ТЕХНОЛОГИИ СТАТИЧЕСКОГО И ДИНАМИЧЕСКОГО АНАЛИЗА УЯЗВИМОСТЕЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

***Аветисян Арутюн Ишханович**, доктор физико-математических наук, доцент*
***Белеванцев Андрей Андреевич**, кандидат физико-математических наук*
***Чукляев Илья Игоревич**, кандидат технических наук, доцент*

Представлены результаты анализа существующих подходов защиты и идентификации дефектов (уязвимостей и ошибок) в исходном и бинарном кодах программ, проводимых на различных этапах их разработки, проектирования и эксплуатации. Обобщены наиболее распространенные дефекты кода и объекты воздействия, функциональные и эргономические требования, предъявляемые к современным системам анализа.

***Ключевые слова**: статический и динамический анализ, исходный и бинарный код, дефекты (уязвимости и ошибки), требования, технологии защиты, объекты несанкционированных воздействий.*

THE TECHNOLOGIES OF STATIC AND DYNAMIC ANALYSES DETECTING VULNERABILITIES OF SOFTWARE

***Arutyun Avetisyan**, Doctor of Science, Associate Professor*
***Andreiy Belevantsev**, Ph.D.*
***Ilia Chucklyaev**, Ph D.., Associate Professor*

Results of the analysis of the existing approaches to protection and identification of defects (vulnerabilities and errors) in the source and binary code of software, provided on the different stages of development and maintenance. The most common defects (vulnerabilities and errors) and objects of destructive programming attacks are generalized, functional and ergonomic requirements to the modern analysis software are presented.

***Keywords**: static and dynamic analyses, source and binary code, defects (vulnerabilities and faults), requirements, technologies of security, objects of destructive programming attacks.*

Высокая сложность современного программного обеспечения (ПО), обусловленная большим объемом (до нескольких миллионов строк) исходного кода, интегрирование методов обфускации (запутывания), деградация производительности процессов, наличие дефектов (уязвимостей и ошибок) является *фундаментальной проблемой*, вызванной текущим состоянием развития информационных технологий [1, 2].

Уязвимости ПО – критические ошибки, не выявленные в ходе тестирования и не декларированные спецификацией разработчика или заложенные преднамеренно, предоставляющие злоумышленникам исключительные возможности по разглашению информации, ее модификации, блокированию использования и безостаточному уничтожению без возможности восстановления (рисунок 1) [3, 4].

Возможность изменения основных свойств защищенности (доступность, целостность, кон-

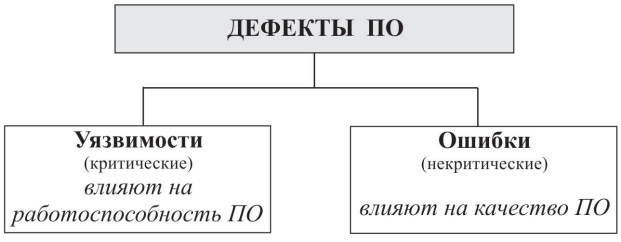

***Рис. 1.** Классификация дефектов ПО*

фиденциальность) информационных ресурсов и дестабилизации процессов функционирования информационно-вычислительных систем различного назначения посредством применения злоумышленниками несанкционированных воздействий деструктивного характера (атак) на уязвимости ПО предопределяют острую потребность в своевременном обнаружении дефектов (уязвимостей и ошибок) на этапах разработки и проектирования ПО, проверки соответствия их заявленной политики безопасности и реализации механизмов защиты [1, 4, 5].

Существующие подходы защиты и идентификации дефектов (уязвимостей и ошибок) ПО

Технологии поиска дефектов (уязвимостей и ошибок) и защиты от них разрабатываются по трем основным направлениям. Во-первых, используются *системы автоматического поиска дефектов с помощью статического анализа исходного кода* программ, которые можно применять на самых ранних этапах разработки, что делает исправление дефектов максимально дешевым. Такие системы промышленного качества обрабатывают программные системы в миллионы строк кода, обладают приемлемым уровнем истинных срабатываний (30-70% найденных ошибок оказываются истинными) и анализируют все возможные варианты выполнения программы одновременно. Возможно применять более точные методы анализа, методы верификации программ для самых важных участков программы за счет уменьшения объема анализируемого кода.

Во-вторых, применяются *системы динамического анализа бинарного кода* программ, позволяющие многократно запускать заданную программу на автоматически генерируемом наборе входных данных и отслеживать ситуации возникновения дефектов. Системы динамического анализа просматривают лишь часть возможных наборов входных данных, но при нахождении ошибки сразу позволяют получить данные, на которых эта ошибка проявляется (т.е. не имеют ложных срабатываний). Применимость этих систем ограничена существенными требованиями к ресурсам и ограничением на максимальный размер анализируемой программы (обычно десятки тысяч строк кода).

Наконец, разрабатываются *технологии защиты программы и ее окружения* от эксплуатации имеющихся в ней уязвимостей во время работы программы (рисунок 4). Это направление является актуальным из-за того, что все системы анализа не гарантируют полного нахождения имеющихся в ПО уязвимостей. Используемые методы данного подхода позволяют существенно затруднить атакующему успешную реализацию своих целей, исключить возможность эксплуатации даже известной уязвимости и сформировать универсальный метод взлома конкретной программы.

