы которых содержат ребро (v,w), которое, будучи ребром в некотором графе G_j, по ранее приведённым рассуждениям однозначно соответствует информационному потоку (v,w,m_j). Множество F_{vw} можно неформально назвать множеством-признаком – если программа принадлежит ему, то она обладает некоторым свойством, и наоборот.

Определение 3. Нетривиальной парой состояний свойств файлов будем называть пару $v^*,w^* \in V$, для которой верно, что $F_{v^*w^*} \neq \emptyset$ и $\bar{F}_{v^*w^*} \neq \emptyset$.

Утверждение 2. Нетривиальная пара состояний свойств файлов существует.

Доказательство утверждения 2.

Возьмём произвольную программу $m_j \in M$ с конечным множеством состояний $S(m_j)$. Справедливо, что $|E_j| \leq 2^{|S(m_j)|}$, что следует из соотношения (2), требования конечности числа шагов исполнения программы, а также свойств множества подмножеств конечного множества. Возьмём произвольные различные $2^{|S(m_j)|}+1$ состояний свойств файлов, полученное множество будем обозначать $p^* \in P$, после чего применим к нему конструирующую функцию ψ, которая возвращает сборочную программу, генерирующую заданные состояния свойств файлов. Из свойств ψ следует, что полученная программа $\psi(p^*)$ гарантировано будет реализовывать в своём графе сборки рёбра, инцидентные с вершинами из множества p^*.

Таким образом, произвольно взятая программа m_j содержит не более, чем $2^{|S(m_j)|}$ рёбер в своём сборочном графе, а сконструированная программа $\psi(p^*)$ в силу свойств конструирующей функции ψ имеет не менее, чем $|p^*| = 2^{|S(m_j)|}+1$ ребёр. Из мощностных соображений очевидно существование нетривиальной пары состояний, как показано в соотношении (5).

$$\exists v^*,w^* \in V: \begin{cases} ((v^*,w^*) \in E(\psi(p^*)) \\ ((v^*,w^*) \notin E_j \end{cases} \quad (5)\ )$$

Конец доказательства утверждения 2.

С использованием утверждения 2 можно доказать утверждение 1, обратившись к теореме Райса.

Доказательство утверждения 1.

Задача выявления информационного потока в ходе исполнения сборочной программы является тривиальным следствием теоремы Райса.

Теорема Райса гласит, что если имеется нетривиальное свойство вычислимых функций (непустое множество с непустым дополнением), то задача отнесения программы к этому множеству либо же его дополнению является алгоритмически неразрешимой[5].

5 Емельченков Е. П., Емельченков В. Е. Вычислимость. Введение в теорию алгоритмов // Математическая морфология: электронный математический и медико-биологический журнал. 2000. № 3–3. С. 121–130. EDN: BJIQOL

Соответствие поставленной задачи условию данной теоремы показано в табл. 1.

Таблица 1.
Удовлетворение условия теоремы Райса, описанной в настоящей работе моделью

Требование теоремы	Удовлетворение требования
Некоторое свойство (множество) функций	F_{vw} для некоторых $v,w \in V$
Свойство нетривиально	Существование нетривиальных $v,w \in V$ показано в доказательстве утверждения 2.
Функции вычислимы	Аргумент функции: введённый фиктивно объект λ Результат функции: характеристический вектор из множества $\{\beta\}$ По изначальному предположению настоящей работы: все сборочные процессы завершают свою работу, т.е. функция вычислима

Конец доказательства утверждения 1.

По соотношению (1) можно совершить обратный переход от доказанного утверждения 1 к (эквивалентной) постановке задачи через понятие информационного потока.

Таким образом, для любого (нетривиального) информационного потока задача его выявления в рамках исполнения сборочной программы является алгоритмически неразрешимой по Теореме Райса.

Эвристический подход к решению поставленной задачи

Решение рассмотренной задачи в том или ином виде является этапом многих этапов обеспечения информационной безопасности при разработке программного обеспечения – к примеру, оно может использоваться для улучшения качества статического анализа [8].

Так как разработка математически корректных методов решения данной задачи не представляется возможной, рассмотрим эвристический подход.