Наиболее распространенные дефекты (уязвимости и ошибки) ПО

Согласно приведенному определению уязвимости [3, 4], будем отличать понятия ошибки программы, наличие которых не приводят к взлому программы, от понятия уязвимость, которые предоставляют возможность взлома программной системы. Исходя из этого, дефекты ПО можно разделить на уязвимости (критические ошибки), приводящие к нарушению работоспособности, отказу в обслуживании, изменению защищенности информационных ресурсов, и некритические, влияющие лишь на качество программной системы (например, программа использует больше памяти для работы, чем необходимо, из-за утечек памяти при работе с ней). Наибольший интерес представляет поиск уязвимостей (критических ошибок), однако и некритические ошибки могут привести к уязвимостям, поэтому современные системы анализа обычно направлены на идентификацию обоих типов дефектов [2, 5].

Актуальная общепринятая классификация дефектов по типам представлена в базе Common Weakness Enumeration [6], список зарегистрированных уязвимостей – в базе Common Vulnerabilities and Exposures организации MITRE [7]. Наиболее распространенными типами дефектов (рисунок 2) являются:

переполнение буфера;

ошибки при работе с динамической памятью (утечка памяти, разыменование нулевых указателей и др.);

ошибки обработки пользовательских данных;

ошибки форматных строк;

ошибки синхронизации (взаимные блокировки, отсутствующие блокировки и т.п.);

утечки памяти и других ресурсов системы;

некорректная работа с временными файлами и другими интерфейсами ОС;

уязвимости безопасности (слабое шифрование, хранение пароля в явном виде и т.п.), не вытекающие непосредственно из дефектов, можно выделить в отдельный класс уязвимостей.

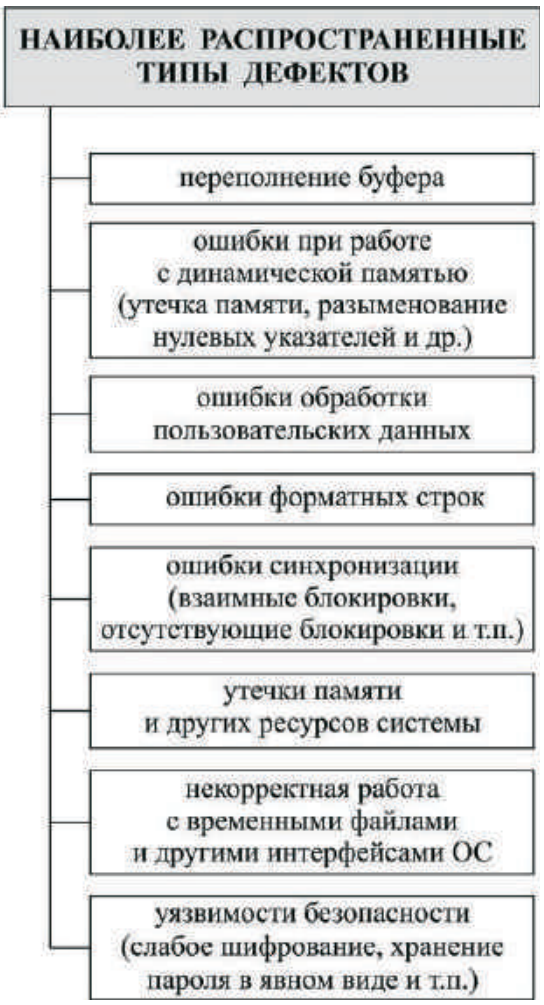

***Рис. 2**. Наиболее распространенные типы дефектов ПО*

Технологии поиска дефектов в исходном коде ПО

Поиск и устранение дефектов в ПО требует больших трудозатрат, при этом многие из них могут остаться незамеченными. По данным исследования, проведенного по заказу Национального института стандартов и технологий США, убытки, возникающие из-за недостаточно развитой инфраструктуры устранения дефектов в ПО (уязвимостей и некритических ошибок), составляют от 22 до 60 миллиардов долларов в год [8], часто являются причиной переноса сроков выпуска программ. Стоимость устранения дефекта, пропущенного на этапах разработки и тестирования, может возрасти после поставки программы от 2 до 100 раз [9].

Как следствие, наибольшее распространение получили методы статического анализа исходного кода ПО, рассматривающие все возможные пути выполнения программы без ее фактического выполнения. Дело в том, что такие методы реализуются в системах, полностью интегрируемые в цикл разработки ПО, применяемые как на этапе тестирования, так и на более ранних этапах – в ходе разработки, причем как во время, так называемых, «ночных сборок», так и непосредственно на аппаратуре разработчика.