Методы эвристического решения поставленной задачи напрямую связаны с методом вычисления введённого ранее множества $S(m_i)$, которое, напомним, содержит все закодированные каким-либо образом промежуточные состояния хода исполнения

сборочной программы, а также, соответственно, с применяемым методом кодирования этих состояний.

Прямолинейным и громоздким решением является симуляция всего программного окружения при помощи соответствующего инструмента с пошаговым анализом преобразований, осуществляющихся над памятью симулируемого устройства, это в достаточной степени решает задачу формирования перечня промежуточных состояний [9, 10].

Для этого может быть использован, к примеру, общедоступный инструмент QEMU международной совместной разработки ([11]), либо его модификация от Института системного программирования им. В. П. Иванникова Российской академии наук, либо же отечественный инструмент Korycat от отечественной компании ООО «Инфорион» [12, 13].

К сожалению, для реализации какого-либо эвристического решения описанный подход является крайне неудобным, так как требует восстановления данных о состоянии из набора двоичных данных, которым является снимок памяти процесса сборочной программы. На основании этого предлагается обратиться к менее доскональным, но более универсальным методам мониторинга – например, к мониторингу обращений к файловой системе.

Избранные методы мониторинга обращений к файловой системе

Мониторинг действий с файловой системой может осуществляться, например, при помощи того же эмулятора QEMU[6]. Альтернативным подходом является мониторинг системных вызовов, который может осуществляться либо через прямой мониторинг при помощи соответствующей утилиты (например, Strace или какое-либо иное решение на базе системного вызова ptrace), либо же через подмену разделяемой динамически линкуемой библиотеки (посредством переопределения переменной окружения «LD_PRELOAD»), которая предоставляет интерфейс системных вызовов программам пользовательского уровня, включая сборочную программу – переопределению могут подвергаться, например, библиотечные функции, являющиеся интерфейсами для системного вызова «execve», аргументы которого во время сборки проверяются на предмет вызова компилятора[7].

Менее популярным решением является прямая подмена файлов, содержащих логику сборочной программы (например, подмена компилятора для языков C и C++) [14, 15].

В результате мониторинга действий над файловой системой может быть сформирована последовательность файлов и доступов к ним – примеры доступов можно почерпнуть, например, из области построения SIEM-систем, где в «усреднённый» перечень видов доступа к абстрактной сущности включают шесть основных действий:

- создание сущности;
- удаление сущности;
- коммуникация с сущностью;
- манипуляция данными сущности;
- манипуляция работой сущности;
- получение дополнительных сведений о сущности (например, размер и время создания).

Подобный перечень видов доступа (в значительно более подробной форме) используется, например, при категоризации событий в MaxPatrol SIEM 10 версии.

Заметим, что классические модели разграничения доступа, опирающиеся на понятие информационного потока, к примеру, модель Белла-ЛаПадулы, вводят другой набор доступов, включающий чтение, запись и, в некоторых моделях, исполнение. Далее будем считать, что «манипуляция данными сущности» соответствует доступу на запись, «манипуляция работой сущности» соответствует доступу на исполнение, «коммуникация с сущностью» соответствует неразделимой совокупности доступов на запись и чтение, а для доступа на чтение аналога в приведённом перечне нет, поэтому дополним его соответствующим седьмым действием.

Описание предлагаемого подхода

Нетрудно заметить, что любой информационный поток имеет два необходимых условия для своего существования:

- субъект-инициатор производит чтение из источника (либо осуществляет коммуникацию с ним);
- субъект-инициатор производит манипуляцию данными приёмника (либо осуществляет коммуникацию с ним).

С учётом приведённых ранее рассуждений, предлагаемый эвристический подход заключается в том, что выполнение необходимого условия считается признаком потенциального существования потока.

В соответствии с предлагаемым подходом следует с использованием некоторой системы мониторинга доступов к файловой системе зафиксировать перечень файлов доступов к ним во время работы сборочной программы, после чего для всех сочетаний

6 Stepanov V. M., Dovgalyuk P. M., Poletaev D. N. Tracing ext3 file system operations in the QEMU emulator. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 5, 2018. pp. 101–108. DOI: 10.15514/ISPRAS-2018-30(5)-6.