Применение систем поиска дефектов (уязвимостей и некритических ошибок) ПО в промышленном масштабе обусловливает следующие *функциональные и эргономические требования*:

минимальные действия пользователя для интеграции инструмента анализа в систему сборки ПО (отсутствие необходимости в изменении, аннотировании, интегрировании в процесс анализа исходного кода ПО);

автоматический поиск дефектов и уязвимостей (без участия пользователя непосредственно в процессе анализа);

масштабируемость анализа (проведение анализа объемом нескольких миллионов строк кода и сотен тысяч функций);

низкий процент ложных срабатываний (значительная часть выдаваемых предупреждений должна быть истинной; приемлемым считается уровень в 30-50% истинных срабатываний, а для важнейших – около 70%);

расширяемость инфраструктуры анализа (дополнение алгоритмами идентификации различными новыми классами уязвимостей (дефектов и критических ошибок);

удобный пользовательский интерфейс просмотра результатов и настройки инструмента анализа.

В свою очередь, описанные требования влекут необходимость реализации следующих технологий системы анализа:

анализ без доступа к полному исходному коду анализируемого ПО (наличие в инструменте анализа внутренних спецификаций, позволяющих описывать действия стандартных библиотечных функций объекта анализа);

выполнение глубокого межпроцедурного анализа (возможность учета влияния разных функций на поведение ПО при поиске заданных ситуаций);

инкрементальный анализ (т.е. анализ только измененной части проекта при наличии полных результатов анализа старой версии проекта);

проведение удаленного анализа, поддержка нескольких разработчиков, ведение истории результатов анализа.

Необходимо отметить, что даже при выполнении всех упомянутых требований методы статического анализа имеют ряд ограничений, не позволяющих в ряде случаев достигнуть высокой точности

анализа. Во-первых, при отсутствии полного исходного кода программы возникает неопределенность, не связанная непосредственно с качеством анализа: в зависимости от свойств недоступного при анализе кода, некоторая операция может как приводить, так и не приводить к ошибке. Например, для библиотечного кода часто возможно построение некорректного вызова из пользовательского кода, приводящего к выполнению некорректной операции в коде библиотеки, но при отсутствии кода этого вызова ошибка в коде библиотеки, как правило, диагностироваться не должна.

Во-вторых, независимо точности статического анализа при обнаружении конструкций, которые могут потенциально указывать на уязвимость, во многих случаях не удается установить, возможен ли в действительности путь исполнения программы и входные данные, приводящие к ошибке. Выдача всех предупреждений в таких ситуациях приводит к тому, что большая их доля оказывается ложной, делая систему статического анализа малополезной для многих приложений.

Наконец, для больших программных систем (в миллионы строк кода) не удается за приемлемое время провести точный межпроцедурный анализ даже имеющегося кода, с учетом необходимости выполнять анализ указателей, интервальный анализ (анализ возможных значений переменных).

Как следствие упомянутых ограничений, промышленные коммерческие анализаторы вынуждены использовать эвристические алгоритмы анализа: при отборе важнейших (с точки зрения анализа) данных о программе среди полученных, при поиске конкретных ситуаций (шаблонов) в потоке данных и управления программы; и последующей выдаче лишь по этим отобранным данным или ситуациям предупреждений о возможных дефектах. Иначе проценты ложных срабатываний инструмента или потребляемые им ресурсы становятся неприемлемо большими. Улучшение алгоритмов анализа или использование других видов анализа, требующих больших вычислительных ресурсов (например, символьного исполнения в комбинации с решателями логических уравнений для отсеивания ложных путей выполнения), повышает точность анализа, но в силу затрачиваемых ресурсов может применяться только к самым важным предупреждениям и сравнительно небольшим частям программы (тысячи строк кода).

Поэтому, в промышленных инструментах анализа возникают ситуации пропуска истинных дефектов как следствие ошибок эвристик, то есть выполняется *нестрогий анализ*. Разные инструменты анализа всегда выдают частично пересекающиеся множества предупреждений для одной и той же программы: часть предупреждений общая, часть – уникальна для каждого инструмента, что обусловлено различием применяемых эвристик.

Всеми заявленными свойствами из коммерческих систем, по всей видимости, обладают системы Coverity Insight [10], Klocwork K9 [11], GrammaTech CodeSonar [12], Svace ИСП РАН [13-15]. Точное суждение об архитектуре и алгоритмах анализа, положенных в основу этих систем, затруднено из-за их закрытости, равно как и сравнение результатов их работы. Проводившиеся в ИСП РАН оценки инструмента Svace на доступном для анализа материале показали качество анализа, сравнимое с остальными коммерческими системами [15].

Существуют и другие классы систем обнаружения дефектов в исходном коде программ, однако точность и требуемые для использования ресурсы ограничивают их область применения (рисунок 3). Данные системы не получили такого распространения, как упомянутые системы автоматического поиска дефектов на основе статического анализа. Из этих классов систем можно упомянуть следующие системы:

автоматизации экспертного аудита;

верификации ограниченного исходного кода;

проверки корректности пользовательских аннотаций.

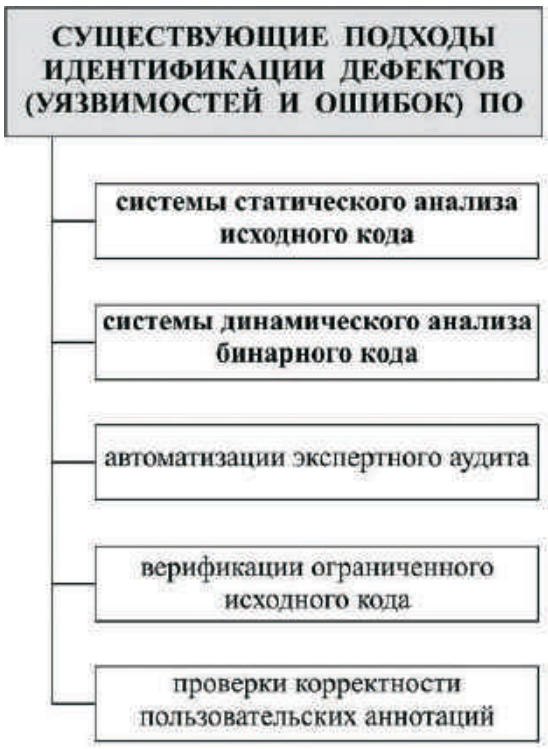

Рис. 3. *– Классы систем обнаружения дефектов в исходном коде программ*

Технологии специализации систем автоматического поиска дефектов в исходном коде программ

Необходимо отметить, что эффективность систем автоматического поиска дефектов ПО зависит от того, на каком наборе тестовых программ нарабатывались эвристики этих систем. По умолчанию эвристики поиска ситуаций в исходном коде, ранжирования собранных данных по важности, параметры точности применяемых алгоритмов настроены на некоторое «среднее» значение поведения анализируемых программ. Так, если программист перед использованием некоторого указателя проверяет его значение на корректность во всех точках программы, кроме одной-двух, то велика вероятность, что и в этих точках программы значение указателя может быть некорректным, и требуется выдать предупреждение о возможном разыменовании нулевого указателя. Конечно, такая эвристика применяется лишь в том случае, если основные алгоритмы статического анализа не смогли с достаточной точностью установить значение указателя.

Следовательно, для специальных классов программ является возможной такая *доработка системы автоматического поиска*, что точность выполняемого системой анализа для этих классов программ повысится. При этом основные виды анализов, выполняемые системой, останутся неизменными.

Перечислим основные способы такой доработки систем автоматического поиска:

определение специфичных для данного класса программ ситуаций, сигнализирующих о дефектах в исходном коде, доработка эвристик системы для выдачи предупреждений о таких ситуациях;

настройка параметров анализа системы или применение дополнительных высокоточных методов анализа (например, чувствительных к путям выполнения) для ключевых, ограниченных по объему участков программы;

ранжирование вычисленной в ходе анализа информации по важности для конкретного класса программы (например, более точное вычисление информации о параметрах вызовов интерфейсов межпроцессного взаимодействия ОС для параллельных программ);

создание спецификаций для используемых программой библиотек, исходный код которых недоступен (как правило, система анализа знает о поведении лишь стандартных библиотек POSIX, C, C++).

Важным специальным классом программы являются системные программы, в частности, ядро и драйверы операционной системы. Как показано выше, объектами воздействия при эксплуатации уязвимостей являются чаще всего именно системные ресурсы, ошибки в привилегированных программах ОС сразу при эксплуатации позволяют добиться необходимой эскалации прав, а также дальнейшего нарушения защищенности информации. Например, при анализе ядра Linux можно создать спецификации для функций выделения и освобождения памяти, используемых в ядре, а также учесть соглашения ядра о возврате кодов ошибок из интерфейсных функций. Для встраиваемых систем, ОС реального времени можно воспользоваться относительно небольшим объемом анализируемого исходного кода и повысить точность алгоритмов анализа. Разработка и реализация таких методов специализации системы автоматического поиска дефектов в настоящий момент является предметом исследований авторов.

Технологии динамического анализа бинарного кода ПО

Системы динамического анализа бинарного кода программ выполняют поиск дефектов (уязвимостей и некритических ошибок) путем генерации различных наборов данных и последующей передачей их на вход исследуемой программе. Возникновение исключительной ситуации, означающей наличие дефекта, отслеживается системой анализа, и текущие входные данные сохраняются для последующего воспроизведения и отладки. Кроме того, собранная информация анализируется на предмет возможности эксплуатации найденной ошибки. Уязвимости, лишь позволяющие провести атаку типа «отказ в обслуживании», считаются менее критическими, чем эксплуатируемые уязвимости, так как существуют технологии, позволяющие найти возможность для эксплуатации уязвимости и автоматической генерации взламывающей программы (эксплоита) [16, 17].

Основными задачами, которые решаются при динамическом анализе, являются: задача генерации наборов входных данных, покрывающих интересующие пути выполнения программы; запуск и трансляция программы; отслеживание возникающих уязвимостей. Часто исследование программы затрудняется применением приемов антиотладки, упаковщиков кода и навесными системами защиты от обратной инженерии. Снятие подобных защит и обход приемов антиотладки является отдельной задачей, поэтому в дальнейшем изложении будем считать, что программа не пытается препятствовать анализу.

Процесс генерации различных наборов входных данных с последующей передачей их программе получил название «фаззинг» (fuzzing), его целью является получение набора данных, выявляющих дефекты работы целевой программы. При этом не все пути исполнения программы представляют интерес, и для более интеллектуальной генерации входных данных требуется учитывать трассу выполнения программы. Для решения этой задачи программа исполняется неким транслятором, который позволяет анализировать пути выполнения, инструментировать исполняемый код или снимать трассу исполнения для последующего анализа.