7 Белеванцев А. А. Многоуровневый статический анализ исходного кода для обеспечения качества программ: дис. ... доктора физико-математических наук 05.13.11 / Белеванцев А. А. – М., 2017. – 229 с.

«чтение»-«запись» зарегистрировать наличие информационных потоков.

После указанных действий предлагается упорядочить перечень потоков по метке времени, – подобные метки можно определить для любого из рассмотренных методов (симуляция, системные вызовы и т.д.), – в паре «чтение»-«запись» следует при этом ориентироваться на второе действие («запись»). В отсортированном перечне информационных потоков можно распознать конкретный интересующий проверяющего сборку эксперта (в рамках некоторой поставленной задачи) посредством валидации прямого или транзитивного наличия такового потока.

В целях фильтрации ложноположительных (или не имеющих смысла) информационных потоков предлагается также дополнительно удалить из перечня потоков те, которые приводят в удаляемый либо создаваемый (пересоздаваемый) объект файловой системы.

Также стоит заметить, что использование необходимого условия в достаточной для эвристического подхода мере снижает риск возникновения ложноотрицательных срабатываний детектирующего алгоритма.

Заключение

В настоящей работе рассмотрены предпосылки задачи выявления информационных потоков в рамках работы сборочной программы, доказана фундаментальная невозможность точного решения этой задачи, а также предложен перечень шагов, которые могут быть предприняты при организации эвристического решения.

В рамках предложенного эвристического решения рассмотрены популярные методы перехвата состояний хода исполнения сборочной программы и подходы к интерпретации полученных данных.

Полученный в данной работе результат может быть также дополнен иными моделями, однако все они также будут носить эвристический характер по оговоренным причинам.

Потенциальным развитием настоящей работы может являться реализация описанного подхода и его дальнейшее расширение.

Литература

1. *Фигловский К. С., Никифоров И. В., Юсупова О. А. Использование Gradle build cache для оптимизации времени сборки // Современные Технологии в Теории и Практике Программирования. Сборник материалов научно-практической конференции. – СПб: Федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский политехнический университет Петра Великого», 2021. С. 127–129.*
2. *Арустамян С. С., Антипов И. С. Интеллектуальные методы фаззинг-тестирования в рамках цикла безопасной разработки программ // Безопасные Информационные Технологии. Сборник трудов Двенадцатой международной научно-технической конференции. – М.: Московский государственный технический университет имени Н. Э. Баумана (национальный исследовательский университет), 2023. С. 11–15.*
3. *Poeplau S., Francillon A. Symbolic Execution with SymCC: Don't Interpret, Compile! // Proc. of 29-th USENIX Security Symposium, 2020, pp. 181–198.*
4. *Леонов Н. В. Противодействие уязвимостям программного обеспечения. Часть 2. Аналитическая модель и концептуальные решения // Вопросы кибербезопасности. 2024, № 3 (61). С. 90–95. DOI: 10.21681/2311-3456-2024-3-90-95.*
5. *On the prevalence of software supply chain attacks: Empirical study and investigative framework / Andreoli A., Lounis A., Debbabi M., Hanna A. // Proceedings of the Tenth Annual DFRWS Europe Conference, 2023. № 44. DOI: 10.1016/j.fsidi.2023.301508.*
6. *Практические аспекты выявления уязвимостей при проведении сертификационных испытаний программных средств защиты информации / В. В. Вареница, А. С. Марков, В. В. Савченко, В. Л. Цирлов // Вопросы кибербезопасности. – 2021. – № 5(45). – С. 36–44. – DOI 10.21681/2311-3456-2021-5-36-44. – EDN TBQOCG.*
7. *Kotlin с точки зрения разработчика статического анализатора / Афанасьев В. О., Поляков С. А., Бородин А. Е., Белеванцев А. А. // Труды Института системного программирования РАН, 2021. № 33 (6). С. 67–82.*
8. *Девянин, П. Н. Формирование методологии разработки безопасного системного программного обеспечения на примере операционных систем / П. Н. Девянин, В. Ю. Тележников, А. В. Хорошилов // Труды Института системного программирования РАН. – 2021. – Т. 33, № 5. – С. 25–40. – DOI 10.15514/ISPRAS-2021-33(5)-2. – EDN WBXBTQ.*
9. *Natch: Определение поверхности атаки программ с помощью отслеживания помеченных данных и интроспекции виртуальных машин / П. М. Довгалюк, М. А. Климушенкова, Н. И. Фурсова [и др.] // Труды Института системного программирования РАН. – 2022. – Т. 34, № 5. – С. 89–110. – DOI 10.15514/ISPRAS-2022-34(5)-6. – EDN JNKSTV.*
10. *Коваленко Р. Д., Макаров А. Н. Динамический анализ IoT-систем на основе полносистемной эмуляции в QEMU // Труды Института системного программирования РАН. 2021. № 33–5. С. 155–166.*
11. *Аристов Р. С., Гладких А. А., Давыдов В. Н., Комахин М. О. Разработка программной платформы Korycat эмуляции сложных вычислительных систем // Наноиндустрия, 2019. № S (89). С. 350–352.*
12. *Гладких А. А., Кемурджиан А. Л., Комахин М. О. Отладка и анализ устройств и приложений с операционной системой на базе Linux в эмуляторе Korycat // Наноиндустрия, 2020. № S5-2 (102). С. 406–408.*
13. *Вишняков А. В. Поиск ошибок в бинарном коде методами динамической символьной интерпретации: дис. ... кандидата физико-математических наук 2.3.5 / Вишняков А. В. – М., 2022. – 131 с.*
14. *Шимчик, Н. В. Irbis: статический анализатор помеченных данных для поиска уязвимостей в программах на C/C++ / Н. В. Шимчик, В. Н. Игнатьев, А. А. Белеванцев // Труды Института системного программирования РАН. – 2022. – Т. 34, № 6. – С. 51–66. – DOI: 10.15514/ISPRAS-2022-34(6)-4.*

DATA FLOW MONITORING PROBLEM IN SOFTWARE BUILDING PROCESS

Tikhomirov N. A.[8], Klyucharev P. G.[9]

Keywords: *Rice's theorem, supply chain security, file-level redundancy, undeclared capabilities, open-source software, build process monitoring, build systems, heuristic approaches to enforcement of information security.*

The purpose of the study *is a formal proof for impossibility of precise identification of data flows, that occur in process of software building.*

Research methods: *analysis of typical building process mathematical model in relation to fundamental problems of mathematics.*

Study results: *in the proposed study a formal proof is suggested, that it is fundamentally impossible to identify data flow in software building process precisely. Practical applications of mentioned precise identification are also covered by this work as well as heuristic resolution steps for the problem are suggested. Implementation means for some of suggested steps overview is also provided. Proposed algorithm is aware of necessary conditions for data flows to exist, which leads to a potentially low level of false negatives.*

The scientific novelty *consists in applicability analysis of a fundamental approach to named problem resolution as well as made suggestion for heuristic algorithm with potentially low level of false negatives. Practical reasons for the problem to be researched are also covered by the study, that strengthens its importance.*

References

1. *Figlovskij K. S., Nikiforov I. V., Jusupova O. A. Ispol'zovanie Gradle build cache dlja optimizacii vremeni sborki // Sovremennye Tehnologii v Teorii i Praktike Programmirovanija. Cbornik materialov nauchno-prakticheskoj konferencii. – SPb: Federal'noe gosudarstvennoe avtonomnoe obrazovatel'noe uchrezhdenie vysshego obrazovanija «Sankt-Peterburgskij politehnicheskij universitet Petra Velikogo», 2021. S. 127–129.*
2. *Arustamjan S. S., Antipov I. S. Intellektual'nye metody fazzing-testirovanija v ramkah cikla bezopasnoj razrabotki programm // Bezopasnye Informacionnye Tehnologii. Sbornik trudov Dvenadcatoj mezhdunarodnoj nauchno-tehnicheskoj konferencii. – M.: Moskovskij gosudarstvennyj tehnicheskij universitet imeni N. Je. Baumana (nacional'nyj issledovatel'skij universitet), 2023. S. 11–15.*
3. *Poeplau S., Francillon A. Symbolic Execution with SymCC: Don't Interpret, Compile! // Proc. of 29-th USENIX Security Symposium, 2020, pp. 181–198.*
4. *Leonov N. V. Protivodejstvie ujazvimostjam programmnogo obespechenija. Chast' 2. Analiticheskaja model' i konceptual'nye reshenija // Voprosy kiberbezopasnosti. 2024, № 3 (61). S. 90–95. DOI: 10.21681/2311-3456-2024-3-90-95.*