Попытка сгенерировать все возможные сочетания входных данных приводит к экспоненциальному росту их объема. А значит, от транслятора требуется возможность: производить анализ выбранных или интересующих для анализа путей (например, применяя технологию символьного исполнения для некоторых путей); генерации входных данных, обеспечивающих переход по интересующим путям для увеличения покрытия; возможность параллельного запуска системы с разными входными данными для ускорения анализа.

Примером систем осуществляющих динамическую бинарную трансляцию, являются QEMU и Valgrind. QEMU – эмулятор процессоров и вычислительных систем, способен эмулировать всю вычислительную систему, в этом случае динамической трансляции подвергается код программы, все библиотеки и операционная система. В режиме эмуляции приложения транслируется только код программы и необходимые библиотеки. В свою очередь, QEMU выполняет обработку системных вызовов [18]. Valgrind является инфраструктурой для отладки и профилирования программ, в которой транслируется лишь пользовательская программа в том же окружении, что и при обычном выполнении. Valgrind включает в себя ряд инструментов, реализованных поверх базовой инфраструктуры трансляции, самым популярным из которых является Memcheck, ориентированный для анализа утечек памяти и обращений к невыделенной памяти, позволяет транслировать программу в промежуточное представление, находящееся в SSA-форме, затем код инструментируется и транслируется в машинный код [19].

KLEE – инструмент для символьного исполнения [20], анализ производится над внутренним представлением компиляторной инфраструктуры LLVM [21]. Инструмент позволяет запускать «символьные» процессы, при этом в ходе интерпретации инструкции внутреннего представления LLVM отображаются в систему уравнений, которые затем решаются с использованием инструмента STP [22] для получения новых путей выполнения, которые требуется обойти. Для ускорения анализа система позволяет выполнять несколько путей одновременно.

S2E – система выборочного символьного исполнения, построенная на базе QEMU и KLEE. S2E основывается на двух базовых идеях: выборочном символьном выполнении, позволяющем автоматически минимизировать количество кода, который будет исполнен символьно, и модели консистентности, обеспечивающей при анализе контроль баланса производительность/точность. Ключевые возможности системы заключаются в одновременном анализе нескольких путей, возможности анализа всей системы (программ пользователя, библиотек, ядра, драйверов), возможность анализа бинарного кода [23].

Avalanche – система динамического анализа, разрабатываемая в ИСП РАН [24]. Avalanche решает задачи отслеживания уязвимостей в ходе выполнения программы и интеллектуальной генерации входных данных для увеличения покрытия путей через собственные инструменты на основе инфраструктуры Valgrind и упомянутый решатель STP. Для поддержки языка Java используется статическая инструментация кода, но сохраняется общая схема итеративного динамического анализа.

Mayhem – система автоматического поиска эксплуатируемых уязвимостей в бинарном коде [25]. Каждая найденная уязвимость сопровождается рабочим эксплоитом. Ключевые особенности инструмента: гибридное онлайн-оффлайн исполнение кода, эмуляция на уровне приложения, а не всей системы, набор различных эвристик для работы с символьными указателями. Система основана на инфраструктуре двоичной трансляции PIN [26], обеспечивающей инструментирование бинарного кода, для перевода во внутреннее представление используется BAP [27], в качестве решателя используется Z3 [28].

Таким образом, системы динамического анализа представляют совокупность нескольких модулей-инструментов (зачастую, с открытым исходным кодом), решающих основные сформулированные задачи анализа. Экспоненциальная сложность анализа и применения в случае нетривиальных алгоритмов обработки входных данных и/или модели исполнения (например, обратные вызовы процедур на мобильных платформах) преодолевается новыми эвристиками, позволяющими производить более глубокий анализ программы.

Объекты воздействия на систему при эксплуатации уязвимостей

Применение систем автоматического анализа дефектов (уязвимостей и некритических ошибок) в ПО на основе статического анализа не гарантирует полного их отсутствия. Поэтому требуются также технологии защиты программ и их окружения от эксплуатации имеющихся в программах уязвимостей (рисунок 4). Для описания этих технологий сначала рассмотрим типичные методы воздействия на уязвимую программную систему.

Как правило, большинство эксплуатируемых дефектов программ заключаются в неправильном использовании системных ресурсов напрямую или через интерфейсы ОС. Например, ошибка переполнения буфера (приводит к неверной работе с памятью на стеке или куче), ошибка использования интерфейсов работы с файлами (отсутствие вызовов функций закрытия файлов, параллельная запись в файл без синхронизации) и т.п. Объектами воздействия при эксплуатации взлома, таким образом, являются разнообразные ресурсы ОС, при этом к взлому может приводить как ошибка в системных, так и в прикладных программах.

Соответственно, эффектом воздействия является либо гарантированный крах программы, либо несанкционированное изменение служебных данных системы, приводящее к эскалации прав атакующего, либо нарушение свойств защищенности информации, обрабатываемой программой.

Технологии защиты ПО от эксплуатации уязвимостей

Большинство технологий защиты ПО от эксплуатации уязвимостей направлены *на защиту системных ресурсов*, которые делают невозможным либо затрудненным их несанкционированное использование (рисунок 4). Так, например, защищенные от записи сегменты памяти и запрет интерпретации сегментов данных как кода позволяют избежать некоторых атак, связанных с использованием уязвимостей переполнения буфера. Использование случайных адресов при расположении программы и её библиотек в памяти динамическим загрузчиком.

Тем не менее, возможна также *защита на уровне компилятора и используемых программой стандартных библиотек*, когда уже на этапе сборки программы используется специально подготовленные системные инструменты, и в собранной программе ряд уязвимостей уже невозможно эксплуатировать. Например, защищенный режим FORTIFY_SOURCE стандартной библиотеки GNU C [29] добавляет легковесные проверки на переполнение буфера в наиболее часто используемых функциях работы с памятью и со строками стандартной библиотеки. В случае, когда в ходе выполнения программы размер копируемой памяти или строки превышает размер буфера, программа будет аварийно завершена с выдачей соответствующего диагностического сообщения. Для использования этого режима требуется поддержка компилятора (GCC) и указание специального флага при компиляции. Аналогичным образом могут быть проверены форматные строки на соответствие фактическим параметрам соответствующих функций.

Примером чисто компиляторного подхода к защите программы является вставка проверок корректности индекса массива перед всеми операциями доступа к массивам (подход, аналогичный применяющимся в языках типа Java). К сожалению, использование таких проверок может замедлять выполнение программы в разы.

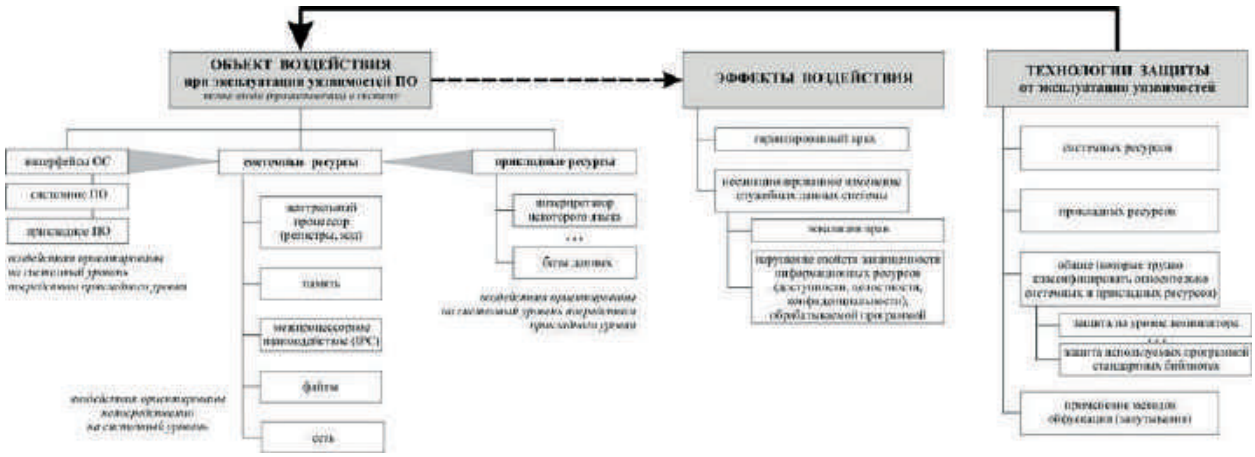

Рис. 4. *Структурно-логическая схема «объект воздействия – эффекты воздействия – технологии защиты»*

Более привлекательным является *применение обфускационных методов*, например, располагающих функции программы при кодогенерации в случайном порядке в создаваемом объектном файле, или вставляющих случайные пропуски в памяти (padding) между локальными переменными функции на стеке. В собранной таким образом программе также затруднена эксплуатация уязвимостей, связанных с некорректной работой памяти, как и при защите от записи и выполнения кода сегментов памяти на уровне ОС. Кроме того, существенным оказывается уникальность каждой бинарной копии распространяемой программы, что не позволяет создать универсальный «взломщик» всех копий программы по результатам анализа одной копии [30].

Наконец, на прикладном уровне возможны специфические технологии защиты, направленные на особенности данной программной системы. Например, при чтении пользовательского ввода база данных может применять фильтрацию потенциально опасных конструкций с учетом знания особенностей языка SQL. Аналогичная фильтрация применяется веб-сайтами при обработке значений, введенных пользователями в формы ввода (например, фильтрация потенциально опасных конструкций языков PHP, JavaScript и т.п.).

Научные результаты получены при продолжении исследований в рамках государственной поддержки Российским фондом фундаментальных исследований и Администрацией Смоленской области инициативных научных проектов № 10-07-97502, 13-07-97518 и Департаментом приоритетных направлений науки и технологий Министерства образования и науки РФ – грантами Президента РФ № МК-755.2012.10, МК-3603.2014.10.