5. *On the prevalence of software supply chain attacks: Empirical study and investigative framework / Andreoli A., Lounis A., Debbabi M., Hanna A. // Proceedings of the Tenth Annual DFRWS Europe Conference, 2023. № 44. DOI: 10.1016/j.fsidi.2023.301508.*
6. *Prakticheskie aspekty vyjavlenija ujazvimostej pri provedenii sertifikacionnyh ispytanij programmnyh sredstv zashhity informacii / V. V. Varenica, A. S. Markov, V. V. Savchenko, V. L. Cirlov // Voprosy kiberbezopasnosti. – 2021. – № 5(45). – S. 36-44. – DOI 10.21681/2311-3456-2021-5-36-44. – EDN TBQOCG.*
7. *Kotlin s tochki zrenija razrabotchika staticheskogo analizatora / Afanas'ev V. O., Poljakov S. A., Borodin A. E., Belevancev A. A. // Trudy Instituta sistemnogo programmirovanija RAN, 2021. № 33 (6). S. 67–82.*
8. *Devjanin, P. N. Formirovanie metodologii razrabotki bezopasnogo sistemnogo programmnogo obespechenija na primere operacionnyh sistem / P. N. Devjanin, V. Ju. Telezhnikov, A. V. Horoshilov // Trudy Instituta sistemnogo programmirovanija RAN. – 2021. – T. 33, № 5. – S. 25–40. – DOI 10.15514/ISPRAS-2021-33(5)-2. – EDN WBXBTQ.*
9. *Natch: Opredelenie poverhnosti ataki programm s pomoshh'ju otslezhivanija pomechennyh dannyh i introspekcii virtual'nyh mashin / P. M. Dovgaljuk, M. A. Klimushenkova, N. I. Fursova [i dr.] // Trudy Instituta sistemnogo programmirovanija RAN. – 2022. – T. 34, № 5. – S. 89–110. – DOI 10.15514/ISPRAS-2022-34(5)-6. – EDN JNKSTV.*
10. *Kovalenko R. D., Makarov A. N. Dinamicheskij analiz IoT-sistem na osnove polnosistemnoj jemuljacii v QEMU // Trudy Instituta sistemnogo programmirovanija RAN. 2021. № 33–5. S. 155–166.*
11. *Aristov R. S., Gladkih A. A., Davydov V. N., Komahin M. O. Razrabotka programmnoj platformy Kopycat jemuljacii slozhnyh vychislitel'nyh sistem // Nanoindustrija, 2019. № S (89). S. 350–352.*
12. *Gladkih A. A., Kemurdzhian A. L., Komahin M. O. Otladka i analiz ustrojstv i prilozhenij s operacionnoj sistemoj na baze Linux v jemuljatore Kopycat // Nanoindustrija, 2020. № S5-2 (102). S. 406–408.*
13. *Vishnjakov A. V. Poisk oshibok v binarnom kode metodami dinamicheskoj simvol'noj interpretacii: dis. ... kandidata fiziko-matematicheskih nauk 2.3.5 / Vishnjakov A.V. – M., 2022. – 131 s.*
14. *Shimchik, N. V. Irbis: staticheskij analizator pomechennyh dannyh dlja poiska ujazvimostej v programmah na C/C++ / N. V. Shimchik, V. N. Ignat'ev, A. A. Belevancev // Trudy Instituta sistemnogo programmirovanija RAN. – 2022. – T. 34, № 6. – S. 51–66. – DOI 10.15514/ISPRAS-2022-34(6)-4.*

8 Nikita A. Tikhomirov, student of the «Information Security» department, Bauman Moscow State Technical University, Moscow, Russian Federation. E-mail: nicktikhomirov02@gmail.com

9 Petr G. Klyucharev, Dr.Sc. Associate Professor of Information Security department, Bauman Moscow State Technical University, Moscow, Russian Federation. E-mail: pk.iu8@yandex.ru