Литература

1. Аветисян А. И. Современные методы статического и динамического анализа программ для решения приоритетных проблем программной инженерии : автореф. дис. … д-ра физ.-мат. наук: 05.13.11/Аветисян Арутюн Ишханович. – М., 2011. – 36 с.
2. Марков А. С. Немонотонные модели оценки надежности и безопасности функционирования программных средств на ранних этапах испытаний // Вопросы кибербезопасности. 2014. №2(3). С. 10-17.
3. Макаренко С. И., Чукляев И. И. Терминологический базис в области информационного противоборства // Вопросы кибербезопасности. 2014. № 1 (2). С. 13-21.
4. Чукляев И. И., Морозов А. В., Болотин И. Б. Теоретические основы построения адаптивных систем комплексной защиты информационных ресурсов распределенных информационно-вычислительных систем: монография / И. И. Чукляев, А. В. Морозов, И. Б. Болотин – Смоленск: ВА ВПВО ВС РФ, 2011. – 227 с.
5. Морозов А. В., Чукляев И. И. Информационная безопасность вычислительных систем боевого управления в аспекте информационного противоборства // Проблемы безопасности российского общества. 2013. № 2-3. С. 85-90.
6. База Common Weakness Enumeration [Электронный ресурс]. – Режим доступа: http://cwe.mitre.org.
7. База Common Vulnerabilities and Exposures [Электронный ресурс]. – Режим доступа: http://cve.mitre.org.
8. Gallaher M. P. and Kropp B. M. Economic impacts of inadequate infrastructure for software testing. Technical report, RTI International, National Institute of Standards and Technology, US Dept of Commerce, May 2002.
9. Forrest Shull, Vic Basili, Barry Boehm, Winsor A. Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fighting defects. In International Software Metrics Symposium. Ottawa, Canada, 2002.
10. Klocwork Insight. Системы анализа исходного кода [Электронный ресурс]. – Режим доступа: http://www.klocwork.com/products.
11. Coverity. Static source code analysis solutions [Электронный ресурс]. – Режим доступа: http://www.coverity.com.
12. GrammaTech, Inc. CodeSonar [Электронный ресурс]. – Режим доступа: http://www.grammatech.com/products/codesonar/overview.html.

References

1. Avetisyan A.I. Sovremennie metodi staticheskogo I dinamicheskogo analiza program dlya reshenia prioritetnih problem programmnoi ingenerii : avtoref. Diss. … d-ra fiz.-mat. nauk: 05.13.11/ Avetisyan Arutyun Ishhanovich. – M., 2011. – 36 p.
2. Markov A.S. Nemonotonnie modeli ocenki nadezhnosti I bezopasnosti funkcionirovaniya programmnih sredstv na rannih etapah ispitaniy // Voprosi kiberbezopasnosti. 2014. №2(3). pp. 10-17.
3. Makarenko S.I., Chucklyaev I.I. Terminologicheskiy bazis v oblasti informacionnogo protivoborstva // Voprosi kiberbezopasnosti. 2014. № 1 (2). pp. 13-21.
4. Chucklyaev I.I., Morozov A.V., Bolotin I.B. Teoreticheskie osnovi postroenia adaptivnih system kompleksnoy zashiti informationnif resursov raspredelennih informacionno-vichislitelnih system: monografia / I.I. Chucklyaev, A.V. Morozov, I.B. Bolotin – Smolensk: VA VPVO VS RF. 2011/ – 227 p.
5. Morozov A.V., Chucklyaev I.I. Informacionnaya bezopasnost vichslitelnih system boevogo upravleniya v aspecte informacionnogo protivoborstva // Problemi bozopasnosti rossiyskogo obshestva. 2013. № 2-3. pp. 85-90.
6. Basa Common Weakness Enumeration [Electronniy resurs]. – Rezhim dostupa: http://cwe.mitre.org.
7. Basa Common Vulnerabilities and Exposures [Electronniy resurs]. – Rezhim dostupa: http://cve.mitre.org.
8. Gallaher M. P. and Kropp B. M. Economic impacts of inadequate infrastructure for software testing. Technical report, RTI International, National Institute of Standards and Technology, US Dept of Commerce, May 2002.
9. Forrest Shull, Vic Basili, Barry Boehm, Winsor A. Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fighting defects. In International Software Metrics Symposium. Ottawa, Canada, 2002.
10. Klocwork Insight. Sistemi analiza ishodnogo koda [Electronniy resurs]. – Rezhim dostupa: http://www.klocwork.com/products.
11. Coverity. Static source code analysis solutions [Electronniy resurs]. – Rezhim dostupa: http://www.coverity.com.
12. GrammaTech, Inc. CodeSonar [Electronniy resurs]. – Rezhim dostupa: http://www.grammatech.com/products/codesonar/overview.html.

13. Аветисян А. И., Белеванцев А. А., Бородин А. Е., Несов В. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды ИСП РАН том 21. 2011. С. 23-38.
14. Аветисян А. И., Бородин А. Е. Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок. Труды ИСП РАН том 21, 2011. С. 39-54.
15. 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 ИСП РАН том 26 вып. 1, 2014. С. 231-250.
16. David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP '08). IEEE Computer Society, Washington, DC, USA, 143-157.
17. T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In NDSS'11, Feb 2011.
18. QEMU – Open Source Processor Emulator [Электронный ресурс]. – Режим доступа: http://wiki.qemu.org/Main_Page.
19. Nicolas Nethercote, Julina Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. Proceedings of ACM SIGPLAN 2007 Conference on Programming Languages Design and Implementation, 2007.
20. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In OSDI, 2008.
21. Компиляторная инфраструктура LLVM [Электронный ресурс]. – Режим доступа: http://llvm.org.
22. Инструмент STP [Электронный ресурс]. – Режим доступа: https://sites.google.com/ site/stpfastprover/STP-Fast-Prover.
23. Инструмент S2E [Электронный ресурс]. – Режим доступа: https://sites.google.com/ site/dslabepfl/proj/s2e.
24. С.П. Вартанов, А.Ю. Герасимов. Динамический анализ программ с целью поиска ошибок и уязвимостей при помощи целенаправленной генерации входных данных. Труды ИСП РАН том 26 вып. 1, 2014. С. 375-394.
25. Sang Kil Cha , Thanassis Avgerinos , Alexandre Rebert , David Brumley, Unleashing Mayhem on Binary Code, Proceedings of the 2012 IEEE Symposium on Security and Privacy, p.380-394, May 20-25, 2012.
26. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In Proc. of the ACM Conference on Programming Language Design and Implementation, Jun. 2005.
27. I. Jager, T. Avgerinos, E. J. Schwartz, and D. Brumley. BAP: A binary analysis platform. In Proc. of the Conference on Computer Aided Verification, 2011.
28. L. M. de Moura and N. Bjørner. "Z3: An efficient smt solver". In TACAS, 2008, pp. 337–340.
29. Расширение FORTIFY_SOURCE стандартной библиотеки Си [Электронный ресурс]. – Режим доступа: https://securityblog.redhat.com/2014/03/26/fortify-and-you/.
30. Иванников В. П., Курмангалеев Ш., Белеванцев А. А., Нурмухаметов А., Савченко В., Матевосян Р., Аветисян А. И. Реализация запутывающих преобразований в компиляторной инфраструктуре LLVM. Труды ИСП РАН том 26 вып. 1, 2014. С. 327-342.

13. Avetisyan A.I., Belevantsev A.A., Borodin A.E., Nesov V. Ispolzovanie staticheskogo analiza dlya poiska uyazvimostei I kriticheskih oshibok v ishodnom kode program. Trudi RAN tom 21, 2011, 23-38 pp.
14. Avetisyan A.I., Borodin A.E. Mehanizmi rasshirenia sistemi staticheskogo analiza Svace detectorami novih vidov uyazvimostei I kriticheskih oshibok. Trudi RAN tom 21, 2011, 39-54 pp.
15. Ivannikov V.P., Belevantsev A.A., Borodin A.E., Ignatiev V.N., Zhurihin D.M., Avetisyan A.I., Leonov M.I. Staticheskiy analizator Svace dlya poiska defectov v ishodnom kode programm. Trudi RAN tom 26, vip. 14, 2014, 231-250 pp.
16. David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP '08). IEEE Computer Society, Washington, DC, USA, 143-157.
17. T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In NDSS'11, Feb 2011.
18. QEMU – Open Source Processor Emulator [Электронный ресурс]. – Режим доступа: http://wiki.qemu.org/Main_Page.
19. Nicolas Nethercote, Julina Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. Proceedings of ACM SIGPLAN 2007 Conference on Programming Languages Design and Implementation, 2007.
20. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In OSDI, 2008.
21. Kompilyatornaya infrastruktura LLVM [Electronniy resurs]. – Rezhim dostupa: http://llvm.org.
22. Instrument STP [Electronniy resurs]. – Rezhim dostupa: https://sites.google.com/ site/stpfastprover/STP-Fast-Prover
23. Instrument S2E [Electronniy resurs]. – Rezhim dostupa: https://sites.google.com/ site/dslabepfl/proj/s2e.
24. Vartanov S.P., Gerasimov A.Y. Dinamicheskiy analiz program s celya poiska oshibok I uyazvimostei pri pomoshi celenapravlennoy generacii vhodnih dannig. Trudi RAN tom 26, vip. 14, 2014, 375-394 pp.
25. Sang Kil Cha , Thanassis Avgerinos , Alexandre Rebert , David Brumley, Unleashing Mayhem on Binary Code, Proceedings of the 2012 IEEE Symposium on Security and Privacy, p.380-394, May 20-25, 2012.
26. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In Proc. of the ACM Conference on Programming Language Design and Implementation, Jun. 2005.
27. I. Jager, T. Avgerinos, E. J. Schwartz, and D. Brumley. BAP: A binary analysis platform. In Proc. of the Conference on Computer Aided Verification, 2011.
28. L. M. de Moura and N. Bjørner. "Z3: An efficient smt solver". In TACAS, 2008, pp. 337–340.
29. Rasshirenie FORTIFY_SOURCE standartnoy biblioteki Si [Electronniy resurs]. – Rezhim dostupa: https://securityblog.redhat.com/2014/03/26/fortify-and-you/.
30. IvannikovV.P., Kurmangaleev Sh., Belevantsev A.A., Nurmuhametov A., Savchenko V., Matevosyan R., Avetisyan A.I. Realizacia zaputivayushih preobrazovaniy v kompilyatornoy infrastructure LLVM. Trudi RAN tom 26, vip. 14, 2014, 327-342 pp.